LiU-ITN-TEK-A--08/050--SE

GPU-optimized generation of normal and color maps

Jimmie West

2008-04-17

Linköpings universitet
TEKNISKA HÖGSKOLAN

Department of Science and Technology
Linköping University
SE-601 74 Norrköping, Sweden

Institutionen för teknik och naturvetenskap
Linköpings Universitet
601 74 Norrköping

LiU-ITN-TEK-A--08/050--SE

GPU-optimized generation of normal and color maps

Examensarbete utfört i medieteknik
vid Tekniska Högskolan vid
Linköpings universitet

Jimmie West

Handledare Ulrik Lindahl
Examinator Björn Gudmundsson

Norrköping 2008-04-17

Abstract

Displaying a virtual world on the computer screen and interacting with objects in the virtual world in real-time requires a lot of processing. In order to handle this kind of processing computers are equipped with a GPU – Graphics Processing Unit. However, even though GPUs possess great processing capabilities there is still a limit to what they can do. As a consequence developers are constantly developing new techniques to reduce the workload on the GPU. One straightforward way to do this is by reducing the number of polygons in the 3D-scene but this implies that the scene will look less detailed. However, by applying color and normal maps to the 3D-models in the scene it is possible to make up for the loss of detail. This thesis presents a ray casting algorithm executed on the GPU that generates color and normal maps for simplified 3D-models. The main difference between the algorithm presented in this thesis and previously presented algorithms is that this algorithm is designed to process 3D-models that do not fit into the RAM – out-of-core meshes.

Contents

1 Introduction

This chapter is supposed to give the reader an introduction to the problem at hand.

1.1 Background

The visual effects of video games are improving at a rapid pace and as a consequence it takes more to impress the gamers. In order for the game developers to meet the demands of the gamers they constantly need to develop new techniques for solving problems associated with rendering. However, the developers are limited to the capabilities that current hardware can offer. This means that as the gamers urge for better looking games, the developers are pushing for more powerful hardware and especially more powerful graphics processing units (GPUs).

The freedom of visual effects programming was heavily increased by the introduction of shading languages. As a consequence the developers were able to modify the fixed function 3D-pipeline of the GPU by writing shader programs, so called shaders. Even though the only purpose of the GPU still was to render graphics, the power and increased flexibility of the GPU made it attractive to solve other types of problems. This gave rise to new concept called General-Purpose computation on GPUs (GPGPU) where the GPU is used to perform other tasks except just rendering. The parallel architecture of the GPU made it perfect to solve problems where a lot of data should be processed in the same way. However, at that time the GPU was still only considered for rendering so the possibilities provided by the shading languages were still rather limited, that is up until recently. Today there are GPGPU-programming languages available where the developers can program and exploit the power of the GPUs in almost any way they like to.

1.2 Problem description

Developers of real time 3D-applications constantly need to put their minds to work in order to use the power of the GPU in the best possible way. More polygons imply more work for the GPU, so one way to achieve better frame rates is to reduce the number of polygons that need to be rendered. However, reducing the number of polygons of a 3D-model, from now on referred to as a mesh, also reduces the amount of detail in the mesh. Using color and normal maps is one way to make the loss of detail less apparent. Normal and color maps can be generated in a number of ways. One way is to retrieve surface information from a detailed, high resolution mesh (HRM) and use this information to make a simplified, low resolution version of the same mesh (LRM) look more detailed. Basically this can be achieved by casting rays from the LRM towards the HRM, fetching surface information from where the ray hits the HRM and then storing the information in a texture map belonging to the LRM. Even though this approach may seem pretty straightforward the ray casting process can be done in several different ways. The processing time and the quality of the end result depend on, among other things, how the rays are being cast and traced.

My principal, Donya Labs, is primarily focusing on developing mesh simplification tools. One of their products is called Simplygon. For instance, Simplygon can be used to speed up the production phase for 3D-artists. Instead of having to create several instances of the same 3D-model with different levels of detail, the artists only need to create the most detailed version of the 3D-model and then use Simplygon to simplify it. By using this kind of tool a

lot of production time can be saved. A natural development of Simplygon would be to provide its users with a normal and color map generation tool. The goal of this project is to explore how such a tool could be developed and then implement a solution that solves the normal and color map generation task. The prerequisites of the project are as follows:

- The HRM consists of around 10^5 - 10^8 polygons.
- The LRM consists of around 10^4 polygons.
- The LRM is parameterized (it has texture coordinates associated with its vertices).

According to the prerequisites the solution should be able to handle extremely large meshes. Those meshes will be referred to as out-of-core meshes - meshes that do not fit into RAM.

The time it takes for a tool to complete a specific task has a major impact on the usefulness of the tool. As a guideline Donya Labs has noted that the processing time for the normal and color map generation tool should be within one minute. Since the ray casting process is highly parallel it should be implemented on the GPU. An important part of this project is therefore to understand the architecture of the GPU and explore how it can be used in order to speed up processing time.

1.3 Method

The workflow of this project is an iterative process which can be divided into following steps:

- Divide the main problem into subproblems.
- For each subproblem:
    - Read how similar problems have been solved.
    - Use an already existing solution or design a new solution.
    - Implement the solution.
    - Test the solution.
    - Evaluate the solution.
- Assemble the solutions to the subproblems in order to solve the main problem.
- Test the solution.
- Evaluate the solution.
- Optimize the solution.

A common way of solving a complex problem is to use the “divide and conquer” approach - dividing the main problem into different subproblems until the subproblems are manageable. When every sub problems have been solved the solutions can be put together to form a complete solution. Even though this solution is not the fastest or most robust solution possible it can still act as a good reference when trying out different ideas. In software engineering it is also preferable to have a complete solution in order to identify which parts of the solution that have the greatest impact on performance and need to be optimized. When those parts of the solution have been improved some other part of the solution might end up being the bottleneck. Then that part is improved. By iteratively work this way, the solution is constantly getting better and eventually it is good enough.

1.4 Delimitations

The main purpose of this project is to do research concerning the generation of normal and color maps on the GPU. A prototype of a normal and color map generation tool will be presented in the thesis. However, to create a fully functional product more time for testing and evaluation would have been needed.

1.5 Document overview

- Chapter 1: Introduction
    - This chapter gives the reader an introduction to the task at hand and explains why the task should be done.
- Chapter 2: Fundamental concepts of 3D-graphics
    - This chapter introduces the fundamental concepts of 3D-graphics and explains notations needed to understand the following chapters.
- Chapter 3: GPGPU
    - This chapter explains how the GPU can be used to perform tasks that are not necessarily associated with rendering.
- Chapter 4: Ray casting
    - This chapter introduces the concept of ray casting.
- Chapter 5: Using ray casting to generate normal and color maps
    - This chapter explains the GPU-accelerated algorithm that has been developed and implemented in this project.
- Chapter 6: Results
    - This chapter presents how well the prototype performs. Some details concerning octree construction and processing time are also presented.
- Chapter 7: Discussion
    - This chapter discusses and analyses the results.
- Chapter 8: Conclusion
    - This chapter concludes the work that has been done in this project and discusses if the goals of the project were met.
- Chapter 9: Future work
    - This chapter presents some ideas that may improve the performance of the algorithm. Some different approaches that might be interesting to examine are also discussed.

2 Fundamental concepts of 3D-graphics

This chapter gives an introduction to 3D-graphics and presents the fundamental vocabulary needed to understand the rest of this thesis. Some notations, such as different coordinate spaces, vary between different authors. However, I have chosen to adopt the notation used by the OpenGL developers since this is the API that has been used in this project.

2.1 Graphics Processing Unit

The Graphics Processing Unit (GPU), which is located on the graphics board, was originally developed in order to render graphics. Today the GPU can be used to solve several tasks which are not all necessarily associated with graphics rendering.

From one point of view GPUs and CPUs work in the same way. They fetch data and perform computations. The major difference between a CPU and a GPU is that the CPU works on data in a serial sequence, while the GPU works on data in parallel which makes it superior compared to the CPU when handling problems where a lot of data should be processed in the same way. The CPU is designed to run general-purpose programs while the GPU is designed to achieve a specific task - render graphics. The fact that the GPU only needs to perform a limited amount of different tasks has made it possible for the GPU vendors to create specialized hardware which is optimized to handle tasks associated to rendering (Owens 2005). This kind of specialized hardware would never be possible to have on a CPU.

The CPU must use a lot of transistors for caching and control flow while the GPU can use more transistors for data processing (NVIDIA 2007). The rapid development of GPU technology during the last couple of years has made the GPU superior compared to the CPU when it comes to the number of floating point operations per second (flops). In November 2006 NVIDIA GeForce 8000 series of GPUs was released. At that time the GPUs could do more than 300 billion flops, around six times more operations compared to an Intel core2 Duo processor. Even though the performance of a processor depends on more than solely the number of flops it can perform, it is apparent that the power within a GPU should not be underestimated.

2.2 Stream programming model

The GPU uses the stream programming model so in order to understand how the GPU processes data it is necessary to understand the concept of stream programming. In the stream programming model the data is represented by streams which is an ordered set of data where every element is of the same data type (Owens 2005). The streams are processed through kernels. A kernel is a program that takes one or more streams as input, processes every element in the stream in the same way and then creates one or more output streams. The elements in the input stream and output stream do not need to be of the same data type. When a data element of a stream is being processed by a kernel it cannot depend on any other element in the stream. If a stream element would need to wait for another element in the same stream to finish executing, this would have a serious negative impact on the performance of the application.

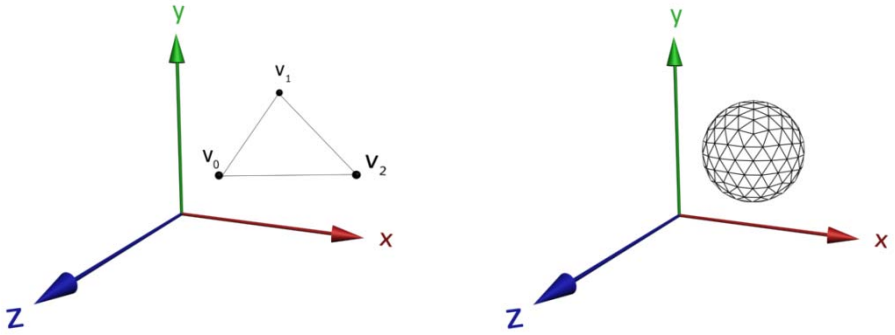
The design of the stream programming model makes data communication very efficient (Owens 2005). Initiating off-chip data transfers implies a performance penalty. By working with streams of elements instead of single elements it is possible to transfer a lot of data at once which means less initiation costs. Furthermore, by chaining together multiple kernels, stream data can be kept locally on chip which implies less off-chip communication and better performance. Another important performance issue is to avoid stalling. Assume that a stream element needs to do some high-latency data fetch. Because of the independence between stream elements the stream processors can continue working with another element in the stream while waiting for the data. In this way latency is hidden with calculations instead of using large data caches which would cost transistors, transistors that can be used for data processing instead. In order for this latency hiding to work it is important that the kernels have a high arithmetic intensity, meaning that the number of arithmetic operations is large compared to the number of memory operations.

By using the stream programming model it is possible to expose the parallelism of applications (Owens 2002). Owens argues that parallelism is exposed at instruction, data and task level. Instruction level parallelism is given by the fact that some operations that are performed on a stream element can be executed in parallel. For instance, assume that a stream of vertices should be multiplied with some constant k. The x, y and z coordinates can then be multiplied with k concurrently. Data level parallelism is given by the fact that kernels can handle several stream elements concurrently. Task level parallelism is exposed by the possibility to divide the work to be done by the kernels among several stream processors.

Apparently, the purpose of using the stream programming model is to process data in parallel. This kind of processing is very suitable for some types of tasks. Some algorithms, such as ray casting and graphics rendering, are very parallel in nature and thereby perfect to implement in a system using the stream programming model.

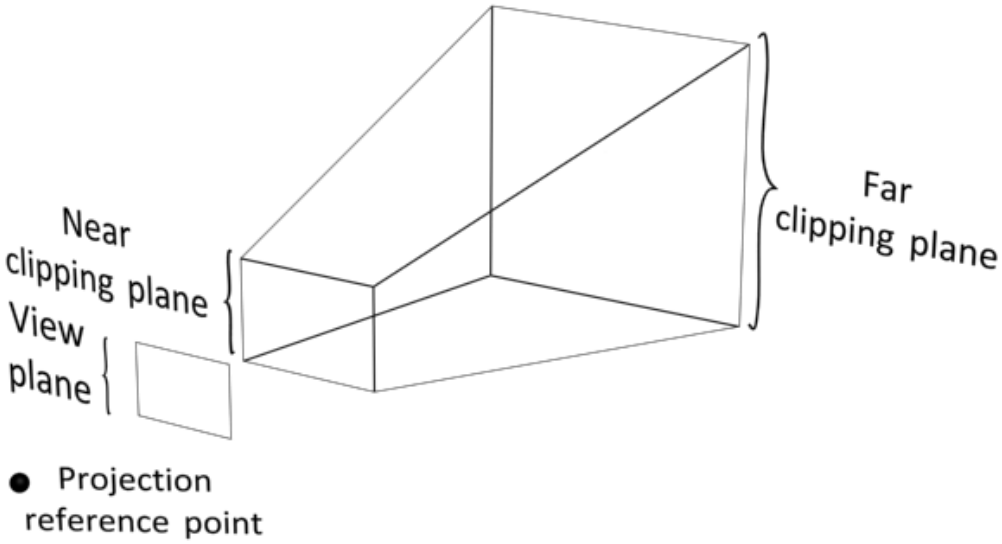
2.3 3D-graphics rendering

This section will present the most fundamental parts of 3D-graphics rendering. Rendering a 3D-scene onto a computer screen involves several steps. A 3D-scene usually consists of thousands of triangular shaped polygons, from now on just called triangles or polygons. A triangle, which is the simplest way to describe a surface, is defined by three points which are referred to as vertices. By connecting several triangles with each other it is possible to create 3D-models, so called meshes, representing things from the real world (see figure 2.1). Other primitives, except triangles, such as lines and points may also be used to fill the 3D-scene with content.



Figur 2.1: To the left is a polygon defined by vertex v0, v1 and v2. To the right is a mesh that consists of several polygons.

Once the content of the 3D-scene has been defined the next step is to define the viewing frustum. The viewing frustum, which is limited by the near and far clipping plane, is a volume that defines what part of the 3D-scene that should be considered for rendering (Hearn & Baker 2004). The view plane, which is located between the projection reference point and the near clipping plane, can be thought of as the computer screen (see figure 2.2).



Figur 1.2: The viewing frustum.

2.4 The 3D-pipeline

When the content of the 3D-scene and viewing frustum is set up the vertices can be sent to the GPU for further processing. The processing steps executed on the GPU are performed by kernel programs which form a chain that is referred to as the 3D-pipeline. The classic view of the 3D-pipeline has changed lately (NVIDIA 2006). However, even though the underlying GPU architecture has gone through some changes lately the classic view of the 3D-pipeline still gives a good picture of how GPUs process data. Basically the classic view of the 3D-pipeline can be divided into seven kernels (Owens 2005). Two of those kernels, the vertex and

fragment shader, may be exchanged by programs written by the user in order to achieve things such as special effects.

