CHALMERS

Ray tracing fully implemented on programmable graphics hardware

Filip Karlsson

Carl Johan Ljungstedt

Master's Thesis

Programme in Computer Science and Engineering

CHALMERS UNIVERSITY OF TECHNOLOGY
Department of Computer Engineering
Göteborg 2004

Abstract

Modern graphics cards are rapidly increasing in computational power and the amount of memory available. Still this computational power is rarely used for anything else than 3D-games and modelling.
This thesis investigates how to implement a ray tracer that executes all of its image rendering operations on a graphics card using pixel shaders and vertex shaders and to compare the performance of this implementation to standard ray tracer.
Implementing general purpose programs for graphics hardware is not trivial because GPU:s have large limitations compared to ordinary CPU:s. Recursion and loops with too many iteration can not be used and there is also the problem of storing data in texture memory.
It will be shown how many of the programming limitations can be worked around and how texture memory can be used to store data needed for the computations.
This thesis will also show how a proximity cloud algorithm can increase performance compared to standard grid traversal when ray tracing on graphics hardware, but still none of the implementations have been able to achieve better performance than 3D-studio's ray tracer, "Mental Ray".

Sammanfattning

Moderna grafikkorts beräkningskraft och dess storlek på texture minnet ökar hela tiden, men fortfarande används sällan grafikkortens beräkningsmöjligheter till annat än 3D-spel eller modelering.
Detta examensarbete undersöker hur man kan implementera en ray tracer, som utför alla sina renderingsberäkningar på ett grafikkort, med pixel- och vertex shaders för att sedan jämföra denna implementation med med befintliga ray tracer.
Att implemetera generella program mot grafikhårdvara är inte trivialt, på grund av att GPU:n har mycket större begräsningar jämfört med en vanlig CPU. Anvädningen av rekursion och loopar med för många interationer kan inte implementeras och det finns även problem när det gäller att lagra data i texturminnet.
Det kommer att visa sig i detta examensarbete hur många av dessa programmeringsbergränsningar kan kringgås och hur texturminnet kan användas för att lagra den data som behövs för att utföra beräkningarna i en ray tracer.
Förutom detta visar vi även hur proximity-cloud algoritmen kan ger ökad prestanda jämfört mot standard grid traversering vid ray tracing på grafikhårdvara. Dock lyckas ingen av implementationerna uppnå högre prestanda än 3D-studios ray tracer ”Mental Ray”.

Table of contents

1 Introduction

The fast development of graphics cards has given us cards whose computational power is equal to or surpasses that of modern CPU:s. It can be argued that modern GPU:s with up to 16 pipelines are a lot more powerful than modern CPU:s but since the types of data that can be processed and output by GPU:s is very restricted it is hard to compare the two.
On a modern computer the graphics card often stand for a large portion of the total cost. Yet graphics cards are usually not used in any other applications than games or 3D-modelling tools [11].
Because modern cards are highly programmable there is little reason not to use the computational power of the graphics card for other applications. For example the latest "graphics processing unit" (GPU) from Nvidia, has up to 16 parallel pipelines that are each able to execute a program of more than 65 000 instructions at a speed of up to 450 MHz.
What we have done in this project is to investigate how to implement a complete ray tracer using pixel shaders and vertex shaders.
During this project we have implemented a ray tracer that runs on a GeForce 6800 GT.
Graphics algorithms are very suitable for implementing on GPUs because they usually consists of doing similar calculations a large number of times, for example calculate the colour of each pixel. Many of the fundamental mathematical functions that are needed are also implemented directly in hardware which allows for high performance.
Because a GPU is a form of stream processor, programs or part of programs that need to perform similar operations on a large number of elements are probably suitable for implementation on the graphics card.
Our goal with this project was to implement a complete ray tracer on a GeForce 6800 that would be able to render scenes with up to 1 million triangles, which we have also succeeded in doing.

2 Problem statement

Today normal ray tracers are implemented like any other program. It uses only the CPU and the internal memory, and can easily render models that have more than 1 000 000 polygons.

The goals of this thesis have been to implement a ray tracer that executes all of its image rendering operations on a graphics card and to se how this implementation performs compared to standard ray tracers.

2.1 Utilizing GPU computational power

For 3D-animators rendering times are a big problem. The animated movie Shrek 2 for example required approximately 10 million computer hours for the final rendering [13]. Therefore is it interesting for us to se how much of the ray tracing algorithm that can be implement on programmable graphics hardware to see how much you can decrease the load on the CPU and use the power of the GPU instead. For 3d-animation studios even a small improvement in rendering time can greatly decrease the cost and time to produce a movie.

2.2 Input Data

The only efficient way of accessing information in large amounts of stored data required for the ray tracing calculations is to use the texture memory of the graphics card. Therefore it is important to investigate how data such as vertex points, vertex normal, rays and grid traversal data can be stored in texture memory without loosing precision and, because high detailed models might require millions of vertex points to be stored, without allocating unnecessary amounts of texture memory.

2.3 Data structure and traversal algorithms

To speed up a ray tracer different types of accelerations algorithms i.e. different types of data structures, are often used. In this implementations a grid structure that divides the scene in to voxels will be used, but what will be the fastest way to traverse this grid, to use the standard way or to use a different algorithm i.e. proximity clouds [12]?

3 Previous work

Ray tracing that only uses the CPU have been around for a long time and the algorithm is well known and has been implemented by thousands. The classic algorithm has been modified many times and to day you can easily render models that contain millions of polygons and new methods that improve ray tracing is developed all the time. Therefore the idea to use the GPU to ray trace models is a natural step for improving it. The idea is not completely new and there have been various papers on the subject of what way would be the best to solve this problem.

The paper on The Ray Engine [2] proposes an idea to reconfigure the geometry engine to a ray engine. The idea is to put the heavy computation of ray-triangle intersection test on to the graphics card by using textures that holds the model and ray data and then use a pixel shader to compute the intersection test. This implementation sounds good except that the result from the ray engine are read back on to the AGP bus. This can still be a bottleneck.

Another idea that has emerged is to view the graphics hardware as a streaming processor and to use it to implement a ray tracer. To use this idea the graphics card and its shader must support branching. As one paper [3] presents this has given good results but this was only a simulation of a streaming processor. At the time when the thesis was presented there were no graphics cards that supported the necessary instructors needed.

There is also a paper on GPU-based nonlinear ray tracing [4]. This solution claims to have succeed in implementing a non linear ray tracer without read back to the AGP bus. However we have found no evidence of any actual ray tracing in the report. The solution seems to be more of a ray caster for curved rays described by differential equations and no classical ray tracing effects such as reflection and refraction seem to be implemented.

4 Graphics hardware

4.1 Programming limitations

There are a lot of problems when programming for graphics card as opposed to ordinary programming for CPU:s. Even modern graphics cards do not allow recursive calls and they have limitations on how many instruction that can be executed in a single pass as well as limited depths for if statements and limited iterations in loops.
For this project an Nvidia GeForce 6800 GT with 128 Mb of memory has been used [5]. This has given some advantages from using earlier cards from the GeForce FX-series [6].
The GeForce 6800 GPU supports the shader model 3.0 [10], which allows us to do conditional branching and execute pixel programs of up to 65 000 instructions.
Another limitation with using graphics cards is the AGP-bus. Although it is very fast at transferring data to the graphics card it is slow at reading data back to the CPU which must be taken into consideration. However this problem can be solved by using PCI-Express [7] cards instead.

4.2 Advantages

The main advantage with using graphics card for programs other than games and 3D-modelling is of course that the CPU could be used for other things during the execution of the shader programs. In our implementation however this is not the case because the CPU is busy running a display loop. It would probably be advisable to use a multithread implementation to be able to utilize the CPU while the graphics card is busy.
Parallelism will be high because modern GPU:s have a large number pixel-pipelines that are executed in parallel, e.g. the GeForce 6800 that has 16 pipelines which makes it very suitable for applications such as ray tracing that need to perform a large number of similar calculations.

