NTNU – Trondheim
Norwegian University of
Science and Technology

Ray Tracing for Simulation of Wireless Networks in 3D Scenes

Lars Espen Strand Nordhus

Master of Science in Computer Science
Submission date: June 2013
Supervisor: Anne Cathrine Elster, IDI

Norwegian University of Science and Technology
Department of Computer and Information Science

Ray Tracing for Simulation of Wireless Networks in 3D Scenes

In this project, we will investigate GPU-based ray tracers for simulation of wireless networks in 3D scenes. This includes using existing methods to generate a scene based on OpenStreetMaps, and using the NVIDIA OptiX ray tracing framework to simulate wireless network (WiFi) coverage, with a focus towards Intelligent Traffic Systems. The solution should be able to simulate WiFi nodes along the streets and in vehicles in motion.

The goal for the project is to build a wireless simulator, although a fairly simple model for wireless communication should first be considered.

Abstract

Simulating WIFI and other similar radio waves in real-time environments has a tremendous potential, and is a hot topic in modern computer science. The Norwegian Road Authority in Trondheim (Norway) has created a physical test road for simulating future road to vehicle communications in a realistic setting. By doing so they get an excellent physical simulator for realistic small scale testing which can be used to verify a computational simulator. IEEE has created a vehicle-to-vehicle communication standard called IEEE 802.11p: Wireless Access for the Vehicular Environment (WAVE) [1]. On the 24th of April 2013, the cross country cooperation Compass4D met in Denmark, where 90 buses will be equipped and tested using this standard for at least one year [2]. These examples are just some of the many great contributions which are aiming towards developing a technology to better the efficiency and safety of roads.

In this project, we develop a comprehensive real-time large scale WiFi simulator. It simulates vehicle-to-vehicle and vehicle-to-road communication, and can be a good supplement to systems like the ones in Denmark and Norway. To make large scale testing possible and affordable, we have created a way to generate simplified versions of real life streets and structures using Open Street Maps. The goal of this thesis is to make it possible to simulate dynamic traffic environments with communication in real-time.

To achieve this, we harness the compute power of graphics cards which is shown to be extremely powerful in solving massive parallel tasks like ray tracing, the core computational method used in our work. In our case, we use it to trace line-of-sight for the mobile WiFi signals (rather than photon rays). This is done by using the NVIDIA OptiX ray tracing engine for most of the heavy calculations. Using the same framework, we also implement a dynamic environment with both static and moving senders/receivers to illustrate a realistic traffic scenario.

Our system has been tested on several benchmarks to examine how it performs in different scenarios. Our results show that it is feasible to created a system capable of simulating medium resolution scenarios with a great number of senders, buildings and moving obstacles in real-time with a frame rate of at least 24fps. We also show that the number of objects, the resolution and even the number of receivers can be increased substantially when simulating vehicle-to- vehicle communication, since it requires lower update rates. Several ideas for how to extend this work is also included.

Norwegian abstract

Å simulere trådløst nett og andre lignende radiobølger i sanntid har et enormt potensial, og er et populært emne innen moderne datateknikk. I Norge har Vegvesenet laget en fysisk test vei for simulering av fremtidens kjøretøykommunikasjon i en realistisk setting. Ved å gjøre dette får de en god fysisk simulator for realistisk småskala testing, som også kan brukes til å bekrefte en virtuell simulator. IEEE har laget en standard for kommunikasjon mellom kjøretøy som heter IEEE 802.11p: Wireless Access for the Vehicular Environment (WAVE) [1]. Den 24. april 2013 møttes den internasjonale samarbeidsorganisasjonen Compass4D i Danmark, hvor 90 busser vil bli utstyrt og teste denne standarden i hvertfall et år [2]. Disse eksemplene er bare noen av de mange gode bidragene som er rettet mot utvikling av teknologi for å forbedre effektivitet og sikkerhet på veiene.

I dette prosjektet har vi laget en omfattende trådløs simulator med mulighet for storskala testing i sanntid. Den simulerer kommunikasjon mellom biler og mellom veistasjoner og biler og kan være et godt supplement til systemer som de funnet i Danmark og Norge. For å lage storskala testing mulig og kosteffektivt har vi laget en måte generere forenklede versjoner av ekte gater og bygninger ved hjelp av Open Street Maps. Målet for denne oppgaven er å simulere dynamisk trafikk miljøer med kommunikasjon i sanntid.

For å oppnå dette har vi utnyttet regnekraften til grafikk kort, noe som tidligere har vist seg å være ekstremt kraftig for å løse massivt parallelle oppgaver som strålesporing (ray tracing). Dette er gjort ved bruk av NVIDIA OptiX sitt strålespringverktøy for å håndtere mesteparten av den tunge kalkulasjonen. Ved hjelp av det samme rammeverket har vi også implementer et dynamisk miljø med både statiske og flyttbare sendere/mottakere for å simulere et realistisk trafikkbilde.

Vårt system har blitt kjørt gjennom en rekke tester for å utforske hvordan det håndterer forskjellige scenarioer. Våre resultater vise at det er gjennomførbart å lage et system som er i stand til å simulere medium oppløsnings scenarioer med et stort antall sender, bygninger og bevegelige objekter i sanntid, og fortsatt opprettholde hvertfall 24 oppdateringer per sekund. Vi har også vist at antallet objekter, oppløsningen og antall mottakere kan bli økt drastisk når man simulerer kommunikasjon mellom kjøretøy, siden det krever lavere oppdateringshastighet. Flere ideer for hvordan man kan utvide dette arbeidet er også inkludert.

Acknowledgements

I would like to thank Dr. Anne C. Elster for her supervision during this project and for the opportunity to test my code on different high-end systems in the HPC-lab.

I would also like to thank Dr. Jo Skjermo from SINTEF and Erik Olsen from The Norwegian Road Authority (Vegvesenet) for their time spent explaining how the system can be used in real life situations in intelligent road systems.

Further, I would like to thank the other students in the HPC-lab for always helping when needed, and for making the lab a good supporting environment with friendly people sharing knowledge.

Rune Erlend Jensen also deserves a big thank for his help with solving difficult computer problems.

I also wish to thank NVIDIA for their support of HPC-Lab through their CUDA Research Center and CUDA Teaching Center programs.

Finally, I would like to give a special thanks to my two brothers Per Erik and John André Nordhus and my sister-in-law Mari Vårdal for their fantastic support and help on this thesis. You are always there when I need it, and I will never forget it.

Lars Espen Strand Nordhus

Trondheim, Norway, June 11, 2013

Contents

List of figures

1 Introduction

Computers are all around us and are rapidly becoming vital elements in everything we own. There are computers in everything, from your car to your refrigerator, and they are now starting to talk to each other to ease everyday life for the user. This makes the potential of intelligent portable systems tremendous but also introduces several challenges.

One of the main challenges for these mobile devices is interacting with dynamically changing environments in a stable manner. Therefore, many researchers are looking for ways to simulate electromagnetic wave propagation and how the environment will affect the data quality. A common and simplified approach is creating sectors based on predicted signal strengths relative to the sender [3][4]. Studies have also tried to estimate the signal strengths in indoor environment, and by using statistical analysis they have found it possible to estimate the signal strength at any given point within 5dB margin of error [5]. Ray tracing has shown even better results, but the cost of computation increases as well [6][7]. As computers are acquiring the sufficient computing power, ray tracing has become a valid alternative for simulating these kinds of complex scenarios more realistically, and in real-time.

Since graphic cards have proven to be the most effective way to handle these kinds of problems, we chose to use GPUs to accelerate the calculations in this project. This is done by using the NVIDIA OptiX ray tracing engine for most of the heavy calculations. We also implemented a dynamically changing environment with both static and moving senders and receivers to simulate a realistic traffic environment.

We added a city generator to provide the wide range of scenarios possible in the real world. This lets you simulate any given location without having to create all the buildings by hand. By combining Open Street Maps and a python script we were able to automatically generate a city segment of our choice, and import it into the ray tracing simulator.

1.1 Motivation

There are countless numbers of accidents yearly in the world caused by the lack of information and endless miles of traffic jams every day. Many of these can be prohibited by just making the roads smarter. An intelligent road system can inform the driver if the road is slippery, or if a traffic jam will erupt the next mile.

NTNU/SINTEF and The Norwegian Road Authority (Vegvesenet) have both worked on developing driving simulators to test how roads perform before they are built [8]. These two systems are both built with high realism, but need a way to simulate road communication. Originally they use a hard coded distance limit to determine if the signal is reached or not, thus being a less realistic solution.

The Snow Simulator Project [9] was started in 2006 at the NTNU HPC-lab, and has since been the subject of many master thesis and projects. As a result the snow simulator has gotten a substantial speed-up and many extensions, and is now utilizing CUDA and high-end NVIDIA graphic cards [10].

We hope this thesis is the start of a similar long term project. In this case, for auto-generating scenarios and looking at complex wave simulations for WiFi and telecommunication. As a benefit, this can also be combined with the snow simulator to create even more realistic renderings of snow and mountains using the OptiX ray tracing engine.

1.2 Thesis outline

Chapter 1, Introduction:

In this chapter, we give a short introduction to the problem at hand and why we want to solve it.

Chapter 2, Background:

This chapter presents an introduction to the useful background information which is helpful to know when reading the rest of the thesis. The background chapter is divided into subsections where the reader can skip subsections already know at a higher level. Many of the later chapters are based on the information found in this chapter. The chapter also looks more thoroughly at the combination of ray tracing and wave simulation.

Chapter 3, Implementation:

This is where we present our implementation of the system in more detail and some notes on what we found when programming the solution. But first we give a quick walk-through of how the system piece together and which parts of the system we have created.

Chapter 4, System scalability test:

This chapter explains the strengths and weaknesses of the simulator and how we tested to find these.

Chapter 5, Conclusion and future work:

The thesis finishes of with a summary of the results and facts found during this project, before exploring some thoughts for further extensions and possible optimizations.

Appendix:

At the end of the thesis you can find a guide for how to use the system, how to install the system, and the raw test results.

2 Background

Simulating WiFi signals in a realistic and efficient manner is a complex problem. Therefore we have added some background material on how electromagnetic waves and image rendering work, before we look at how these two can be combined. This chapter also contains an introduction to the state of the art research done in the field.

2.1 Rasterization

real-time graphics is often defined as an image stream with at least 24 frames per second (FPS in shortened form). Around this point the eye "stops" seeing the change of pictures and starts seeing a more constant flow like in a film. 24 FPS is a demanding requirement to maintain when the level of detail increases. For games where the player can "feel" the latency through game-play the requirements set to the update speed are often even higher.

Rasterization is currently the most popular technique for producing real-time 3D in the world of computer graphics [11]. It is a fairly fast sequence of algorithms which transform a three-dimensional scene, often composed of triangles, into a two-dimensional image of pixels. At run time, rasterization processes the individual primitives through the rendering pipeline and thereby extracts the 2D forms appearing on the screen [12].

This pipeline contains many different elements that have been well optimized and accelerated using the specialized hardware on GPUs. The rasterization process contains the following five algorithms: Clipping, perspective division, back-face culling, viewport transform, and scan conversion.

Clipping selects the parts of the scene that is placed fully or partially inside the view frustum and is implemented in hardware.

Perspective division scales the models to create the illusion of distance. Objects fare away gets scaled down thereby making objects close by look larger in comparison.

Back-face culling can remove the backside of objects if no reflection is planned, thereby saving much rendering.

Viewport transform changes the coordinate system to fit the viewport. This is done by reflecting the scene across the XZ-plane, scaling the image to fit the chosen resolution, and finally finishing off by translating the scene to fit the screen coordinate system where 0,0,0 is in the top left corner.

The final part of rasterization is **scan conversion**, where the primitives are drawn using pixels following a given algorithm.

Systems where the real-time requirement is absent is often referred to as being "pre-rendered" or "offline rendered" and make use of more time-consuming algorithms to achieve better qualities like ray tracing, radiosity and photon mapping.

2.2 Ray tracing

Whereas illumination models like "Flat shading", "Gouraud shading", "Phong shading" and the simplified version of ray tracing called ray-casting are all examples of "local illumination rendering methods", ray tracing is a "global illumination rendering method" [13]. A global illumination rendering method does not only use lights to specify the brightness of a surface, but also take into account the reflection of light emitted by surrounding surfaces. This creates a much more realistic representation of the given scene, but comes at the cost of rendering speed.

Ray tracing is often used when creating 3D-animated movies [14] [15] [16] [17] [18] and have also in recent years become a valid option for use in real-time graphics games like Quake 3 and 4 [19] [20]. This is mainly due to the improvements done to graphic hardware and the massively parallel nature of the ray tracing algorithm. The research done on the subject has also played a significant roll in the progress and still new fields of research look into alternative use-cases for ray tracing. One alternative application for ray tracing is to use it for displaying X-ray data, which is extremely complex and computationally expensive [21].

To calculate shadows, lighting, transparency and reflections realistically in a three-dimensional scenario is a challenging problem. Ray tracing solves this problem by simulating the movement of light through the scene simulating a chosen set of light-rays [22]. This is done by casting a ray from a given camera point and through every pixel on the wanted screen. If a ray does not hit an object within a limited distance on its journey through the scene, a default colour or texture is returned and set as the colour of the given pixel. If the ray hits an object, the colour of the position is noted just as with ray-casting. But this is where the algorithm gets advanced. Instead of just returning the colour of the surface the algorithm spawns a set of new recursive rays that are going to simulate the different effects mentioned earlier.

A shadow ray is spawned for every light source and is traced to determine if the given source contribute with light of any colour or intensity. If the ray collides with a non-opaque object the ray terminates and no light is contributed by it. If the ray hits the light source, the colour and intensity produced taking into account the distance and potential opaque object passed through is returned and combined with the results of the other rays spawned.

A number of reflection rays are spawned. The number and angle are depending on how defuse the object is supposed to be, for example a cone of rays can be sent to make a smooth matt finish. These new rays are all recursive and behave like the original rays sent from the camera.

To simulate the transparency of objects a final ray is spawned when an intersection is found. The ray is sent through the object in a calculated angle given that the surface is opaque enough. This gives a new recursive ray just like the ones coming from the camera.