If the user chooses to exchange neither the vertex nor the fragment shader the fixed-function 3D-pipeline is used instead. In the fixed-function 3D-pipeline the vertex coordinates are transformed between several coordinate systems. Figure 2.3 illustrates the chain of transformations that are performed when the fixed-function 3D-pipeline is being used (Segal & Akeley 2006).

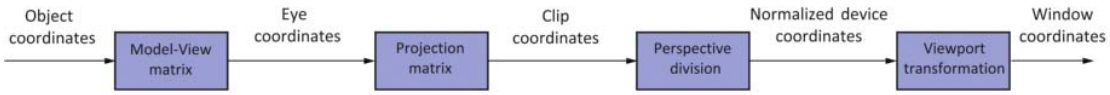


Figure 2.3: Vertex transformation chain taking place on the GPU.

Meshes are given in object space coordinates (Martz 2006). They may be resized and moved around in the 3D-scene by using scaling, translation and rotation operations. Those operations are concatenated and stored in the model-view matrix. The model-view matrix also contains operations that transform the vertex coordinates into eye space which is defined by the orientation of the viewing frustum. Then the eye space coordinates are transformed into clip coordinates by using the projection matrix. The projection matrix is just like the model-view matrix affected by the properties of the viewing frustum. Clip coordinates are used to simplify the process of determining if a polygon is located inside or outside the viewing frustum. Polygons located outside the viewing frustum are discarded and polygons which are part outside part inside the viewing frustum are clipped against the border of the frustum. The clip coordinates are transformed to normalized device coordinates by doing a perspective division. This transformation implies that meshes close to the viewer appear larger and meshes far from the viewer appear smaller. Then finally, by doing a viewport transformation, the normalized device coordinates are transformed to window coordinates which correspond to pixels on the computer screen.

In figure 2.4 the classic view of the 3D-pipeline is presented (Owens 2005). The first kernel in the 3D-pipeline is the vertex shader which processes a stream of incoming vertex coordinates. The fixed-function vertex shader transforms the incoming object coordinates to clip coordinates by multiplying them by the model-view matrix and then by the projection matrix. In the next kernel the transformed vertices are assembled to triangles or other primitives such as lines and points.

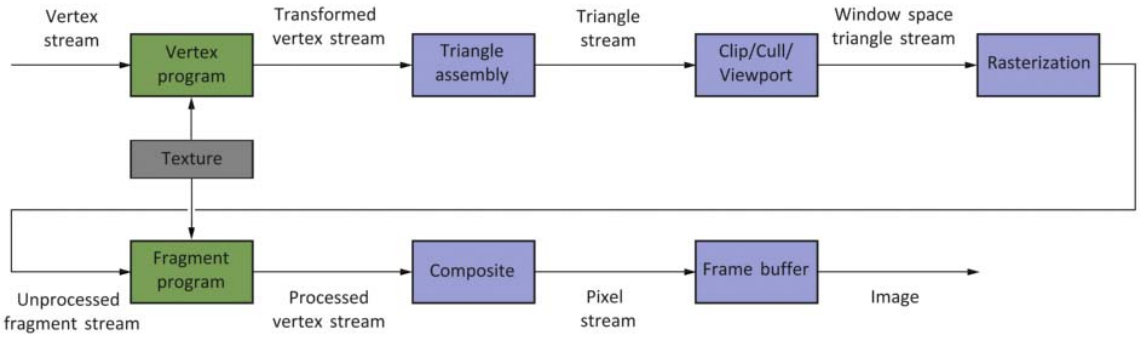


Figure 2.4: The 3D-pipeline.

In the third kernel the primitives are clipped against the viewing frustum. Backface culling, which is a method to neglect polygons facing away from the viewer, is also performed in this kernel. Then perspective division and viewport transformation is performed outputting a stream of triangles given in window space. The next kernel determines which pixels of the screen that should be filled with color information. This is called rasterization and results in a stream of fragments (see figure 2.5). The fragments are then processed by a fragment shader.

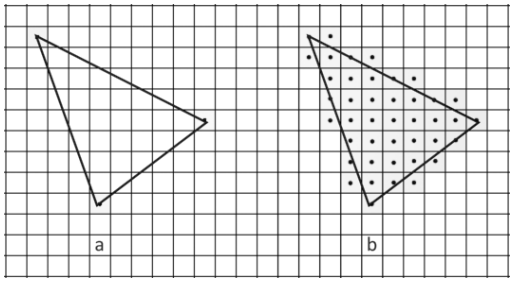


Figure 2.5: Before (a) and after rasterization of a triangle (b).

In order to make a mesh look more realistic it is possible to add color to the fragments by using texture maps. Basically a texture map is an array of values which is loaded and stored onto the GPU. A texture map may be 1D, 2D or 3D but 2D-textures are most common. The containers holding the texture values in a texture are referred to as texels. A texel usually consist of four channels - R, G, B, A. The first three channels stand for red, green and blue and specify the color and the fourth channel is called the alpha value and specifies the opacity of the color. In order to give the polygons the correct color each vertex is associated with a 2D-texture coordinate given in the interval [0, 1]. By interpolating the texture coordinates the GPU can access the correct color values in the texture and add color to the fragments representing the different polygons (see figure 2.6).

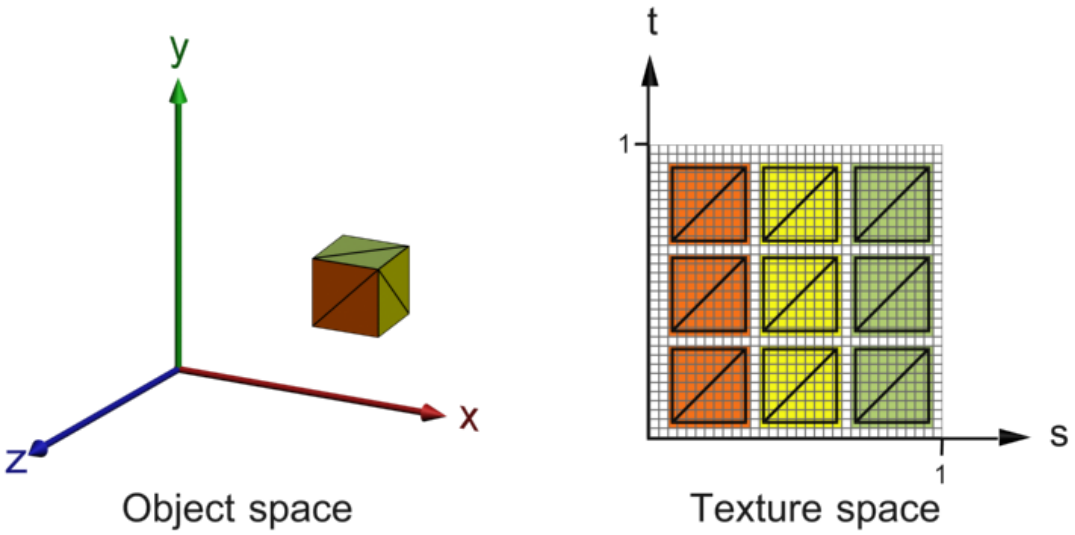


Figure 2.6: A textured cube defined in object space and the associated texture defined in texture space.

Usually some parts of a texture map do not contain any color values. Such parts may cause some serious artifacts when a mesh is being rendered. Figure 2.7 illustrates the problem. The teapot has some visible seams where the different texture areas, referred to as charts, are stitched together. This seam problem is discussed further later on in the thesis.

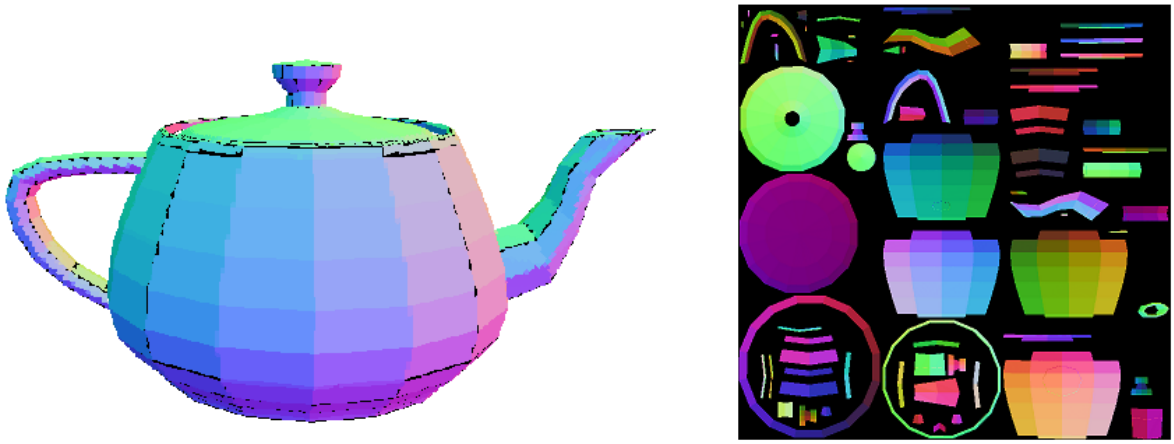


Figure 2.7: A textured teapot with visible seams and the associated texture.

Another way to make a mesh look more detailed is by applying lighting effects. Different rendering methods apply different lighting effects. Three common rendering methods are flat surface rendering, Gouraud surface rendering and Phong surface rendering (Hearn & Baker 2004). Flat surface rendering only uses the normal of a polygon to calculate how much light polygon receives. The amount of light that the polygon receives, referred to the surface intensity, is calculated by taking the dot product between the normal of the polygon and the direction to the light source. When Gouraud surface rendering is being used each vertex is assigned a normal value by averaging the normals of the surrounding polygons. Then the surface intensity for each vertex is calculated by taking the dot product between the vertex

normals and the direction to the light source. The surface intensities for the vertices are then interpolated across the polygon or polygons they define. The Gouraud surface rendering method yields a more convincing result compared to flat surface rendering. However, Phong surface rendering is even more convincing. When using the Phong surface rendering the vertex normals are interpolated across the polygons and then, when the fragments are being processed, the dot product between the interpolated normal and direction towards the light source is calculated to obtain the surface intensity (see figure 2.8).

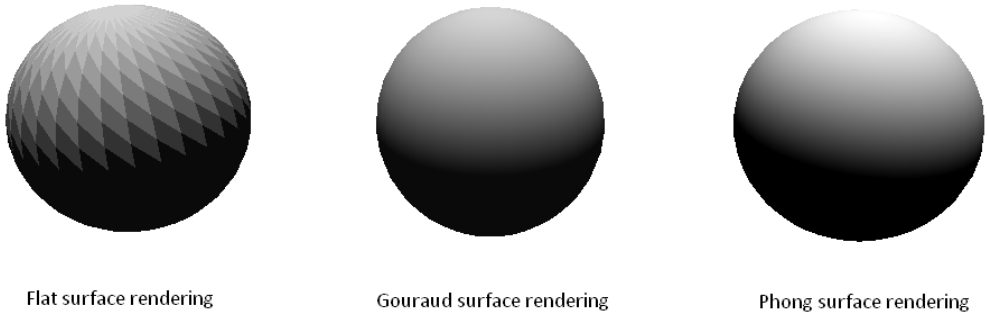


Figure 2.8: Three common surface rendering techniques.

There are many other methods to enhance the appearance of a mesh. One more such technique will be mentioned. It is called bump mapping and today it is a very common technique used in commercial games. Bump mapping is a general term for techniques in which a 2D-texture is used to add 3D-details to a mesh (Tarini et al 2000). One variant of bump mapping is normal mapping. A normal map is a 2D-texture that contains one normal for each texel. Figure 2.6 shows a color mapping example illustrating the mapping between a triangle in object space and a triangle in texture space. The same mapping concept is used for normal maps. As you can see in figure 2.6 each triangle occupies around 40 texels. This means that a triangle that usually has three normals, one for each vertex, may be described by 40 normals. During rendering, the normals stored in the normal map are used instead of the normals defined for the mesh. In this way smooth surfaces may be rendered having a lot more detail than before (see figure 2.9).

Rendering a simplified mesh with a normal map has some benefits compared to rendering a high resolution mesh that has the same amount of normals as the normal map. One benefit is that the normals stored in the normal map do not need to be processed through the whole 3D-pipeline like the vertices. The normal map may be applied directly in the fragment program. Another benefit is that a normal mapped mesh generally requires less data compared to a high resolution mesh.

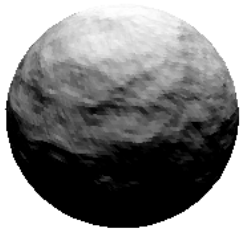


Figure 2.9: A sphere rendered using a normal map.

The composition kernel (see figure 2.4) determines which fragments should be visible in the final image by doing a depth test. The depth test involves comparing the depth values of fragments with the same window coordinates and then determining which of the fragments that is closest to the view plane – this fragment is kept. Some other operations such as blending and stencil tests are also performed in this part of the 3D-pipeline, however those operations will not be explained any further in this thesis. The output of the composition kernel is a stream of pixels representing the image. In the seventh and last kernel the pixels are assembled into the final image and stored in the framebuffer.

2.5 Communication between the CPU and the GPU

In order to simplify the communication between the CPU and the GPU it is necessary to use an API. The two largest APIs available for 3D-graphics rendering are OpenGL and Direct 3D. OpenGL, which is a cross-platform API, was originally developed by Silicon Graphics but since 1992 the development of OpenGL has been supervised by the OpenGL review board consisting of people from the major graphics vendors (Astle and Hawkins 2004). Direct3D is a part of the DirectX API developed by Microsoft and is thereby bound to the Windows platform.

Both OpenGL and Direct3D provide the user with hundreds of functions to change different rendering specific variables. The choice of API is, from my point of view, just a matter of taste and whether you would like to run your applications on other operating systems than windows. Even though there are some differences between the APIs most of the tasks that can be done using Direct3D can also be done using OpenGL and vice versa.

OpenGL uses a shading language called GLSL (OpenGL shading language) and Direct3D uses a shading language HLSL (High-Level Shading Language). The shading languages are used to write kernel programs which can be used to exchange the vertex and fragment shaders in the 3D-pipeline. Another common language is Cg (C for graphics) developed by NVIDIA which can be used together with both OpenGL and Direct3D. All three shading languages have a syntax that is rather similar to the C programming language.

2.6 Introduction to NVIDIA GeForce 8000 series

Up until recently the GPUs have had a fixed number of dedicated processors working with vertex processing and a fixed number of dedicated processors working with fragment processing. Since the release of NVIDIA GeForce 8800 GPU this has changed. Instead of having fixed amount of dedicated vertex and fragment processors, the processors used by the

GeForce 8800 GPUs can be used for both vertex and fragment processing (NVIDIA 2006). The most powerful GPU in the GeForce 8800 series, Geforce 8800 Ultra, has 128 stream processors. Some applications may need a lot of vertex processing while other types of applications need to process many fragments. In the older generations of GPUs this non-balanced situations would cause some processors to be idle, this is not likely to happen with GeForce 8800.