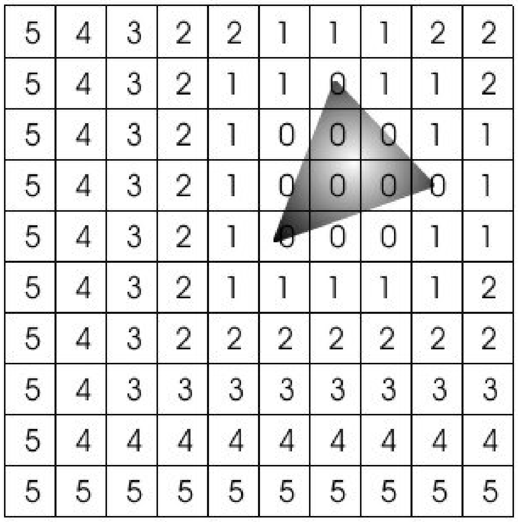
5 Data structures / Texture memory management

Before we can start rendering an image we need to transfer all the information needed to the texture memory of the graphics card. In practice this means that all the information about rays, positions, meshes, lights, materials, textures, normals and grid traversal must be organized into two-dimensional float-arrays that can then be stored in texture memory, by using for example OpenGL's glTexImage2D. The reasons that we use 2D-textures for storing data are that we are able to use the texture-coordinates from the vertex-pipeline for some of the texture-lookups required during the execution.

Depending on what type of information we want to store in the texture memory we must use different types of memory data structure.

5.1 Grid data structure

To speed up the ray-triangle intersection test a uniform grid structure has been used. In our implementation the grid size and also the number of voxels can be varied in every dimension, in the range from 2^2 to 2^8. This is used because the grid can then be built recursively instead of linearly.
Two variations of the grid structure have been implemented, 1) one normal [8] and, 2) one that builds a proximity cloud [12].
A proximity cloud is equal to a normal grid with the difference that when the grid has been built every voxel is processed and given a value that corresponds to the distance, in voxels, to the nearest voxel that contains polygons, see Fig.1. These values can then be used to skip empty voxels when traversing the grid.



5	4	3	2	2	1	1	1	2	2
5	4	3	2	1	1	0	1	1	2
5	4	3	2	1	0	0	0	1	1
5	4	3	2	1	0	0	0	0	1
5	4	3	2	1	0	0	0	1	1
5	4	3	2	1	1	1	1	1	2
5	4	3	2	2	2	2	2	2	2
5	4	3	3	3	3	3	3	3	3
5	4	4	4	4	4	4	4	4	4
5	5	5	5	5	5	5	5	5	5

Fig.1 shows an example of a proximity cloud in 2-dimensions.

5.2 Scene texture structure

A scene will be stored on the graphics card in different textures. This is done to get good access to every voxel and its meshes. The concept is to first store the mesh information, coordinates, normals, texture coordinates and materials in textures.
Fig 2 shows a graphical representation of some of the information stored in textures.

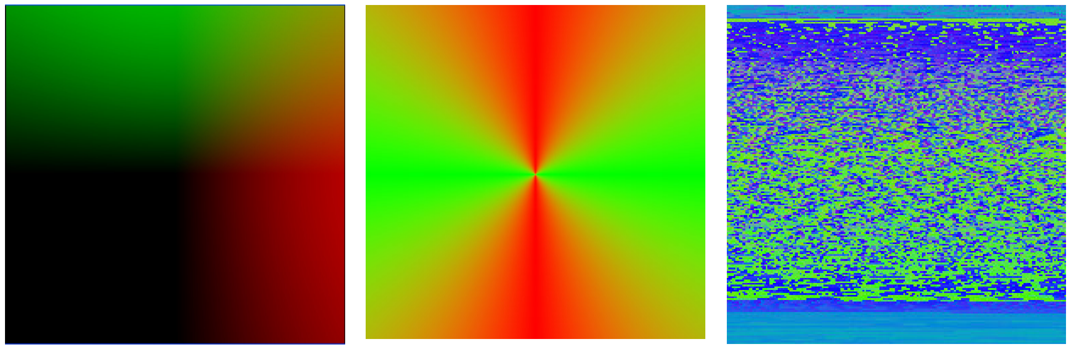


Fig.2 Left: The texture for the generated eye-rays.
Middle: The texture holding information for grid traversal.
Right: A texture representing part of the mesh.

All data concerning rays are stored in textures with the same dimensions as the final picture. This makes it possible to make use of the texture coordinates provided by the rasterizer.
For data concerning the mesh it is a bit more complicated. Grid, triangle and vertex data are stored in large textures for which texture coordinates must be calculated during execution of the pixel shaders.

The grid, triangles and vertex data are represented as described in Fig.3. Voxels contain only a pointer to the index of the first triangle it contains and triangles contain a pointer to the index of the vertex points that describes the triangle. These index values are used as in-data for calculating texture coordinates and values are fetched using texture lookups.

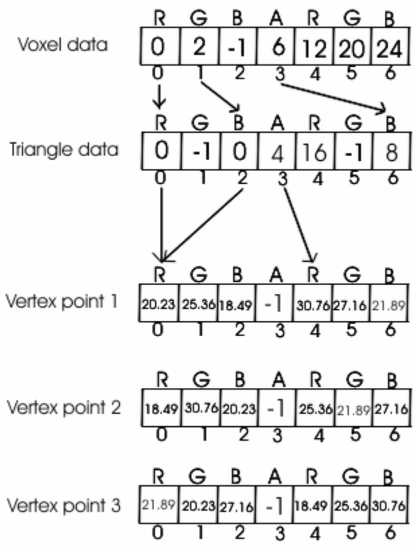


Fig.3 A scheme showing how information for grid, triangles and vertices are stored.

6 Algorithm

Because there is no support for recursive functions on any graphics cards today we have been forced to develop an iterative ray tracing algorithm. Our algorithm is a multipass algorithm that runs each shader program several times. Because the number of instructions that can be executed in a shader program is limited we have divided the algorithm into three separate programs, intersection calculations, grid traversal and shading. We also have a fourth program that display the partial results but this is not necessary and could also be replaced with OpenGL's texture2D and will not be explained further in this text.

6.1 Setup

By setting up a single square polygon in OpenGL and positioning the viewport and camera so that the polygon exactly fills the entire screen we are able to store eye rays, ray start positions and so on in textures with the exact same resolution as the screen and then use the texture coordinates produced by the vertex shader to access a single ray every time a pixel shader program is executed. This is because the size of a single pixel is the same size as a single texel in for example the ray texture.
Before any rendering can be done all the information about the scene, grid and initial values for the rays must be processed by the CPU and stored in textures on the graphics card.
The single square polygon is then rendered using OpenGL which will cause the pixel shader programs to be executed once per ray.

6.2 Execution

6.2.1 Intersection tests

First, every ray is tested for intersection against the triangles in the ray's current voxel in the grid. Intersection testing is done using the "Moller-Trumbore ray-triangle intersection test" [9]. If an intersection is found within the voxel, the triangle index and information about the point of intersection are stored in a separate texture to be accessed in later steps of the algorithm.

6.2.2 Grid traversal

If an intersection is not found in the previous step the ray will be traversed through the grid to the next voxel it intersects in the grid. Depending on whether standard grid traversal or proximity cloud is used, the next voxel will be either the nearest next voxel or the voxel at the distance implied by the proximity value.
If an intersection has been found, no grid traversal is done because a new ray will have to be spawned with its starting point in this voxel.
Last, the current voxel indices are stored in a separate texture.

