NTNU
Norwegian University of
Science and Technology

Parallel Methods for Projection on Strongly Curved Surfaces

Joel Eelaraj Chelliah

Master of Science in Computer Science
Submission date: June 2011
Supervisor: Torbjørn Hallgren, IDI

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

Problem Description

At the department of Computer and Information Science at NTNU, there is a virtual reality system known as conCave, consisting of a strongly curved projection surface. In order to view an image projected onto such a surface, it is necessary to first deform the image according to the geometry of the surface, so that it appears correctly when projected onto the curved surface. In this thesis we investigate two different mathematically correct methods for transforming the projection of a scene, such that it can be displayed on the strongly curved surface of the conCave system. We develop massively parallel solutions for both these methods on the GPU, and aim to achieve real-time stereoscopic projection of the transformed images.

Assignment given: January 17 2011
Supervisor: Torbjørn Hallgren

Abstract

Using the parallel architecture of the graphics processing unit for general purpose programming has become increasingly common in the recent years. The process of creating a mathematically correct transformation of a scene for curved stereoscopic projection is a very expensive task, which would greatly benefit from a massively parallel solution implemented on the GPU.

In this thesis, we first investigate two different methods for obtaining a mathematically correct transformation of images intended for stereoscopic projection on strongly curved surfaces. One method revolves around transforming a pre-rendered image, pixel by pixel, while the other method applies the transformation to the projection of the vertices in the scene before they are rendered as an image. We then develop massively parallel solutions for both these methods on the GPU, striving to a reach a real-time rate for the stereoscopic projection of the transformed images.

We test both methods for different problem areas, and compare the results to map their strengths and weaknesses. From the obtained results, we conclude that they are both useful in different areas. The vertex transformation performs poorly when the number of vertices in the scene is very high, but for a moderate number of vertices it achieves excellent results, even for exceptionally large image resolutions. The pixel transformation is far less affected by the number of vertices in the scene; however its performance declines rapidly as we increase the size of the image. Both methods were able to execute in real-time for relevant problem sizes.

Acknowledgements

This report is the result of the Master's thesis by Joel Chelliah as part of the course TDT4900. It was written at the Department of Computer and Information Science at the Norwegian University of Science and Technology.

I would like to extend my gratitude to several people for making this thesis possible. First of all, I would like to thank my supervisor Torbjørn Hallgren for providing the idea for this thesis, and for his invaluable assistance throughout the semester by providing interesting ideas, valuable feedback and helpful guidelines, on my thesis work, and the structuring this report. I would also like to thank my friends and fellow students at the HPC and Graphics lab for technical support, many stimulating discussions of relevant topics, and motivation throughout this semester.

Trondheim, Norway, June 1 2011

Joel Chelliah

Contents

List of Figures

List of Tables

Listings

CHAPTER 1

Introduction

In this chapter we describe the motivations behind this thesis, state the main goals that we want to achieve and how we wish to approach them, and provide a short outline of the structure of this report.

1.1 Motivation

The concept of viewer-dependent projection consists of projecting a scene, where the position and angle of the projected scene is dependent on the position of the viewer. When the viewer moves to the left or right, the projection of the scene is rotated and shifted based on the new position, such that the viewer gets the feeling of standing in front of the actual elements in the scene. When this is done using a strongly curved surface, the sensation of reality is enhanced even more. This is due to the curved surface partly surrounding the viewer, and providing the impression of standing inside the projected environment. The perception of depth is also increased due to the curvature of the surface. Stereoscopic projection on a curved surface enhances the feeling of depth even more by providing additional depth cues. However, performing a mathematically correct transformation of the scene for such a surface is a very expensive task.

In recent years it has become more and more common to utilize the massively parallel architecture of modern graphics processing units for general purpose programming. Using the parallel computational capabilities of the GPU, programs can be parallelized to run across hundreds of thousands of threads concurrently, thus greatly improving their performance. Transforming an image for curved stereoscopic projection is a task often consisting of performing many expensive operations across different portions of the image, which makes it a very good candidate for such massive parallelization.

1.2 Goals and Problem Definition

The main goal of this thesis is to investigate two different methods for transforming an image, such that it can be projected onto a curved surface, and to develop massively parallel solutions for both of these methods on the GPU. The purpose of this is to also compare the strengths and weaknesses of these methods for various problems, examining the levels of performance as well as the visual aspects. The desired outcome of this thesis is to find a well-balanced parallel solution for stereoscopic projection on curved surfaces that performs well, both visually and in regards to execution time. This is a theoretically based thesis, focusing primarily on the calculations of the transformations and the development of parallel solutions, thus the scope of this thesis does not cover any physical experimentation of the transformations on the conCave surfaces.

The first method will revolve around performing a mathematically correct pixel-by-pixel transformation of a pre-rendered image of the scene, where we reposition each pixel in the image according to the geometry of the surface. This will be implemented on the GPU using the CUDA framework. The second method will apply the transformation to the actual projection of the scene, by calculating new positions for the projection of each of the vertices in the scene. This will be implemented on the GPU using GLSL code for the vertex and fragment shaders in the graphics pipeline.

The following requirements are established for both solutions:

- Projection of the curved image should run in real-time, which we define as at least 24 frames per second as this is the de facto standard for animated motion pictures.
- The code should utilize the GPU for all calculations pertaining to the transformation of the scene, and maximize the number of operations that can be performed in parallel.
- There should be support for stereoscopic rendering of the transformed scene.
- The application should be scalable for relevant problem sizes.

1.3 Outline

The remainder of the report is structured as follows:

Chapter 2: Background provides background information on several subjects within virtual reality and stereoscopy that are relevant to this thesis. Topics concerning GPGPU programming and the NVIDIA CUDA framework are also discussed.

Chapter 3: Related Work gives an overview of earlier projects and theses that were done concerning stereoscopic projection on strongly curved surface, and also covers other popular approaches that are often considered in these cases.

Chapter 4: Method presents a detailed and thorough description of the two transformation methods that are the focus of this thesis.

Chapter 5: Implementation describes the parallel implementations of the two transformation methods investigated in this thesis, including the stereoscopic rendering of the transformed scene.

Chapter 6: Benchmarking and Results covers the benchmarking routines performed on both methods, presents the results that were obtained, and provides a discussion based on these results.

Chapter 7: Conclusions and Future work summarizes what was achieved during the course of this thesis, draws conclusions based on the results and discussions, and provides some ideas for possible future work.

Appendices: include an excerpt from our fall specialization project report [5], covering background information on the CUDA framework, and some example kernel and shader code of the transformation methods developed throughout this thesis.

CHAPTER 2

Background

This chapter provides background information on several subjects within virtual reality, stereoscopy, GPU programming and the CUDA framework that are relevant to this thesis.

Firstly, Section 2.1 gives an introduction to virtual reality, provides an overview of various VR systems, and gives a more detailed explanation of the conCave VR system. Secondly, Section 2.2 takes a brief look into stereoscopy, covering some basic knowledge of its functionality and how it is used in VR systems today. Thirdly, Section 2.3 describes the concept of parallel programming on the GPU and looks into GPGPU computing. Finally, Section 2.4 presents the CUDA framework and programming model from NVIDIA.

2.1 Virtual Reality

Virtual Reality (VR) is a term used to describe a graphical computer-generated environment that can simulate the presence realistic elements, and provide the experience of places in the real world. VR systems are used in a range of applications such as flight simulation, games and for therapeutic uses. A VR system can consist of anything from a simple computer screen to more complex constructions including special stereoscopic displays, position and orientation sensors, and surround-sound speakers. In a VR system, the possibilities of interaction available to the user depends on the complexity of the system and the different components it consists of. The user can interact with the virtual environment or a virtual object using a simple keyboard or mouse, or through special multimodal devices such as wired gloves, touch screens, or devices that register head and body movements. The simulated environment can be a copy of real world locations, which is the case for various vehicle simulation programs, or the simulated environment can be completely fictional, such as in virtual reality games.