MIMD and SIMD are two common concepts associated with parallel data processing. MIMD stand for multiple-instruction, multiple-data and SIMD stand for single-instruction, multiple-data (Owens et al 2007). The stream processor in NVIDIA GeForce 8800 uses a form of MIMD processing called SPMD, single-program, multiple data. SPMD means that each processor executes the same program but every processor does not need to execute the same instruction concurrently as in SIMD processing. Even though GeForce 8800 uses the SPMD processing model it still processes some data in SIMD manner by dividing the data into SIMD groups.

Since graphics processing involves handling a lot of vector data it seems to be a natural choice to use specialized vector processing units on the GPU (NVIDIA 2006). Nevertheless, this is not as efficient as it may seem, at least not any more. Before NVIDIA designed the GeForce 8800 they analyzed hundreds of shader programs and discovered an increasing usage of scalar operations. Running complex shader programs having a mix of scalar and vector operations makes it difficult to get full utilization of the processor units. However, using scalar processing units would, according to NVIDIA, imply full utilization of the processors at hand. NVIDIA expects two times better performance using 128 scalar processing units instead of using 32 four-component vector processing units. Apparently GeForce 8800 has adopted the scalar processor design.

The GeForce 8000 series also introduced a new programmable shader called the geometry shader (Blythe 2006). Even though geometry shaders are not used in the solution presented in this thesis, it should be mentioned in order to have a more accurate picture of the 3D-pipeline. The geometry shader is located after the vertex shader and from our point of view it replaces the kernel performing the primitive assembly. The geometry shader takes vertices associated with the same primitive as input and then outputs a stream of primitives (Blythe 2006). The geometry shader adds flexibility to GPU programming by adding the possibility to transforming the incoming primitives, removing primitives, copying primitives and adding new vertices.

3 GPGPU

GPGPU, which stands for General-Purpose computation on GPUs, is getting more widespread and recognized. Basically it includes everything that has to do with computations performed on the GPU which are not necessarily associated with rendering. Ever since the introduction of shader languages, programmers have been able to change the fixed function 3D-pipeline and today a lot of non-rendering associated algorithms can take advantage of the parallel architecture provided by the GPU. This section will shed light on the GPGPU concept and explain how the GPU can be used to solve some tasks much faster compared to if they were handled on the CPU.

3.1 GPGPU with CUDA

Writing, compiling, linking and running shader programs requires a rather good knowledge of computer graphics in general, knowledge of a shader language and familiarity with some graphics API such as OpenGL or Direct3D. In order to reach a wider public NVIDIA developed a GPGPU-programming language called CUDA that run on the GeForce 8000 series of GPUs (NVIDIA 2007). CUDA is a C-like programming language that simplifies communication between the CPU and the GPU and provides the programmer with much more freedom when programming the GPU compared to if he or she would use a shader language such as GLSL. However, with freedom comes responsibilities and in order to create fast running applications using CUDA the programmer must be familiar with the pitfalls that exist. There are also other GPGPU-programming languages available but none of them has been used in the solution presented in this thesis, mainly because the problem at hand maps very well to the 3D-pipeline.

3.2 GPGPU with shaders

Creating general applications that takes advantage of the power within a GPU by using ordinary shading programs involves a some steps (Owens et al 2007). First it is necessary to determine which part or parts of the application that can be parallelized and would benefit from being executed on the GPU. Then the parallelizable parts are implemented as fragment programs. Then it is necessary to specify the input data that should be processed by fragment programs. Usually the input data is stored in textures which are transferred to the GPU. The fragment programs need an input stream to start executing and in order to create an input stream a stream of vertices must be passed to the GPU. The most common approach is to create a quad, a polygon having four sides, which is parallel to the view plane (see figure 2.2). When the quad is rasterized it generates an input stream of unprocessed fragment to the fragment program. To process n^2 elements the quad should cover n^2 pixels of the screen. The same fragment program is then executed for each element in the input stream.

The input values stored in the texture are accessed by passing texture coordinates to the vertex program (Owens et al 2007). The texture coordinates are then interpolated across the quad which means that every unprocessed fragment gets a unique set of texture coordinates which can be used to access the input values. It is important to note that the fragment program is not restricted to read the values stored at the specified texture coordinates, it may read any other

values from the texture as well. However, the writing capabilities in the fragment program are very limited. Since each fragment is associated with a certain pixel on the screen the fragment program can only output values to precomputed memory positions.

Usually the fragments are eventually transformed into pixels and stored in the fragment buffer. Another possibility is to store the output from the fragment program in a texture. Actually, it is possible to specify multiple render targets so that the fragment program can write to several textures. This makes it possible to reuse the output values as input values in subsequent GPU-processing. Another way to do this would be to read back values from the framebuffer to the CPU-side of the application, put those values in a texture and then issue another GPU-call. However, this is far less efficient compared to writing to a texture immediately and thereby keeping the data in GPU memory. The technique of writing to a texture instead of the fragment buffer is simply referred to as render-to-texture (Harris 2005).

Independence between the elements that are to be processed on the GPU is crucial to even consider implementing a solution on the GPU (Harris 2005). In other words, there cannot be any restriction concerning the order in which the fragments are being processed. Another important fact is that the algorithms implemented on the GPU should have high arithmetic intensity.

Branching efficiently on architectures using the SIMD processing model requires some considerations. If the branching condition evaluates differently on the fragments currently being processed the fragment program needs to process every instruction in both branch paths (Harris 2005). In other words, divergence between fragments being processed concurrently has a negative impact on performance and should therefore be avoided if possible. Moving the branching condition to an earlier stage is one solution to this problem.

For instance, assume we want to perform a fluid simulation on the GPU. The fluid is represented by fluid cells, that only contain fluid, and boundary cells, that contain both gas and fluid. The cells should be handled differently depending on the type. One solution is to have a single fragment program that branches over the different type of cells. However, if any of the fragments being processed concurrently are not of the same type we need to run both the code for the fluid cells and the code for the boundary cells - this is not efficient. A better solution is use two fragment programs, one for each cell type and then put cell type branching at a higher level.

4 Ray casting

Essentially ray casting is about finding the intersection points between a ray and a 3D object (Hearn & Baker 2004). The concept of ray casting should not be mixed up with ray tracing. Both ray tracing and ray casting involves finding intersection points between rays and 3D object, however, ray tracing also includes calculating new directions for the rays as they hit the objects in the scene. Ray tracing is a common method for rendering photo realistic scenes. By tracing millions of rays originating from the pixels of the screen it is possible to achieve reflection and refraction effects yielding very convincing images. Even though ray casting is inferior to ray tracing when it comes to rendering images, it is still useful for some other type of problems such as visibility detection.

4.1 Scene management

To check for the intersection point between a ray and every polygon in a mesh would be a very time consuming task. To reduce the processing time the search space should be reduced by using some kind of spatially dependent data structure that consists of bounding volumes. By storing indices to the polygons which reside within a bounded volume it is no longer necessary to check every polygon against a single ray. Instead, it is possible to start off by doing ray intersection tests against the bounding volumes. If a bounding volume is hit the algorithm can continue to check which if any polygon inside the bounded volumes is intersected by the ray. In this way it is possible to reduce the number of ray-polygon intersection test needed before finding a hit. Several scene management structures have been developed over the years. Uniform grids and octrees are two rather common data structures.

A uniform grid is a 3D-grid in which every grid cell, also referred to as voxel, has the same size (see figure 4.1). Creating a uniform grid is a very basic approach to subdivide a 3D-scene and it is rather easy to trace a ray through a uniform grid. However uniform grids consume a lot of memory. An octree is a hierarchical 3D-grid data structure in which the voxels either are leaves or contain eight child voxels. It requires more work to trace a ray through an octree compared to a uniform grid. The advantage of using an octree is that it usually consumes much less memory compared to a uniform grid.

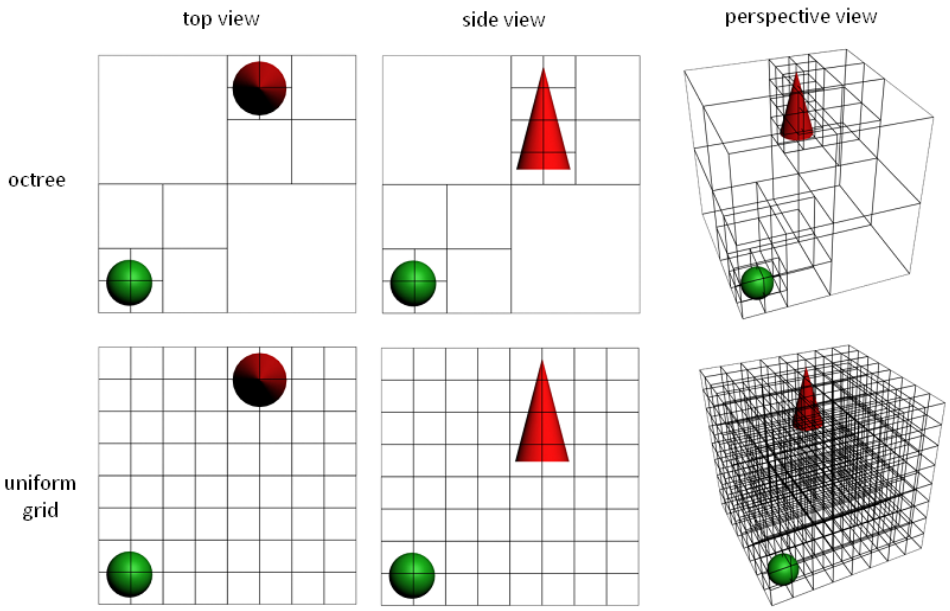


Figure 4.1: An illustration of an octree and a uniform grid.

4.2 Traversal algorithm

In 1987 Amantides and Woo presented a simple, yet fast, algorithm that finds every voxel in a uniform grid that is intersected by the path of a ray. Let us look at how this algorithm, which is a variant of the DDA line drawing algorithm, can be used to trace a ray in 2D space. Extending the algorithm to 3D space is then rather straightforward. A ray *R* is given by the following equation:

$$R: u + tv$$

Variable *u* specifies the origin of the ray, *v* is the direction of the ray and *t* specifies the length of the ray. Before the uniform grid can be traversed it is necessary to determine which cell that contains *u*. This is done by rounding off the coefficients of *u* downwards to the nearest integer value (see appendix A). Then four distance variables, tMaxX, tMaxY, tDeltaX and tDeltaY, should be calculated (see figure 4.2).

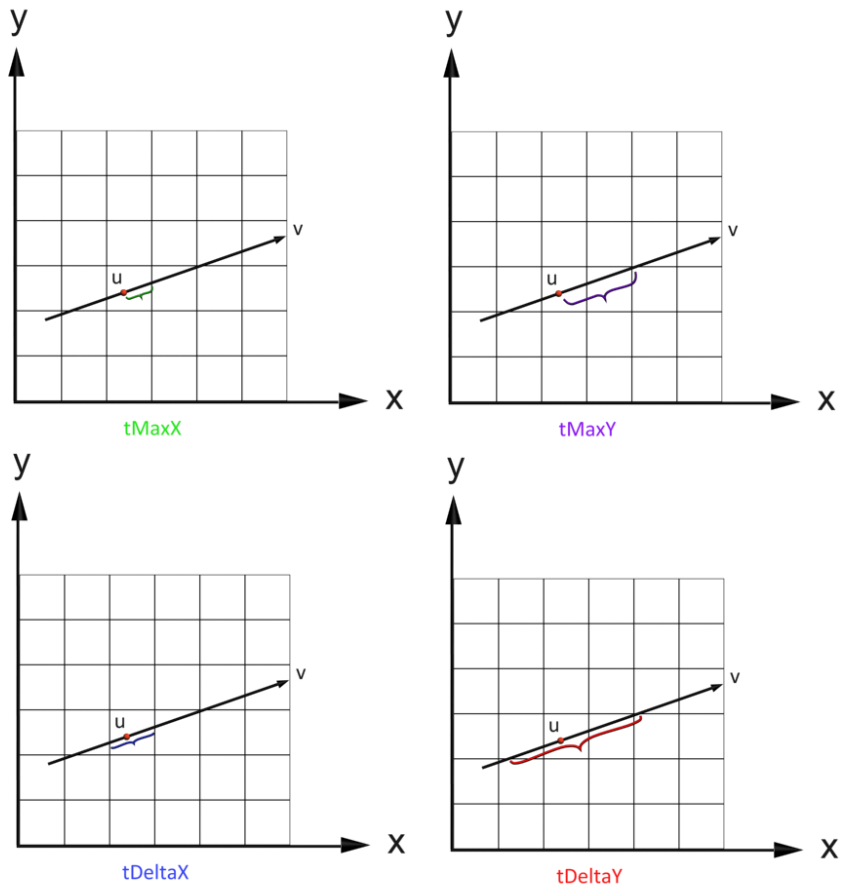


Figure 4.2: A 2D-uniform grid with a ray originating from the red dot.

The distance variables measured in units of *t*. The value of tMaxX is given by distance between *u* and the point where *R* intersects the first vertical line of the grid. The value of tMaxY is given by distance between *u* and the point where *R* intersects the first horizontal line of the grid. The distance the ray needs to travel to cover one cell horizontally is stored in tDeltaX and the distance the ray needs to travel to cover one cell vertically is stored in tDeltaY. The algorithm also needs to store two stepping variables, stepX and stepY. The value of stepX and stepY depend on the value of *v*. If the x-coefficient of *v* is negative then stepX is -1 otherwise it is 1. The value of stepY is given in correspondingly by observing at the y-coefficient.

When every variable is set the uniform grid can be traversed (see appendix A for details). The loop continues executing until a non-empty voxel is found or until the ray leaves the uniform grid. By modifying this algorithm it is possible to traverse other type of data structures such as octrees.

As mentioned above ray tracing and ray casting may involve tracing millions of rays. The fact that every ray should be processed in the same way makes it great to be handled by the stream processors located on the GPU. The simplicity of the algorithm presented by Amantides and Woo (1987) makes it rather easy to implement and debug on the GPU.

Some other traversal algorithms have been proposed over the years. For instance Revelles et al (2000) developed a recursive octree traversal algorithm that outperformed other octree algorithms available at that time. Naturally this hierarchical traversal algorithm is a bit more complicated compared to the traversal algorithm presented by Amantides and Woo. I cannot certainly say that it would not map well to the GPU, however the fact that the algorithm seem to be rather complex and that current generation of GPUs does not support recursion makes the thought of implementing it on a GPU unpleasant.