6.2.3 Shading

The third step uses "multiple render targets" (MRT) to output values into four separate buffers that are then stored in textures.
If an intersection has been found, the pixel is shaded using the hit point on the ray, light position and material, texture and normals of the hit triangle. The calculated colour is then

blended with the colour calculated in previous steps to be able to produce the reflection and refraction effects. The colour value of the pixel is written to a buffer and the previously calculated colour texture is overwritten.
If the hit triangle is not diffuse and the maximum reflection depth has not been reached, a new ray is calculated and written to two separate buffers. The previous ray direction and ray start position textures are then overwritten. Also one texture with data for grid traversal is overwritten.

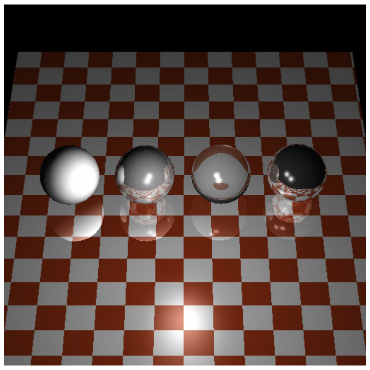
6.2.4 Iteration

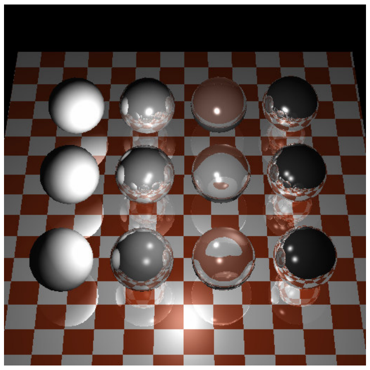
These three steps are executed repeatedly until all rays have been reflected out of the grid, have hit a diffuse object or the maximum reflection depth has been reached.
The texture containing the temporary colour values of the pixel will now be the finished image and can be used to texture the screen filling polygon to produce the final result.

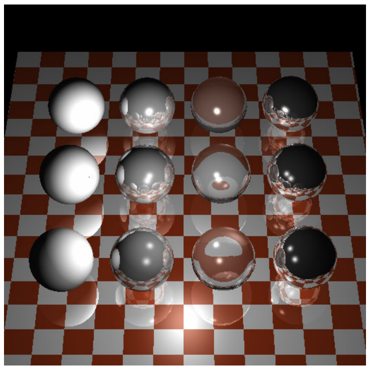
7 Results / Discussion

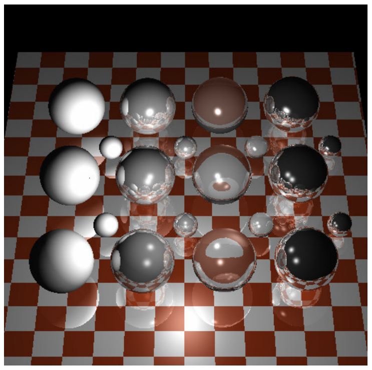
To test the efficiency of our different implementations vi have made a number of test renderings on 7 test scenes consisting of between 1792 and 328192 polygons. We have measured rendering times for the standard grid traversal and proximity cloud implementations for different image resolutions. To put our times in perspective we have also done time measurements on the same scenes and resolutions using 3d studio max's standard renderer "mental ray".

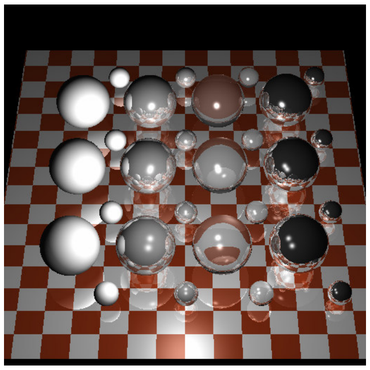
The test scenes are shown below in fig.4.

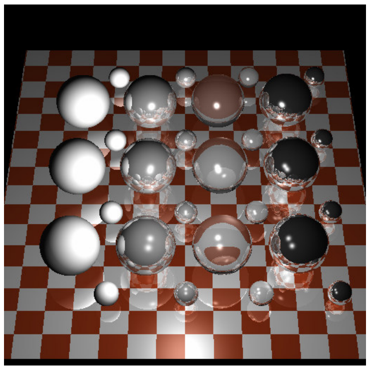

Scene 1


Scene 2


Scene 3


Scene 4


Scene 5


Scene 6

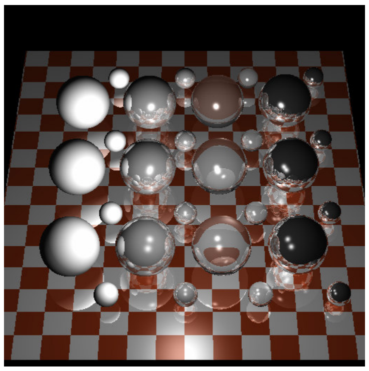

Scene 7

Fig.4 Showing the seven test scenes used in the rendering time measurements.

7.1 Performance tests

The times measured do not include time for preparing data structures or readbacks from the graphics card. This is because 3d studio max can build it's data structures while the scenes is being created and because readback times can be greatly reduced through the use of different hardware like PCI-Express.
Fig 5-7 present the results of the time measurements.

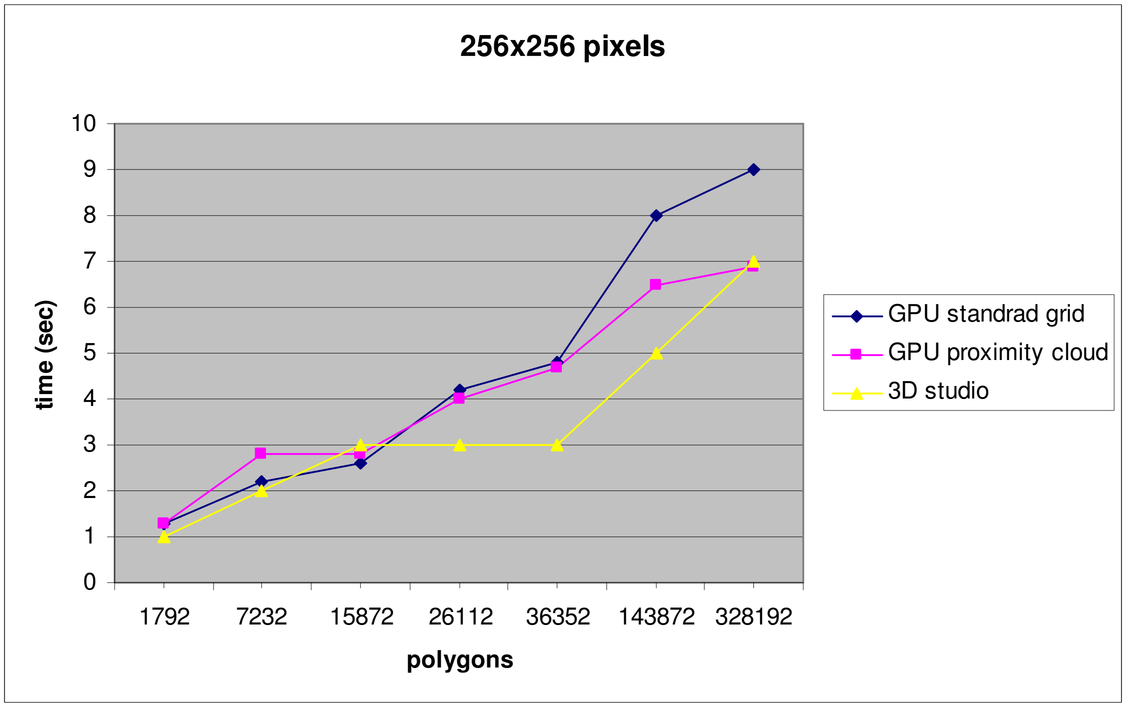


Fig.5 Rendering times for the seven test scenes with an image resolution of 256x256 pixels.