When all the rays have terminated either by hitting the scene walls or by reaching the maximum number of recursions, the rays and the surface colour are scaled, summed, and set as the pixel colour of the original pixel on the screen.

The ray tracing algorithm scales exceptionally well with the number of vertexes in the scene compared to for example rasterization [23]. In best case it scales logarithmically with the number of vertexes, and this is very important when we look at the development of 3D scenes where both the size and the number of details tends to increase.

In many cases, increasing the number of GPUs has also proven to scale excellent. H. Ludvigsen and A. C. Elster [24] found that as long as the computational complexity outweighed the cost of transferring the data, great speed-ups may be achieved.

2.3 Radio waves

To get at complete picture of the problem of simulating electromagnetic waves like in WiFi, we first need to look at how radio waves behave in the real world. When waves interact with the environment three distinct effects occurs: reflection, refraction and diffraction. These effects are described in more detail in the subsections below [25].

2.3.1 Reflection

When a wave encounters a surface, a part of the wave gets reflected. A similar effect is seen when playing pool and a ball hits the table wall. Some of the energy of the ball is lost due to the material absorbing some of the energy. In addition to this effect, a wave hitting a material also passes a portion of the wave through the material, dependent on the density/opacity of the material. This effect is like a glass window which is both reflective and transparent at the same time.

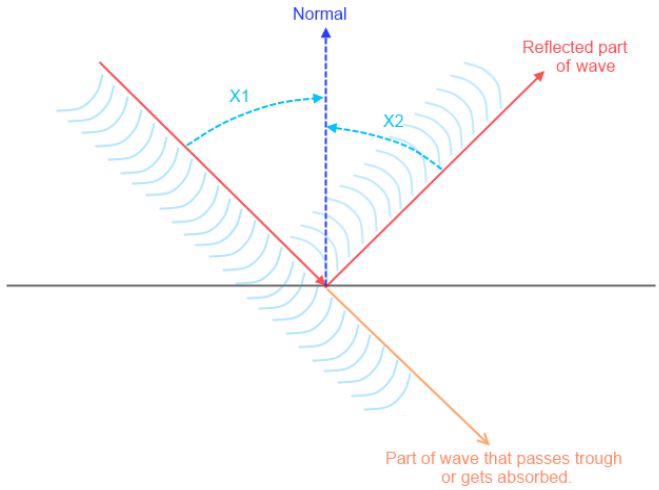


Figure 1: Illustration of how parts of the wave get reflected, absorbed or pass through the object it hits.

As you see in Figure 1, the angles x1 and x2 are always the same for wave reflection, just like a perfect mirror. The amount of energy absorbed by hitting a surface depends on the material of the surface and often (in the real world) the amount of water in the material.

2.3.2 Refraction

Refraction comes to play when a wave moves into a material with another density. This is shown in Figure 2. This effect is utilized in lenses to bend the light in the desired direction. According to Snell's Law [13] shown in (1), the relationship between the variables is constant.

$$\frac{refractive\,index1}{refractive\,index2} = \frac{velocity1}{velocity2} = \frac{\sin(angle\ of\ incidence)}{\sin(angle\ of\ refraction)} \quad (1)$$

$$R = \frac{n1 * I}{n2} - normal * \left(\frac{n1}{n2} * (normal * I) + \sqrt{\left(1 - \left(\frac{n1}{n2}\right)^2 (1 - (normal * I)^2)\right)}\right) \quad (2)$$

This fact can be used to calculate the wanted angle of refraction. With some reorganizing of (1), the formula can be written like in (2). R is short for refraction vector, I for incidence vector, n1 and n2 refers to refractive index 1 and 2 respectively.

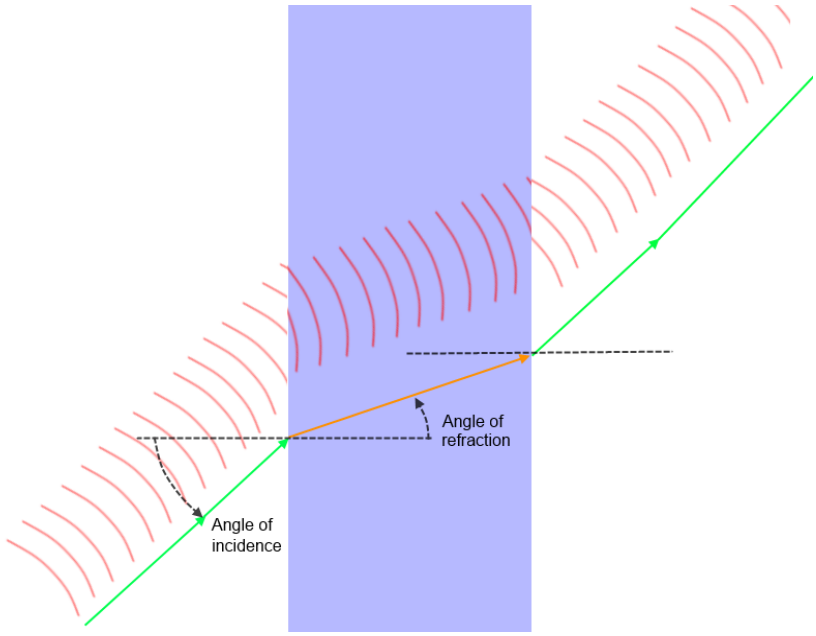


Figure 2: Waves going through a medium with a higher density.

2.3.3 Diffraction

A wave diffract when it passes an object or through a small opening. This effect is seen when a wave enters the inside of a breakwater and spreads out. One of many exploitations of diffraction is to send communication and TV signals into valleys which have no line of sight to the sender like shown in Figure 3.

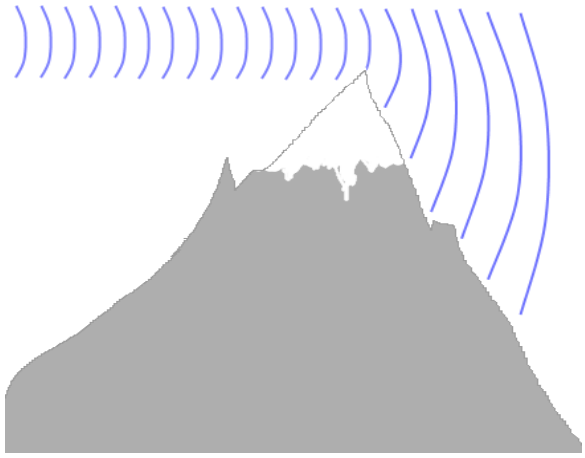


Figure 3: Diffraction of waves passing a mountain.

2.3.4 Simulating electromagnetic waves using ray tracing

In this thesis, we look at simulating the electromagnetic waves sent from a WiFi sender and received by a mobile receiver using ray tracing. Although both ray tracing and electromagnetic waves are based on the concept of wave physics, the wave length of ray tracing is much smaller in comparison. As a result of this, ray tracing does not simulate diffraction. This is estimated to have a minor impact on the outcome of this simulation since the wavelength of the IEEE 802.11p standard (specified for vehicle-to-vehicle communication) is quite small. The extension is mentioned as potential improvement in further work 5.1.4.

Visual light	390nm – 750nm	790THz - 405THz
IEEE 802.11p	51282 µm - 50632 µm	5.85 GHz - 5.925GHz
FM radio	2,8 m – 3,4 m	Usually 87.5 to 108.0 MHz
TV and AM radio	higher wave lengths	even lower frequencies

Figure 4: Comparison of commonly used wave lengths and IEEE 802.11p (vehicle-to-vehicle communication).

2.4 Acceleration structures

To ray trace a large scene in high resolution, without doing some sort of optimization, will result in a pretty slow system. Turner Whitted, the creator of ray tracing, stated that he used up to 95% of the checking for intersections between rays and objects [22]. As a result of this, much research have been done on optimizing this search [26] [27]. These algorithms scale logarithmically at best with the number of vertexes, which is exceptionally well compared to rasterization. A simplified version of the two most commonly used acceleration structures is therefore explained in the following subsections.

2.4.1 Bounded volume hierarchy

The goal of this algorithm is to create a hierarchy of bounding volumes which can help to reduce the time spent testing for intersections. When the hierarchy is built, you should be able to calculate the intersection point with any object by just using the process of elimination. This is possible since you know that no intersection is possible with objects within a box if there are no intersections with the outer box in the first place. And by using boxes or other simple structures as binding volumes the time used to calculate if an object is hit or missed is reduced substantially.

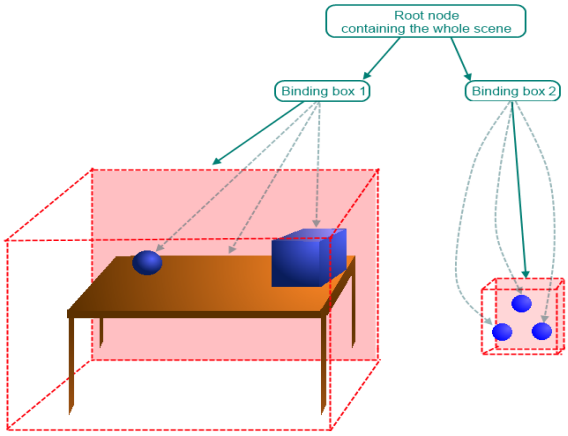


Figure 5: Illustration of a scene with two binding boxes. This Figure is inspired by [26].

When creating a bounding volume hierarchy, you first start with a scene of objects as the root in your tree structure. Part one of the algorithm is to split the scene into large boxes containing collections of objects. These boxes should not intersect with each other. This way, objects closer together will be in the same box. These boxes are then added to the root node as children. The scene is now split into smaller scenes/boxes and can be reduced like this recursively until a goal condition or the recursion depth is met.

2.4.2 Kd-tree

The goal of this algorithm is to create a hierarchy to reduce time spent testing for intersections [27]. The essential difference between creating a bounded volume hierarchy (bvh) and a kd-tree is that the kd-trees use splitting planes, not bounding volumes.

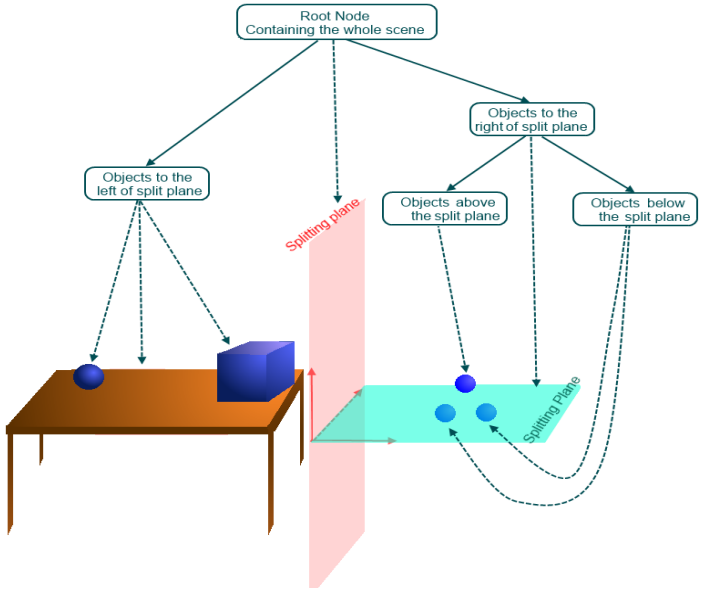


Figure 6: Illustration of a scene with two splitting planes. This Figure is inspired by [26].

To create a Kd-tree you first start with a single node containing the whole scene and all the objects within it. Then you split the scene in two with a plane, save the splitting plane to the node and create two child nodes. All objects to the left of the split are moved to the left child node and the objects to the right are moved to the right child node. This split is done recursively on the new child nodes until the goal conditions are met or the recursion depth is reached.

2.5 The background of NVIDIA OptiX

In September 2009, NVIDIA released its ray tracing engine named OptiX [28]. It uses the NVIDIA CUDA GPU computing architecture for GPU programming and can therefore only be run on NVIDIA hardware. OptiX has proven to be a good ray tracing engine and have the capability of using multiple GPUs simultaneously [24]. It uses Glut and OpenGL to display the output on screen.

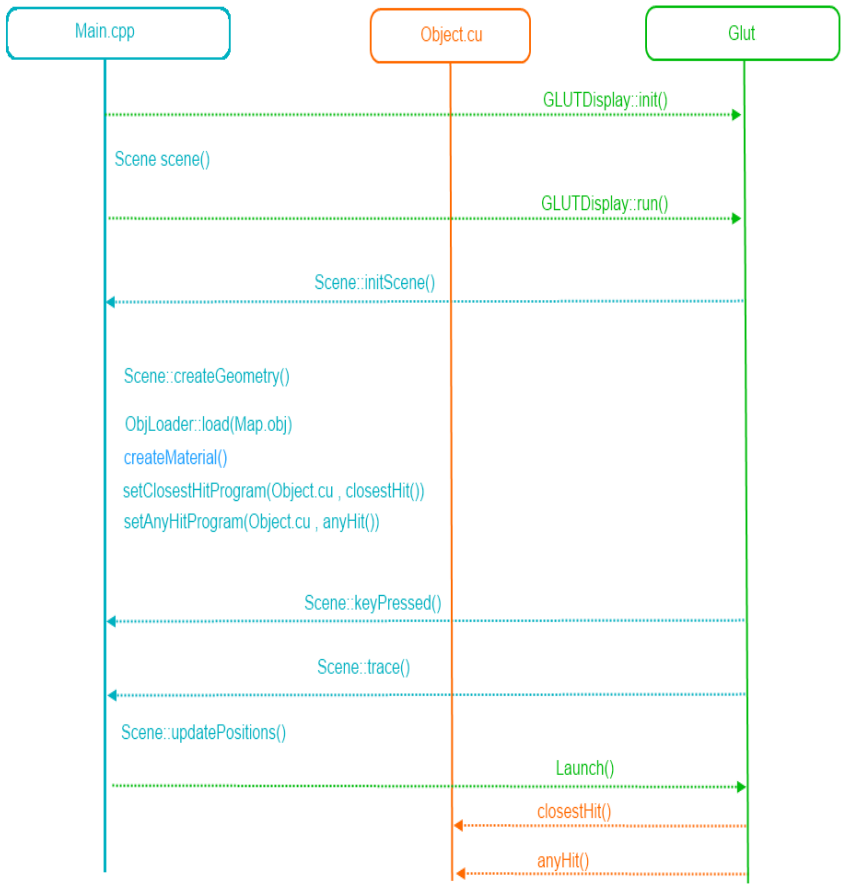


Figure 7: Illustration of the call sequence in OptiX at .