5 Using ray casting to generate normal and color maps

A lot of research has been done concerning the preservation of appearance during mesh simplification. In 1998 Cohen et al presented an interesting article on the subject. In their algorithm they use chart boundary constraints during the mesh simplification process. Simplygon, mentioned in chapter 1, does not take any chart boundaries into consideration during the simplification process so this method is not feasible together with Simplygon. Another approach (Cignoni et al 1998) is to use ray casting to generate the color and normal maps. This method is more general than the method described by Cohen et al since it does not add any constraints on the simplification process. Sander (2003) has continued to work on the ray casting approach and presents some interesting contributions such as seam removal methods.

In GPU Gems 3 Teixeira (2007) presents a method for creating normal maps by using the GPU. The method has served as a great inspiration source to this thesis, however, it does not handle out-of-core meshes. The algorithm uses a uniform grid to store the polygons of the HRM. Then rays are cast from the LRM towards the polygons of the HRM extracting and storing data from the intersection point between the ray and the polygons. The fact that every polygon of the HRM should be kept in GPU memory concurrently makes this approach impossible when handling out-of-core meshes.

There are some free texture map generation tools available online. NVIDIA Melody and AMD GPU MeshMapper are two such tools. NVIDIA Melody is a tool for generating normal maps but unfortunately it is very unstable and it crashes time and again. AMD GPU MeshMapper can create normal, displacement and ambient occlusion maps and it has several preferences to adjust the quality of the maps. However, neither Melody nor MeshMapper does support out-of-core meshes.

This thesis presents a method to extract surface information from high resolution meshes using ray casting executed on the GPU. The timing constraint and the fact that the algorithm should be able to handle out-of-core meshes have had a large impact on the design of the algorithm. Basically the algorithm can be divided into the following steps:

1. Load the LRM and the HRM.
2. Voxelize the HRM and transfer the voxel data to 2D-textures stored on the GPU.
3. Use the GPU to cast rays from LRM towards the voxelized HRM and extract surface information such as normals and colors.
4. Remove seams from the color and normal map.

5.1 Loading meshes

There are various kinds of file formats that may be used to store mesh data. In this project the obj file format (Murray & vanRyper 1996) is used. This file format makes the mesh loading procedure a straightforward task. The obj files contain vertex positions, vertex normals, texture coordinates and triangle information. For each vertex we must load three floating point values representing the position of the vertex, three floating point values representing the normal of the vertex and two floating point values representing the texture coordinate of

the vertex. In addition, it is necessary to store indices to vertex positions, texture coordinates and normals. Representing a triangle requires three integer values indexing the vertices of the triangle, three integer values indexing the texture coordinates of the triangle and three integer values indexing the vertex normals of the mesh.

Both a floating point value and an integer value are usually given by 4 bytes each. Let us assume that the number of polygons is twice as large as the number of vertices. The amount of data that should be loaded and stored to represent a mesh that consists of *x* polygons is:

$x \cdot (3 + 3 + 2) \cdot 4 + 0.5 \cdot x \cdot (3 + 3 + 3) \cdot 4 = 50x$ bytes of data

Let us continue and assume that there is one gigabyte of RAM available to store mesh data. How large mesh is possible to store by using one gigabyte?

$1024^3/50 = 21\,474\,836$ polygons

This is a quite large mesh. However, the solution should support even larger meshes. A solution to this problem would be to load a chunk of data, process it and then continue loading and processing another chunk of data. However, this kind of processing demands that the mesh data in the obj file is read in a specific order. It may be several hundreds or even thousands of rows between the vertices needed to specify a face and to search for those vertices would be too time consuming. To support out-of-core meshes the data stored in the obj file should be reordered so that the data can be loaded in chunks. Even though the current solution is not able to load out-of-core meshes it is important to note that everything in the algorithm generating normal and color maps supports out-of-core meshes.

When the meshes are loaded the solution keeps track of the maximum and minimum x-,y- and z-coefficients of the vertices. Those values are then used to create bounding boxes around the meshes. The bounding boxes are used when the meshes are scaled to fit within the unit cube.

5.2 Voxelization

In order to speed up ray casting the polygons of the HRM need to be partitioned into some spatially dependent data structure. However, since the solution should support out-of-core meshes it is impossible load and store every polygon of the HRM – at least not at once. One solution to this problem is to approximate the mesh by having each voxel store one normal vector and one RGB-color representing the polygons it contains. The normal vector is calculated by first summing up the normals of the polygons located inside the voxel. When all polygons have been processed the summed up normal is normalized. Initializing the voxels with RGB-colors requires a bit more work. First of all the HRM must be associated with a texture map from which the algorithm can gather RGB-colors. Then the algorithm needs to calculate the color polygon currently being processed. One simple way to do so is to take the average of the values stored at the texture coordinates associated with the vertices defining the polygon. Then the same summing procedure as with the normals is used but instead of normalizing the result the algorithm takes the average of the RGB-colors inside a voxel. This approximation method works very well with the mesh chunk loading procedure since it does not require that all polygons of the mesh are in memory at the same time. Next step is to

decide what kind of spatial dependent structure that will work with this kind of approximation method.

Let us consider the case of placing a mesh within a uniform grid containing x^3 voxels. Let us assume that the mesh has a somewhat cubical form it will occupy around $6x^2$ voxels, that is x^2 voxels for each side of the of the cubical grid. Also assume that we want to create a color or normal map that consists of 4096^2 texels and that one ray is cast for each texel. In order to have the possibility that any two rays do not hit the same voxel the side of the cubical grid would have to be:

$$1 = 6x^2/4096^2$$

$$x = \sqrt{\frac{4096^2}{6}} \approx 1672$$

How much GPU-memory is needed to represent a uniform grid having 1672^3 voxels? Let us continue doing some assumptions. For every voxel in the uniform grid the algorithm needs to store a value saying whether it is empty or not. This does only require one bit of data. Nevertheless, if the voxel not is empty the algorithm must store an index to a normal and RGB-color representing the polygons inside the voxel. The highest index value will be around:

$$6x^2 = 6 \cdot 1672^2 = 16773504$$

This means that 24 bits is needed to describe the indices to the normals and RGB-colors. To store a normal 16 bits for each coefficient is needed - 48 bits in total. Storing a RGB-color requires 24 bits of data, 8 bits per color channel. To summarize, approximately the total amount of GPU-memory needed will be:

$$(6 \cdot 1672^2 \cdot (24 + 48 + 24) + 1672^3)/(8 \cdot 1024^2) \approx 750MB$$

It is important to note that this requires perfect packing of the bits. This kind of packing is not realistic. A uniform grid is apparently not sufficient for our needs. Some kind of hierarchical structure to handle the voxelization step will be needed. Since octrees are far less memory consuming compared to a uniform grid this would be one way to go. However, to reach a resolution that is at least as high as 1672^3 requires eleven depth levels. Traversing an octree that has eleven depths on the GPU requires many texture lookups and for each depth level the algorithm would need to initialize and keep track of traversal variables. Another approach is to create a dynamical hierarchical uniform grid. In this kind of structure the user may specify one grid size for each depth level. This gives a higher degree of flexibility compared to using an octree.

Let us assume we would like to create a hierarchical grid having the same resolution as a uniform grid of 256^3 voxels. This could be achieved in a number of ways. For instance, one solution would be to have 8^3 voxels for the grid at the first depth level, 8^3 voxels for the grids at the second depth level and then finally 4^3 voxels for the grids at the third depth level.

Another solution would be to have 32^3 voxels for the grid at the first depth level and 8^3 voxels for the grid at the second depth level. Apparently this approach is much more flexible than if the algorithm would use an octree. Nevertheless, having large grids still requires a lot of memory so it is a matter of finding a good balance between memory consumption and maximum depth level of the hierarchical structure.

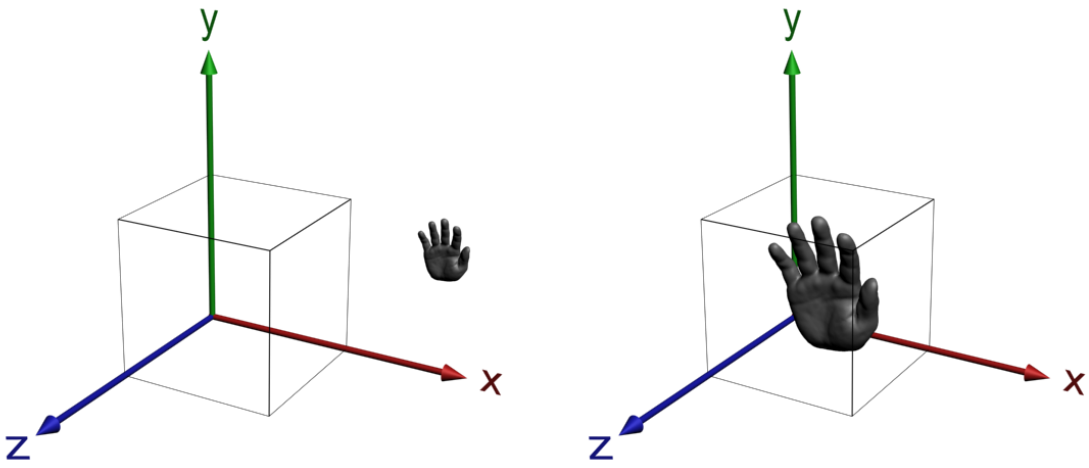


Figure 5.1: Rescaling the HRM so it resides within the unit cube

The dynamic hierarchical data structure is created recursively in depth first order on the CPU side of the application. Before the HRM can be voxelized the HRM must be translated and scaled so it resides within the unit cube (see figure 5.1). This operation is straightforward since the HRM is associated with a bounding box. The algorithm also needs to create the voxels at the first depth. Then each polygon of the HRM is processed sequentially. First the algorithm retrieves the normal and RGB-color associated with the polygon. The polygon normal is calculated during the mesh loading process and can be easily retrieved. The RGB-color requires a bit more work. First off the HRM needs to have texture coordinates and a texture associated to it. Then the texture coordinates for each vertex of a polygon can be used to get corresponding color values. Then those values are averaged to give the polygon a RGB-color.

The easiest way to explain the voxelization algorithm is probably by showing an example. The processing of a single polygon from the HRM is illustrated in figure 5.2. The resolution of the grid is $2 \cdot 4 \cdot 4 = 32$. Before performing any intersection tests the active grid should be translated to the origin of the coordinate space and then rescaled so that the positions of the voxels can be given in integer values. This simplifies addressing the voxels. The polygon currently being processed should be translated and rescaled in the same way as the grid (see figure 5.3). Then it is easy to check which voxels the polygon might intersect by rounding off the vertex positions of the polygon to the closest integer value downwards. Then the algorithm loops through the voxels that might be intersected by the polygon and perform an accurate intersection test (Ericson 2005) between those voxels and the polygon. If a voxel contains the polygon a new grid is created for that voxel and then the algorithm continues processing it. Assume that the algorithm is processing a translated and rescaled voxel grid that consists of 4^3 voxels and a translated and rescaled polygon is given by:

$$v0 = \begin{pmatrix} 0.5 \\ 0.7 \\ 1.2 \end{pmatrix} \; v1 = \begin{pmatrix} 2.5 \\ 1.7 \\ 1.2 \end{pmatrix} \; v2 = \begin{pmatrix} 3.5 \\ 0.8 \\ 1.9 \end{pmatrix}$$

Rounding off the coefficients of *v0*, *v1* and *v2* downwards to the closest integer results in:

$$v0 = \begin{pmatrix} 0 \\ 0 \\ 1 \end{pmatrix} \; v1 = \begin{pmatrix} 2 \\ 1 \\ 1 \end{pmatrix} \; v2 = \begin{pmatrix} 3 \\ 0 \\ 1 \end{pmatrix}$$

This means that the algorithm must perform a voxel-against-polygon intersection test for every voxel within the following interval:

$x = [0,3], \qquad y = [0,1], \qquad z = [1,1]$

The number of intersection tests required is 3·2·1 = 6 instead of 4·4·4 = 64.

Since the grid at depth 0 is already located at the origin no translation is needed but the algorithm still needs to rescale the grid according to the side of the grid (figure 5.3.a). The top right voxel contains the polygon so the algorithm creates a grid having 4^3 voxels. Then the grid and polygon are translated and rescaled (see figure 5.3.b) and the algorithm continue by performing intersection tests on depth 1. On depth 1 there are four different voxels that contain the polygon. They are processed in the same way as previous voxels, that is create voxel grid, translate, rescale and perform intersection tests. The translation and scaling for the voxel located down to the left is illustrated in figure 5.3.c.

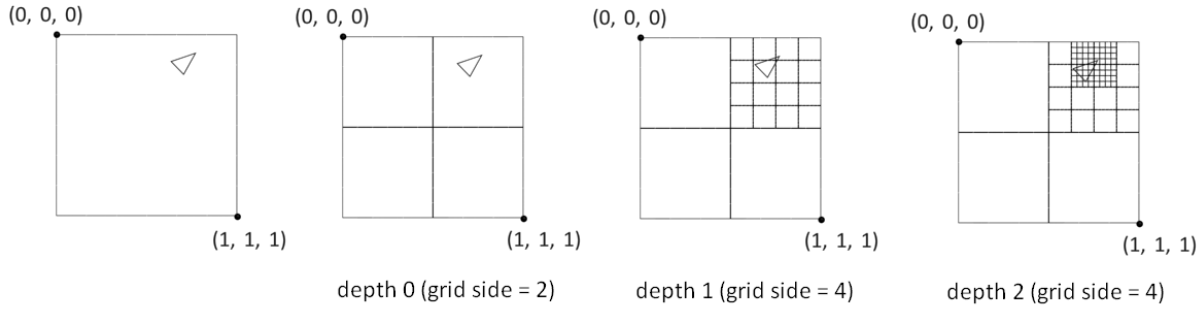


Figure 5.2: Voxelization process of a single polygon from left to right. The hierarchical uniform grid is shown from above.

When the algorithm reaches the maximum depth of the hierarchical grid the normal and RGB-color associated with polygon currently being processed need to be stored “within” the voxel or the voxels that contain the polygon. A voxel at the maximum depth may contain several polygons but since each voxel may only represent one normal and one RGB-color the normals within the voxel are summed up, and so are the RGB-colors. When every polygon has been processed the sum of the normals are normalized and the sum of the RGB-colors are averaged resulting in one normal and one RGB-color representing the voxel.

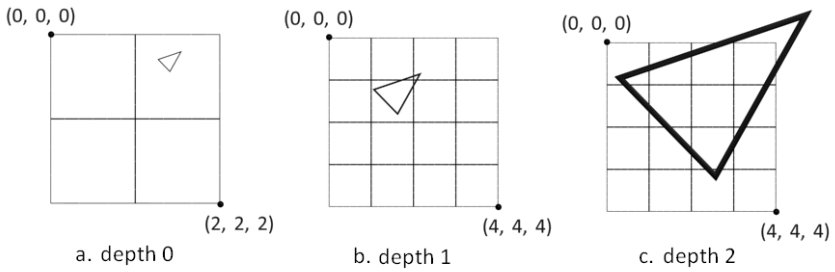


Figure 5.3: Translation and scaling of the polygon for different depths.