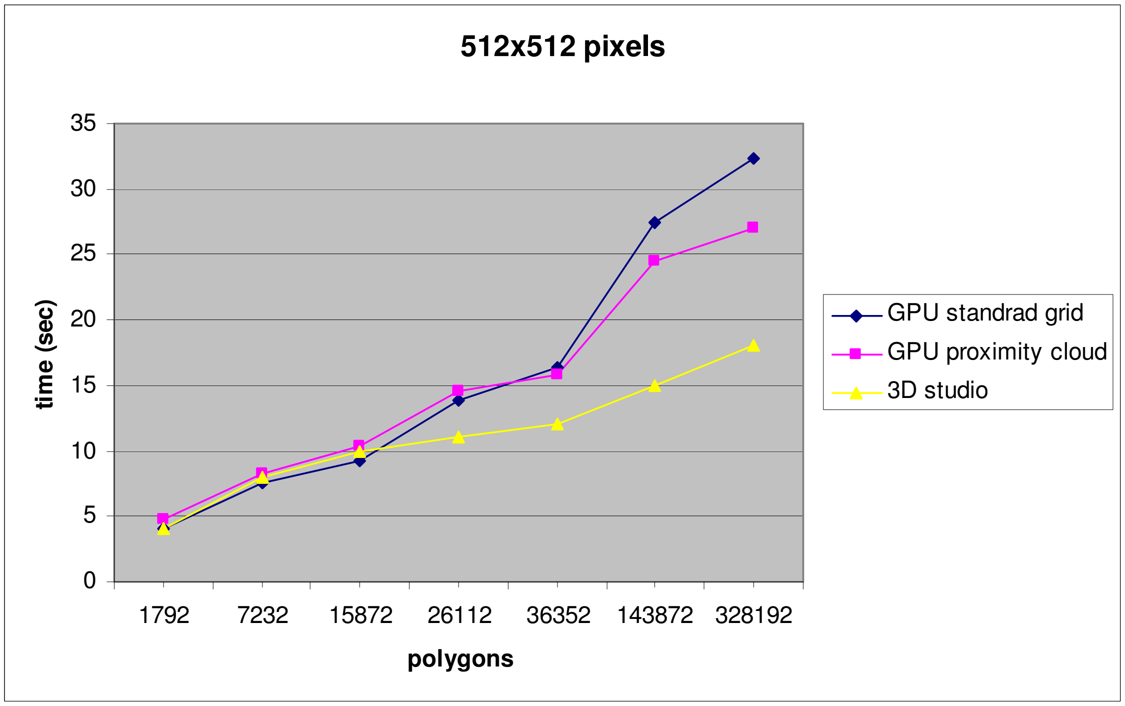


Fig.6 Rendering times for the seven test scenes with an image resolution of 512x512 pixels.

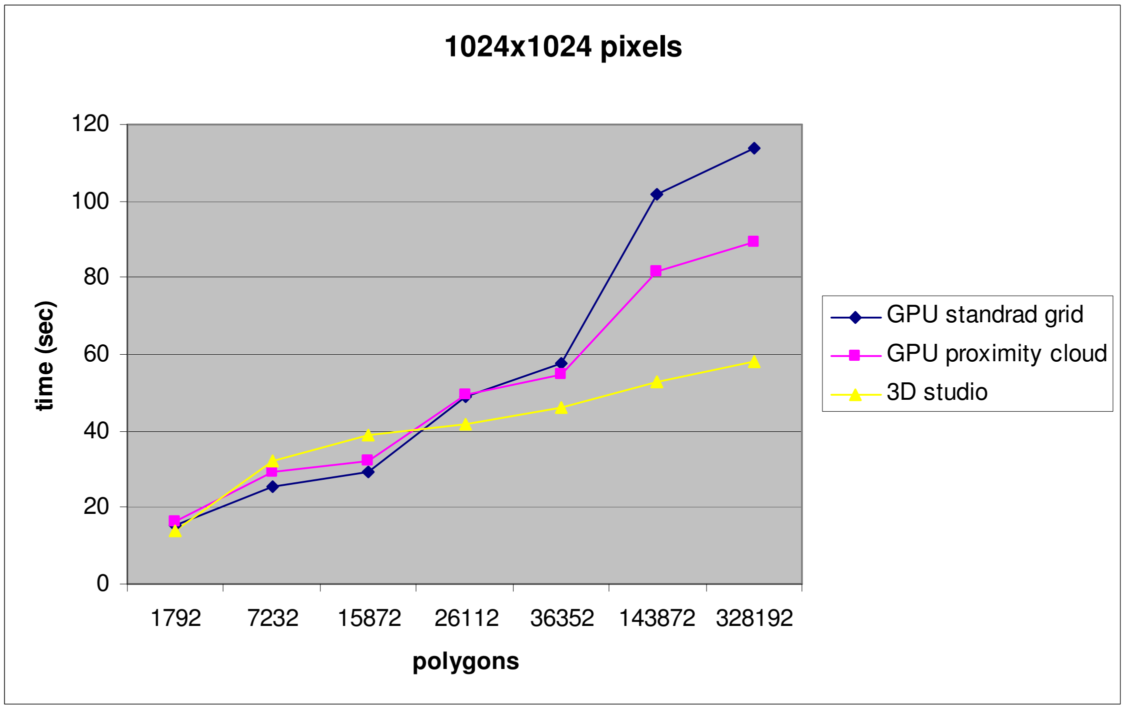


Fig.7 Rendering times for the seven test scenes with an image resolution of 1024x1024 pixels.

As we can se from Fig. 5-7 a significant reduction in rendering times is achieved with the proximity cloud implementation over the standard grid traversal in scenes with more than about 40 000 polygons. This is probably mostly because these scenes require a grid with high resolution which causes the standard grid traversal algorithm to do a lot more iterations than the proximity cloud algorithm. For the scenes with more than 36 352 polygons a grid of 128x128x128 voxels have been used. These scenes have a maximum proximity distance of 63 voxels which means that there is at least one voxel in the grid from which the proximity cloud algorithm can traverse at least 63 voxels in any direction. The maximum proximity distance might not be a very good measurement because it is also very important to know how many voxels in the scene that have a large proximity distance but it can at least give us a hint about when the proximity cloud can give us an increase in performance. For the scene with 36 352 polygons we do not get any noticeable advantage with proximity clouds. For this scene a grid of 64x64x64 voxels was used with a maximum proximity distance of 32 voxel which leads us to the conclusion that *for this scene* we need a maximum proximity distance of more than 32 voxels to get any performance increase with the proximity cloud algorithm.

7.2 Advantages of proximity clouds

The sparser a scene is the more apparent the advantage of the proximity cloud becomes. In sparse scenes large areas of the grid can be skipped in a single iteration which gives significantly better rendering times with proximity clouds compared to standard grid traversal. To show this we have done a measurement on a single sparse scene, see Fig. 9. The results are presented in Fig. 8 below.