2.1.1 Types of VR Systems

VR systems can be divided into several different types depending on their purpose, what kinds of equipments are used for visualization, and how the user is able to interact with the virtual environment. The most common types of VR systems are listed below [6].

Window on World System

This method has its roots all the way back to the 1960s, and is one of the first ways of viewing computer-generated virtual reality environments; it consists of using a conventional computer monitor to display the visual world. Today, this type of display is often not considered to be VR since they hardly block out the real world, do not present virtual objects in life size, and do not create the illusion of immersion [7].

Video Mapping

This is a variation of the Window on World VR system, but also includes merging a video input of the user's silhouette into the scene, thus allowing the user to see himself moving around and interacting with the surroundings in the virtual world.

Immersive Systems

In these systems the user's view is completely immersed in the virtual world. It is common to use special helmets or masks that block out the view of the real world, enabling the user to fully experience the virtual environment through small displays and speakers placed inside the headgear.

A variation of immersive systems uses several projection surfaces or a curved projection surface that surround the user, providing a cave-like room. Images are projected onto all the surfaces giving the experience of standing in the middle of a virtual environment. The principle advantages of surround projections in CAVE systems are a wide, surrounding field of view and the ability to give a shared experience to a group of users, where one or several can be tracked [7]. The principle disadvantages are that it may be very costly if multiple projectors are needed, requires a lot of space especially for rear projection, brightness limitations, and reduced contrast and color saturation due to light scattering. Of course, all these issues depend on the size and type of the CAVE system. For small screen sizes the brightness limitations will not be as hindering as for larger ones.

Telepresence

Telepresence is the concept of linking remote sensors to a device in the real world, so that the device is linked with the senses of a human operator. The perceived view of the device is displayed to the user through various displays and speakers, and the user controls the actions of the device by sending different commands. Tracking and pressure sensors are used to give the user as much control over the device as

possible. This comes close to the definition of augmented reality, which we will cover later in this chapter.

Mixed Reality

A mixture of Telepresence and VR where the computer-generated input is merged with Telepresence input. The virtual environment either partially consists of real-time data taken from Telepresence input, or is constructed from earlier scans, images, etc.

2.1.2 Augmented Reality

In contrast to traditional virtual reality, augmented reality (AR) is a term describing live direct or indirect view of a real-world environment. While virtual reality deals with a simulated environment, AR lets the user observe and interact with the real world through various sensory inputs. Real world elements are often combined with virtual elements through different technologies, such as object recognition, in real time to help enhance the user's perception. For example, objects can appear as if they have been inserted into the real environment, such as text and markings appearing over areas of interest. Most of the components that are used in traditional VR are also present in AR systems, such as head-mounted displays and tracking devices.

2.1.3 ConCave

ConCave is a cave-like projection surface that is designed for passive, depth-enhanced projection, and was created by FakeSpace Systems in 1998 [8]. It was originally de-

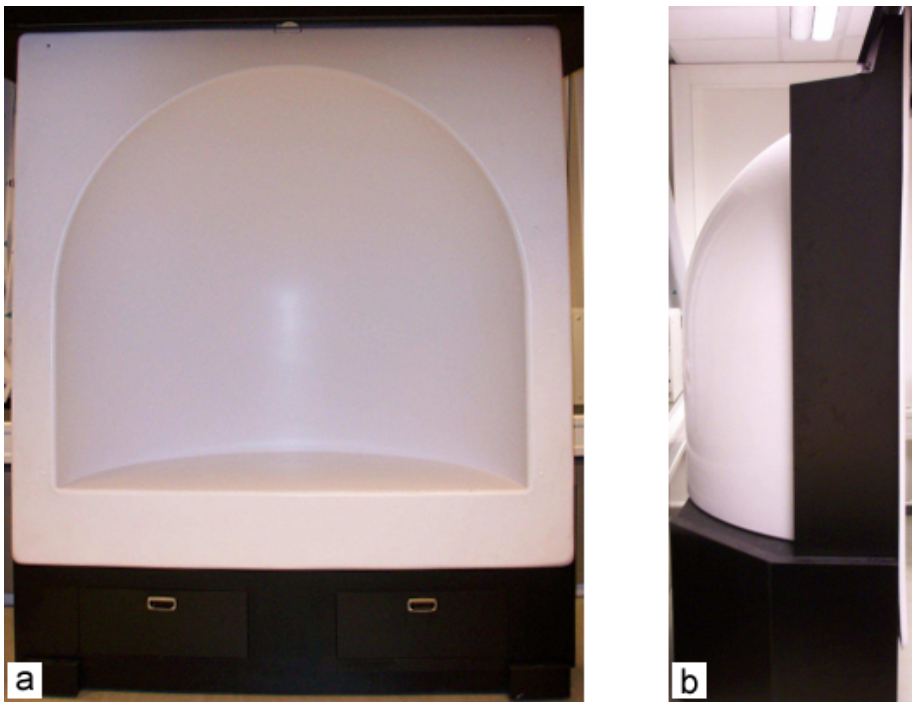

Figure 2.1: (a) ConCave front view. (b) ConCave side view.

veloped for Phillips Petroleum for the purpose of studying volumetric data under correct spatial visualization. The shape of the surface gives a certain illusion of depth even without the use of any special stereoscopic glasses, which is one of the reasons that cave-like projection surfaces are a popular choice for oil and gas industries for studying geologic and seismic data. The small size of conCave makes it very useful especially for small groups of people, and for rooms that are too small to fit large projection equipment. FakeSpace later merged with another immersive display development company known as Mechdyne. We refer to their website [8] for more detailed information on the history of the conCave system.

The conCave unit at NTNU was also delivered by FakeSpace. This system is a variation of an immersive VR system, as discussed in Section 2.1.1. It does not completely immerse the user's view into the virtual world, but it comes pretty close. When standing close enough, the strongly curved surface of the cave will still provide the user with the illusion of standing somewhat inside a virtual environment. A front and side view of the cave can be seen in Figure 2.1.

This system can also be compared to a dome. Although not completely similar, they work in the same manner, and in most cases can be used for the same purposes. Dome environments are spherical rooms where a single user or multiple users stand in the middle and witness the virtual environment being projected around them. However, dome systems can also be configured to only show a hemispheric view, which is quite similar to what the conCave system does. The hemispheric view is enough to wrap around the viewer's peripheral vision, giving a very wide field of view and realistic perception of distance to objects.

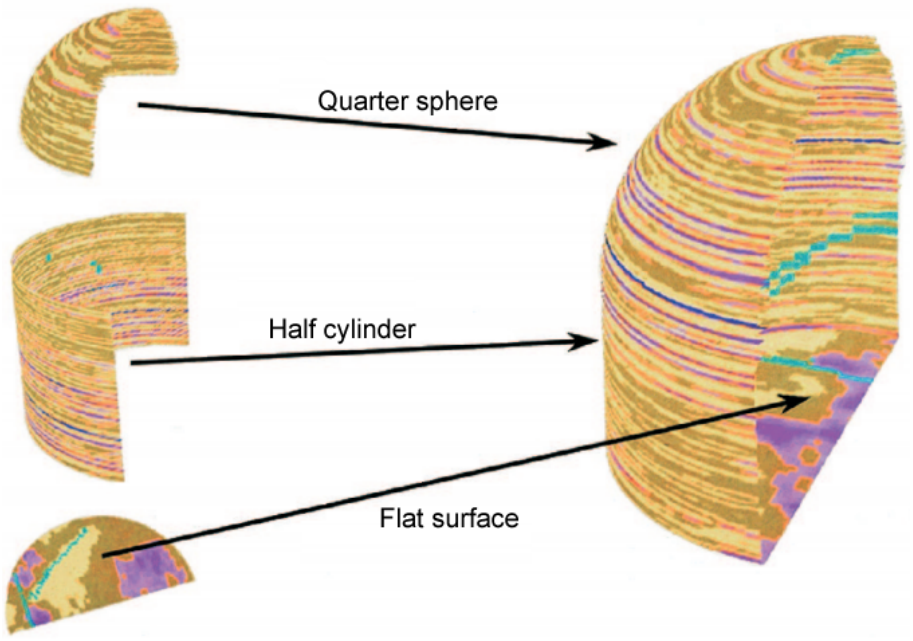

Figure 2.2: ConCave geometry, [8].