The fact that a voxelization involves a very specific task, that is performing intersection tests, and that it should be applied to a very large amount of elements, indicates that it might be a suitable task to be handled on the GPU. However, the hierarchal data structure must be created whilst the polygons are being processed. This creates a dependency between the polygons, which makes it less appealing to perform this processing on the GPU.

5.3 Memory handling

During the voxelization step an array is created that represents the voxels in the hierarchy – the voxel array. For empty voxels the algorithm stores -1, for voxels that contain a voxelgrid it stores an index to the voxelgrid and for voxels that contain a normal and a RGB-color value an index to where the normal and RGB-color are located is stored. The array is, just like the hierarchical uniform grid, created in depth-first order (see figure 5.4). The normals are stored in a separate array and so are the RGB-colors. Since the number of RGB-colors and the number of normals are the same the normal located at position *x* in the normal array will be associated with the RGB-color located at position *x* in color array. This means that the algorithm only needs to store one index for each voxel.

array position	0	1	2	3	4	5	6	7	8	9	10	11	…	72	73	74	75	…
voxel ID	0	1	2	3	4	5	6	7	3.0	3.1	3.2	3.3	…	3.2.0	3.2.1	3.2.2	3.2.3	…
indices	520	1096	-1	8	-1	-1	1736	2760	-1	-1	72	-1	…	-1	-1	-1	-1	…

Figure 5.4: The voxel array.

In order to perform ray casting on the GPU the voxel data need to be stored in 2D-textures and then transferred to the GPU. The array containing the indices (see figure 5.4) is stored in a texture which will be referred to as the voxel texture. Each texel in the voxel texture represents four voxels, one for each channel. The voxel texture (see figure 5.5) is created by copying the values stored in the voxel array. The first four voxels indices are stored in the texel positioned at (0, 0), the next four voxels are stored in the texel positioned at (1, 0) and so on. When the border of the texture is reached the following voxel indices are added the next row. The indices may be rather large so 32 bits is required for each channel. The current

solution uses 32 bit floating point values to store the indices. Eventually this should be changed so that the texture uses 32 bit integer values instead.

The normals and RGB-color also need to be transferred to the GPU. The RGB-color may have an alpha value associated with the color so it requires four values for each texel but the normal does only require three values so it does not need the alpha channel. The alpha channel may be removed from the texel representation by choosing another internal format for the texture. Another way to reduce the memory consumption is to reduce the precision of the values stored in the texture. Neither the normal nor the RGB-color need 32 bit precision floating point values, 16 bit precision floating point values is sufficient. The precision of the values used in the color texture could actually be reduced even further but because of implementation details it still requires 16 bit precision values.

The sizes of the textures containing the indices, normals and colors are determined dynamically. In the current solution quadratic textures are being used. The side of those textures is 256*n*, *n* being an integer. By restricting the side of the texture to a multiple of some static value, such as 256*n*, makes it easier to transfer the data into the textures.

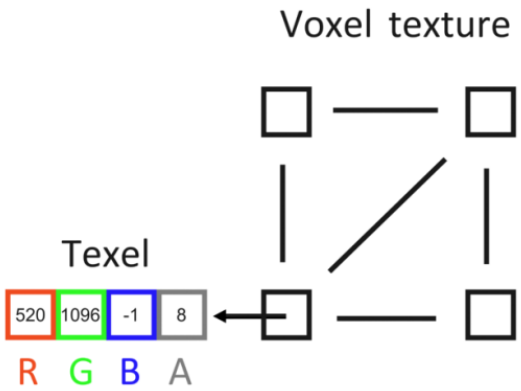


Figure 5.5: An illustration of the voxel texture. The first texel contains indices for the first four voxels in the voxel array.

5.4 Ray casting on the GPU

The voxel indices, normals and RGB-colors have been stored in textures and transferred to the GPU so the next step is to cast rays against the grid of voxels representing the HRM and extract normals and RGB-colors.

First the LRM must be translated and scaled so it resides within the unit cube. In this way the LRM and the voxelized HRM will be at approximately the same position (see figure 5.6), which is crucial to retrieve the correct values during the ray casting procedure. The ray casting algorithm is executed in a fragment program. During ordinary graphics rendering the processed fragments are assembled into an image and stored in the framebuffer. However, during the ray casting procedure the output of the fragment program should not be stored in the framebuffer, the output should be stored in a texture. The redirection of the output from the fragment program may be performed by using an OpenGL feature called FBO - Frame

Buffer Object. By binding a FBO with a texture and activating the FBO before rendering the output of the fragment program is written to the texture instead of the framebuffer.

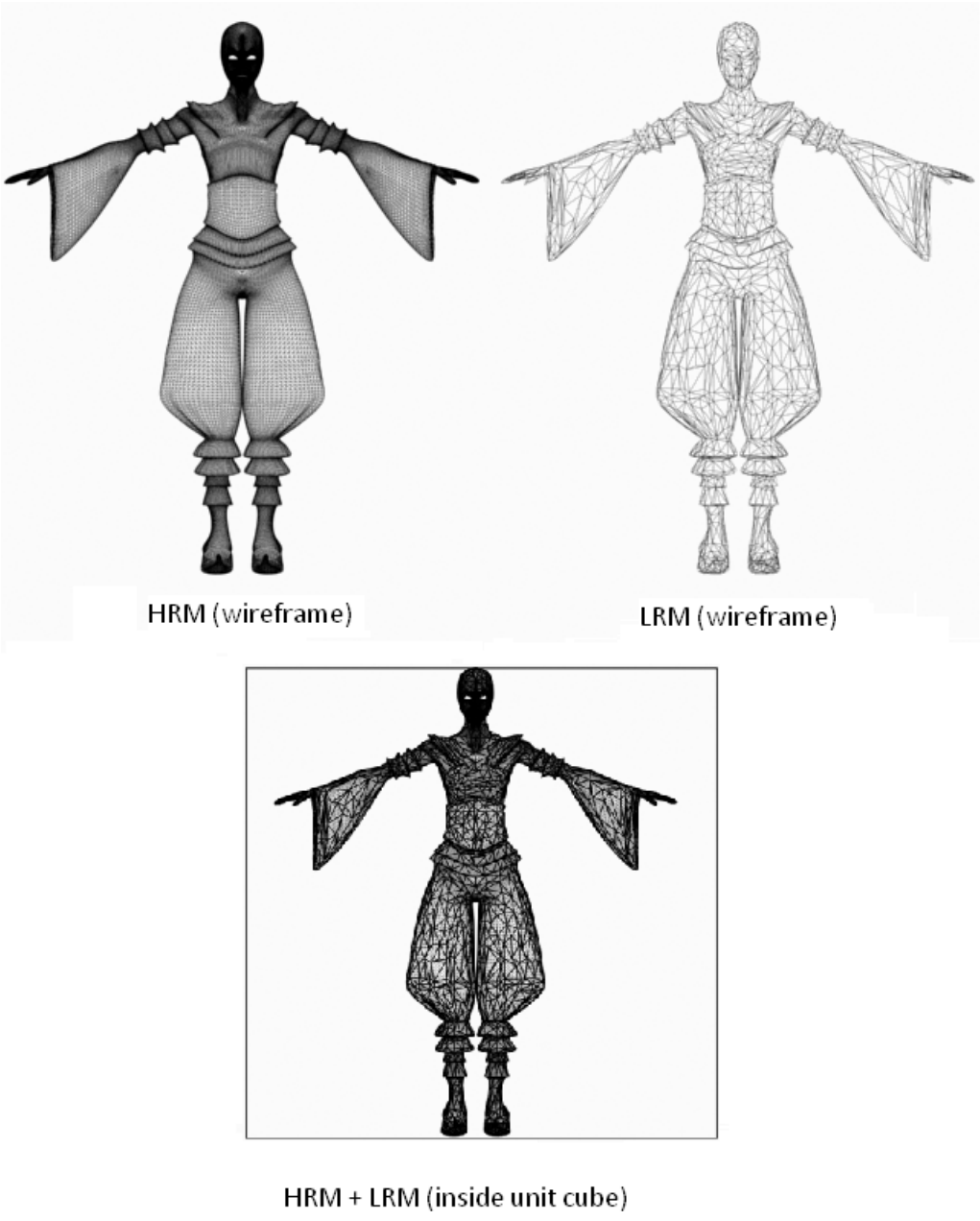


Figure 5.6: The HRM and the LRM placed within the unit cube.

The ray casting procedure requires that every polygon of the LRM is associated with a certain area in texture space (see figure 5.7). Consequently every vertex must be associated with a point in texture space. Before the ray casting procedure is executed the areas defined in texture space are empty. The goal is to fill those areas with information by casting rays towards the voxelized HRM.

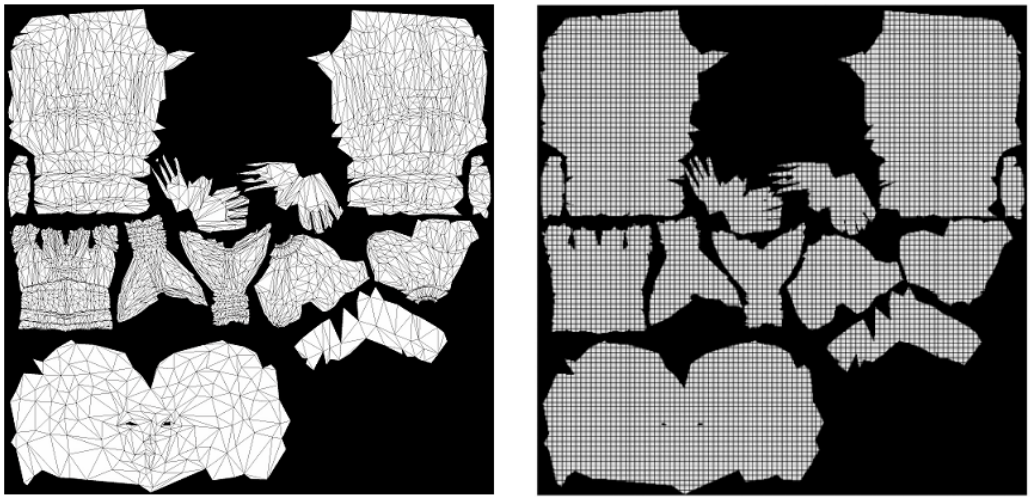


Figure 5.7: An empty texture associated with the LRM with visible triangle edges (a) and with visible texels (b).

The fragment program requires an input stream in order to start casting rays against the voxelized HRM. The input stream is generated by feeding the vertex program with a stream of vertices, texture coordinates and normals belonging to the LRM. In ordinary rendering the vertices define the surface to be rendered, however in this case the texture coordinates define the surface. Since texture space is defined in 2D this results in a flat surface that consists of several triangles in a plane (see figure 5.7). Before passing the texture coordinates to the GPU they need to be scaled according to the size of the map that should be generated. For instance, if we would like to generate a map with 4096^2 texels then we need to multiply each texture coordinate by 4096. When the texture coordinates have been scaled they are passed to the vertex program, transformed into clip coordinates and eventually they are assembled into 2D-triangles which are rasterized. The vertex positions and normals are interpolated across the rasterized surface. The interpolated vertex positions define the starting positions of the rays and the interpolated normals define the directions of the rays (see figure 5.8).

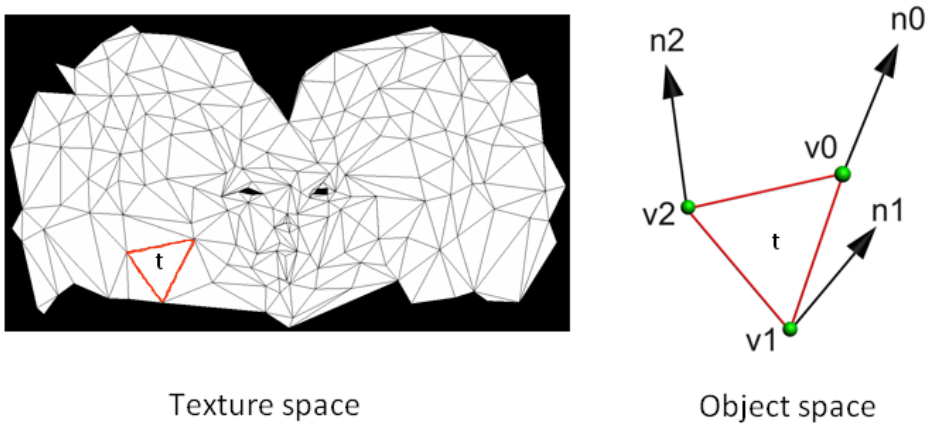


Figure 5.8: Triangle *t* is rasterized and for each pixel that *t* covers one ray is cast from the LRM towards the voxelized HRM. The starting position of the ray is given by interpolating v0, v1 and v2 which define *t* in object space. The direction of the ray is given by interpolating n0, n1 and n2.

The traversal algorithm developed by Amantides and Woo (1987) can be modified and used to traverse a hierarchical grid. The modification of the algorithm involves adding a translation and scaling operation just like the method used to reduce the number of intersection tests between polygons and voxels (see section 5.2). In order to traverse the grid it is necessary to know the direction and origin of a ray and the maximum depth of the hierarchical voxel grid. The traversal algorithm (see algorithm 5.1) begin by translating the ray origin $-v$. This corresponds to the translation needed to position the voxel grid, currently being processed, at the origin of the coordinate space (see figure 5.9). Then the ray origin is scaled according to the size of the voxel grid. The traversal parameters described in section 4.2 must also be calculated (see appendix A for details). Then the actual traversal part of the algorithm is executed.

```
bool TraverseVoxelGrid(Ray ray, SurfaceInfo surfaceInfo, integer maxDepth, integer depth,)
1. - Translate the ray origin –v and scale it according to the size of the voxel grid
2.  - Calculate traversal parameters
2.     - Traverse the grid while ray is inside the grid defined for the current depth
3.        - If the ray hits a non-empty voxel VE
4.           - If depth is equal to maxDepth (It will contain surface info)
5.              - Store surface information in surfaceInfo
6.              - Return true
7.        - Else
8.           - If (TraverseVoxelGrid(new_ray,  surfaceInfo, maxDepth, depth + 1))
9.              - Return true
10. - Return false
```

Algorithm 5.1: Pseudo code to traverse a hierarchical uniform grid recursively.

If the ray hits a voxel that is non-empty the algorithm checks whether the maximum depth is reached. If the maximum depth is not reached the algorithm calculates a new ray origin that depends on how far the ray has traveled. Then the algorithm performs a recursive call and proceeds traversing the voxel grid on the next depth level. If the ray hit a non-empty voxel on the maximum depth level it implies that the voxel contains surface information. The surface information is stored and then the algorithm is terminated by returning true.