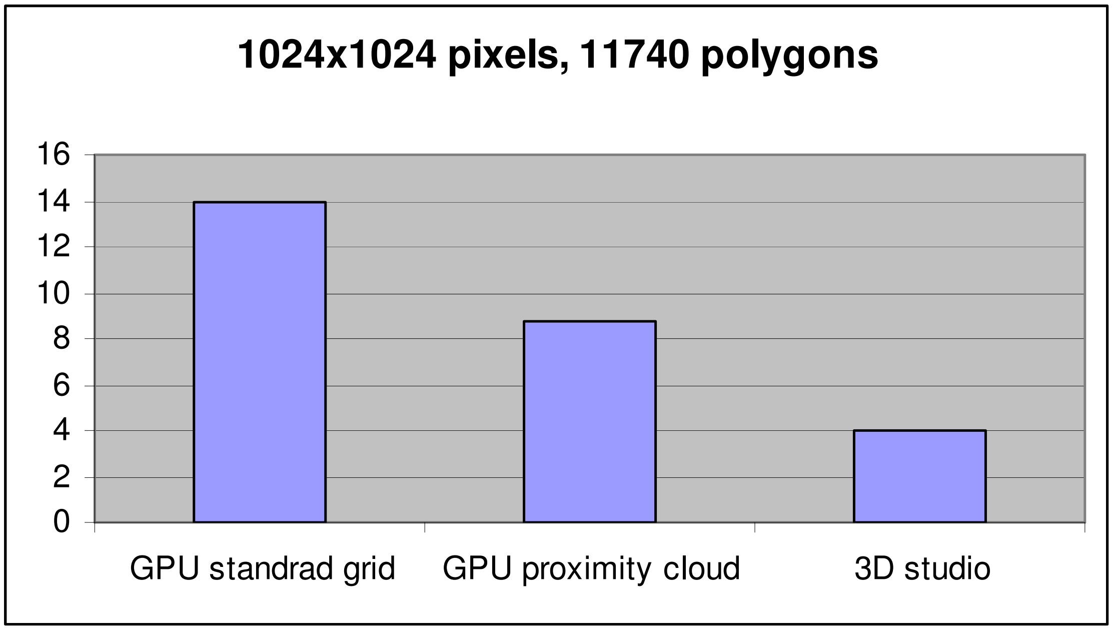


Fig.8 Rendering times for a single sparse test scene with an image resolution of 1024x1024 pixels.

For this test rendering a grid of 32x32x32 voxels was used with a maximum proximity distance of 24 voxels and still the proximity cloud implementation has about a 37% lower

rendering time than the standard grid traversal. Although the maximum proximity distance is only 24 voxels for this scene we most likely have a large amount of voxels with more than 20 voxels in proximity distance which gives this large performance increase. If the same scene is rendered with a grid of 128x128x128 voxels we get a maximum proximity distance of 96 voxels and a performance increase of more than 50%.
Although the time for the proximity cloud is much lower it is still far from the 3d studio rendering time which shows that a uniform grid probably isn't a very good data structure for these types of scenes at all.
For dense scenes the proximity cloud will not give any increase in performance and might even perform a bit worse than the standard grid traversal because the shader code for proximity cloud traversal is slightly more complex. The standard grid traversal algorithm computes 2-3 lines of code per iteration while our proximity cloud implementation needs to compute 8 lines of code that includes 2 divisions.

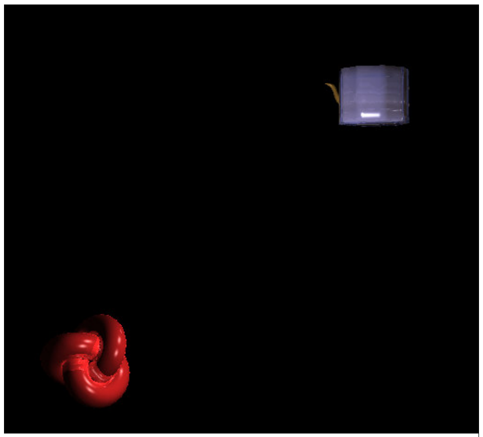


Fig.9 The test scene for the proximity cloud test.

8 Future work

We have presented a fast GPU implementation of a ray tracer but there are still more features to add to it, for instance shadows and reflecting refractions. This feature will demand the possibility to generate more than one new ray at intersection points but this would demand more memory because these rays must be stored. Shadows can be achieved by adding another rending pass to our implementation but it would not be correct shadows because the shadows would not be reflected.

In this implementation we use the ATI predefinition use of buffer reading, atiDrawbuffers. This instruction has given us some limits on the size of the buffers we can use and therefore we would like to see if there is a way to circumvent the use of it in the future.

A problem with the proximity cloud algorithm is that only one proximity value per voxel is calculated. This value is the minimum number of voxels to the nearest voxel that contains polygons but does not take into account the direction of the ray. One possibility to increase performance would be to calculate proximity values for every positive and negative axis of the grid. This would allow for better accuracy when skipping empty voxels [14].

9 Conclusion

The work on this thesis has definitely given us great insights to how graphics card programming works. We are very pleased with the results we have acquired as far as answering our problem statements is concerned. However using the graphics card as an extra processor for running programs might not yet be fully appropriate for mainstream programs. What we have learned during our work is that it is definitely possible to write many types of programs for graphics cards but since there is no real standard for how the code will be executed this might not always be the best idea. Code execution is dependent on what driver and hardware is being used and the performance might change simply because the graphics card driver is updated. This poses obvious concerns with security and functionality of programs since a program can only be guaranteed to run properly when run on a computer with the exact same hardware and driver configuration. This is a major problem that will have to be addressed by the graphics card manufacturers before their cards might be used in a more wide area of applications.

Although it has taken a lot of work to get our ray tracer up and running the final result is great success. Not only have we been able to show that it is possible to implement a ray tracer algorithm fully on a modern graphics card we have also been able to show that using proximity clouds for grid traversal can greatly enhance.
So far we have not been able to acquire rendering times that compete with established ray tracers like mental ray but then again this has never been the objective of this thesis. Writing fast ray tracers with the help of graphics hardware is something that we believe is definitely possible though. A good idea is probably to use the graphics card only for some parts of the rendering like for example intersection tests and shading and letting the CPU take care of the traversal of data structures and such.

10 References

[1] T. Akenine-Möller, E. Haines, “Real-Time Rendering, second edition”,
A K Peters Ltd, USA, 2002

[2] N.A. Carr, J.D Hall, J.C. Hart, “The Ray Engine”,
Graphics Hardware 2002, pp. 1-10

[3] T.J Purcell, I. Buck, W.R Mark, P. Hanrahan, “Ray Tracing on Programmable Graphics Hardware”, ACM Transactions on Graphics 21 (3), pp. 703-712, 2002

[4] D. Weiskopf, T. Schafhitzel, T. Ertl, ”GPU-Based Nonlinear Ray Tracing”,
EUROGRAPHICS, Volume 23, number 3, 2004

[5] “Nvidia Product Info Geforce 6800 series”,
http://www.nvidia.com/page/geforce_6800.html, (2004)

[6] T. Nuydens, “Delphi3D, 3D Hardware info”
http://www.delphi3d.net/hardware/listreports.php, (2004)

[7] L. Weinand, “Future Promise for Graphics: PCI Express”,
Tom’s Hardware
http://graphics.tomshardware.com/graphic/20040310/index.html

[8] J. Amantides, A. Woo, ”A Fast Voxel Traversal Algorithm for Ray Tracing”,
In proceedings of Eurographics’87, pp. 3-10, New York, 1987

[9] T. Moller, B. Trumbore, “Fast, Minimum Storage Ray/Tringle Intersection”,
Journal of Graphics Tools, 2(1):21--28, 1997

[10] R. Fernando, “Shader Model 3.0 Unleashed”,
NVIDIA Developer Technology Group,
ftp://download.nvidia.com/developer/presentations/2004/SIGGRAPH/Shader_Model_3_Unleashed.pdf