As we can see in Figure 2.2, the conCave surface consists of three basic shapes. The top half has the shape of a quarter-sphere, whereas the bottom half has a semi-cylindrical shape. In addition, there is a half-circular flat surface that covers the bottom. Measurements of the surface can be found in Table 2.1.

Width	154cm
Height	154cm
Depth / radius	72cm
Height of cylindrical part	72cm
Height of spherical part	72cm

Table 2.1: ConCave measurements

2.2 Stereoscopy

Stereoscopy, also known as stereoscopic imaging, is a technique that is used to enhance the illusion of depth in images and animations, and is a very useful technique when trying to add realism to a scene. We are used to always perceiving the world around us in three dimensions, so a scene that has no or very little sign of depth will immediately seem artificial.

2.2.1 Cues for Depth Perception

The human visual system uses several cues to determine the depth in a scene. These can be divided into monocular cues and stereo cues [9]. Monocular cues are present in most two dimensional images, and lets us judge depth from a single image. Some examples of these are:

- Texture variations and gradients.
- Objects being occluded by other objects.
- The size of an object changing as it moves towards or away from the viewer.
- Haze and desaturation

Stereo cues are based on two different viewpoints or images. One such cue is called stereopsis. As our eyes are a small distance apart from each other, each eye will give a slightly different view of the scene we are looking at. As we can see in Figure 2.3, the same object appears slightly different to each eye. The brain then combines the images projected onto the retinas of the two eyes to give us a perception of depth. The retinal disparity, which is the distance between the two perceived images, varies with the distance to the object we are looking at.

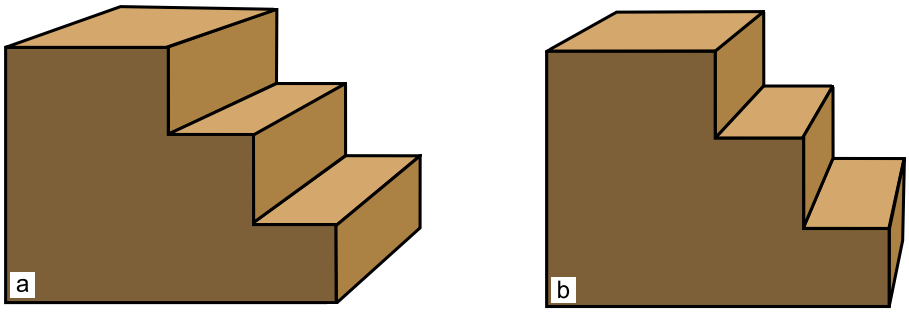

Figure 2.3: Same object seen from two different perspectives: (a) seen from the right eye, (b) seen from the left eye.

Another cue that can also be classified as a stereo cue is eye accommodation. This term refers to when objects that are not in focus will appear blurred, and we are required to readjust the focal length of the eye lens to bring the new object into focus [10]. This helps provide even more depth information, especially when looking at objects in close range.

A third stereo cue is convergence, which is what we do when we rotate our eyes so that they are both looking at the same point. A result of this is that everything that is not on the point of convergence appears doubled as each eye is providing a different view of them, being in slightly different locations.

2.2.2 Using Stereoscopy

Stereoscopic imaging is done by creating two offset images of the same scene (also known as a stereo pair), then displaying one image to the left eye and the other image to the right eye. Doing so, it is possible to artificially provide stereopsis to some extent by simulating the way we view the world with our own eyes. As we can see in Figure 2.4, we create the two images of the stereo pair by placing the viewpoints a distance d from each other. A typical average value for this distance d is 1/30 of the distance from the observer to the nearest object of the scene [11].

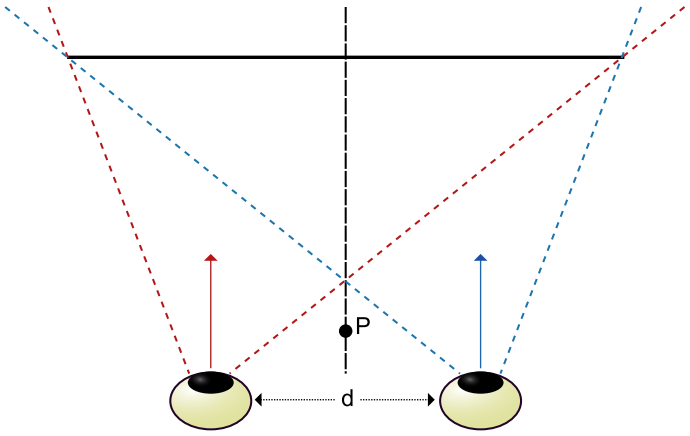

Figure 2.4: Creating a stereo pair from a reference point P and distance d between two eye/camera positions.

The viewpoints should also be equal distances to the reference point P, which is the point right between the eyes. When d is zero, both viewpoints are positioned at P and both images of the stereo pair will be identical. It should also be noted that to get best possible visual results, it is important that the viewing directions of both viewpoints are parallel to each other. The thick horizontal line represents the scene, and the dotted lines visualize how the scene is displayed to each eye. The two images we get from the offset viewpoints are combined by our brain into one image and we will get an illusion of depth.

Although stereoscopy enhances the illusion of depth by adding stereopsis, it does not provide eye accommodation or convergence, which are the second and third stereo cues we discussed earlier. Hence the illusion of depth is still not as good as reality. This is because when using stereoscopy it is necessary to keep the viewing axes parallel, and not let them converge towards a specific point on the scene as we do in real life. When looking at a specific point to which our eyes converge to, this point appears perfectly clear but everything else appears blurry. When we then change our view, our eyes will both focus and converge on the new location. In stereoscopy, the entire image needs to be in focus so that whatever part you are looking at can be viewed clearly [11]. Focusing on a specific point would only make the image comfortable to look at for all the points in front of the converging point, while making the rest of the image in the stereo pair difficult to fuse. By keeping the viewing axes parallel (converging at infinity), the whole image can be easily fused.

2.2.3 Stereo in VR

Stereoscopy is often used in VR systems to enhance the feeling of reality in the virtual environment. This is done by rendering the view twice, once for each eye, with a tiny offset. There are different methods to make sure that each eye only sees its corresponding image, which we can categorize as passive or active stereo.

Passive stereo refers to using two projectors, one for each eye, using different polarized filters. Smaller filters matching the projectors' filters also need to be placed in front of the eyes, for each eye to pick up the projected image from its corresponding projector.

Active stereo is the term used for alternatively projecting different perspectives for each eye, and using special shutter glasses that alternately darkens over one eye, and then the other, in synchronization with the frequency of the projection. When the brain receives the images in such rapid succession it fuses them into one single image and perceives depth.