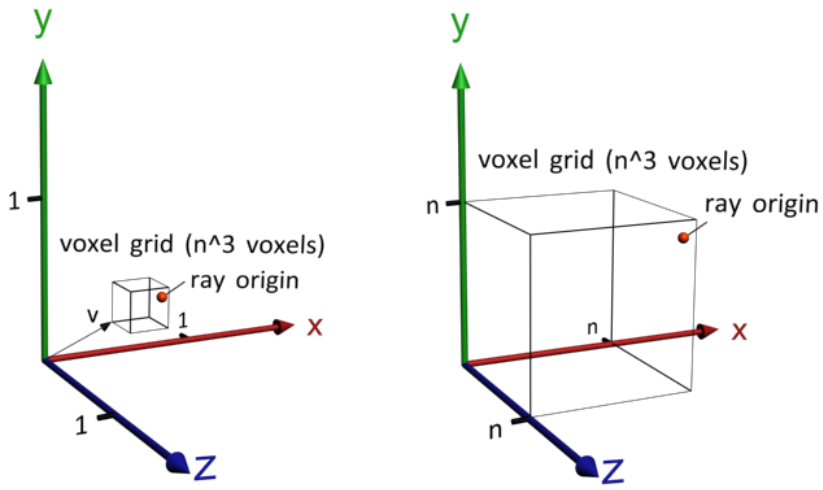


Figure 5.9: Before the algorithm may begin traversing the hierarchical voxel grid it must translate and scale the ray origin according to the position and size of the voxel grid.

The traversal algorithm does either extract RGB-colors or normals. It would be possible to extract both values simultaneously by using multiple render targets. However, having the normal texture, color texture, one texture to hold extracted normals and one texture to hold the extracted RGB-colors on the GPU at the same time would require too much memory.

In order to extract the correct value of the HRM the ray must be cast in two directions (see figure 5.10.a). First the ray is cast in the direction of the interpolated normal and then the ray is cast in the opposite direction. The traversal algorithm keeps track of how far the first ray has traveled before the HRM was hit. This value is used when the second ray is cast. If the second ray has traveled further than the first ray the algorithm can terminate because no better value will be found. Teixeira (2007) presents another approach in which the rays only need to be cast in one direction. This is done by extruding the LRM so every point on the HRM lies within the LRM. However, performing this extrusion process automatically on meshes with a complex topology is far from simple. Since the solution Donya Labs needs cannot demand user interaction Teixeira's solution is not sufficient

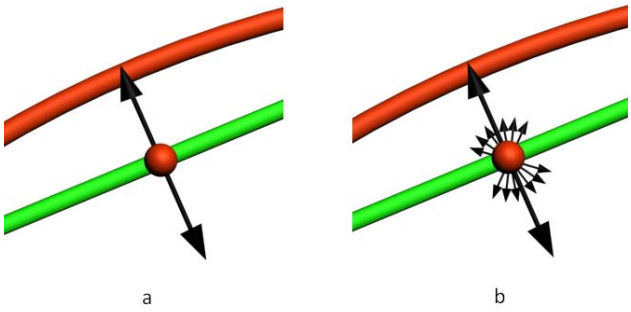


Figure 5.10: Single sampling (a) and multisampling (b). The green line represents the LRM and the red line represents the voxelized HRM. The red dot is the origin of the ray.

One way to improve the visual quality of the color and normal maps is to cast even more rays for each texel in the map. For each ray origin the algorithm could cast rays in several directions (see figure 5.10.b), accumulate the gathered values and then take the average of those values. Another approach is to have a primary ray that has a greater influence on the end result compared to the other rays that are cast from the same position. Nevertheless, casting several rays imply a lot more work for the GPU so as usual it is a matter of balancing quality and performance. The current solution uses single sampling to reduce processing time. Experiments show that multisampling does improve the visual quality but it has a serious negative impact on execution time.

GLSL does not support recursion (Kessenisch 2006) so in order to implement the traversal algorithm (see algorithm 1) on the GPU it must be rewritten to an iterative form. This does not require much extra work; nevertheless it implies a lot of duplicate code. Another negative consequence is that the number of depth levels the hierarchical voxel grid should be able to adopt must be specified before the fragment program is compiled. Every depth in the voxel grid needs a traversal function of its own which means that there will be several copies of more or less the same code.

5.5 Remove seams

Section 2.4 shows an illustration of a teapot having problems with visible seams. The problem often arises because the hardware uses bilinear interpolation (Sander 2003). There are different ways to handle the seam problem. Sander (2003) describes one solution in which the empty texels of the texture copies values from the closest sample point using a Dijkstra-like flood fill algorithm. Another approach to eliminate visible seams is to extrapolate the chart colors by mixing different mipmapping levels of the texture (Lefebvre et al 2005).

The current solution uses a fragment program to reduce the worst seam artifacts. Every texel that does not contain any data gathers data from the surrounding texels and calculates and stores the average of those values. This means that that texels that are surrounded by empty texels will remain empty. However, by executing this algorithm in multiple passes every texel will eventually be filled with data. Another approach to handle texels that only have empty neighbor texels is to extend the search area, that is gather values from non-neighboring texels. However, this approach requires many texture lookups which implies a performance penalty. A flood fill algorithm executed on the CPU is probably a better solution to the seam problem. More elaboration should be done in this area.

6 Results

This section presents what the prototype presented in this thesis is capable of. The prototype has been executed on a computer with 2 GB RAM, an Intel Core 2 Duo CPU T5250 at 1.5 GHz and a NVIDIA 8600M GT with 512 RAM.

6.1 Generating normal maps

Table 6.1 presents information concerning some normal map experiments. The HRM and LRM columns contain the number of polygons these versions of the mesh consist of. The voxel grid column presents the design of the voxel grid and the resolution of the voxel grid (see section 5.2 for more information). The texture column presents the size of the normal map that is to be generated and the next two columns show the total amount of voxels in the voxel grid and the amount of voxels containing normals respectively. In figure B1-B3 (Appendix B) the experiments are illustrated.

Mesh	HRM	LRM	Voxel grid	Map size	Amount of voxels	Voxels with normals
Hand	200000	1600	4·4·4·4·4 = 1024	1024^2	9416832	2207292
Cxandra	190000	5600	4·4·4·4·4 = 1024	1024^2	4749504	1122890
Neptune	225000	6000	6·4·4·4·4 = 1536	1024^2	10902912	2559205

Table 6.1: Some example meshes and normal map generation details

6.2 Generating color maps

Figure B4 (Appendix B) illustrates the generation of a color map. The HRM consist of around 100k polygons, the LRM consist of 5600 polygons, the voxel grid is 4·4·4·4·4 = 1024, the generated color map is of size 1024^2, the total amount of voxels is 3 852 736 and the amount of voxels that contain color information is 909 942. As you can see some parts of the color map for the HRM does not exist in the color map for the LRM. The reason for this is that those parts are not directly associated with the mesh representing the body of Cxandra, they are associated with accessory meshes to Cxandra.

6.3 Voxel count growth

Figure 6.1 and table 6.2 show the voxel count growth for an octree when a mesh representing a sphere is being voxelized. This result can be used as a reference when trying out different variants of hierarchical voxel grids. As the resolution of the octree doubles the total amount of voxels and the amount of voxels containing surface information quadruples.

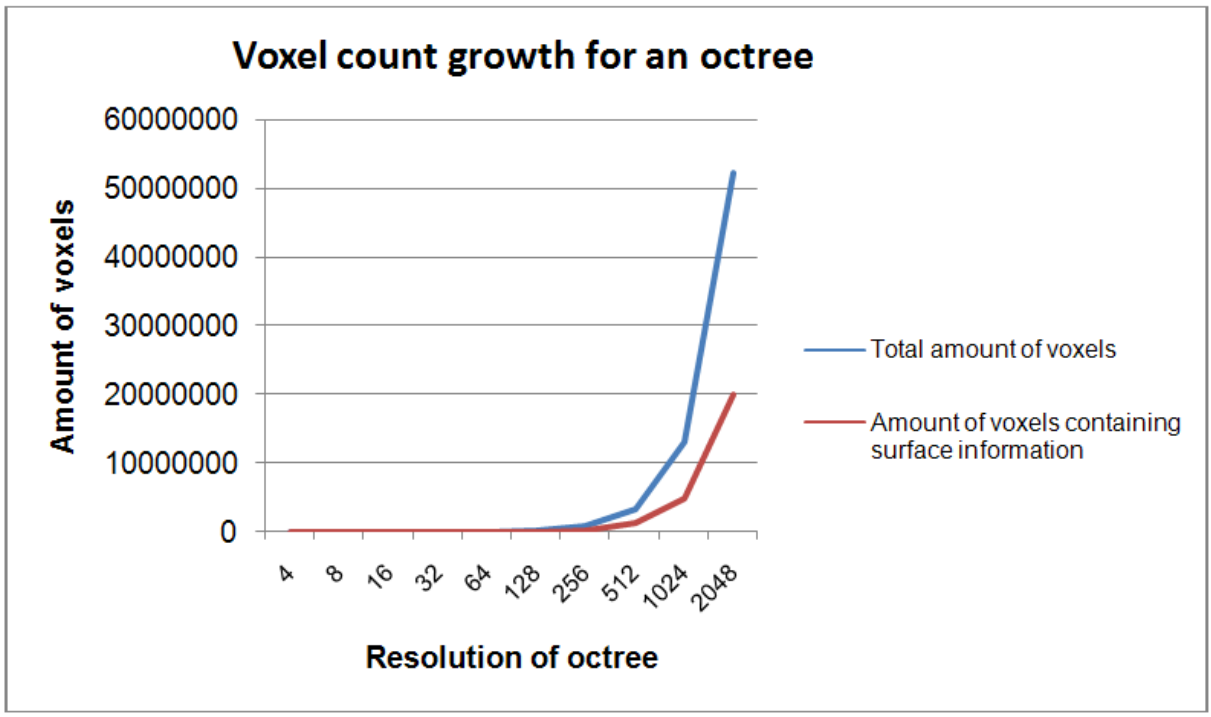


Figure 6.1: A diagram that illustrates how fast the number of voxels grows when an octree is being used to voxelize a HRM shaped as a sphere.

Resolution of octree	Total amount of voxels	Amount of voxels containing surface info
$2^2 = 4$	72	56
$2^3 = 8$	520	272
$2^4 = 16$	2696	1160
$2^5 = 32$	11976	4760
$2^6 = 64$	50056	19136
$2^7 = 128$	203144	76760
$2^8 = 256$	817224	306848
$2^9 = 512$	3272008	1227368
$2^{10} = 1024$	13090952	4909668
$2^{11} = 2048$	52367496	20051987

Table 6.2: Table associated with the diagram that illustrates the voxel count growth for an octree.

6.4 Processing time

The time it takes to generate a normal or color map depends on many factors. Some factors have a greater influence on the execution time than others. The size of the HRM, the size of the map to be generated and the resolution and number of depths in the voxel grid seem be the most important factors that affect the execution time. The size of the HRM affects the mesh loading time and the voxelization time. In fact the mesh loading time is the bottleneck of the solution at the moment. Donya is currently working on a fast mesh loading procedure that is able to load out-of-core meshes so no effort has been made to reduce the mesh loading time for the algorithm presented in this thesis. The voxelization time does not only depend on the number of polygons in the HRM it also depends on the topology of the HRM. Table 6.3

illustrates the voxelization time (in seconds) for three versions of Neptunus. The first version of Neptunus consists of 400k polygons, the second version consists of 225k polygons and the third version consists of 112k polygons.

Resolution of voxel grid	Voxel grid depth	Voxelization time (400k)	Voxelization time (225k)	Voxelization time (112k)
4·4·4·4·4·2 = 2048	6	14.062	11.844	10.844
4·4·4·4·4 = 1024	5	4.750	3.563	3.000
4·4·4·4·2 = 512	5	3.031	2.063	1.453
4·4·4·4 = 256	4	1.672	1.031	0.641
4·4·4·2 = 128	4	1.453	0.875	0.500
4·4·4 = 64	3	1.031	0.594	0.312

Table 6.3: Voxelization time for HRM version of Neptunus that consist of 200 000 polygons.

Table 6.4 presents the time (in seconds) it takes to cast rays and remove seams for Neptunus. The HRM and LRM version of Neptunus presented in table 1 have been used in this experiment.

Map size	Ray casting time	Remove seams time
1024^2	2.563	0.047
2048^2	2.469	0.141
4096^2	2.515	8.641

Table 6.4: Processing time for Neptunus using a voxel grid with resolution 6·4·4·4·4 = 1536.

Table 6.5 presents the compile and link time (in seconds) for the fragment program containing the traversal code for different voxel grid depths. For every new depth the compile and link time is increased by a factor of three.

Number of depths in voxel grid	Compile and link time
1	0.375
2	0.610
3	1.188
4	3.156
5	10.203
6	34.250
7	118.781

Table 6.5: Compile and link time for the fragment program containing the traversal code.

7 Discussion

This section presents some ideas concerning the results presented in section 6.

7.1 Map resolution

The number of voxels that contain normals affect how large normal and color maps that can be created without experiencing artifacts. There is no point in creating a map of n^2 texels if the number of values stored from the HRM is much less than n^2. However, it is important to note that even though the amount of texels in a map is n^2 it does not mean that the amount of texels that will contain data is n^2 – rather the opposite. As you can see in figure B1-B4 the texture atlases have a lot of areas that are black which mean that they are empty. Consequently a more accurate rule is that the number of texels that will contain data should not exceed the amount of values stored in the voxel grid.

7.2 Normal map generation

The normal map adds very much detail to the LRM. However, as we get closer to the mesh we can see that some seam artifacts still are present, at least when lower resolution normal maps are applied (see figure B5). If a high resolution normal map, 4096^2, is created from a voxel grid having far less normals compared to the amount of texels in the normal map, the mesh appear to be a bit blocky. The blocky appearance does most likely depend on the fact that there are several rays that hit the same voxel.

7.3 Color map generation

The generation of color maps has had a lower priority compared to generating normal maps. The primary reason for this is that it is not very likely that out-of-core meshes have a texture bound to them and if there is no texture bound to the HRM it is impossible to generate color maps. Figure B5 (Appendix B) illustrates the creation of a color map for the LRM version Cxandra. The resulting color map is not great but by increasing the resolution of the voxel grid and by having a HRM that consists of a larger amount of polygons the quality of the color map will increase.

7.4 Voxelization time

Table 6.3 presents the time it takes to voxelize two versions of a mesh representing Neptunus. As you can see the difference in processing time between the two meshes is rather small. This indicates that the most time consuming part of the voxelization algorithm is rather the process of creating new voxels than handling many polygons.

7.5 Ray casting time

There is hardly any difference between the ray casting time for the different map sizes but the seam removal time increases dramatically when a 4096^2 map is to be generated (see table 6.4). This performance penalty has most likely something to do with the fact that the seam removal program requires many reading operations which may stall the GPU. As explained earlier, handling the seam problem on the CPU is probably a better solution.