Figure 7 illustrates the interaction made in a typical OptiX program. A material can be created for each object to specify its behaviour. A range of settings can be linked to the material to simulate effects like reflection, refraction, colour, shadow and lighting. By for example setting the closest hit program, the programmer may choose what action OptiX should take when the selected object intersects with a ray in the future. This is shown in the lower right corner, where Glut calls the closestHit program of the hit object. ObjLoader::load() lets us load objects from file and place them in the system hierarchy. The Launch() command starts the process of rendering a new frame.

2.6 The Obj format

This format consist of a list of textures, vertices, lines and normals [29]. This list can be translated into objects in systems like OptiX and 3D-Studio Max. Figure 8 shows how the objects are split into four lists. Each vertex, normal and texture then only needs to be listed once, even if used to create numerous polygons. Each polygon is simply a combination of the vertices, textures and normals listed earlier in the format: vertex/texture/normal.

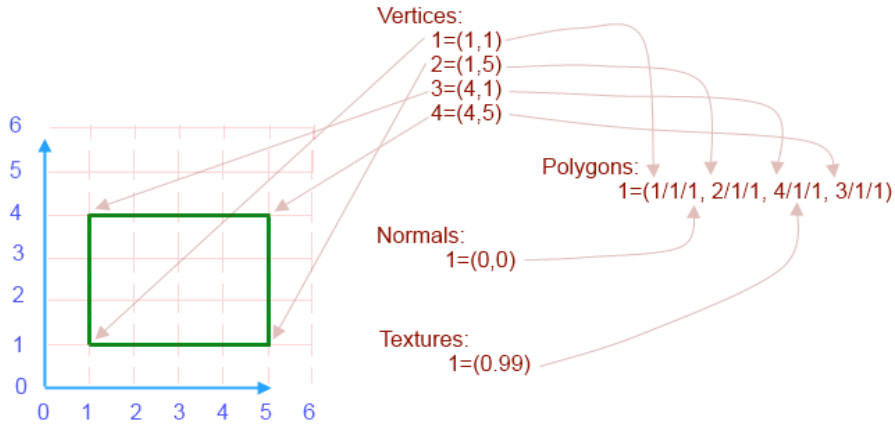


Figure 8: Example of a polygon represented in an object-file.

2.7 State of the art

Much research has been done in the recent years on the related subjects, and we have therefore chosen to give a quick walk-through of how far research has come in this field.

2.7.1 Automatic city generation

The techniques of generating cities has come far and is essential in many systems like games, films and simulators where large scenarios will have to be created [30]. As a result of the users' expectations for the quality of 3D modelling to always improve, the creation of a realistic 3D environment may take years to create, even for hundreds of artists. This is extremely expensive and can be improved substantially by automating this process.

Georges Nakhle [31] has produced an advanced city generator, which automatically creates a three-dimensional map of buildings based on GPS information derived from OpenStreetMaps.org. He has also created a building generator which focuses on creating more detailed buildings [32]. In [33], Claus Brenner implemented an algorithm for automatically reconstructing buildings based on digital ground planes and laser scanned digital surface models.

David Gary [34] has also created an advanced city generator, which focuses more on creating detailed cities. This implementation depends on users placing the buildings, setting the parameters, and is therefore not as automatic as the others mentioned above. Pearl Goswell and Jun Jo [35] researched the possibility of creating a grammar-based real-time city generator. Their system utilized shape grammar rules to automatically generate buildings with high levels of variety in assets, while keeping the memory cost low. These two generators create new fictive structures instead of deriving data from an external map.

2.7.2 Intelligent Transport Systems

The construction of intelligent transport systems (ITS) have in the recent years become a hot field of research. Some of the many uses for ITS is shown in Figure 9.

Emergency warning system for vehicles
Cooperative Adaptive Cruise Control
Cooperative Forward Collision Warning
Intersection collision avoidance
Approaching emergency vehicle warning (Blue Waves)
Vehicle safety inspection
Transit or emergency vehicle signal priority
Electronic parking payments
Commercial vehicle clearance and safety inspections
In-vehicle signing
Probe data collection
Highway-rail intersection warning
Electronic toll collection

Figure 9: Some of the many uses for ITS [36].

The Norwegian Road Authority in Trondheim (Norway) has created a physical test road for simulating future road-to-vehicle communications in a realistic setting. This test road is based on cooperative vehicle-infrastructure systems (CVIS [37]) and is an excellent simulator for realistic small scale testing [38]. Likewise the cross country cooperation Compass4D met in Denmark on the 24th of April 2013, where 90 buses will be equipped and trailed for at least one year.

ERTICO - Intelligent Transport Systems and Services for Europe [2] is also a cross country cooperation where the United Kingdom - Department for Transport and Norwegian Public Roads Administration are two of the over 100 members. Their goal is to have zero accidents, zero delays, reduced impact on the environment, and fully informed people driving the roads by utilizing technology.

In [39], they have implemented a simplified open source two-dimensional simulator with vehicle-to-vehicle and vehicle-to-road communication which realistically simulates traffic. Although using a fairly simplified transmission range simulator, their system has an advanced traffic simulator and proved to scale excellent.

A vehicle-to-vehicle communication standard called IEEE 802.11p: Wireless Access for the Vehicular Environment have been created in July 2010 to hinder miscommunication [1]. It is created and maintained by the IEEE LAN/MAN Standards Committee as part of the IEEE 802 standard. According to [40], this standard appears to be a great basis for future traffic communication systems. The article also points at the need for improvements to secure a controlled network load for safety applications.

2.7.3 Optimizing Ray tracing

Since ray tracing was first described in 1980 [22], the run time has gone from taking days to calculate down to potential real-time rendering. This improvement is much related to the improvement in graphic hardware, which specializes in solving massively parallel problems like these. Another reason for the drastic improvement in run time is the implementation of for example acceleration structures like mentioned in Section 2.4. Some of the newer optimizations look into suboptimal solutions through progressive rendering [41].

2.7.4 Alternative ray tracing engines

In addition to NVIDIA OptiX, there are other highly optimized ray tracing implementations out on the market. Some alternatives are: RTSL, Rtfact, RTRT/OpenRT and OpenRL.

OpenRL is an open source ray tracing library created by Caustic Professional which is a division of Imagination Technologies [42].

The RTRT/OpenRT ray tracing engine was created by The Saarland University in 2001 [43]. In 2008 this open source engine was used to implement a ray traced version of Quake 4 which proved to scale by a factor of 15.2 when run on 16 cores [44].

The ray tracing shading language RTSL was built in 2007 as an open source project and builds on the GLSL language that is a part of the OpenGL specification [45]. RTSL have shown the potential to compete with hand optimized ray tracing implementations.

Rtfact is a template library consisting of packet-centric components and was created in 2008 [46]. Unlike the others, **Rtfact** uses only the CPU to accelerate the ray tracing and provides the building blocks for creating custom ray tracing-based solutions instead of supplying a stand-alone ray tracing engine.

3 Implementation

As part of this project, we have implemented a simulator based on the background found in the previous chapter. This simulator is capable of simulating vehicle-to-vehicle communication in real-time, and includes movement, multiple cameras, reflections, refraction, illumination, 360 degree vision, and output analysis. In addition to creating the simulator, we also established ways to obtain or alter test scenarios, and ways to handle the output. In the following section, the data-flow of the system will be explained followed by a more detailed explanation.

3.1 Information flow

In this thesis, the system is split into smaller and more standardized parts which follow known formats to stay as dynamic as possible. To get a better understanding of how the different pieces of the system work together, a quick overview will be given before the next sections go into more detail. This section will describe the flow of information through the system from the online map on OpenStreetMaps.org to the final product shown on the screen.

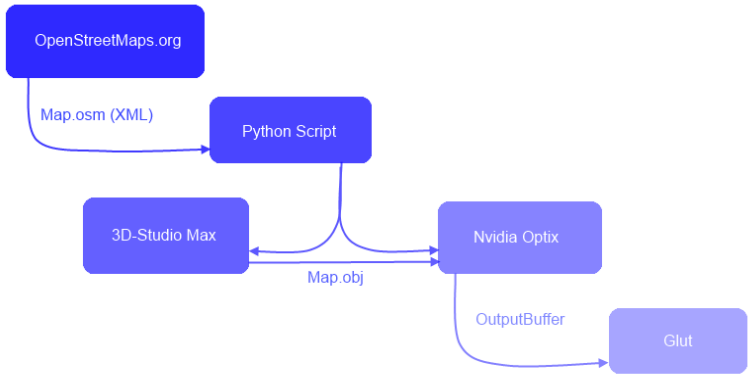


Figure 10: Information flow through the simulator system.

3.1.1 OpenStreetMaps.org

OpenStreetMaps.org is a free to use online map database that gives you the possibility to export information of a selected environment of your choice with ease [47]. The data is then saved in an osm file, which is formatted using XML syntax [48]. Unfortunately, OpenStreetMaps.org calculates the altitude data using GPS, which is lacking precision, and is only saving the data for road elevation and not the terrain. This comes to show especially in areas without drivable roads, like in between large buildings. The data is so corrupt in some cases, that it gives better results ignoring the altitude data ending in one flat area.

3.1.2 XML translator

When the XML formatted map is derived from OpenStreetMaps.org, we extract the needed information. In our case, it is of how tall a building is, its shape and where it is located. This is done using a python script based on Georges Nakhles tutorials [31] which imports an osm file, and translates it to a recipe for constructing the given buildings in a 3D environment. The python script outputs this build recipe to an obj file like described in Section 2.6. All the buildings created using the script are set to the same height and the floor is not modelled. This is done to conserve resources while preserving the details that are important for the simulation. This scripts was provided through doctor Jo Skjermo.

3.1.3 Implementations made using 3D-Studio Max

We used the 3D-modelling program 3D-Studio Max to import the obj file produced in the previous section to swap the axis for a better fit of the OptiX standard coordinate system. We also added a flat ground floor to simulate the ground of the city while not increasing the number of vertices in particular.

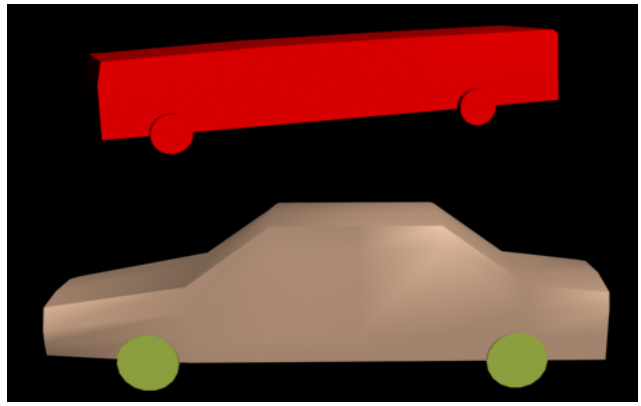


Figure 11: A car and a bus created using 3D-studio Max to simulate traffic.

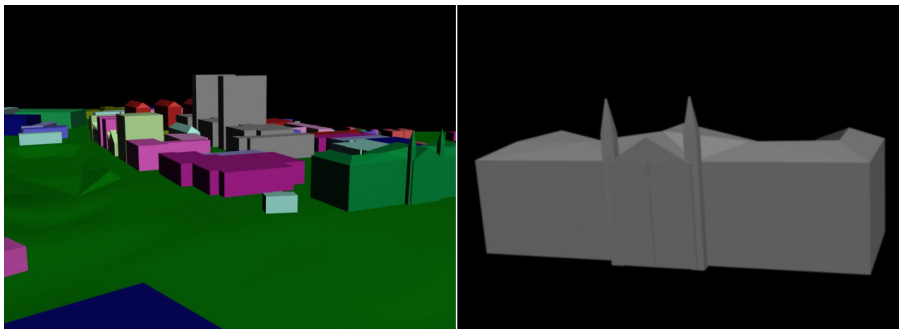


Figure 12: NTNU Gløshaugen and the main building modelled in more detail.

To improve the scene, we made a more realistic testing scenario with more detail and differences in elevation. Cars and buses were also made and exported to improve the simulation even further. This is a very important aspect since traffic itself can prove a hinder for vehicle-to-road communication. These new objects follow the same physics as the buildings when it comes to reflection and refraction, but can in the future get their own refraction index to simulate the difference in density.

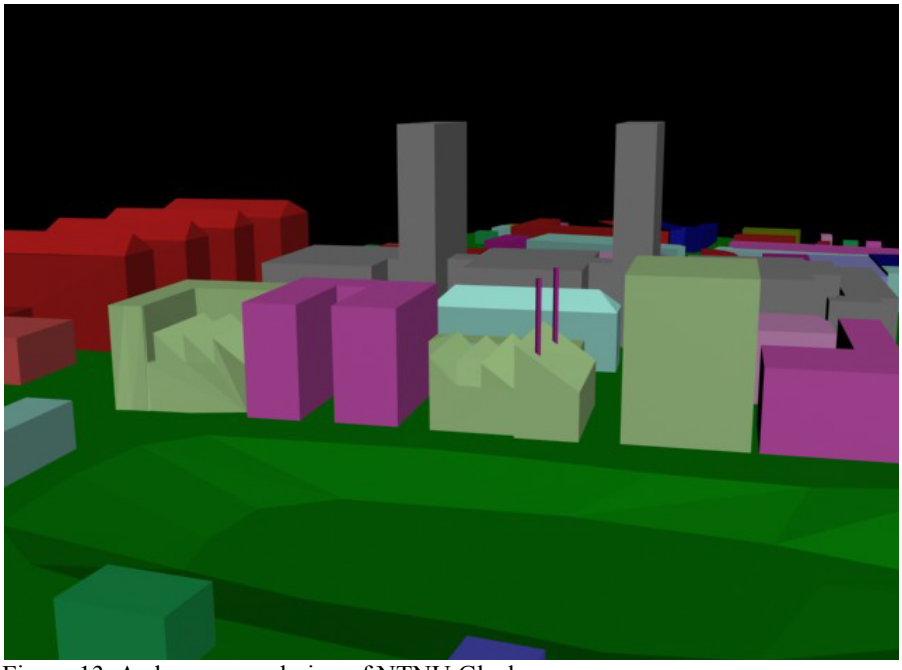


Figure 13: A close-up rendering of NTNU Gløshaugen.