2.3 Parallel Computing on the GPU

Parallel computing is the concept of many instructions being carried out simultaneously, and it involves dividing larger problems into smaller ones which are then solved in parallel across several computing units. The most preferable scenario would be if it were possible to divide these problems into completely independent parts that

require no interactions with each other. This way, each task can be completed at its own pace without having to wait for any of the other tasks. However, this is not always the case, as sometimes there is need for communication between the different processes, whether it is the need for synchronization or data transfer. The way parallel programs are written to account for these communication needs, and the underlying memory architecture can greatly affect the performance of the program.

2.3.1 Evolution of Parallel Computing

For a long time, gain in performance has come from improving the single-processor design, but today parallelism has become the standard way to increase overall performance. The most common way of increasing single processor performance was by increasing the clock speed in processors, which lead to faster execution time of single processes. This has, however, reached its limit due to reasons such as increased heat generation and the power needed for further improvement. As a result, we see that parallel systems are quickly becoming more and more common. Some examples of these are:

- **Multi-core CPUs:** Putting several low power processor cores on the same chip.
- **Clusters:** Connecting several commodity PCs through a network.
- **GPUs:** Using graphical processing units as accelerators to perform intense computations.

In the early stages of parallel computing, these systems were mostly seen in supercomputers and heavy workstations; however, they soon spread out to consumer PCs, and multi-core CPUs are now common in most new desktop and laptop PCs. Additionally, many large applications such as modeling and image editing software are taking advantage of the GPU for accelerating computationally intense operations.

2.3.2 Graphics Processing Unit

The Graphics Processing Unit (GPU) is a specialized component that helps accelerate the rendering of 3D graphics. GPUs are able to perform highly intense computations in parallel, and they are designed such that more transistors are devoted to processing data. They are different from the CPU in the sense that only a very small part of the GPU is devoted to things like caching and flow control, as depicted in Figure 2.5. With a very coherent memory access pattern and simple flow control, the GPU is not designed to take into account things such as branching, memory access, and extraction of instruction level parallelism at the same level of the CPU. However, today the raw computational power of the GPU is enormous compared to some of most powerful CPUs, and this gap is steadily growing. The highly parallel structure of the GPU enables it to perform several instructions simultaneously, which makes the GPU very useful in the fields of computer graphics rendering as well as high performance computing.

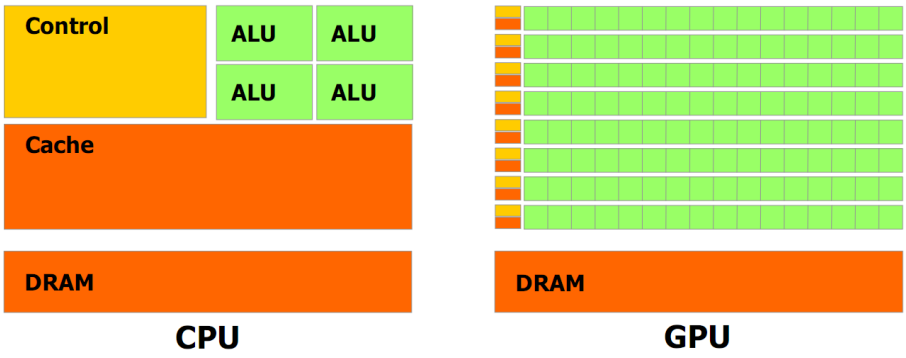

Figure 2.5: The GPU devotes more transistors to processing data, [14].

By the end of the 1990s, nearly every new computer contained a GPU that was dedicated to providing high performance, interactive 3D graphics. This was the consequence of increasing consumer demand for video games and various advances in manufacturing technology [12]. Today, modern GPUs can be seen as commodity data-parallel processors, and their computational capacities are still growing.

2.3.3 General Purpose GPU (GPGPU) Programming

In the beginning, the primary focus behind the development of GPUs was to achieve more advanced and realistic graphics for games. However, in recent years this development has also been influenced by the idea of using GPUs for general purpose programming. Consequently, GPUs have slowly moved away from the traditional fixed-function 3D graphics pipeline, and towards the interest of general purpose computation.

For a long time, GPGPU programming was seen as a difficult task with a very steep learning curve, and was not regarded as a relevant programming method for some time. This was because general purpose programs needed to be written using graphics APIs, which meant that programmers needed to first learn these APIs really well, and then try to figure out ways to write their programs under their given limitations. Writing programs using the graphics API included using shaders, which are the programmable parts of the graphics pipeline. They are responsible for tasks such as generating vertices, drawing lines, and creating polygons. The first languages that were used to program these shaders were High Level Shader Language (HLSL) and the OpenGL Shading Language (GLSL). As the idea of GPGPU became more popular, improvements were made to the GPUs to expand the programmability beyond shaders. This resulted in new programming models and GPGPU-friendly frameworks being created to hide the overheads from graphical APIs and simplify the task of general purpose GPU programming. Some of the most recent frameworks are OpenCL and CUDA.

2.4 CUDA

Compute Unified Device Architecture (CUDA) is a general purpose parallel computing architecture for GPGPU programming. Programmers can write their code in languages such as C and Fortran using CUDA, and then have the program translated into bytecode that can be run on NVIDIA Graphics processors. Together with OpenCL, CUDA is one of the most recent GPGPU-friendly frameworks designed to relieve the burden of having to write general programs using graphics APIs.

As extensive research was done on the CUDA framework and programming model in an earlier project [5], the same information will not be covered in this report. Instead we refer to an excerpt from that report which is included in Appendix A. This should cover the details of the CUDA framework that is relevant to this thesis.

CHAPTER 3

Related Work

This chapter provides an overview of popular ideas, and earlier projects and theses concerning stereoscopic projection on curved surfaces. The strengths and weaknesses of these approaches will also be explained briefly.

There are several methods for projection of data on curved surfaces. Some approaches focus mainly on the accuracy and correctness of the visualization, while others are more concerned with getting a good approximation at high frame rates. Sections 3.1 and 3.2 cover some popular approaches that are often considered in these cases. Sections 3.3 - 3.5 present various methods that have been investigated as part of project and thesis work at NTNU during recent years.

3.1 Ray Tracing

Ray tracing is done by projecting a ray from the viewer's position through every pixel on the screen and into the scene. If the ray hits an object in the scene, then the color of that object is added to the pixel. Once each pixel has been colored this way, the final image is rendered. Ray tracing gives very good visual results, but the performance may be too slow for real time visualization due to the vast number of calculations that must be performed. However, the speed and power of today's consumer PCs are growing at a rapid pace, meaning that this method may become more relevant in the future, especially when one can harness the power of the GPU for parallelizing all the heavy computations involved.

3.2 Use of Voxel Data

Voxels are cubic subdivisions of objects that store the information of different parts of that object. They can be considered as three dimensional pixels. For projecting scenes onto a curved surface, the objects in the scene are divided into smaller regions consisting of voxels, which are shaped in accordance with the shape of the surface.

This gives the impression of the objects surrounding the viewer. Voxel data are often used for such visualizations in the oil industry and within medical studies, but are generally a very poor choice for polygon based models.

3.3 Approximation to Several Planes

This is one of the two approaches investigated by Akeren during his Master's thesis in 2003 [1]. The idea was to divide the strongly curved surface into many tiny flat segments that approximate the curved shape, and then use a regular method for simple plane projection to project parts the image on each of these segments. The subdivision of the surface depends on the overall geometry. As this was done on the concave system described in Section 2.1.3, different patterns were needed for dividing the cylindrical part and the spherical part of the surface. The cylindrical surface was divided into uniform rectangular strips, while the spherical surface was divided into uniform triangular planes. The corners of the planes were positioned so they intersect the curved surface, as depicted in Figure 3.1. The number of segments is the deciding factor in the accuracy and smoothness of the image.