7.6 How well does the dynamical hierarchical voxel grid perform?

To create color and normal maps of size 4096^2 the voxel grid should have at least 16 million voxels that contain surface information. This requires a rather large grid and a large grid require a lot of memory. The least memory consuming hierarchical voxel grid that can be created with the algorithm presented in this thesis is an octree. Figure 6.1 illustrated the voxel count growth for an octree when a sphere is being subdivided. To create a normal or color map of size 4096^2 would require an octree resolution of 2048. An octree of this resolution consist of 52.4 million voxels of which 20.1 million voxels contain surface information. Consequently, to create a normal map of size 4096^2 implies that the voxelized HRM would require:

$$(52.4 \cdot 10^6 \cdot 4 + 20.1 \cdot 10^6 \cdot 3 \cdot 2)/1024^2 \approx 315\text{MB}$$

This kind of memory consumption is manageable for current GPUs. However, since the current algorithm requires one traversal function for each grid depth this result in a very large fragment program. I have experienced that compiling and linking large fragment programs can take very long time (see table 6.5) and in some cases even crash the system. The NVIDIA 8600M GT is able to handle five voxel grid depths without much delay but as new depth levels are added the compile time increases dramatically. At six depth levels the compile and link time is around 30 seconds, which is still rather manageable, but for every new depth level the compile and link time increases by a factor of three. To understand this dramatic increase in compile and link time one must understand how the GLSL compiler works. This kind of research is beyond the scope of this thesis.

The only way to reach a high resolution hierarchical voxel grid when using only six grid depths is to increase the size of the voxel grids at each depth level. Nevertheless, this implies more empty voxels and more empty voxels entail a need for more memory. One way to create a hierarchical voxel grid of resolution 2048 with only six depth levels is to let the voxel grids at the first five depth levels consist of 4^3 voxels and the voxel grids at the sixth level consist of 2^3 voxels. Voxelizing a sphere with this kind of hierarchical voxel grid does actually not result in a particularly large increase in memory consumption. The total amount empty voxels is 60.2 million and the number of voxels that contain surface information will remain the same as for the octree. Voxelizing a HRM with this kind of structure results in the following memory consumption:

$$(60.2 \cdot 10^6 \cdot 4 + 20.1 \cdot 10^6 \cdot 3 \cdot 2)/1024^2 \approx 345\text{MB}$$

The increase in memory consumption compared to the octree example is more or less insignificant. It is important to note once again that this example does only concern a sphere mesh. However, by doing the same experiments with the other meshes presented in this thesis (see appendix B) shows that the memory gap between an octree and a hierarchical grid is even less compared to the experiment concerning the sphere. The number of voxels that contain surface information is approximately half of the total amount of voxels for those meshes.

To summarize, even though an octree consumes less memory compared to any other hierarchical voxel grid our algorithm can create, the difference in memory consumption is still

rather low provided that the sizes of the voxel grids at the different levels are kept rather close to 2^3.

8 Conclusion

The main purpose of this project was to do research concerning the creation of a tool that can generate color and normal maps on the GPU and develop a prototype of such a tool. There were also some constraints concerning the solution. The algorithm should be able to handle out-of-core meshes and it should finish the map generation procedure within one minute.

8.1 Processing time

The processing time is affected by many factors so there is no guarantee that the algorithm will finish processing within one minute. By looking at the execution time for the experiments presented in appendix B the timing constraint is met with a large marginal. However, it is important to note that the meshes used in those experiments are not out-of-core meshes. Handling out-of-core meshes will affect the mesh loading time and the voxelization time. The mesh loading time will take more time for out-of-core meshes compared to small meshes but this is inevitable. Experiments presented in section 6.4 indicate that the size of the HRM does not inflict very much on the voxelization time. Consequently I believe the algorithm will be fast enough even for out-of-core meshes.

8.2 Quality

Speed is not everything. The algorithm must also generate high quality color and normal maps. As with the processing time there are many factors that affect the quality of the normal and color maps. One crucial factor that affects the quality is the resolution of the voxel grid. Higher resolution voxel grids imply that the algorithm can store more data from the HRM and more data means better preservation of details. More data also implies that more memory is needed. Thus, the quality of the maps that can be generated by our algorithm directly depends on the amount of RAM in the GPU running the algorithm. By judging from the images presented in appendix B the quality of the maps are pretty good. However, the result from generating a color map is not as satisfying as when generating a normal map. This is because a lot of color detail is lost during the voxelization process.

9 Further work

There is still some work left to be done before this prototype can be turned into a product ready for sale. This section presents some areas which would be interesting to examine further.

9.1 Code optimization

For instance, the fragment program executing the traversal algorithm should be optimized. In the current solution each depth requires a traversal function of its own. It should be possible to rewrite this code so that the traversal code is executed within a single function. The traversal algorithm must still be able to handle many variables but reducing the amount of function calls could still imply better compile and link time and maybe even a noticeable performance boost.

9.2 Multisampling

In order to enhance the visual quality of the normal and color maps the possibility to use multisampling should be further examined. In the current solution multisampling implies a large performance penalty. Rewriting the traversal code so that it may be executed within a single function could hopefully also mean that it is possible to use multisampling more efficiently. Another area that should be examined further is the process of removing visible seams. Some different approaches have already been proposed by Sander (2003) and Lefebvre (2005).

9.3 Non-cubic voxel grids

An extension to the algorithm presented in this thesis would be to support non-cubic voxel grids. For instance, assume that the algorithm should process a mesh representing a human. Such a mesh is not very cubic since the width of a human usually is less than her height. Instead of starting the voxelization process by placing the HRM human within a unit cube the algorithm could start the voxelization process immediately by subdividing the bounding box around the HRM. This would most likely imply less empty voxels but it is difficult to say how much gain there is to make. As long as the resolutions of the voxel grids at the first depth levels are kept low the amount of empty voxels will remain rather low with a unit cube as well. Nevertheless, it would be interesting to examine this modification of our algorithm.

9.4 Simplifying the HRM before normal map generation

Another approach to generate normal maps from out-of-core meshes is to start off by reducing the polygon count in the mesh to a reasonable amount that does fit into the RAM and then proceed to generate color and normal maps. Even though polygon reduction implies a loss of details so does the approximation method used in the algorithm presented in this thesis. Donya’s polygon reduction tool Simplygon has a function which allows the user to specify a maximum deviation between the original and simplified mesh. By using this function, important characteristics in the mesh may be preserved which is necessary to create high quality normal maps. Another nice thing about using this approach is that instead of storing a normal for each voxel it is possible to store every polygon that lie within a certain voxel.

When a voxel is hit by a ray the algorithm must perform intersection tests between the ray and the polygons within the voxel to see whether the ray actually hits the mesh. If a polygon is hit the vertex normals associated with the polygon are interpolated giving a very accurate normal sample. This would also mean that any two rays are not likely to retrieve the same normal value, which is the case when using the method presented in this thesis. In GPU Gems 3 Teixeira presents a normal map generation algorithm executed on the GPU that stores the polygons of the HRM in a uniform grid, transfers it to the GPU and performs a ray casting procedure. It would be interesting to see how well Simplygon and Teixeira's algorithm is able to handle out-of-core meshes and compare the result with the algorithm presented in this thesis.

Bibliography

Amanatides, J. & Woo, A., 1987. *A Fast Voxel Traversal Algorithm for Ray Tracing.* Eurographics '87. Amsterdam, The Netherlands, August 1987, p. 1-10.

Astle, D & Hawkins, K., 2004. *Beginning OpenGL Game Programming*. Boston: Premier Press.

Blythe, D., 2006. *The Direct3D 10 system*. ACM Transactions on Graphics (TOG), 25 (3), p. 724-734.

Cignoni, P., Montaniy, C., Rocchiniz, C. & Scopignox, R., 1998. *A general method for preserving attribute values on simplified meshes.* IEEE Visualization 1998, p. 59-66.

Cohen, J., Olano, M. & Manocha, D., 1998. *Appearance Preserving Simplification*. SIGGRAPH 1998, p. 115-122.

Ericsson, C., 2005. Real-Time Collision Detection. Boston: Morgan Kaufmann.

Haines, E, 2006. *An Introductory Tour of Interactive Rendering*. IEEE Computer Graphics and Applications, 26 (1), p. 76-87

Harris, M, 2005. *Mapping Computational Concepts to GPUs*. *GPU Gems 2*, p. 493-508.

Hearn, D. & Baker, M. P., 2004. *Computer Graphics with OpenGL*. 3rd ed. Upper Saddle River,

Kessenisch, J, 2006. *The Open GL Shading Language*. [Online].
Available at: http://www.opengl.org/registry/doc/GLSLangSpec.Full.1.20.8.pdf [accessed 3 mars]

Martz, P, 2006. *OpenGL Distilled*. New York: Addison-Wesley Professional.

Murray, J.D. & vanRyper, W., 1996. *Encyclopedia of Graphics File Formats*. Bonn: O'Reilly.

NVIDIA, 2006. *Technical Brief: NVIDIA Geforce 8800 GPU Architecture Overview.* [Online].
Available at: http://www.nvidia.com/page/8800_tech_briefs.html [accessed 11 February 2008].

NVIDIA, 2007. *NVIDIA CUDA Compute Unified Device Architecture Programming guide* (version 1.1). [Online].
Available at: http://www.nvidia.com/object/cuda_develop.html [accessed 11 February 2008].

Owens, J.D., 2002. *Computer graphics on a stream architecture*. Ph. D. Stanford University.

Owens, J.D., 2005. *Streaming architectures and technology trends*. *GPU Gems 2*, p. 457-470.

Owens, J.D. et al., 2007. *A Survey of General-Purpose Computation on Graphics Hardware*. Computer Graphics Forum, 26 (1), p. 80-113.

Purcell, T.J., Buck I., Mark W. R. & Hanrahan, P., 2002. *Ray Tracing on Programmable Graphics Hardware.* ACM Transactions on Graphics. *21 (3)*, p. 703-712.

Sander, P, 2003. *Sampling-Efficient Mesh Parametrization*. Ph. D. Harvard University.

Segal, M & Akeley, K., 2006. *The OpenGL Graphics System: A* specification (version 2.1). [Online].
Available at: http://www.opengl.org/documentation/specs/ [accessed 11 February 2008].

Tarini, M., Cignoni, P., Rocchini C. & Scopigno R., 2000. *Real Time, Accurate, Multi-Featured Rendering of Bump Mapped Surfaces*. Computer Graphics Forum, 19 (3), p. 119-130.

Teixeira, D., 2005. *Streaming architectures and technology trends. GPU Gems 3*, p. 491-512.

Appendix A: Traversal code

```
bool TraverseVoxelGrid(rayDirection, rayOrigin, activeVoxel)
x = floor(rayOrigin.x)
y = floor(rayOrigin.y)
z = floor(rayOrigin.z)

t1 = (x - rayOrigin.x) / rayDirection.x;
t2 = (x + 1.0 - rayOrigin.x) / rayDirection.x;
tMaxX = max(t1, t2);

t1 = (y - rayOrigin.y) / rayDirection.y;
t2 = (y + 1.0 - rayOrigin.y) / rayDirection.y;
tMaxY = max(t1, t2);

t1 = (z - rayOrigin.z) / rayDirection.z;
t2 = (z + 1.0 - rayOrigin.z) / rayDirection.z;
tMaxZ = max(t1, t2);

stepX = 1
stepY = 1
stepZ = 1

if (rayDirection.x < 0.0)
             stepX = -1
if (rayDirection.y < 0.0)
             stepY = -1
if (rayDirection.z < 0.0)
             stepZ = -1

tDeltaX = abs(1.0 / rayDirection.x)
tDeltaY = abs(1.0 / rayDirection.y)
tDeltaZ = abs(1.0 / rayDirection.z)

activeVoxel = getVoxel(x,y,z)

loop while activeVoxel is within the uniform grid
{
             if (activeVoxel is non-empty)
                          return true

             if(tMaxX < tMaxY)
             {
                          if (tMaxX < tMaxZ)
                          {
                                       tMaxX = tMaxX + tDeltaX
                                       x = x + stepX
                          }
                          else
                          {
                                       tMaxZ = tMaxZ + tDeltaZ
                                       z = z + stepZ
                          }
           }
           else
           {
                          if (tMaxY < tMaxZ)
                          {
                                       tMaxY = tMaxY + tDeltaY
                                       y = y + stepY
                          }
                          else
                          {
                                       tMaxZ = tMaxZ + tDeltaZ
                                       z = z + stepZ
                          }
           }
           activeVoxel = getVoxel(x,y,z)
}
return false
```

Appendix B: Illustrations of results

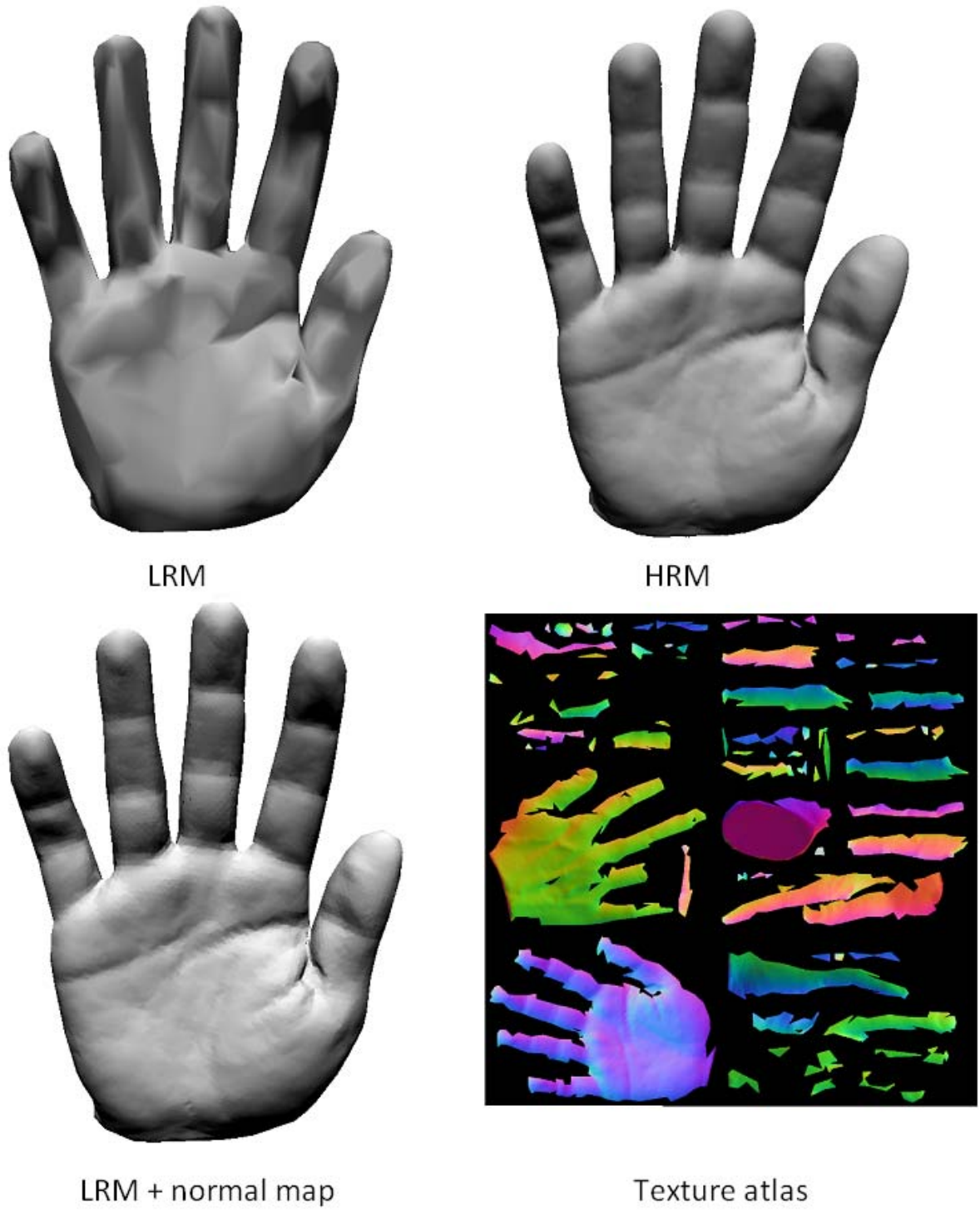


Figure B1: Three different versions of the hand and the normal map stored in a 2D-texture.