3.1.4 Reading input using OptiX

To create useful 3d models from the given obj file we created a file-reader, which read useful 3d models and outputted a list of OptiX commands. This reader was created by first reading the file, then interpreting and storing the data in lists of the given type. Since the buildings used more vertexes than needed the list was optimized to only contain the minimum number of vertices per surface. These lists where then used to create a set of OptiX commands for building the read structure.

The negative consequence of reading and assembling the structures one surface at the time is that it would create a large amount of individual surfaces which would have to be checked separately for collision. We would then have a suboptimal solution, but with fewer vertices than originally. Luckily we found a function within OptiX which read the obj files and outputted a complete model structure, resulting in a radical reduction in number of objects thus giving a much better solution.

3.1.5 The simulation output from OptiX

The ray tracer uses the object given as input and outputs a buffer containing the colour information for each pixel on the screen. The buffered colour information, which can be thought of as a picture of a landscape seen through a camera lens, is what we are interested in. By searching for the brightest point in the picture we find the precise signal strength of the radio, or in this case the light intensity hitting the receiver. This maximum value can be used as input for another system to act as a real-time WiFi signal simulator.

A side effect of the simulation is an image stream which can be displayed on screen using GLUT. This makes it easier to find where the strongest signal comes from to potentially send the reply in this direction, amplify the signal strength or to do other optimizations.

3.2 Implementing ray tracing using OptiX

To simulate the electromagnetic waves from a sender to a receiver realistically many factors needed to be taken into account. As said in Section 2.3.4 we have chosen to use the NVIDIA ray tracing engine since it is a great tool for realistic simulations of light in real-time. The jump from short wave electromagnetic waves to light is luckily not very far and we were able to mimic most of the effects.

3.2.1 View area

To simulate a radio antenna, our system needed to be able to receive signals from all possible directions. Achieving this using a camera was no easy case, but we ended up with distributing the rays in both axes equally throughout the 360 degrees view, like shown in Figure 14.

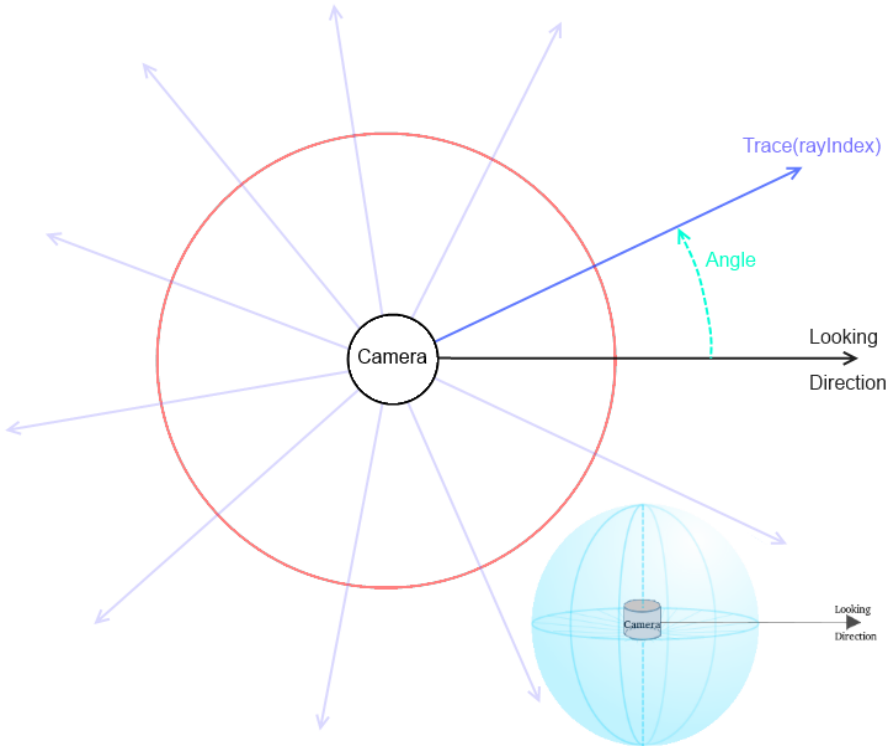


Figure 14: 360 degree camera, where every ray is traced in the direction calculated according to the given ray-index.

As this is done in massive parallel, some calculation needs to be done to estimate the direction of every single ray. The CUDA method for starting and initializing a single ray is shown in Figure 15. The variable launch_index contain two integers which indicate the index of the given thread in the two-dimensional array. These values range from zero to the wanted camera output width and from zero to the wanted output height respectively.

```
RT_PROGRAM void env_camera(){
  float2 d = launch_index/launch_dim*make_float2(2.0f*PI,PI)
                                                  +make_float2(PI, 0);
  float3 angle = make_float3(cos(d.x)*sin(d.y),-cos(d.y), sin(d.x)
                                                       * sin(d.y));
  float3 ray_direction = normalize(angle.x*normalize(U[cameraIndex])
                               +angle.y*normalize(V[cameraIndex])
                               +angle.z*normalize(W[cameraIndex]));
  float3 ray_origin = eye[cameraIndex];
  optix::Ray ray(ray_origin, ray_direction,
                          radiance_ray_type, scene_epsilon);
  PerRayData_radiance prd;
  prd.importance = 1.f;
  prd.depth = 0;
  rtTrace(top_object, ray, prd);
  uint2 LI = make_uint2(launch_index.x+launch_dim.x*cameraIndex ,
                                                    launch_index.y);
   output_buffer[LI] = make_color( prd.result );
}
```

Figure 15: The code for handling a single ray.

The value *launch_dim* is set as the wanted camera output width and height. The eye, W, U, and V variables are used to pass current camera info to the ray generation program at render time. The four buffers all contain a three float vector for each of the cameras to be displayed. Eye contains the position of the camera. W is used to define the viewing direction where the length of W is set as the focal distance [49]. The focal distance of the camera is an estimate of how far away the lens has to be for all light to hit the focal point and is shown in Figure 16.

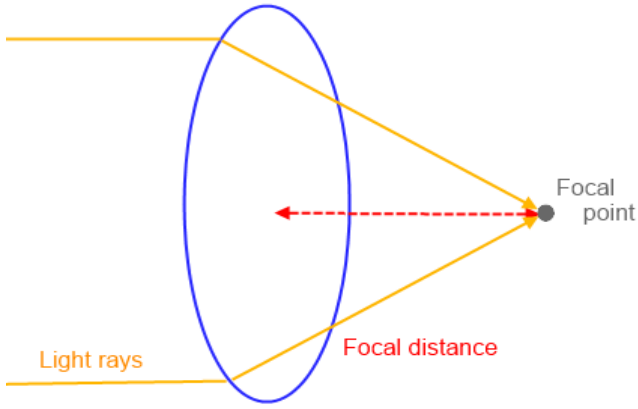


Figure 16: The focal distance and focal point of a lens.

The U vector describes the horizontal axis of the view plane, with the length of U being the width of the view plane at focal distance. V is the vertical equivalent of U where the length describes the height of the view plane at focal distance.

3.2.2 Our implementation of multiple cameras

To simulate large scale vehicle-to-vehicle communication we implemented the possibility for multiple cameras at the same time. By repeating the command for launching a set of rays like shown in Figure 17, we can simulate as many cameras as we want. It does not seem like NVIDIA have intended for OptiX to be used in this way since ray tracing tends to be complex enough on its own. Therefore we had to write the return values for all cameras to the same array, but with an offset dependent on the given *cameraIndex*. By sending the camera data in arrays instead of single values and with the help of the camera index, we are able to change the variable sets used without having to send new information when changing camera.

```
for(int i = 0; i<numberOfCams ;i++) {
       camUpdate(camera_data,i);
       m_context[ "cameraIndex" ]->setInt(i);
       m_context->launch( i, m_WIDTH,  m_HEIGHT  );
}
```

Figure 17: Code for launching multiple cameras.

3.2.3 Our implementation of reflections and transparency

As mentioned in Section 2.2 there are many ways to implement ray tracing. Since we in this thesis look at simulating WiFi communication, we chose to implement mirror-like reflections (which act like wave reflection found in Subsection 2.3.1) in combination with refraction.

```
RT_PROGRAM void closest_hit_radiance(){
  float3 result = make_float3(0.0f);
  if( prd.depth < 10 ){
    float reflectRefractRatio = 0.3f;
    float3 hit_point = ray.origin + t_hit * ray.direction;
    float3 world_geometric_normal =normalize( rtTransformNormal(
                              RT_OBJECT_TO_WORLD,geometric_normal));
    float3 dir = ray.direction;
    PerRayData_radiance new_prd;
    new_prd.importance = prd.importance;
    new_prd.depth = prd.depth + 1;

    if(dot(world_geometric_normal,dir)<0){ //if outer surface;
         Reflect
         const float3 refl = reflect( dir, world_geometric_normal);
         const optix::Ray refl_ray = optix::make_Ray( hit_point,
                                    refl, 0, 1e-3f,RT_DEFAULT_MAX );
         //cast a reflection ray in this direction
         rtTrace( top_object, refl_ray, new_prd );

         result = (1-reflectRefractRatio) * new_prd.result;
    }
    // refraction
    float3 t; // transmission direction
    float refraction_index =  1.1f; //n1/n2
    if( refract(t, dir, world_geometric_normal, refraction_index) ){
         float cos_theta = dot(dir, world_geometric_normal);
         if (cos_theta < 0.0f)
              cos_theta = -cos_theta;
         else
              cos_theta = dot(t, world_geometric_normal);

         optix::Ray ray( hit_point, t, 0, scene_epsilon );
         PerRayData_radiance refr_prd;
         refr_prd.depth = prd.depth+1;
         rtTrace( top_object, ray, refr_prd );
         result += reflectRefractRatio * refr_prd.result;
    }
  }
  prd.result = result;
}
```

Figure 18: The code for handling reflection and refraction.

By using the variable *reflectRefractRatio* to scale the two separate effects, we are able to combine both reflection and refraction in the simulator. One ray is sent into the material and one is reflected. This also takes into account some loss of intensity for every time a light is reflected or seen through an object. To simulate the loss in signal strength when sending waves across long distances we also subtract a portion of the light. This is done as the last action before the ray returns the found signal strength. The exact formula for simulating the loss of signal strength in vacuum is to divide the signal strength by four every time the travel distance doubles. As this function depend on being in vacuum and knowing the complete distance travelled, the solution was simplified to using linear signal loss as shown in Figure 19. The fact that ray tracing is a recursive algorithm introduces much complexity and the exact computation of signal strength is therefore beyond the scope of this thesis.

```
RT_PROGRAM void chrome_ch_radiance(){
       float3 hit_point = ray.origin + isect_t * ray.direction;
       float distanceTravelled = length(hit_point-ray.origin);
       float signalLoss = distanceTraveled/1000;
       prd_radiance.result = make_float3( 1-signalLoss );
}
```

Figure 19: Code for calculating the light received form hitting a light sphere.

To define the ratio between the density of air and the density of buildings (that cause the effect shown in Figure 2 Section 2.3.2) we used a constant we called *refraction_index*. Optimally the *refraction_index* should be set for every building dependent on construction material and so on. This is discussed further in Subsection 5.1.1.

To hinder waves from reflecting on the inner surface of buildings we included a check *dot(world_geometric_normal,dir)<0*. By dotting the surface normal and the ray direction we are able to sort out the back sides. This is because vectors pointing in the same direction return a positive value when dotted.

3.2.4 Lighting

In normal ray tracing systems, lights and background colours (the miss program) would be used to illuminate the scene. In ray tracing, lights are not implemented as objects which can be intersected by rays and illuminate objects it hit, independent of the view direction. In our system, where the light sources are supposed to simulate senders and the cameras are supposed to simulate receivers, this creates some problems. To simulate a sender we need to be able to see the light source, not only the effects of it being there. The solution is to remove all lights usually used, and create a special sphere which returns the wanted light (independent of the illumination) at the point of the intersection. Doing this saves us much ray tracing since no shadow rays needs to be cast, and we also get the wave simulation effects we are looking for.

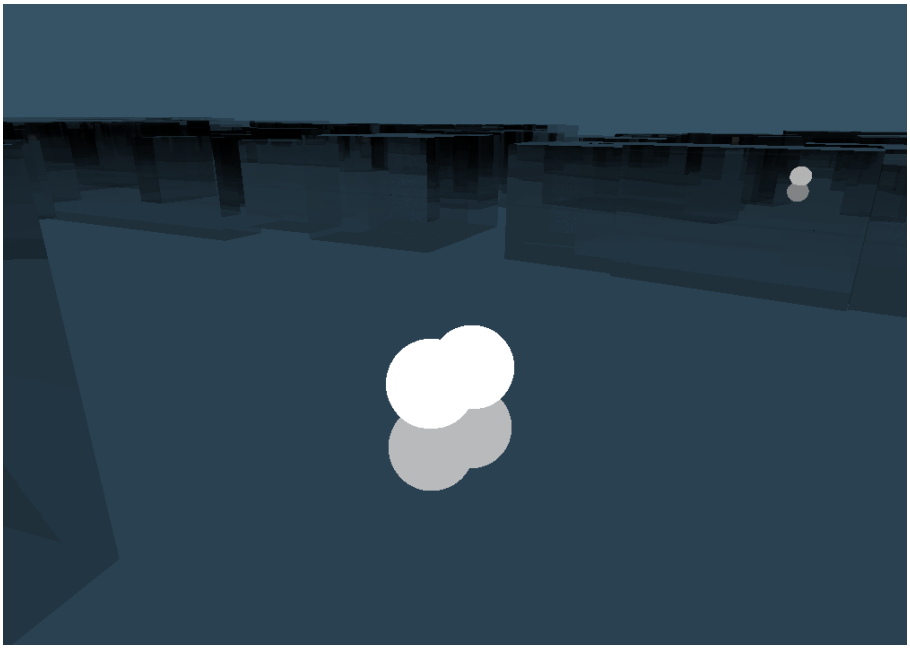


Figure 20: Two moving light spheres in the simulator.