The advantages of this method is that it is very fast for a small number of segments, as plane projections are pretty simple to perform and require very little computational power. But there are several disadvantages. The visual quality is

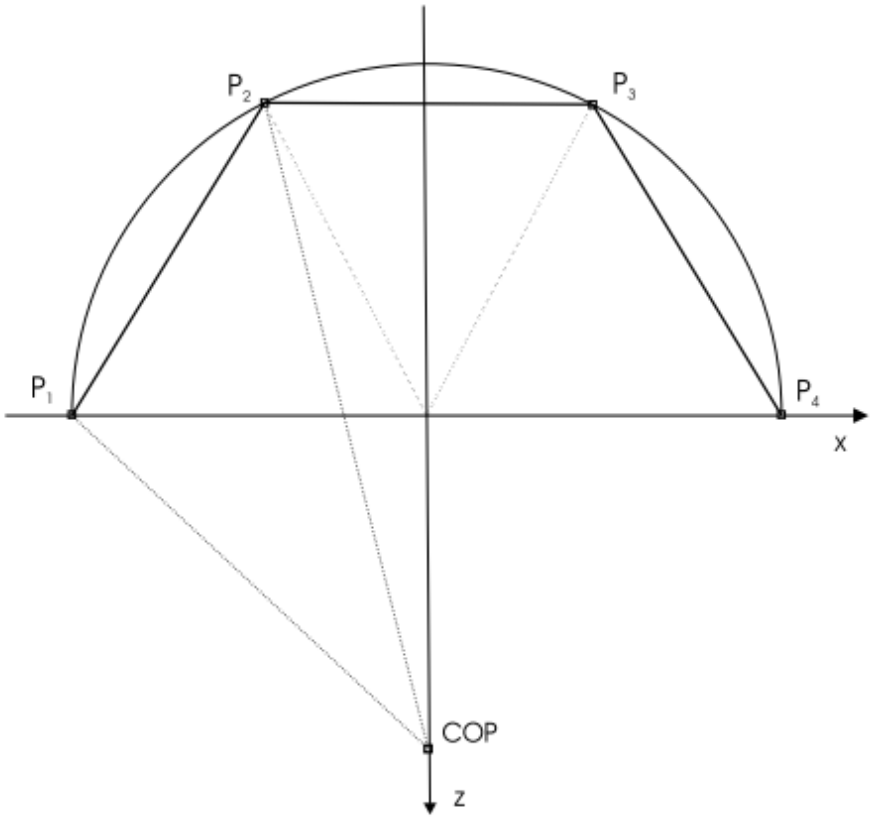

Figure 3.1: Plane approximation of the cylindrical part of the conCave surface, using three segments (seen from above). Image taken from [1].

highly dependent on the number of segments that are used, so a high number of segments is necessary for good quality. However, dividing the surface into too many segments will affect the performance. Too few segments also lead to the displayed image being very jagged in the areas where the segments connect, due to the segments not overlapping enough.

3.4 The Grid Method

This approach was investigated by Djønne and Solheim during their combined specialization project in 2003 [2]. The idea here is to render the image to a texture, which is then laid out on a grid. This grid is then shaped to fit the interior shape of the curved surface, as depicted in Figure 3.2. The texture that is attached to the grid will follow the same transformation, and will be displayed correctly on the curved surface.

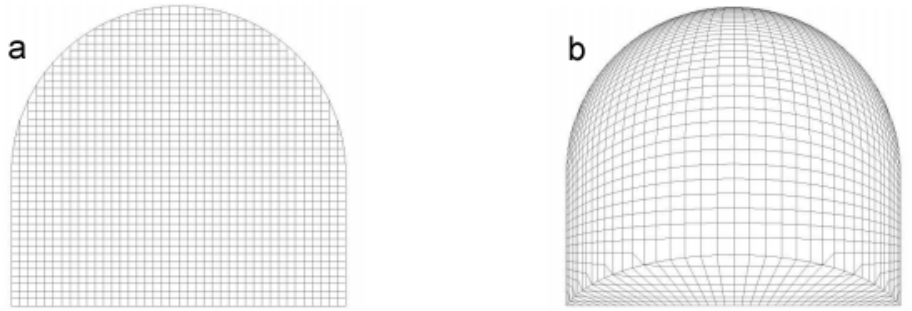

Figure 3.2: (a) Grid placed in front of the conCave surface before transformation, (b) grid after transformation (seen from the front). Images taken from [2].

Although this method proved to work well in real-time and gave good visual results, it had problems showing bigger models correctly. The visual quality suffered especially at larger perspectives or when viewing from very slant angles.

3.5 Using Polygon Triangulation

This approach was investigated by Djønne during his Master's thesis in 2004 [3]. The method consists of dividing larger polygons into simple triangles, and then moving all the corners according to a transformation table. The transformation table is created based on the position of the viewer and the projector. It is used to curve the corners of the polygons, such that the image is displayed correctly on the curved surface for the given viewer and projector position.

This method provided a lot clearer and better quality projections than the grid method, especially for larger models. However, the performance of this approach was somewhat slower than the grid method.

CHAPTER 4

Methods

This chapter provides a detailed and thorough description of the two transformation method that are the focus of this thesis.

Section 4.1 describes the pixel transformation method, where the idea is to pre-render an image through regular planar projection, and then transform this image into one that can be projected onto a curved surface. Then Section 4.2 describes the vertex transformation method, which deals with transforming the projection based on the vertex data of the scene.

4.1 The Pixel Transformation Method

A simplified version of this approach was investigated by Akeren during his Master's thesis in 2003 [1]. Here we look into a complete mathematically correct transformation. The idea consists of first rendering an image meant for regular planar projection, in this case a perspective projection, and then transforming this image pixel by pixel before projecting it, so that it displays correctly on the curved surface. Two different transformations are needed. One for the part of the image projected onto the spherical surface, and one for the part projected onto the cylindrical surface.

4.1.1 Overview of the Transformation

The following list provides an explanation of the main steps in the transformation procedure.

1. A flat image is first rendered through regular perspective projection. The pixels of the image are then read and stored in an array.

2. A Cylinder transformation procedure is performed on the bottom half of the image. The new values of these pixels are calculated based on the position of the viewer, the projector, and the projection surface.

3. A Sphere transformation procedure is performed, doing the same thing for the top half of the image. The two transformation procedures are independent of each other; hence they do not need to be called in any specific order.

4. After both transformations have taken place, the new image is rendered using the transformed array of pixels, and is ready to be projected onto the curved surface.

4.1.2 Cylinder Transformation

Here we describe the mathematical process of transforming the part of the image that is meant for the cylindrical surface. The basic process of the transformation is to map every pixel (i, j) of the output image to a pixel (u, v) from the input image, which is rendered using regular perspective projection. The corresponding pixel coordinate (u, v) for each (i, j) is found using the geometry between the position of the projector, the viewer's position, and the geometry of the curved surface. This results in a pixel perfect, mathematically correct transformation of the original image, so that it appears correctly on the curved surface.