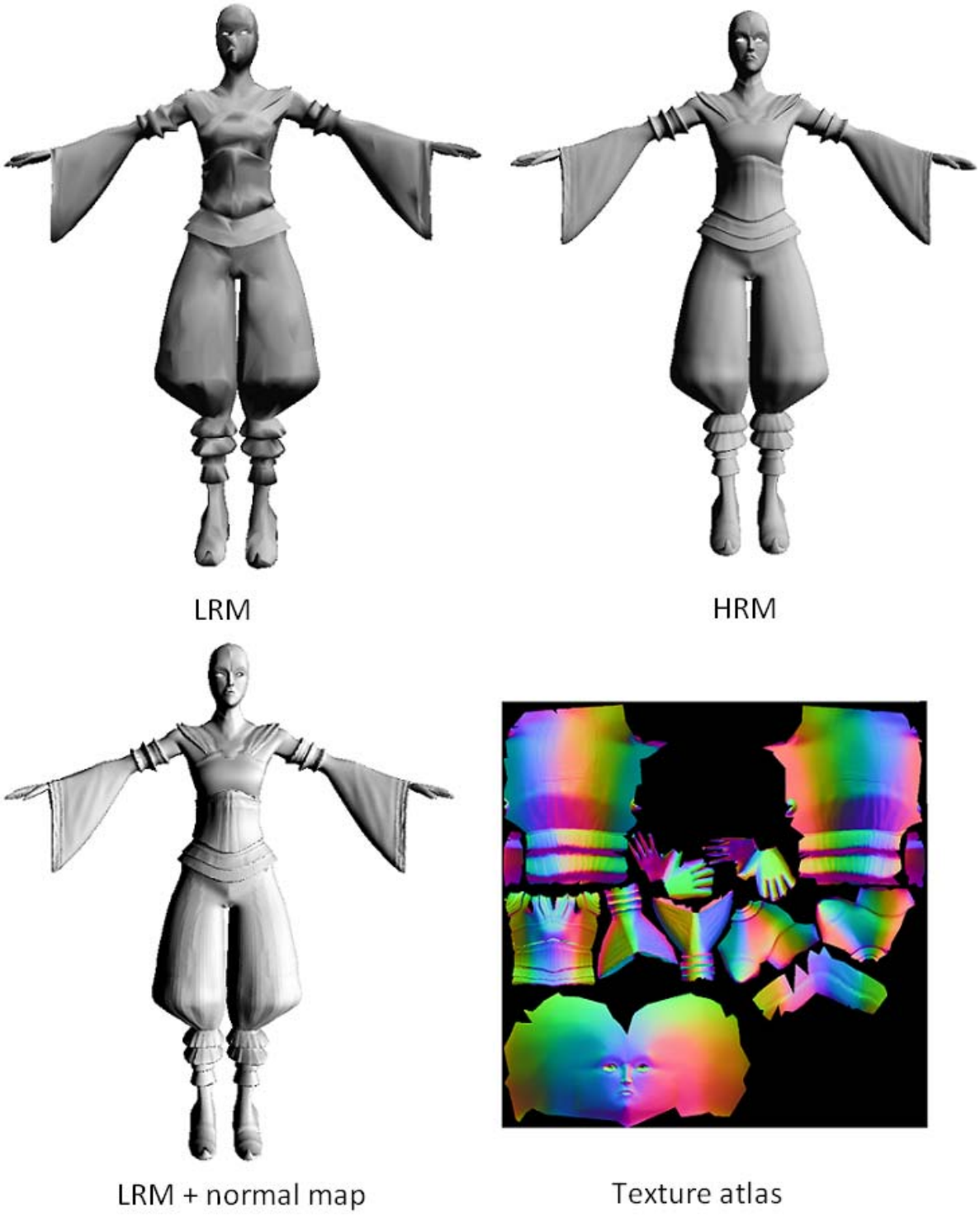


Figure B2: Three different versions of Cxandra and the normal map stored in a 2D-texture.

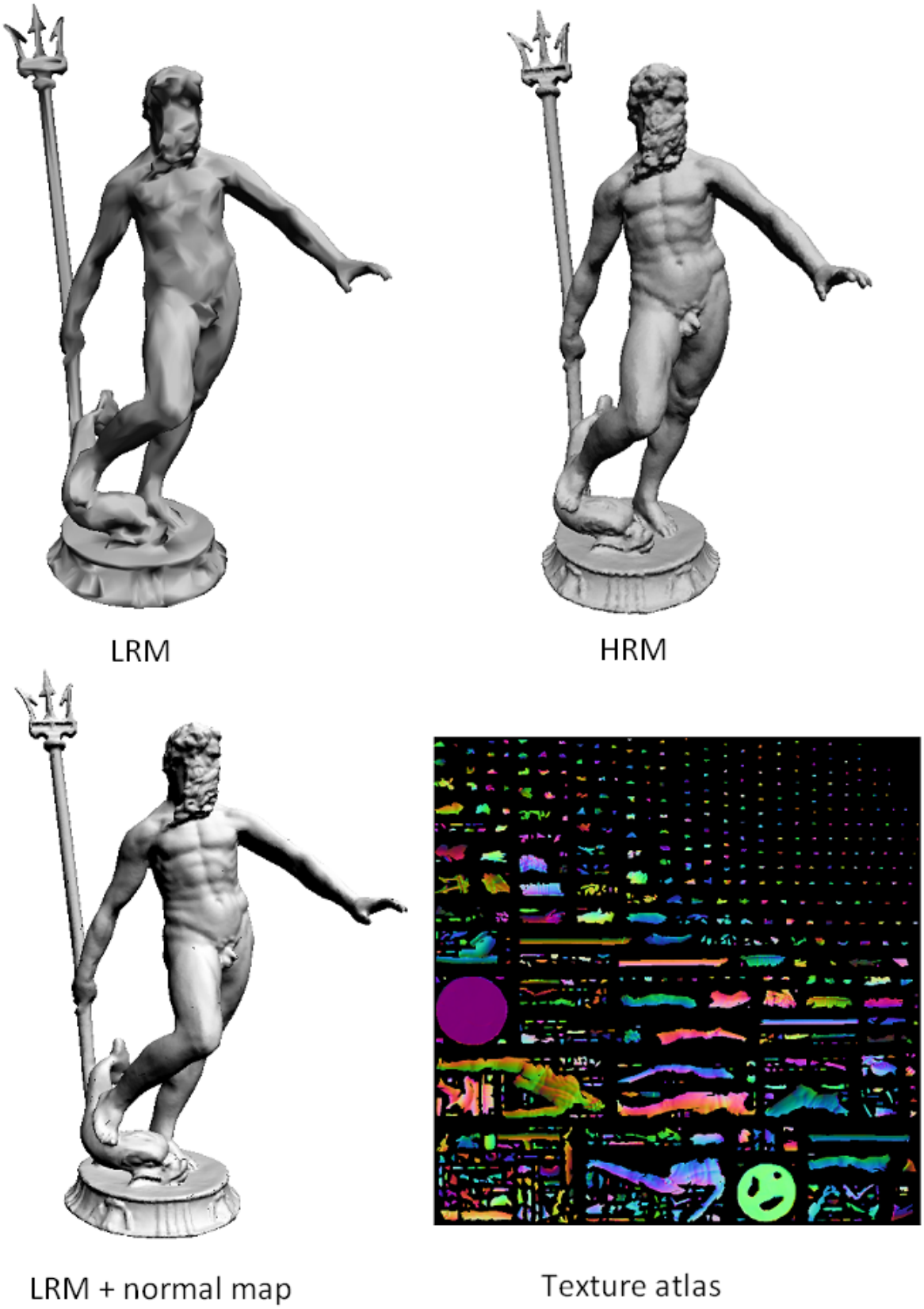


Figure B3: Three different versions of Neptunus and the normal map stored in a 2D-texture.

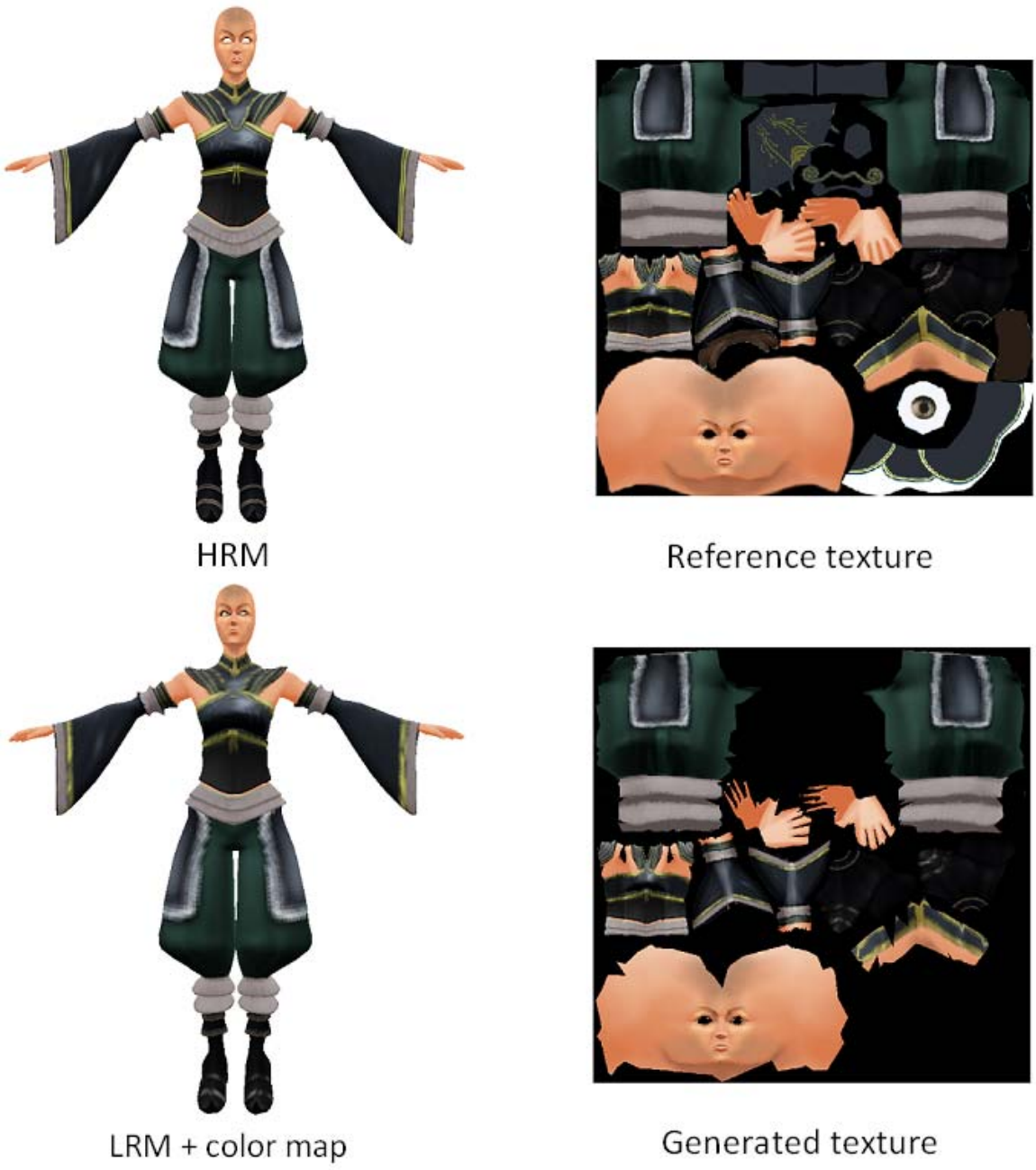


Figure B4: HRM version of Cxandra rendered with the reference texture and LRM version of Cxandra rendered with the generated color map.

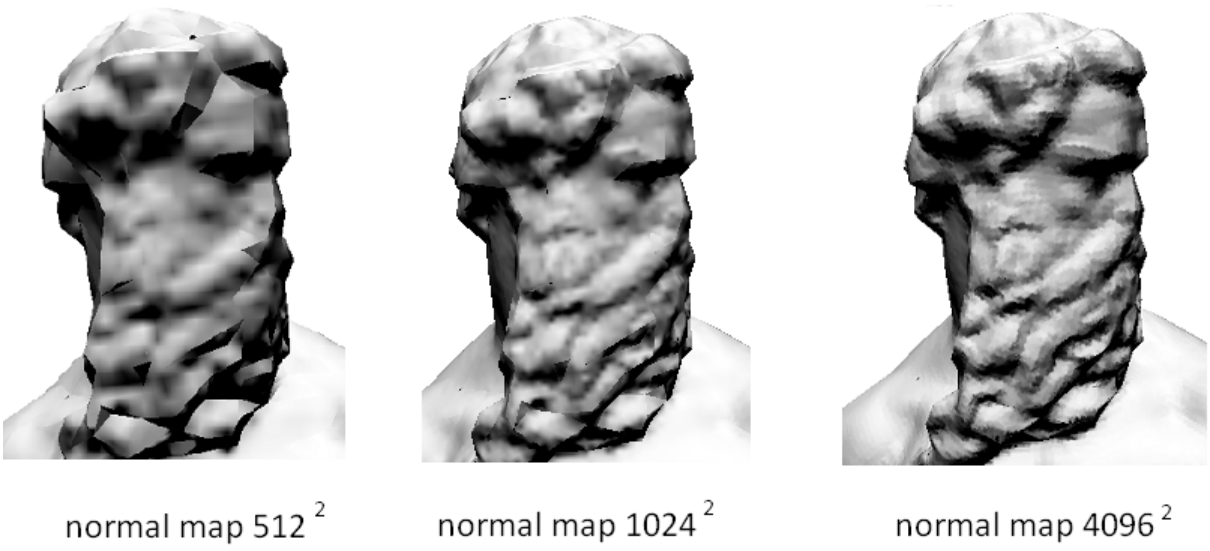


Figure B5: LRM version of Neptunus rendered with normal maps of different resolutions.