By using light to simulate the WiFi waves, we may use different light colours to simulate different signals in the same environment without interference.

3.2.5 Geometry hierarchy

To keep track of all the different objects, materials and acceleration structures, OptiX utilize a storage hierarchy. Figure 21 shows the hierarchy of our implemented WiFi simulator. **Top object** contains the whole scene and is the root node of the hierarchy. The hierarchy is split into different elements with different capabilities. A **Geometry group** is a collection of geometry instances. **Group** is a collection of Transforms, Geometry groups and Geometry instances. Top object is an example of a Group. An **Acceleration structure** can be attached to a Geometry group or a Group. All nodes below a **Transform** node will be moved, rotated and scaled according to its transform matrix. A **Geometry** node is created for each object in the scene which contains the vertex, normal and texture information of the object. A **Geometry instance** links a geometry and its given Material. A **Material** describes the properties of the given object and can be reused for different objects to behave alike.

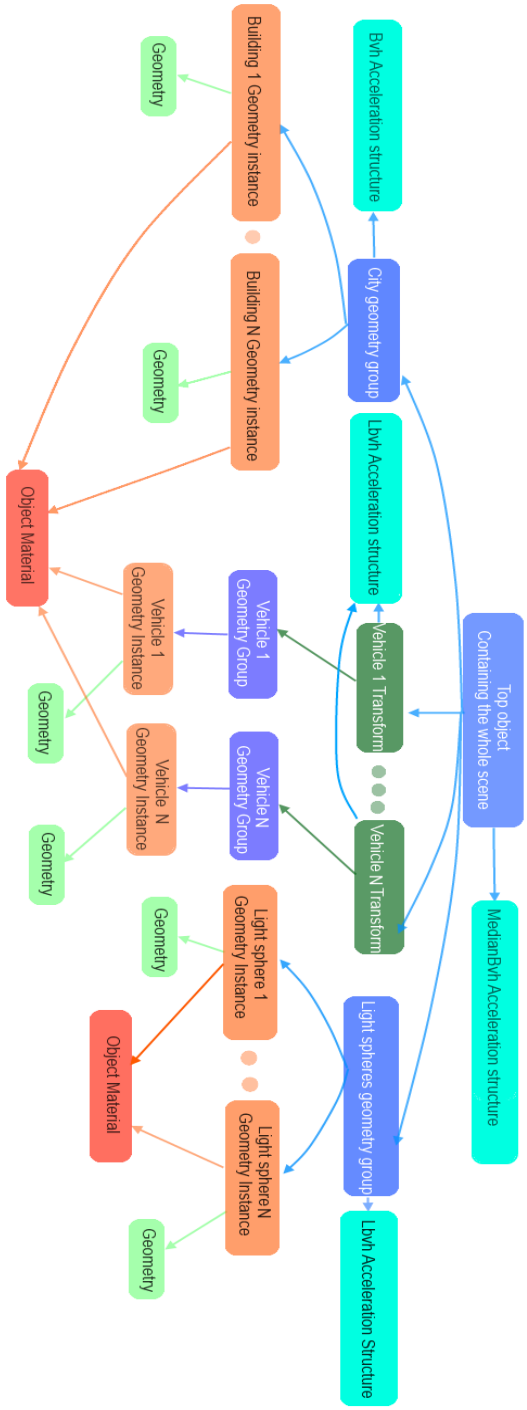


Figure 21: A breakdown of the scene hierarchy.

3.2.6 Acceleration structures

As explained earlier in Section 2.4, choosing the right acceleration structures is a very important part of a ray tracing system. Therefore NVIDIA have implemented a set of acceleration structures in OptiX [50]. These acceleration structures have different applications and have to be benchmarked to find the best fit for our system. This is shown in Subsection 4.3.5.

To let OptiX iterate through the primitives without any acceleration structure, you can set the geometry group to use *NoAccel.* This is best suited for nodes with few children, often positioned at a higher level in the hierarchy. Normally, you should select a more specific builder and traverser combination. OptiX has two traversers and five different implementations of the corresponding builders included by default. These algorithms build on the theory found in Section 2.4.

Builder	Traverser	Construction speed	Traversal speed
NoAccel	NoAccel	-	Very slow
Bvh	Bvh	Slow	Fast
Sbv	Bvh	Very slow	Very fast
MedianBvh	Bvh	Medium	Medium
Lbvh	Bvh	Fast	Slow
TriangleKdTree	KdTree	Medium	Medium

Figure 22: Table of building and traverser combinations with subsequent run speeds. The table data originates from the OptiX documentation.

Sbv is used to build a high quality BVH variant for optimal performance. It is great for ray tracing buildings and such, but have a slower build speed and slightly higher memory footprint than for example normal Bvh. Lbvh is on the other hand a good choice for moving objects which need to rebuild the structure for each iteration.

3.2.7 Movement

The moving light spheres update their positions by altering the position variables sent to the GPU once for each frame. As a result of reading the buildings from file, and thereby not knowing the starting positions of each object, this approach could not be reused for these objects. Instead a transform node in combination with matrix multiplication was implemented for moving the cars and buses as shown in Figure 21.

3.2.8 Input and Output

To send information from the CPU to the GPU before rendering a new frame we use RT_BUFFER_INPUT. This type of buffer is used for sending information to the GPU and cannot be used to return values. The camera values U,V,EYE and W plus the array of sphere positions are all sent through such buffers. The output buffer is the opposite type, which can only return a two-dimensional array from the GPU to the CPU once the trace call returns. This two-dimensional array is shared between all the cameras when using multiple cameras. Each camera uses its unique offset parameter to distinguish the results.

4 Scalability Tests

To look at the strengths and weaknesses of the system, we ran some tests using three different machines with different hardware. The following sections will give a walk-through of these tests and the results we found.

4.1 Test beds

To get a good scenario for testing different parts of the system, we created a path of way points for both the cameras and the light spheres to follow. This way all simulations behave the same, except from the intended changes made for the specific benchmark. The test scenario is set to use a map with 438 buildings extracted from OpenStreetMaps.org, displaying an area around the NTNU.

We have used two estimates when evaluating the simulator; the and the average number of frames per second of the system when following the way points from start to end. All movement is updated in the same loop as the rendering, resulting in being a good estimate for the efficiency of the algorithm. We also used 24FPS as a requirement for real-time graphic and 2FPS as requirement for vehicle-to-vehicle communication. The latter is based on the transmission intervals of the communication interface used in the SINTEF driving simulator which is specified as 1-2 Hz, although, for safety critical applications, communication at 10Hz is required.

A set of default settings have been chosen so if nothing else is noted, the maximum reflection depth is set to 10 times. Two glowing spheres act as moving light sources, and an analysis of the output buffer is done to estimate the maximum intensities found. This is done individually for every camera and frame during the simulation. Unless otherwise stated, these maximums are not displayed since they are meant as input for another system.

We had our doubts to whether the implementation of refraction rays would be too challenging and time consuming to manage during the limited time of a master thesis, but luckily we managed to complete a simplified version. Sadly, as this was a high risk section of the project, this version did not make it in time to be a part of the testing phase of the thesis. It can therefore be regarded as turned off unless otherwise stated.

One camera with a 1024x768 resolution and 360 degree view angle is used. The output buffer is not displayed on screen if not specified in the benchmark. The acceleration structure of the root node containing the whole scene is set to use builder=MedianBvh and traverser=Bvh. The moving light sources use builder=Lbvh and traverser=Bvh. For more details on how acceleration structures work; see Section 2.4.

OptiX uses a stack to keep track of the massive number of threads used when running a ray tracing scenario. This stack is set to a fixed maximum size, which cause errors if exceeded. The stack size should in theory be width*height*maximum recursion depth * number of different ray types cast. For the standard resolution, this would be 1024*768*10*1= 7 864 320. The stack size is set using an unsigned integer, which is defined as a positive number lower than 2^16 = 65536. The implementation of integer can vary much dependent on operation system and compiler. On all the three computers used, the maximum size of unsigned integer was found to be 2^32 -1= 4 294 967 295 which fits well with norm of 64 bit operating systems.

Much research was done to find documentation which could help us create a more scientific formula. The hope was to create a formula which calculates a lower, but still valid, stack size than the original theory, but no documentation was found. After much testing and tweaking the formula “width*2+1000” was used in the final system. In the future, more research should be done on the subject, since this formula might depend on race conditions.

To test the scalability of the simulator a second test city was produced, where all houses contain the same number of vertices. This “BoxCity” contains a varying number of boxes in the range from 2 to 2048, where each box contains 12 polygons.

4.2 Test benches

To run our simulations we have chosen three different computers, which range from high performance to medium. Computer1 and Computer2 are fairly new machines and provide a good estimate of how well the simulator runs on normal computers. The CPUs on these two computers are rated in the "High End CPUs" list on the benchmark created by "PassMark R Software Pty Ltd" on the 27th of May 2013 [51]. By being on this list shows us that we are using hardware from modern commercial computers. Intel Xeon E5-4650 can be found on the top, scoring 14,969 points and the Intel Core i7-3930K CPU rates as the top model of the commonly used CPUs scoring 12,081 points in comparison. To look at how the powers of modern computers have developed we also tested the simulation on an older computer. The CPU in Computer3 is rated in the “Medium-High End” class. A comparison of these five CPUs is shown in Figure 23.

The four graphic cards used were also rated in the "High End Video cards" list at the same web page, although Computer3 is ranked in the bottom part of this list. Tesla C2070 scored between 3,478-4,305 comparing to the top model being GTX Titan scoring 8,390 points and the top commonly used GPU being GTX 680 with 5.685 points.

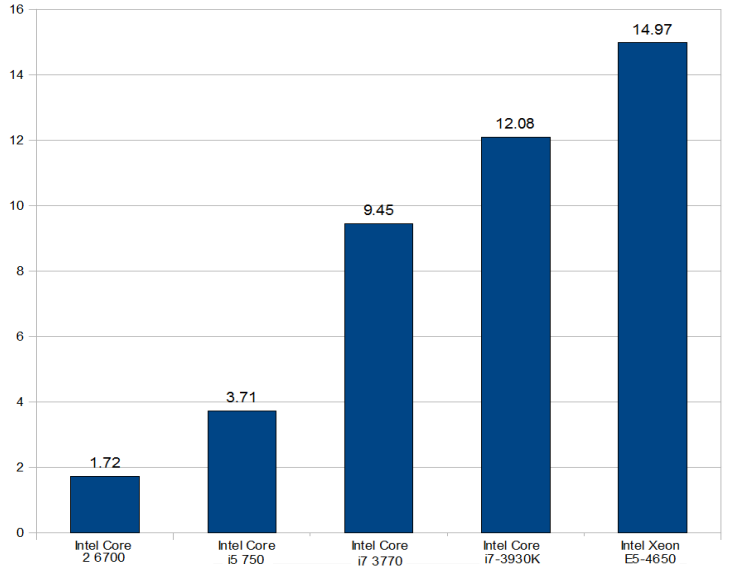


Figure 23: Benchmark score for the three CPUs used, the best scoring model, and highest rated commonly used CPU. These benchmarks are created by "PassMark R Software Pty Ltd" on the 27th of May 2013 [51]. Permission was granted to create subsets of their data for this thesis.

Computer 1	**GPU 1** GeForce GTX 480 Total Memory: 1610153984 bytes Clock Rate: 1401000 kilohertz Max. Threads per Block: 1024 SM Count: 15 Max. HW Texture Count: 128	-	-
	CPU Intel Core i5 CPU 750 2,67GHz	**RAM** 8GB	**Operating system** Ubuntu 64 bit
Computer 2	**GPU 1** GeForce GTX 480 Total Memory: 1609760768 bytes Clock Rate: 1401000 kilohertz Max. Threads per Block: 1024 SM Count: 15 Max. HW Texture Count: 128	**GPU 2** Tesla C2070 Total Memory: 6442123264 bytes Clock Rate: 1147000 kilohertz Max. Threads per Block: 1024 SM Count: 14 Max. HW Texture Count: 128	**GPU 3** Tesla C2070 Total Memory: 6442123264 bytes Clock Rate: 1147000 kilohertz Max. Threads per Block: 1024 SM Count: 14 Max. HW Texture Count: 128
	CPU Intel Core i7 3770 3.4GHz	**RAM** 32GB	**Operating system** Ubuntu 64 bit
Computer 3	**GPU 1** GeForce GTX 285 Total Memory: 107374824 bytes Clock Rate: 1476000 kilohertz Max. Threads per Block: 512 SM Count: 30 Max. HW Texture Count: 128	-	-
	CPU Intel Core 2 6700 2.66GHz	**RAM** 4GB	**Operating system** Windows 7 64 bit

Figure 24: The hardware specifications for the three test benches used in the thesis.

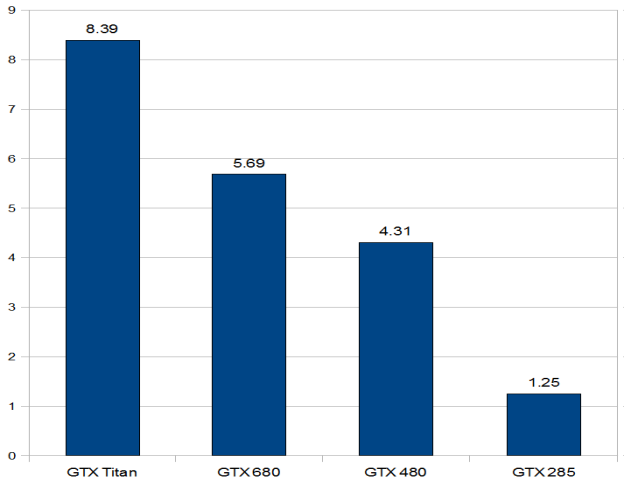


Figure 25: Benchmark score for two of the GPUs used, the best scoring model, and highest rated commonly used GPU. These benchmarks are created by "PassMark R Software Pty Ltd" on the 27th of May 2013 [51]. Permission was granted to create subsets of their data for this thesis.