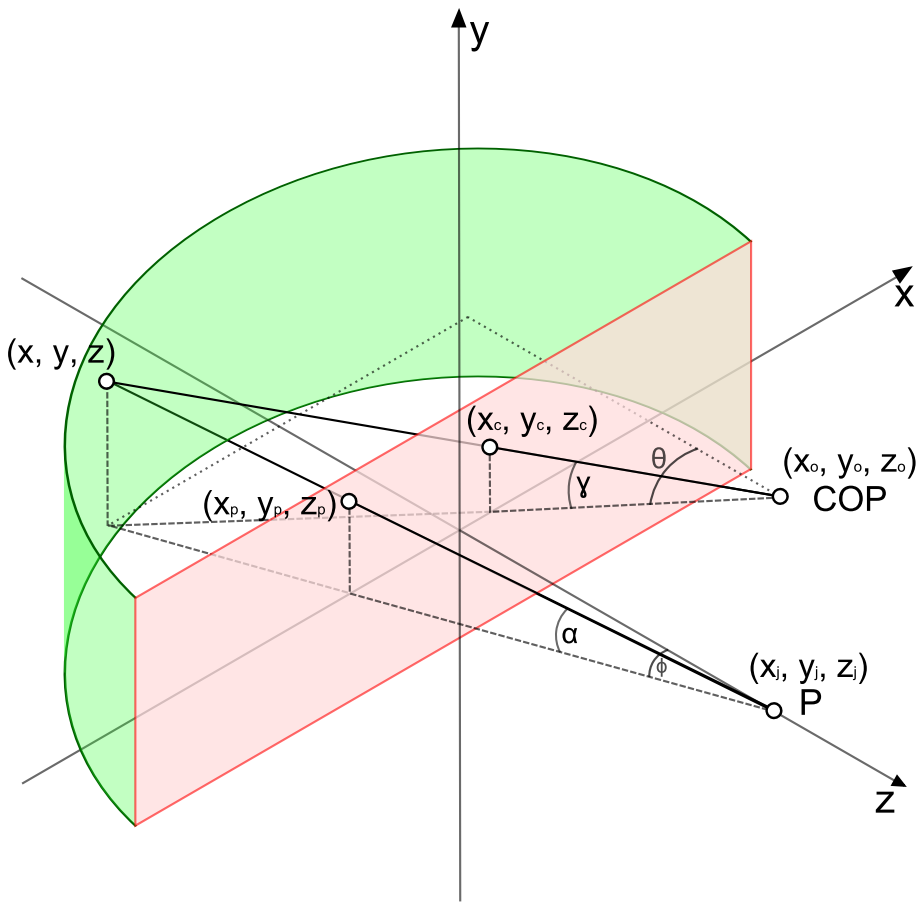

Figure 4.1: Overall view of the projection on the cylindrical part of conCave, for the pixel transformation.

Projection on the Cylindrical Surface

In Figure 4.1 we look at the setup from an angular bird's-eye view. The cylindrical surface is displayed in green, while the red plane represents an imaginary flat surface on which the original planar projection would have been displayed. As this method is about transforming an image rendered for a flat surface into one that fits on a curved surface, we must take into account where each pixel sent from the projection point hits both of these surfaces. However, note that the flat surface is not really there during the actual projection, but it is displayed in this image for the purpose of visualizing the concept.

The point P indicates the position of the projector, and has the coordinates (x_j, y_j, z_j). The Center of Projection (COP) is the position of the viewer, and is given by the coordinates (x_o, y_o, z_o). The coordinates (x_p, y_p, z_p) correspond to where a pixel (i, j)

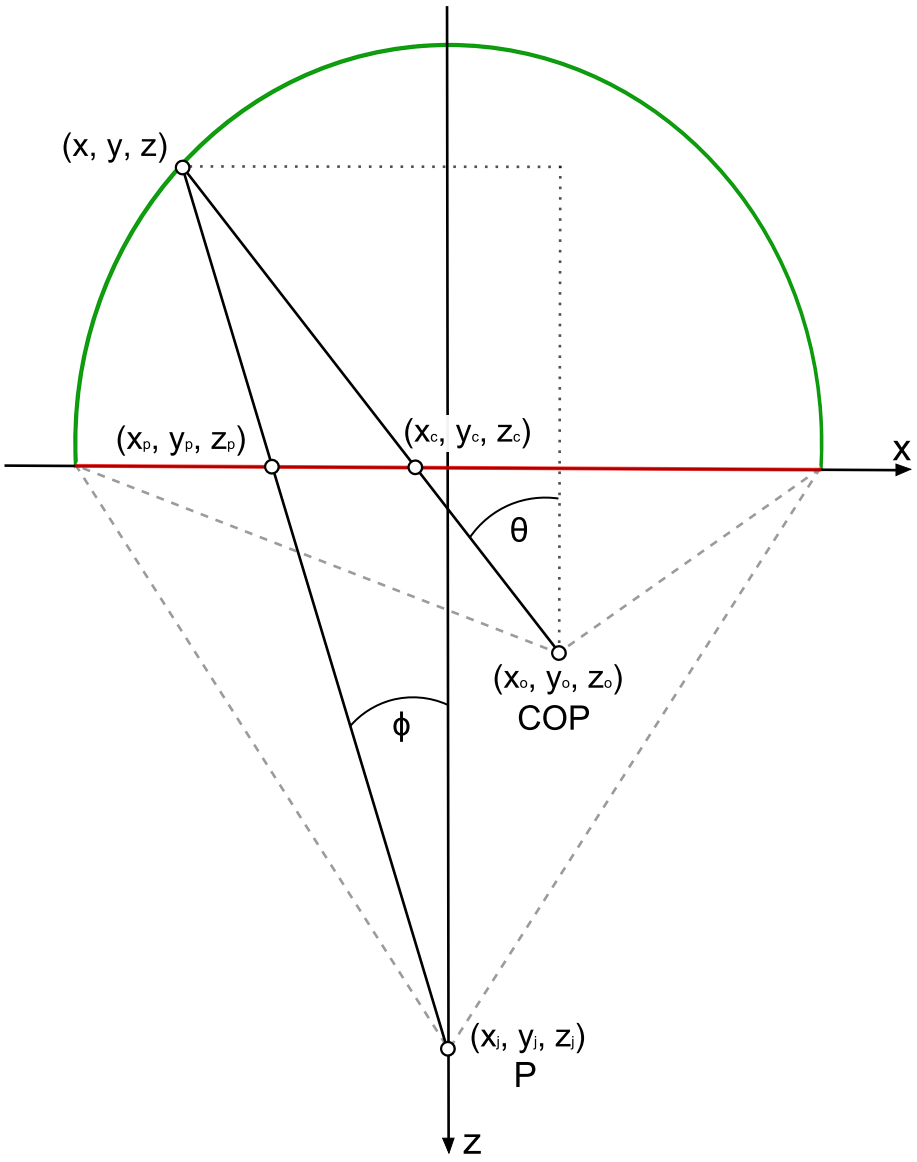

Figure 4.2: Top view of the projection on the cylindrical part of conCave, for the pixel transformation.

of the image that is projected from P, hits the plane; point (x, y, z) indicates where the same pixel lands on the cylindrical surface. The coordinates (x_c, y_c, z_c) correspond to the point on the plane that intersects the viewer's line of sight when the viewer is looking directly at point (x, y, z) . The idea behind the transformation is to make sure that what the viewer would have observed at point (x_c, y_c, z_c) on the plane is the same as what is projected towards (x_p, y_p, z_p), and also (x, y, z).

To clarify, first note that both the projection path from P and the viewer's line of sight intersect at (x, y, z). At this point we want to project the pixel the viewer would have seen at (x_c, y_c, z_c) if a regular planar projection was done on the flat plane, thus making it seem to the viewer as if he is observing this part of the image at point (x_c, y_c, z_c). The coordinates (x, y, z) must first be calculated in order to find the point (x_c, y_c, z_c). We then find the pixel coordinates (u, v), which is the pixel position on the image that corresponds to the point (x_c, y_c, z_c) on the plane. The value of (i, j) is then replaced by the value of (u, v), so that the pixel that would have been seen by the viewer on the flat surface at (x_c, y_c, z_c) is the same as the pixel projected onto the curved surface at (x, y, z). Once this is done for every single pixel on the bottom half of the image, the cylinder transformation of the image is complete.