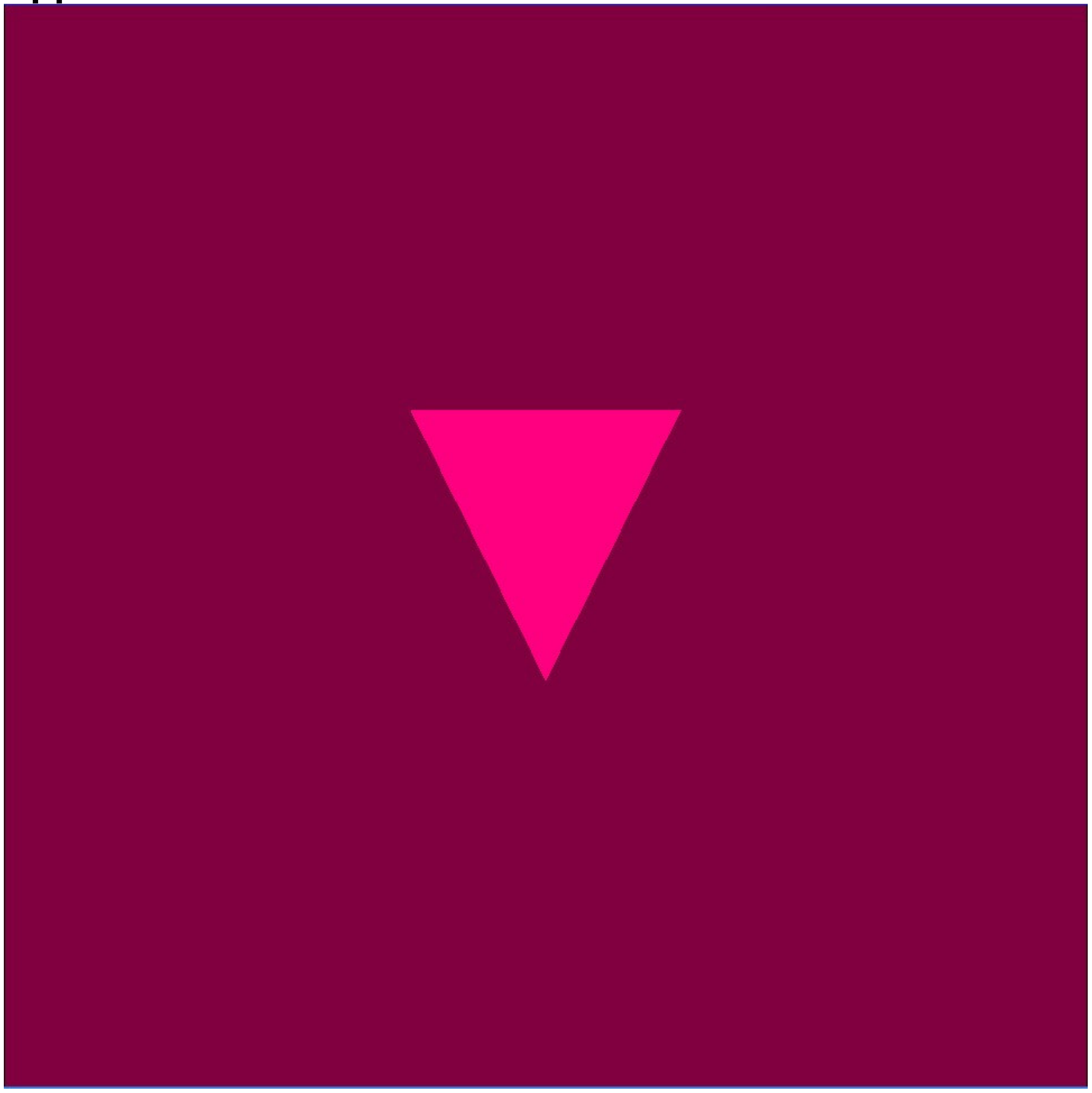
[11] “General-Purpose Computation Using Graphics Hardware”,
http://www.gpgpu.org

[12] D. Cohen, Z. Sheffer, “Proximity Clouds – An Acceleration Technique for 3D Grid Traversal”, Department of Computer Science Ben Gurion University, Beer-Sheva 84105, Israel

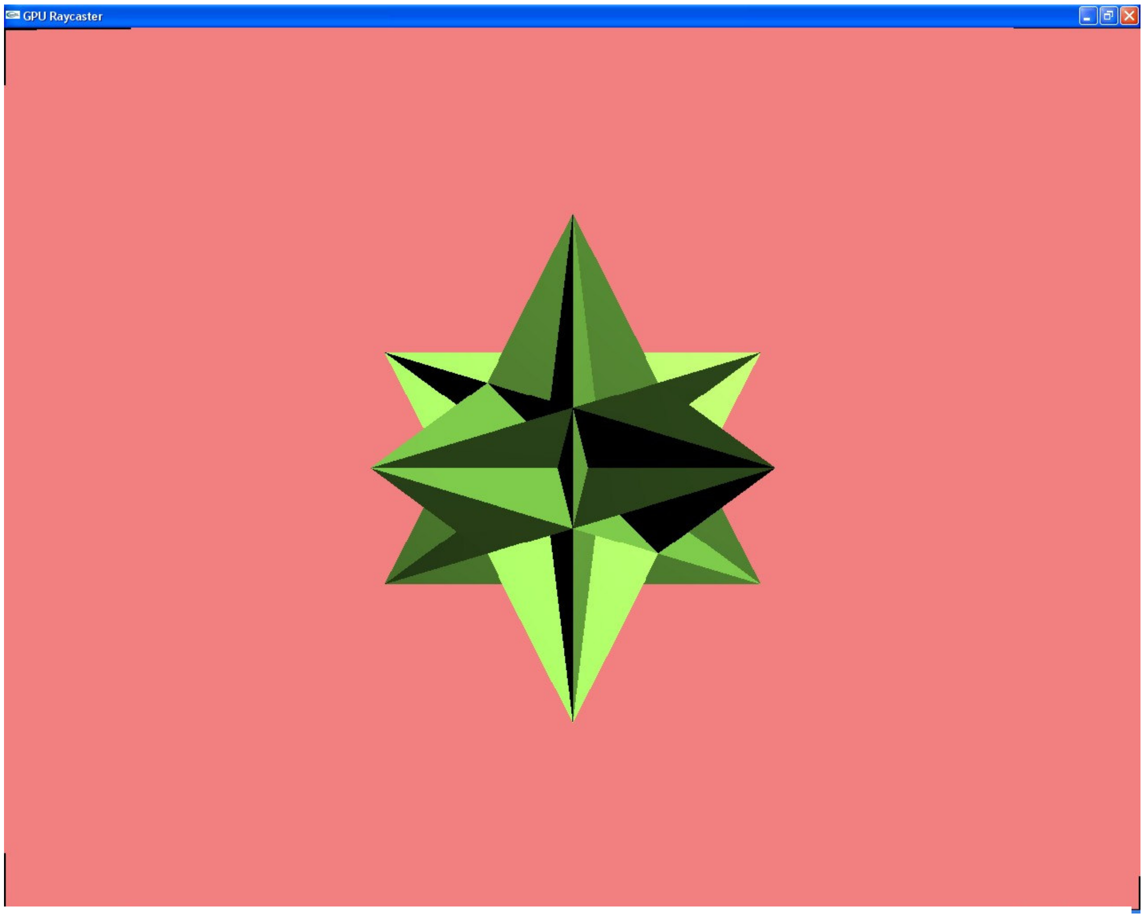
[13] D. Takahashi, “HP’s Shrek Factor”,
Corwallis Gazette-Times,
http://www.gazettetimes.com/articles/2004/11/16/news/business/monbiz01.txt

[14] S. K. Semwal, H. Kvarnstrom, “Directed Safe Zones and the Dual Extent Algorithms for Efficient Grid Traversal during Ray Tracing”,
Department of Computer Science University of Colorado, Colorado Springs