All the test machines use OptiX version 3.0.0 and Cuda 5.0.35 for running all simulations. Computer number one and two both run 64bit Ubuntu and Computer3 runs 64bit Windows 7.

If nothing else is mentioned in the sections below, computer1 is used.

4.3 Benchmarks

We ran simulations to check whether printing the maximum values to the screen had any effect on . Although it had no notable effect, we still chose to leave this feature turned off during the rest of the tests to limit the sources of errors. As arguments to run the simulations we used “-B” for benchmarking without display, “-sizeI=2” for 1024x768 resolution and “-D=1” for the simulator to run on one GPU.

4.3.1 The three computers in comparison

To compare the three computers fairly, we used only one of their GPUs while running the simulation. Although the first two computers use the same graphic card, there is a significant improvement in as a result of the difference in CPU and RAM. As shown in Figure 26 the increase in compute power shows tremendous results in run time and FPS. If hardware continues to improve at this rate we will have real-time ray tracing systems simulating large scale traffic communication scenarios smoothly very soon.

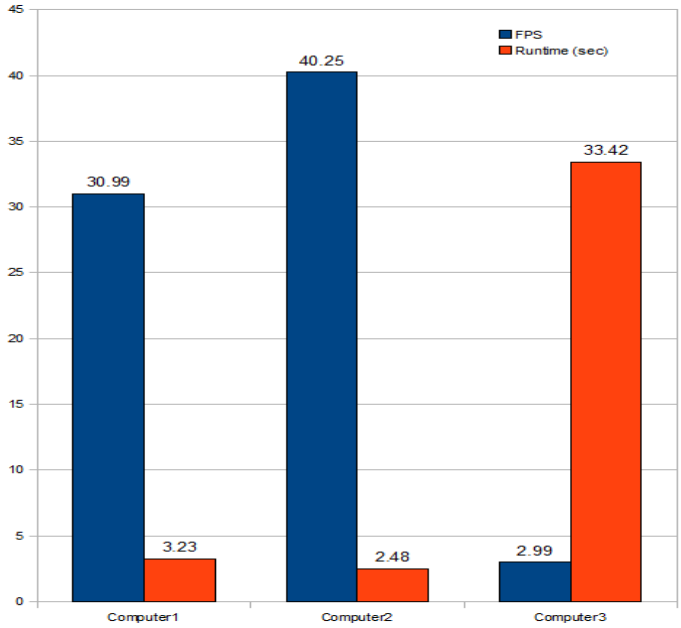


Figure 26: Compares the run time and FPS of the three test benches.

4.3.2 Different numbers of rays.

As described in Section 3.2.1 each camera sends a chosen number of rays every frame. In our system, the rays are distributed throughout both axes equally in the full 360 degrees view. Since this view field does not change size, the resolution of an area is directly dependent on the number of rays sent in the given direction. This is why it is very important to test how the system reacts to changes in resolution.

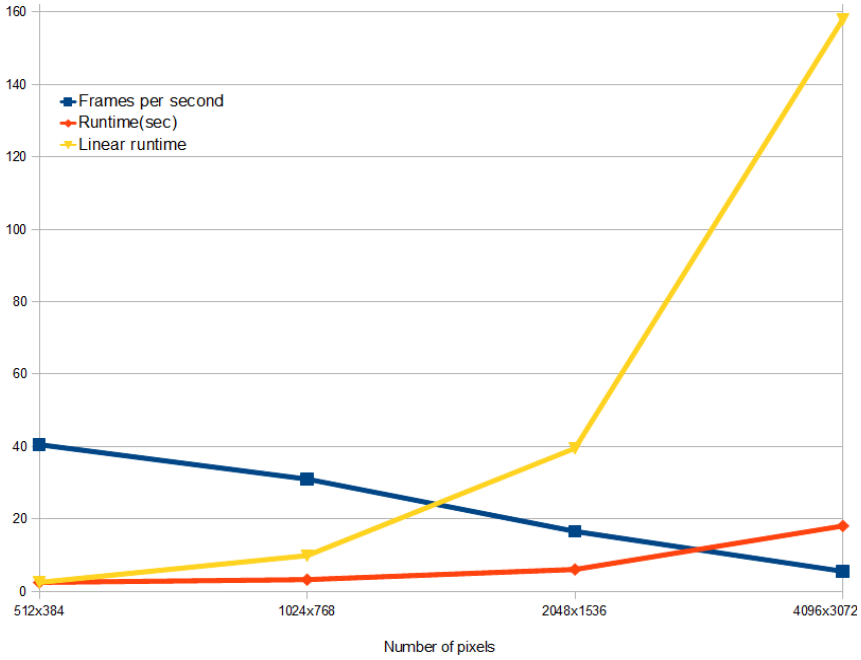


Figure 27: Shows the s and FPS when changing the resolution of the camera.

As shown in Figure 27, the system scale well with changes in resolution compared to linear scalability shown in yellow. Even when the amount of rays have increased 64 times the has only increased by a factor of 7.31.

Although the system scales well, the starting cost of the algorithm is so high that only the two smallest resolutions uphold the requirements for real-time graphics of 24PFS. Luckily, systems like the driving simulator only need a minimum of 2 updates per second to keep up with the flow of communication, and this holds for all four resolutions with the largest having 5.5 FPS. Keep in mind that the resolution doubles the size in both directions and thereby increase by a factor of four for each iteration.

4.3.3 Different numbers of cameras.

For large scale simulations where dozens of cars will communicate it is very important that the system allows as many cameras as possible. As you see in Figure 28, the system dos not scale well enough at this time to run large scale traffic simulations where much more than 16 vehicles communicate with each other. This is not surprising as OptiX is not built for running multiple cameras simultaneously. The potentially for optimization at this area may be huge, but is a complex and unknown area of research, and is therefore noted as future work.

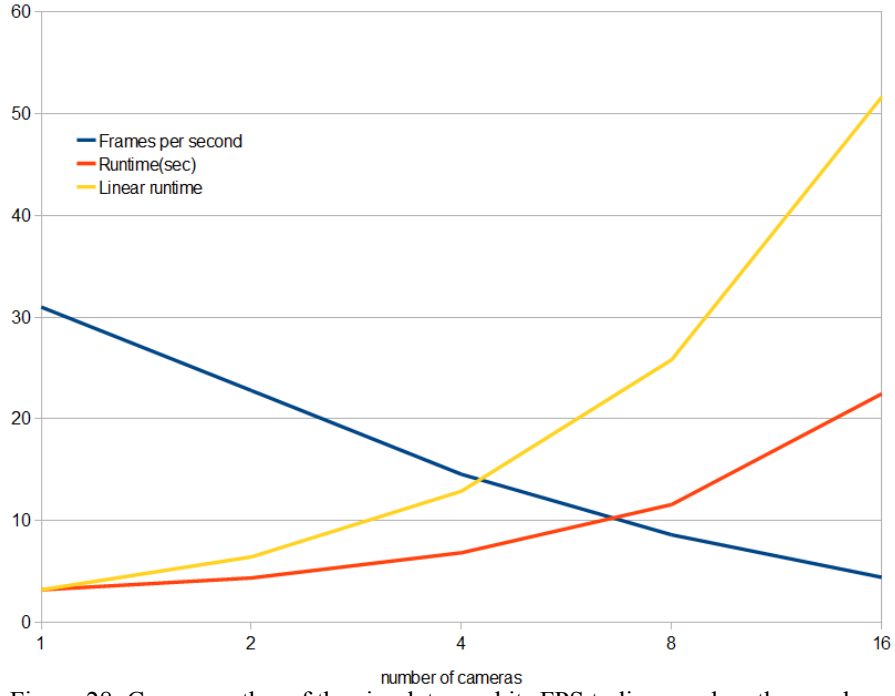


Figure 28: Compares the of the simulator and its FPS to linear when the number of cameras increase.

It is worth noting that there are many more combinations of settings in the simulator which is not yet tried. There may be much to save in optimizing these variables to fit a realistic setting, for instance changing the resolution, the recursion depth, the view area and much more. More optimized settings may make larger simulations possible.

4.3.4 Different numbers of moving objects.

As explained in Subsection 3.2.4, the simulator uses moving light spheres to simulate light sources (WiFi senders). The simulator scales excellent with increasing number of senders, and is not far from scaling logarithmic as shown in Figure 29. This shows that the system has reached its goal of being able to run large scale simulations with more than 1024 senders at above 2fps. The simulator is also capable of displaying up to 64 senders in a real-time system where the requirement is set to 24 FPS.

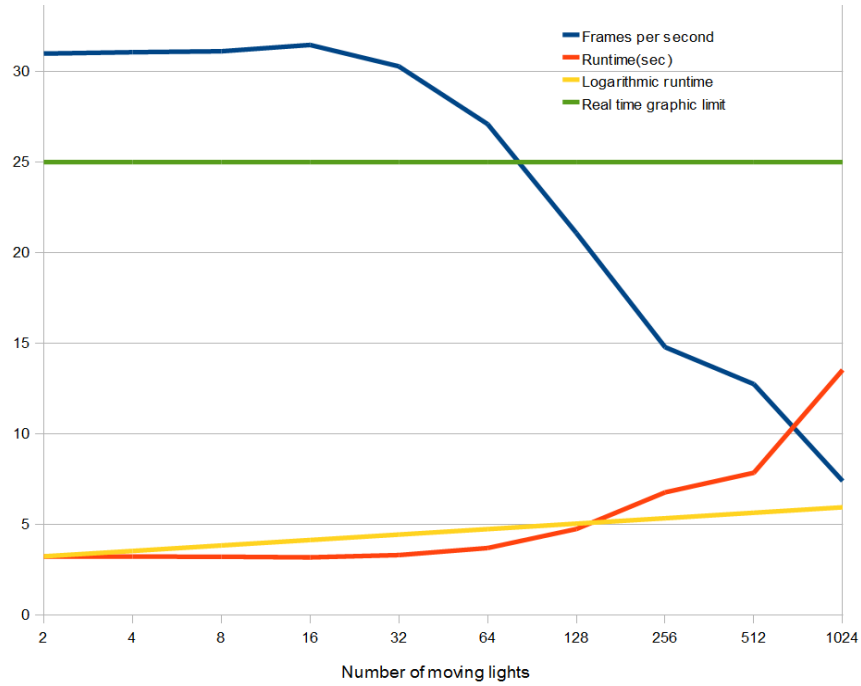


Figure 29: Shows the correlation between the number of light sources, the and the FPS. Logarithmic run time is also drawn in comparison.

4.3.5 Different acceleration structures.

To further optimize the simulator, some of the acceleration structures implemented in OptiX were tested. The details of the implementation are shown in Subsection 3.2.6.

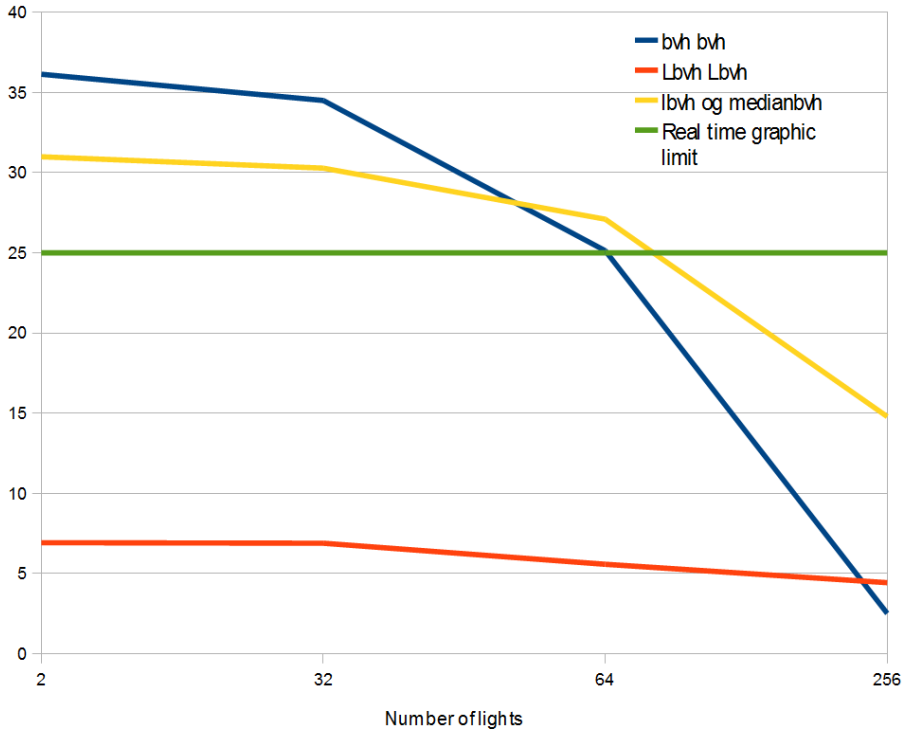


Figure 30: Compares the different acceleration structures tested. The first variable is used for moving objects and the second is used for stationary objects.

Figure 30, shows the correlation between the s of the tested acceleration structures and the number of light spheres used. As expected, *bvh* preforms best with few objects and worse as the number rises. This is because the slow build time of the stationary city only occurs once. Since the city does not move, its acceleration structure only have to be altered, not rebuilt. The fast traverse speed makes up for the time lost while the number of stationary buildings (438) is far superior to the number of moving objects (2-32). When the number of light spheres increase, the time spent rebuilding the acceleration structures increase rapidly. Resulting in a drop in for the whole simulator.

The second simulation use Lbvh to build, but then suffers from the opposite problem. The fast build time never catches up with the high cost of the slow traverse speed. It scales well for achieving the requirement of having above 2 FPS but cannot be used for real-time graphic simulations.

We achieved the best results while using Lbvh as acceleration structure for the moving objects and MedianBvh as acceleration structure for the stationary objects. This performs better on average and will therefore be easier to work with as the different factors vary.

To show how much compilation the system have to do as a result of having moving objects, we ran a simulation with resolution 2048 where we left out the moving objects. This also removes the *make_dirty* call to the acceleration structures which in turn cause them not to rebuild for every frame. The result was a 1.76 sec simulation running at 56.8 FPS in comparison to the normal 16.6 FPS.