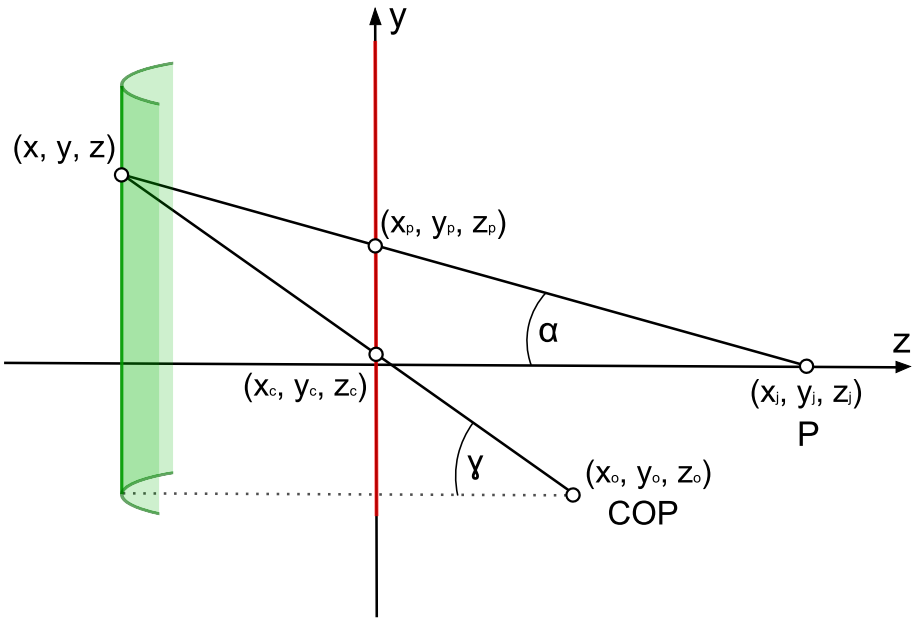

Figure 4.3: Side view of the projection on the cylindrical part of conCave, for the pixel transformation.

We assume that the planar surface is always parallel to the xy-plane of the coordinate system, so that we can always expect z_c and z_p to be equal. From here on we assume that $z_c = z_p$ holds for all equations. To be able to more clearly observe the scene from different angles, a top view of the same setup is provided in Figure 4.2 and a side view can be seen in Figure 4.3. All the coordinates are the same as in Figure 4.1.

Finding x,y and z

We start by finding expressions for (x, y, z), which is the point at which the projection of a certain pixel hits the curved surface. From the details in Figure 4.2 we obtain the following equations:

$$\tan\phi = \frac{x_p - x_j}{z_j - z_p} = \frac{x - x_j}{z_j - z} \tag{4.1}$$

$$\tan\theta = \frac{x_c - x_o}{z_o - z_c} = \frac{x - x_o}{z_o - z} \tag{4.2}$$

From Equation 4.1 we derive the following expression for x:

$$\begin{aligned} x &= x_j + \frac{(z_j - z)(x_p - x_j)}{z_j - z_p} \\ &= \frac{z_j(x_p - x_j) - z(x_p - x_j) + x_j(z_j - z_p)}{z_j - z_p} \\ &= \frac{x_p z_j - x_j z_j + x_j z_j - x_j z_p - x_p z + x_j z}{z_j - z_p} \\ &= \frac{x_p z_j - x_j z_p + z(x_j - x_p)}{z_j - z_p} \end{aligned} \tag{4.3}$$

From the side view of the scene provided in Figure 4.3 we can derive these additional equations:

$$\tan\alpha = \frac{y_p - y_j}{\sqrt{(z_j - z_p)^2 + (x_p - x_j)^2}} = \frac{y - y_j}{\sqrt{(z_j - z)^2 + (x - x_j)^2}} \tag{4.4}$$

$$\tan\gamma = \frac{y_c - y_o}{\sqrt{(z_o - z_c)^2 + (x_o - x_c)^2}} = \frac{y - y_o}{\sqrt{(z_o - z)^2 + (x_o - x)^2}} \tag{4.5}$$

The overall depiction provided in Figure 4.1 gives a much more accurate view of how these equations are obtained, as it shows all three axes at the same time.

In the same manner as for x, we derive an expression for y based on Equation 4.4:

$$\begin{aligned} y &= \frac{(y_p - y_j)\sqrt{(z_j - z)^2 + (x - x_j)^2}}{\sqrt{(z_j - z_p)^2 + (x_p - x_j)^2}} + y_j \\ &= (y_p - y_j)\sqrt{\frac{(z_j - z)^2 + (x - x_j)^2}{(z_j - z_p)^2 + (x_p - x_j)^2}} + y_j \end{aligned}$$

We then substitute the x in this equation with the expression we obtained in Equation 4.3, and we are left with the following expression for y:

$$y = (y_p - y_j)\sqrt{\frac{(z_j - z)^2 + (\frac{x_p z_j - x_j z_p + z(x_j - x_p)}{z_j - z_p} - x_j)^2}{(z_j - z_p)^2 + (x_p - x_j)^2}} + y_j \tag{4.6}$$

The (x_j, y_j, z_j) coordinates are already known, as they describe the position of the projector. Since we know the dimensions of the image and the projection surface, we can compute the (x_p, y_p) coordinates corresponding to any pixel position (i, j). We choose z_p based on where on the z-axis we want to place the planar surface. This means that the only unknown variable in Equations 4.3 and 4.6 is z. Since the surface we are projecting on has a cylindrical shape, and the vertical axis of the cylinder is parallel to the y-axis of the coordinate system, we can make use of the following equation:

$$(\frac{x}{R})^2 + (\frac{z}{R})^2 = 1 \tag{4.7}$$

This equation pertains to an ordinary circular cylinder with a radius R. We insert the expression for x derived in Equation 4.3 into Equation 4.7 to get the following quadratic equation:

$$\begin{aligned}
0 &= x^2 + z^2 - R^2 \\
&= (\frac{x_p z_j - x_j z_p + z(x_j - x_p)}{z_j - z_p})^2 + z^2 - R^2 \\
&= \frac{x_p^2 z_j^2 - 2x_p z_j x_j z_p + 2z(x_j - x_p)x_p z_j + x_j^2 z_p^2 - 2z(x_j - x_p)x_j z_p + z^2(x_j - x_p)^2}{(z_j - z_p)^2} \\
&\quad + z^2 - R^2
\end{aligned}$$

Multiplying by $(z_j - z_p)^2$ on both sides gives:

$$\begin{aligned}
0 &= x_p^2 z_j^2 - 2x_p z_j x_j z_p + 2z(x_j - x_p)x_p z_j + x_j^2 z_p^2 - 2z(x_j - x_p)x_j z_p + z^2(x_j - x_p)^2 \\
&\quad + (z_j - z_p)^2(z^2 - R^2) \\
&= z^2((x_j - x_p)^2 + (z_j - z_p)^2) + z(2(x_j - x_p)(x_p z_j - x_j z_p)) + x_p^2 z_j^2 - 2x_p z_j x_j z_p \\
&\quad + x_j^2 z_p^2 - R^2(z_j - z_p)^2
\end{aligned} \tag{4.8}$$

We can then solve for z using the quadratic formula:

$$z = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where:

$$\begin{aligned}
a &= (x_j - x_p)^2 + (z_j - z_p)^2 \\
b &= 2(x_j - x_p)(x_p z_j - x_j z_p) \\
c &= x_p^2 z_j^2 - 2x_p z_j x_j z_p + x_j^2 z_p^2 - R^2(z_j - z_p)^2
\end{aligned}$$

There is no need to calculate both roots of z, as the answer we are interested in always comes from the negative root. This comes from the fact that the semi-cylindrical surface is located on the negative side of the z-axis, meaning that the value of z we are looking for should always be negative.