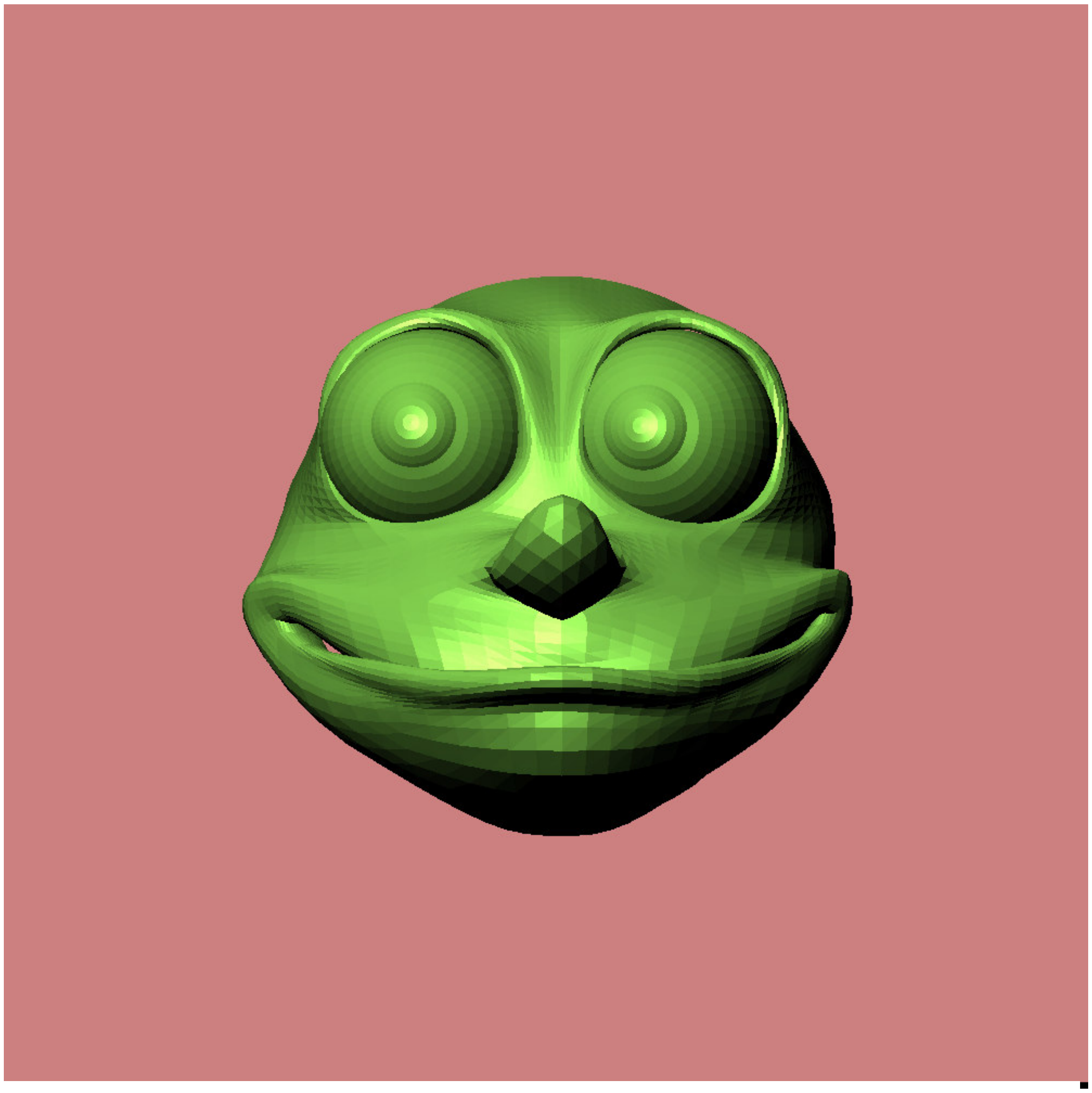
Appendix A: Screenshots

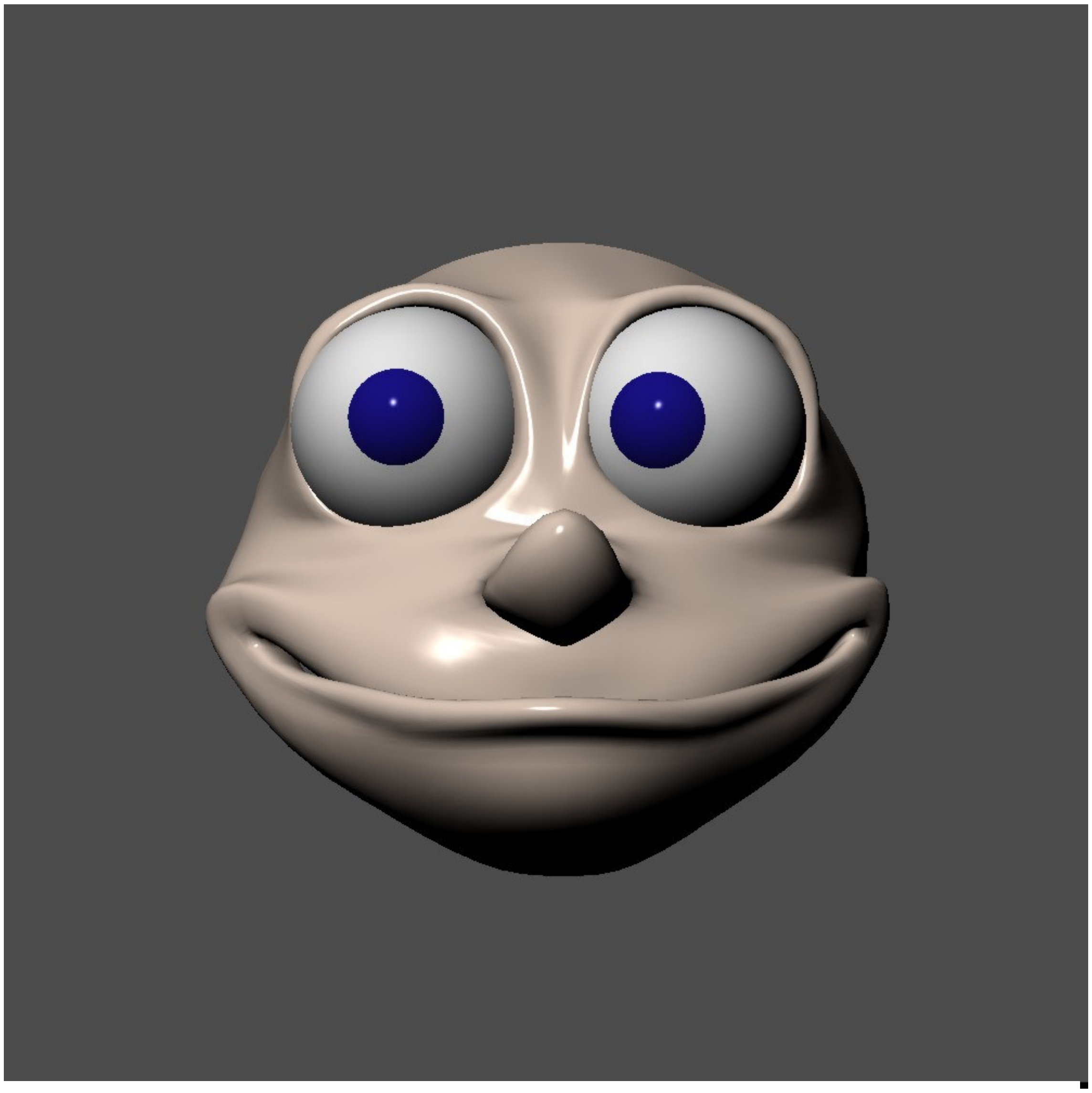


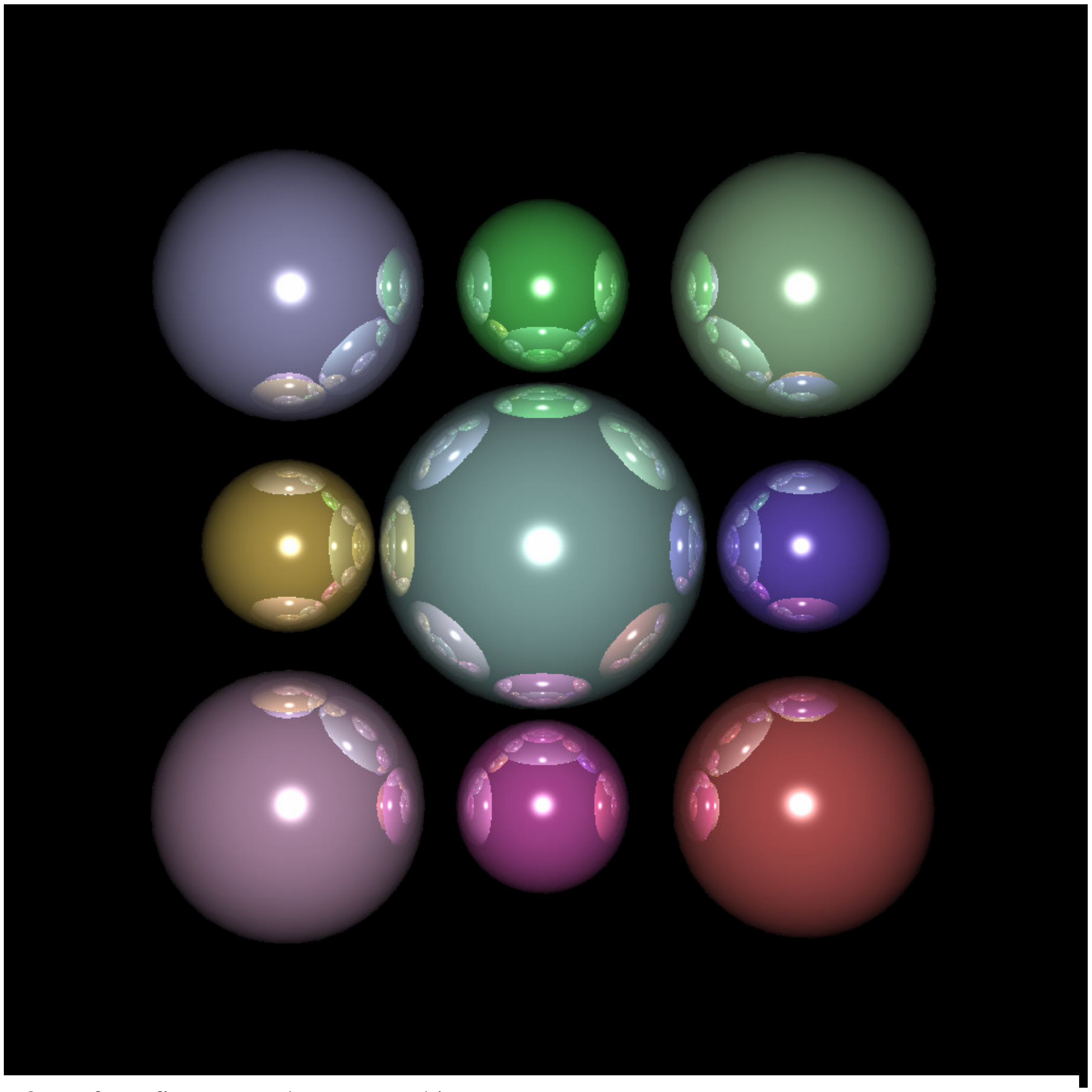
Our first raycasted triangle

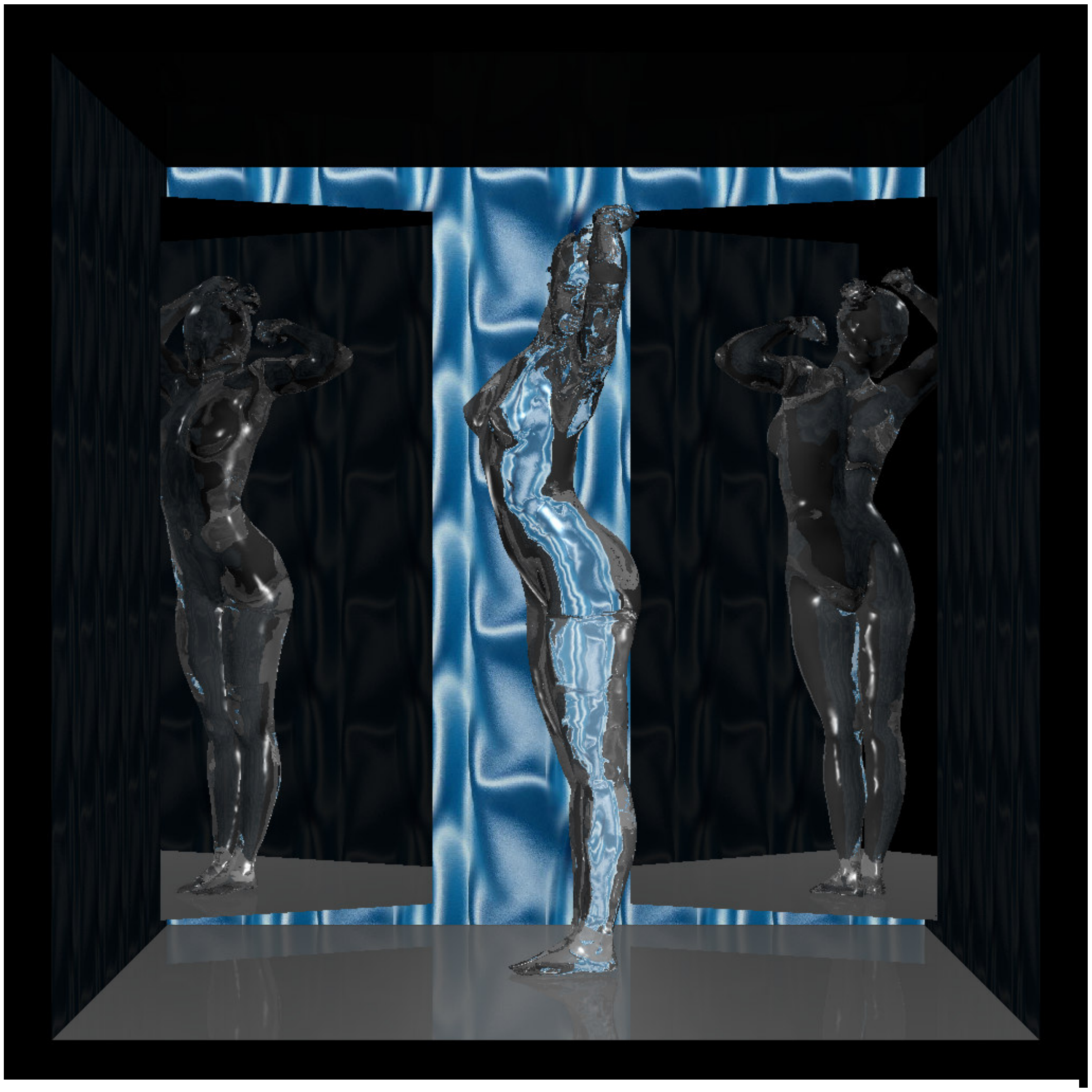


Or first model from a 3ds-file, 20 polygons.


A face, about 2000 polygons


Face again but this time with interpolated normals and materials.


One of our first correctly ray traced images.


Glass statue with about 45000 polygons.

Happy Buddha, more than 1 000 000 polygons!