4.3.6 Different number of GPUs.

NVIDIA OptiX is a great tool for multi GPU ray tracing. It handles all of the work and there is no visual difference for the programmer, that is if he programs for one or more GPUs. With this said, there are some down sides to this as well. When the programmer knows that multiple cameras can be traced in parallel on different GPUs, he could have achieved a great speed-up. But, since OptiX hides all the "magic" from the users they have no way of knowing that each camera is independent. As a result, we ended up with the s for 16 cameras on one, two and tree GPUs being 19,4 – 16,4 – 15,4 respectively using a resolution of 1024x768. This is a 15,4% and 20,6% better for two and three GPUs compared to using just one. There are clearly potential for improvements here, and as we selects a smaller resolution, 512x384, we get a 14,6% and 25,1% slowdown in run speed for two and three GPUs compared to using just one.

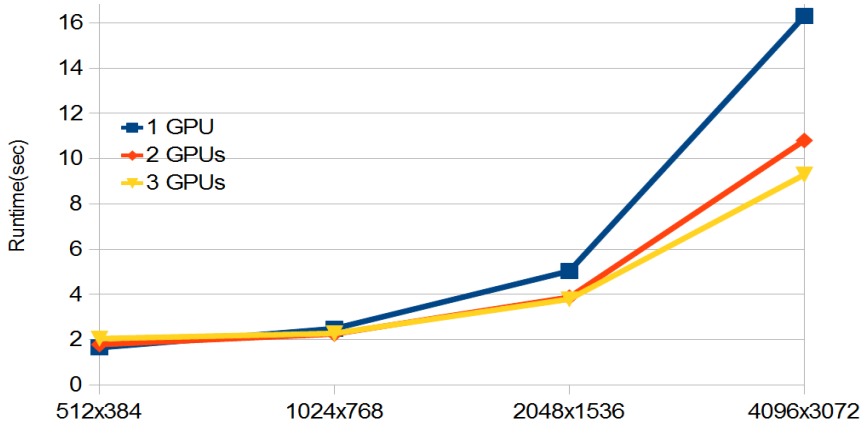


Figure 31: The changes in , dependent on resolution and number of GPUs.

Also when we simulated with a large amount of moving object we found that the use of multiple GPUs was inefficient in our system. The simulation containing 1024 moving light spheres resulted in a 39,6% and 79,0% slowdown when using two and tree GPUs respectively.

The third test which utilizes multiple GPUs is shown in Figure 31. This time, the simulator was run at the four different resolutions. It gave a negative speed-up for the lowest resolution, but sped up as the number of threads increased up to a 43% faster . As noted in [24], scenes with little computational complexity do not scale optimally on multiple GPUs. This seems to fit good with the results we have found as well. When the amounts of data needing to be distributed is at its lowest, compared to the amount of computation and number of threads, OptiX scales at its best with multiple GPUs. We therefore have a suspicion that turning on and adding more effects, like refraction, will in the future let the simulator scale better with multiple GPUs.

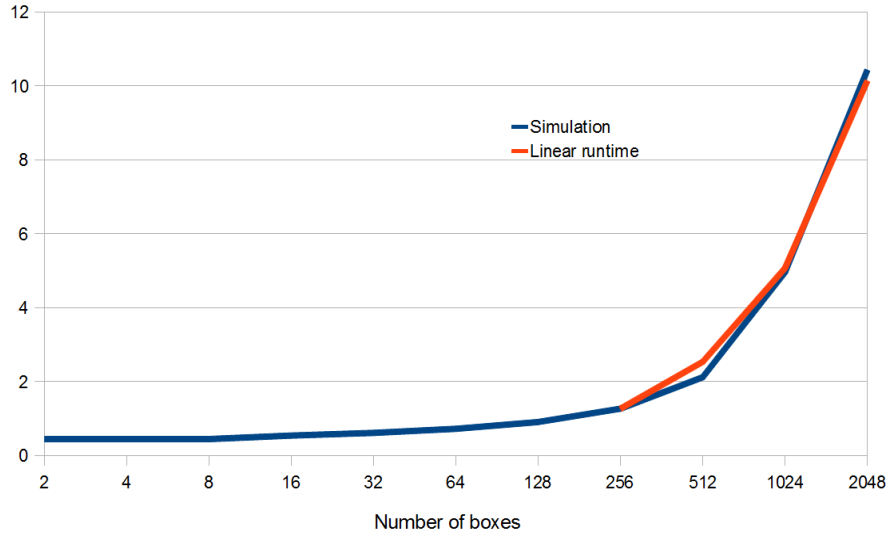


Figure 32: Shows the comparison between (sec) and number of buildings in the scene.

4.3.7 Different number of vertices.

As mentioned earlier, a second test city was created, where all houses contain the same number of vertices. This “BoxCity” contains a varying number of boxes in the range from 2 to 2048, where each box contains 12 polygons. The BoxCity is an excellent scene to test the scalability of the simulator and as shown in Figure 32 the simulator scales very well up to 2048 buildings. Keep in mind that the number of buildings double for every step in the graph and that the choice of acceleration structure can alter the outcome of this graph significantly. When the number of buildings rise from 512 to 1024 the rise by a factor of 2,34 which is a bit high. Around this number of buildings is also where the simulator first encounter FPSs lower than 24.

Another interesting fact found when simulating was that the auto generated city (containing 438 houses) had a 52% worse than the BoxCity (containing 512 buildings). This shows that it may be possible to optimize the city generator a bit further.

5 Conclusion and future work

The research and development of new hardware is opening for new and interesting uses of known algorithms. Ray tracing is one of these, and have gone from being a theoretical curiosity in 1980 to being a possible successor for rasterization in the time to come.

From our experiments (Chapter 4), we found that the use of multiple GPUs in our simulation implemented in NVIDIA´s OptiX framework, did not scale as much as our theoretical estimates. Our best multi-GPU speed-up was 43% on 3 GPUs with 4096x3072 resolution over a single GPU. This may be due to the downside of having a framework which automatically takes care of all the heavy lifting for the programmer. The scalability of using multiple GPUs may also improve as the complexity of the simulator increases and the massive parallel sections becomes the dominant computational element.

Although our original goal was to build a simplified prototype, few simplification were actually made, so our results can be used as part of a complete and fairly realistic simulator. Overall our system shows great scalability with most of the settings and can sustain calculating over 6144 polygons distributed among 512 objects in real-time. The number of rays sent, moving spheres and boxes in the scene scaled a lot better than linearly. This is likely a result of the high initial cost of ray tracing and the use of acceleration structures to eliminate objects from the search.

In the range of 2 to 128 light spheres (Section 4.3.4), the simulator even scaled better than logarithmic which was the optimal goal estimated in Section 2.4. The run time only increase from 3.23 seconds to 4,74 seconds while the number of lights doubles 6 times. As with the other variables this is likely a result of the high high initial cost of ray tracing and the use of acceleration structures.

5.1 Future work

During the course of this project we have found many possible extensions which can improve the realism or the of the simulator. Here are five of the concrete ideas we have come up with.

5.1.1 Define building density

By defining the materials for each individual building according to pictures from sources like google street view, the simulator could take the difference in building densities into account. Some research on car and bus densities would also be a great improvement for the simulator. This way the simulator can get much more realistic results.

5.1.2 Optimized ray tracing for multiple cameras

The results found in Subsection 4.3.6 show that our implementation of multiple cameras in OptiX may be optimized further. By running the different cameras in parallel instead of in series the run time may decrease substantially, especially for multiple GPUs. This can lead to run times which allow large scale testing of receivers as well.

5.1.3 Find a new stack size formula

As noted in Section 4.1 OptiX uses a stack to keep track of its rays. The stack size of the simulator is a crucial element of the system and needs to be as low as possible and as high as necessary. The relationship between this stack size and the number of potential rays needs to be thoroughly explored to guarantee the safety of the stack, while keeping the overhead as low as possible.

5.1.4 Diffraction can be added

By creating a shell of glass which bend/split light in the wanted direction the simulation of diffraction may be possible without much altering of the simulator. The diffraction can also be calculated at edges like done in [52] using the OptiX ray tracing engine. The theory of diffraction is shown in SubSection 2.3.3.

5.1.5 Implement photon mapping

To simulate communication speed and packet delay Arne Schmitz and Leif Kobbelt suggest using a Photon Path Map [53]. This extension is possible to achieve in OptiX, and can be based on the SDK example called ProgressivePhotonMap. This feature would be a great supplement to the existing simulator.

References

[1]: Standard Number: IEEE 1609.3-2010 , "*IEEE standard for wireless access in vehicular environments (WAVE) networking services, IEEE Std 1609.3-2010 (Revision of IEEE Std 1609.3-2007)*", July 15 2010, 1 - 144, Institute of Electrical and Electronics Engineers,

[2]: "*http://www.ertico.com/compass4d-project-consortium-meets-in-denmark-where-90-buses-will-be-equipped-and-trialed-for-at-least-one-year/*" , ,

[3]: Goel, S. Dept. of Comput. Sci., Rutgers Univ., New Brunswick, NJ, USA Imielinski, T. Ozbay, K. , "*Ascertaining viability of WiFi based vehicle-to-vehicle network for traffic information dissemination*", 3-6 Oct 2004, pages 1086 - 1091, Intelligent Transportation Systems, 2004. Proceedings. The 7th International IEEE Conference on, 0-7803-8500-4

[4]: David B. Johnson , David A. Maltz , "*Dynamic Source Routingin Ad Hoc Wireless Networks*", 1996, pages 153-181, In Mobile Computing, volume 353. Kluwer Academic Publishers,

[5]: M. Hassan-Ali and K. Pahlavan , "*A new statistical modelfor site-specific indoor radio propagation prediction basedon geometric optics and geometric probability*", Jan. 2002, Pages 112–124, IEEE Transactions on Wireless Communications (Volume:1 , Issue: 1), 1536-1276

[6]: M. Hassan-Ali and K. Pahlavan , " *Site-specific widebandand narrowband modeling for indoor radio channel using ray-tracing*", 8-11 Sep 1998, pages 65 - 68 vol.1, Personal, Indoor and Mobile Radio Communications, 1998. The Ninth IEEE International Symposium on (Volume:1), 0-7803-4872-9

[7]: Balamati Choudhury and R M Jha , "*A Refined Ray Tracing approach for Wireless Communications inside Underground Mines and Metrorail Tunnels*", 2011, pages 1 - 4, Applied Electromagnetics Conference (AEMC), IEEE, 978-1-4577-1098-8

[8]: Jo Skjermo and Cato Mausethagen , "*Software Intergration for Implementation*

and Testing of CVIS ITS Application in a Driving Simulator Environment, ITS Europe, Dublin", 6 june,2013, , ,

[9]: Ingar Saltvik, Anne C. Elster and Henrik R. Nagel , "*Parallel Visualization of Snow*", 2006, pages 218-227, PARA'06 Proceedings of the 8th international conference on Applied parallel computing: state of the art in scientific computing, 3-540-75754-6 978-3-540-75754-2

[10]: Kjetil Babington, Anne C. Elster , "*Terrain Rendering Techniques for the HPC-Lab Snow Simulator*", 2012, , NTNU,

[11]: Heiko Friedrich, Johannes Günther, Andreas Dietrich, Michael Scherbaum, Hans-Peter Seidel, Philipp Slusallek , "*Exploring the Use of Ray Tracing for Future Games*", 2006, Pages 41-50, Sandbox '06 Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames, 1-59593-386-7

[12]: JungHyun Han et al., 3D Graphics for Game programming, 2011, Pages 53-81, a Capman & Hallbook

[13]: JungHyun Han et al., 3D Graphics for Game programming, 2011, Pages 122-125, a Capman & Hallbook

[14]: Per H. Christensen, George Harker, Jonathan Shade, Brenden Schubert, Dana Batali , "*Multiresolution Radiosity Caching forEfficient Preview and Final QualityGlobal Illumination in Movies*", July, 2012, , SIGGRAPH '12 ACM SIGGRAPH 2012 Talks, Article No. 47 , 978-1-4503-1683-5

[15]: Per H. Christensen , "*Adjoints and Importance in Rendering:An Overview*", JULY-SEPTEMBER 2003, pages 329-340, IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS, VOL. 9, NO. 3, 1077-2626

[16]: Per H. Christensen, David M. Laur, Julian Fong, Wayne L. Wooten, Dana Batali , "*Ray Differentials and Multiresolution Geometry Cachingfor Distribution Ray Tracing in Complex Scenes*", Volume 22 (2003),Number 3, pages 543-552, EUROGRAPHICS,

[17]: Henrik Wann Jensen, Per Christensen , "*High Quality RenderingusingRay Tracing and Photon Mapping*", August 5, 2007, , SIGGRAPH '07 ACM SIGGRAPH 2007 courses, rticle No. 1, 978-1-4503-1823-5

[18]: Per H. Christensen , "*Point-Based Global Illuminationfor Movie Production*", July 2010, , Pixar,

[19]: Sven Woop, Jörg Schmittler, Philipp Slusallek , "*RPU: A Programmable Ray*

Processing Unit for Realtime Ray Tracing", 2005, pages 434-444, ACM Transactions on Graphics (TOG) - Proceedings of ACM SIGGRAPH 2005,

[20]: Heiko Friedrich , "*Ray tracing techniques for computer games and isosurface visualization*", 2009, http://scidok.sulb.uni-saarland.de/volltexte/2012/4917, Saarlaendische Universitaets- und Landesbibliothek,

[21]: Thomas L Falch , "*3D Visualization of X-ray Diffraction Data*", June 2012, , NTNU,

[22]: Turner Whitted et-al. , "*An Improved Illumination Model for Shaded Display*", 1980, pages 343-349, Communications of the ACM, Volume 23 Issue 6, June 1980,

[23]: "*www.pcper.com/reviews/Processors/Ray-Tracing-and-Gaming-One-Year-Later/Ray-tracing-faster-rasterization-Example-1*" , , Intel

[24]: Holger Ludvigsen and Anne Elster , "*Real-Time Ray Tracing Using Nvidia OptiX*", May 3-7 2010, pages 65-68, presented and published at EuroGraphics in Norrköping,

[25]: "*Electromagnetic waves - reflection, refraction, diffraction*" , www.radio-electronics.com/info/propagation/em_waves/electromagnetic-reflection-refraction-diffraction.php, Adrio Communications Ltd

[26]: Kjetil Babington, Anne C. Elster , "*Real-Time Ray Tracing for the HPC-lab Snow Simulator*", 17.jun 2011, Pages 6-13, NTNU,

[27]: Niels Thrane, Lars Ole Simonsen , "*A Comparison of Acceleration Structuresfor GPU Assisted Ray Tracing*", August 1, 2005, , ,

[28]: "*http://www.nvidia.com/object/optix.html*" , ,

[29]: Gogueny, Joseph A., et al. , "*Introducing obj.*", 1993, , ,

[30]: B. Watson, P. Müller, O. Veryovka, A. Fuller, P.Wonka, and C. Sexton , "*Procedural Urban Modeling in Practice*", May 2008, pages 18-26, Journal of IEEE Computer Graphics and Applications, vol. 28, no 3,

[31]: "*http://cmivfx.com/store/233-Houdini+XML+Based+Procedural+Cities*" , ,

[32]: "*http://cmivfx.com/store/253-Houdini+Building+Generation*" , ,

[33]: Claus BRENNER , "*TOWARDS FULLY AUTOMATIC GENERATION OF*

CITY MODELS", 2000, pages 85-92, Institute for Photogrammetry (ifp), Stuttgart University, Germany, In: IAPRS ,

[34]: "*http://cmivfx.com/store/47-Houdini+Procedural+Cities*" , ,

[35]: Pearl Goswell and Jun Jo , "*Real-Time 3D City Generation using Shape Grammars with LOD Variations*", 2012, , WASET 2012 : World Academy of Science, Engineering and Technology, Issue 61, 2010-376X

[36]: "*http://en.wikipedia.org/wiki/Dedicated_short-range_communications*" , ,

[37]: "*http://www.cvisproject.org/*" , , Cooperative vehicle-infrastructure systems

[38]: "*http://www.vegvesen.no/Fag/Trafikk/ITS* " , ,

[39]: Jérôme Härri, Marco Fiore, Fethi Filali,Christian Bonnet , "*Vehicular Mobility Simulation withVanetMobiSim*", 2009, pages 275-300, eurecom, Simulation Volume 87 Issue 4, April 2011,

[40]: Sebastian Gräfling, Petri Mähönen and Janne Riihijärvi , "*Performance Evaluation of IEEE 1609 WAVEand IEEE 802.11p for Vehicular Communications*", 2010, pages 344 - 348, Institute for Networked Systems, RWTH Aachen UniversityKackertstrasse, 2010 Second International Conference on Ubiquitous and Future Networks (ICUFN), 978-1-4244-8088-3

[41]: Irena Notkin, Craig Gotsman , "*Parallel Progressive Ray-tracing*", March 1997, pages 43-55, The Eurographics Association, Computer Graphics Forum, Volume 16, Issue 1,

[42]: "*https://www.caustic.com/openrlsdk.php*" , ,

[43]: Ingo Wald , "*Realtime Ray Tracing andInteractive Global Illumination*", 2004, , Computer Graphics GroupSaarland UniversitySaarbrücken, Germany,

[44]: "*http://www.pcper.com/reviews/Processors/Ray-Tracing-and-Gaming-One-Year-Later/Progress*" , , pcper

[45]: S. G. Parker, S. Boulos, J. Bigler, and A. Robison , "*RTSL: A ray tracing shading language*", Sep. 2007, Pages 149-160, RT '07 Proceedings of the 2007 IEEE Symposium on Interactive Ray Tracing, 978-1-4244-1629-5

[46]: I. Georgiev and P. Slusallek , "*RTfact: Generic concepts for flexible and high performance ray tracing*", Aug. 2008, pages 115 - 122, IEEE Symposium on

Interactive Ray Tracing, 978-1-4244-2741-3

[47]: "*http://www.openstreetmap.org/copyright/en*" , ,

[48]: Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, François Yergeau, John Cowan , "*Extensible Markup Language (XML)*", 16 August 2006, , W3C® (MIT, ERCIM, Keio), ID: fdd000075

[49]: John E. Greivenkamp , "*Field guide to geometrical optics*", 20 January 2004, , SPIE Optical Engineering Press,Volume: FG01, 9780819452948

[50]: , "*NVIDIA-OptiX-SDK-3.0.0-B1-OptiX_API_Reference*", 2013-01-03, Chapter 1.6, NVIDIA,

[51]: "*Passmark^AR software pty ltd, www.cpubenchmark.net*" , ,

[52]: Okada, M. ,Onoye, T. , Kobayashi, W. , "*A Ray Tracing Simulation of Sound Diffraction Based on the Analytic Secondary Source Model*", Nov. 2012, pages 2448 - 2460, IEEE Transactions on Audio, Speech, and Language Processing (Volume:20 , Issue: 9), 1558-7916

[53]: Arne Schmitz, Leif Kobbelt , "*Wave Propagation Using the Photon Path Map*", 2006, pages 158-161, In Proceedings of the 3rd ACMInternational Workshop on Performance Evaluation ofWireless Ad hoc, Sensor and Ubiquitous networks(PE-WASUN ’06), 1-59593-487-1

6 Appendix

In the following sections you can find a guide for how to use the system, how to install the system, and the raw test results from the scalability test in Chapter 4.

6.1 How to use the system

To define some of the often used variables without having to recompile we have implemented the possibility of sending them in as input.

Some of the input arguments at start-up:

'-D=Number' can be used to define the number of GPUs the simulator should run at. The default value is the computers maximum number of GPUs.

'-B' and '-b' both tell the simulator to run a benchmark. The upper case version also turns the display off while benchmarking.

'-sizeI=Number' defines the resolution used for each camera to be Number*512 wide and Number*384 high. The default value is set to 1024x768.

At run time these commands and more can be used:

'a': Pushing 'a' will change the camera mode of camera one from guided movement to free movement using the mouse. The left mouse button rotates the camera around the “look at” point. The right mouse button zooms the camera, while the middle one moves it.

'c': Pushing 'c' will change from the 360 degree camera to a normal ratio camera and can be very useful when debugging.

'q': Push 'q' to quit the simulator.

The number of light spheres and the number of cameras used can be changed altering the numberOfSpheres value and the numberOfCams value found in the constructor of JuliScene at the top of Julia.cpp.

To change the maximum number of reflections and refractions set the maxDepth value found in closest_hit_radiance() in one_bounce_diffuse.cu.

6.2 How to set up the system

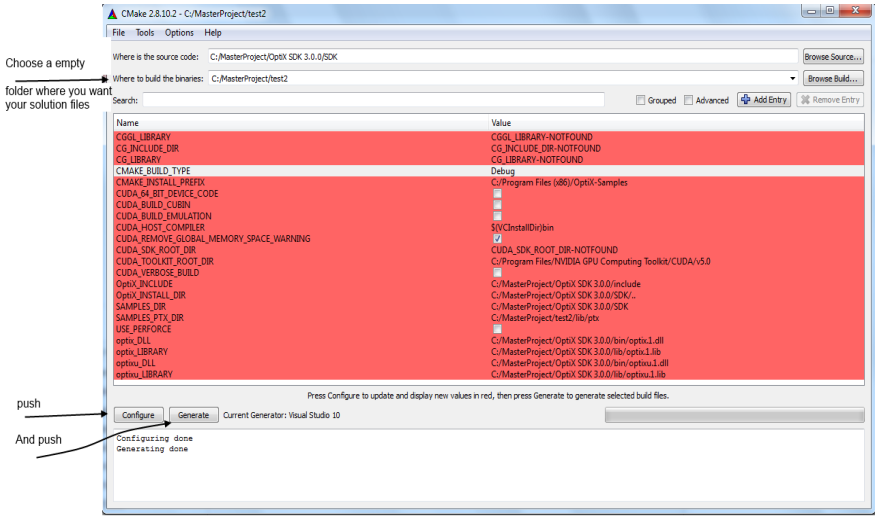
Install OptiX and Cuda.

Copy the files our julia folder to the newly installed folder. For example: *C:\MasterProject\OptiX SDK 3.0.0\SDK\julia.*

For Windows:

Be sure **not** to put the OptiX files in the default program files folder. That folder is often read only for programs and will cause problems.

At http://www.cmake.org/cmake/resources/software.html you can download cmake, which is used to create a working solution.

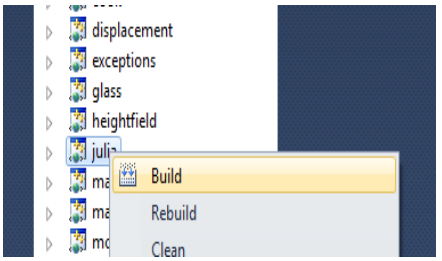


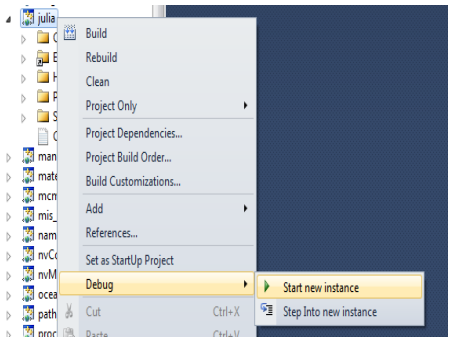
Copy the object files into the solution folder. For example:

C:\MasterProject\test2\julia

We used Microsoft Visual C++ 2010 Express to open the solution file.

Use build and debug as shown below.





For linux:

sudo apt-get install cmake

create a empty folder and change directory to inside it. Type cmake and the directory of the SDK.

C:\MasterProject\test2: **Cmake** *C:\MasterProject\OptiX SDK 3.0.0\SDK*

Copy the object files into the solution folder. For example:

C:\MasterProject\test2\bin

cd julia && make

cd ../bin && ./julia

6.3 Raw test result data

This following tables contain the test data received when benchmarking the simulator. Although such amounts of data cant be displayed in other parts of the thesis it can be useful to have the details of both s and FPS of the different scenarios tested for comparisons in the future.

Default settings				
Width	Height	FPS	Run time(sec)	ms/frame
512	384	40.51	2.47	24.68
1024	768	30.99	3.23	32.26
2048	1536	16.55	6.04	60.41
4096	3072	5.54	18.05	180.5

Displaying the output on screen using -b instead of -B				
Width	Height	FPS	Run time(sec)	ms/frame
512	384	30.86	3.24	32.4
1024	768	27.86	3.59	35.9
2048	1536	15.03	6.66	66.55
4096	3072	5.44	18.4	183.96

Without output analysis				
Width	Height	FPS	Run time(sec)	ms/frame
512	384	43.7	2.29	22.88
1024	768	36.23	2.76	27.6
2048	1536	20.64	4.85	48.46
4096	3072	7.97	12.55	125.54

2 cameras				
Width	Height	FPS	Run time(sec)	ms/frame
512	384	33.87	2.95	29.52
1024	768	22.79	4.39	43.87
2048	1536	9.97	10.03	100.34
4096	3072	2.93	34.11	341.1

4 cameras				
Width	Height	FPS	Run time(sec)	ms/frame
512	384	25.27	3.96	39.57
1024	768	14.57	6.86	68.63
2048	1536	5.17	19.34	193.4
4096	3072	1.5	66.56	665.56

Resolution 1024x768			
Number of cameras	FPS	Run time(sec)	ms/frame
8	8.62	11.59	115.95
16	4.45	22.47	224.68

Resolution 1024x768			
Number moving spheres	FPS	Run time(sec)	ms/frame
4	31.07	3.22	32.19
8	31.12	3.22	32.19
16	31.47	3.18	31.78
32	30.29	3.3	33.02
64	27.1	3.69	36.9
128	21.08	4.74	47.43
256	14.78	6.76	67.65
512	12.74	7.85	78.48
1024	7.39	13.53	135.27

Using bvh as acceleration structure on all objects. Resoultion 1024x768			
Number moving spheres	FPS	Run time(sec)	ms/frame
2	36.14	2.77	27.67
32	34.5	2.9	28.98
64	25.13	3.98	39.79
256	2.51	39.92	399.18

Using Lbvh as acceleration structure on all objects. Resoultion 1024x768			
Number moving spheres	FPS	Run time(sec)	ms/frame
2	6.92	14.46	144.56
32	6.89	14.52	145.21
256	5.57	17.95	179.46
1024	4.42	22.6	226.02

Resolution	Number of GPUs	FPS	Run time(sec)	ms/frame
4096x3072	3	10.76	9.29	92.91
	2	9.26	10.8	108.04
	1	6.14	16.3	162.99
2048x1536	3	26.43	3.78	37.83
	2	25.84	3.87	38.7
	1	19.89	5.03	50.27
1024x768	3	43.99	2.27	22.73
	2	44.51	2.25	22.47
	1	40.25	2.48	24.85
512x384	3	49.02	2.04	20.4
	2	56.18	1.78	17.8
	1	60.78	1.65	16.45

1024 light spheres. Resolution 1024x768			
Number of GPUs	FPS	Run time(sec)	ms/frame
3	4.57	21.86	218.58
2	5.87	17.05	170.48
1	8.19	12.21	122.11

16 cameras				
Resolution	Number of GPUs	FPS	Run time(sec)	ms/frame
1024x768	3	6.48	15.44	154.36
	2	6.09	16.43	164.32
	1	5.15	19.42	194.16
512x384	3	10.31	9.7	96.97
	2	11.26	8.88	88.83
	1	12.9	7.75	77.5

Resolution 1024x768			
Number of buildings	FPS	Run time(sec)	ms/frame
2	227.43	0.44	4.4
4	227.72	0.44	4.39
8	224.39	0.45	4.46
16	184.53	0.54	5.42
32	162.87	0.61	6.14
64	138.04	0.72	7.24
128	110.28	0.91	9.07
256	78.92	1.27	12.67
512	47.2	2.12	21.19
1024	20.17	4.96	49.58
2048	9.59	10.43	104.31

2048 if no objects move and the not making dirty.
Resolution 1024 x 768 | 56.8367 fps | 1.75943 sec | 17.5943 ms/f

Computer3
Resolution 1024 x 768 | 2.99178 fps | 33.4249 sec | 334.249 ms/f