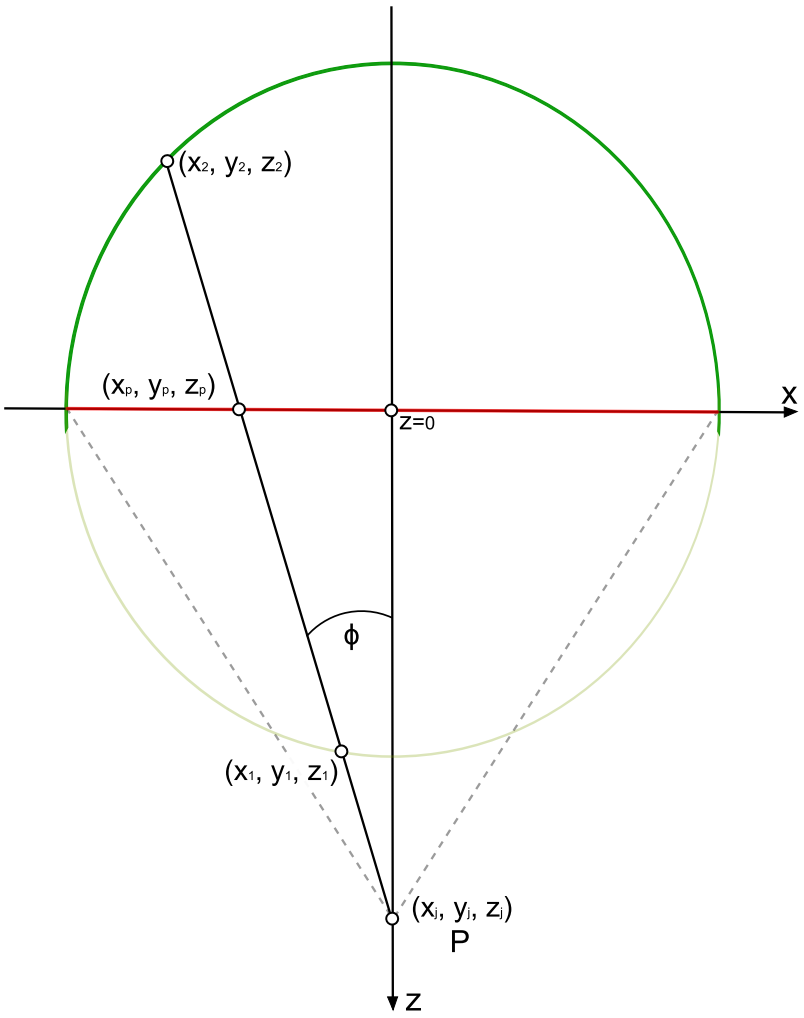

Figure 4.4: Visualizing both solutions of z.

In Figure 4.4 we also see that since the cylinder is centered at $z = 0$, when solving the quadratic formula, we will always end up with one positive solution and one negative solution, because the line from the projection point intersects the cylinder once before and once after crossing the z-axis. Of course, if the projector's position is offset in the x direction by a very large amount, it will be theoretically possible to get two negative solutions. However, this is irrelevant to us since in this case parts, of the projected image would be landing on the outside of the surface. Since the solution we need is always on the negative side of the z-axis, we can always disregard the positive solution. Looking at the expression for a in the quadratic formula, we see that it is always a positive value. We also know that everything inside the square root is always positive. This means that we only need to solve the formula with the

negative root, as we can be sure that it will always give us the negative solution for z:

$$z = \frac{-b - \sqrt{b^2 - 4ac}}{2a} \tag{4.9}$$

Once we have a solution for z, we insert this into Equations 4.3 and 4.6 to find the corresponding values of x and y.

Finding x_c, y_c and z_c

Next we find the coordinates (x_c, y_c, z_c), which is, as described earlier, the point where the viewer's line of sight intersects with the plane when looking at (x, y, z). As we already discussed in Section 4.1.2, z_c is equal to z_p. Therefore, since z_p is already known to us prior to projection, we only need to find x_c and y_c. From Equation 4.2 we derive the following expression for x_c:

$$\begin{aligned} x_c &= \frac{(z_o - z_c)(x - x_o)}{z_o - z} + x_o \\ &= \frac{(z_o - z_c)(x - x_o)}{z_o - z} + \frac{x_o(z_o - z)}{z_o - z} \\ &= \frac{xz_o - xz_c - x_oz_o + x_oz_c + x_oz_o - x_oz}{z_o - z} \\ &= \frac{x(z_o - z_c) + x_o(z_c - z)}{z_o - z} \end{aligned} \tag{4.10}$$

We can then solve for x_c using the solutions for x and z obtained from Equations 4.3 and 4.9. Finally, we derive the following expression for y_c from Equation 4.5:

$$\begin{aligned} y_c &= \frac{(y - y_o)\sqrt{(z_o - z_c)^2 + (x_o - x_c)^2}}{\sqrt{(z_o - z)^2 + (x_o - x)^2}} + y_o \\ &= (y - y_o)\sqrt{\frac{(z_o - z_c)^2 + (x_o - x_c)^2}{(z_o - z)^2 + (x_o - x)^2}} + y_o \end{aligned} \tag{4.11}$$

We solve for y_c using the solutions for x, y and z obtained from Equations 4.3, 4.6 and 4.9.

Pseudocode for the Cylinder Transformation

The pseudocode in Listings 4.1 describes the order in which the different computations of the cylinder transformation are performed.

```
// Input: P, COP, Input_image, Output_image
for(int i = 0; i< Input_image.width; i++)
{
    xp = compute_xp(i, Input_image.width);

    // Choose zp based on desired z-position
    // for planar surface.
    zp = 0.0;

    // Solve for z using quadratic equation
    z = Solve_z(P, xp, zp);

    x = Compute_x(P, z, xp);
    xc = Compute_xc(COP, x, zp);
    u = Compute_u(xc, Input_image.width);
    for(int j = Input_image.height/2; j< Input_image.height; j++)
    {
        yp = compute_yp(j, Input_image.height);
        y = Compute_y(P,  x, z, xp, yp);
        yc = Compute_yc(COP, x, y, z, xc, zp);
        v = Compute_v(yc, Input_image.height);

        Output_image[ i ][ j ] = Input_image[ u ][ v ];
    }
}
```

Listing 4.1: Cylinder transformation pseudocode

The first step is to solve the quadratic equation to find z. The projector's position P, which we are given as input, and the intersection point (x_p, y_p, z_p), which we get from scaling the current pixel position (i, j), are enough to solve this equation. Once this is done, we can use the value of z to calculate the corresponding x and y values, and thereafter find the point (x_c, y_c, z_c) which we use for transforming the pixels. Note that the cylinder transformation is only being applied to the bottom half of the image.

The cylindrical shape of the surface enables us to first compute a value for x and x_c and then find all corresponding y and y_c coordinates for that particular pair of x and z. This is possible because the x and z coordinates of a cylinder do not depend on the y coordinate, as we can see in Equation 4.7, meaning that for each pair (x, z), we can compute all values of y without having to recalculate the value of x or z. This is taken into account in the example in Listings 4.1 by computing x, x_c and u outside the second **for**-loop. We find u and v by scaling x_c and y_c back to the resolution of the image, and use them to determine the pixel values of the output image.

4.1.3 Sphere Transformation

Here we describe the mathematical process of transforming the part of the image that is to be projected onto the spherical surface. The overall procedure of the transformation is very similar to the cylinder transformation discussed earlier. The difference is in the mathematical details of finding the unknown variables and in which order the different computations need to be performed.

Projection on the Spherical Surface

In Figure 4.5 we look at the same setup as earlier from a bird's-eye view, but this time the projection is done on the spherical part of the surface. The remaining details are the same as the ones describing the positioning of the cylindrical surface in Section 4.1.2. A side view of the same scene is shown in Figure 4.6.

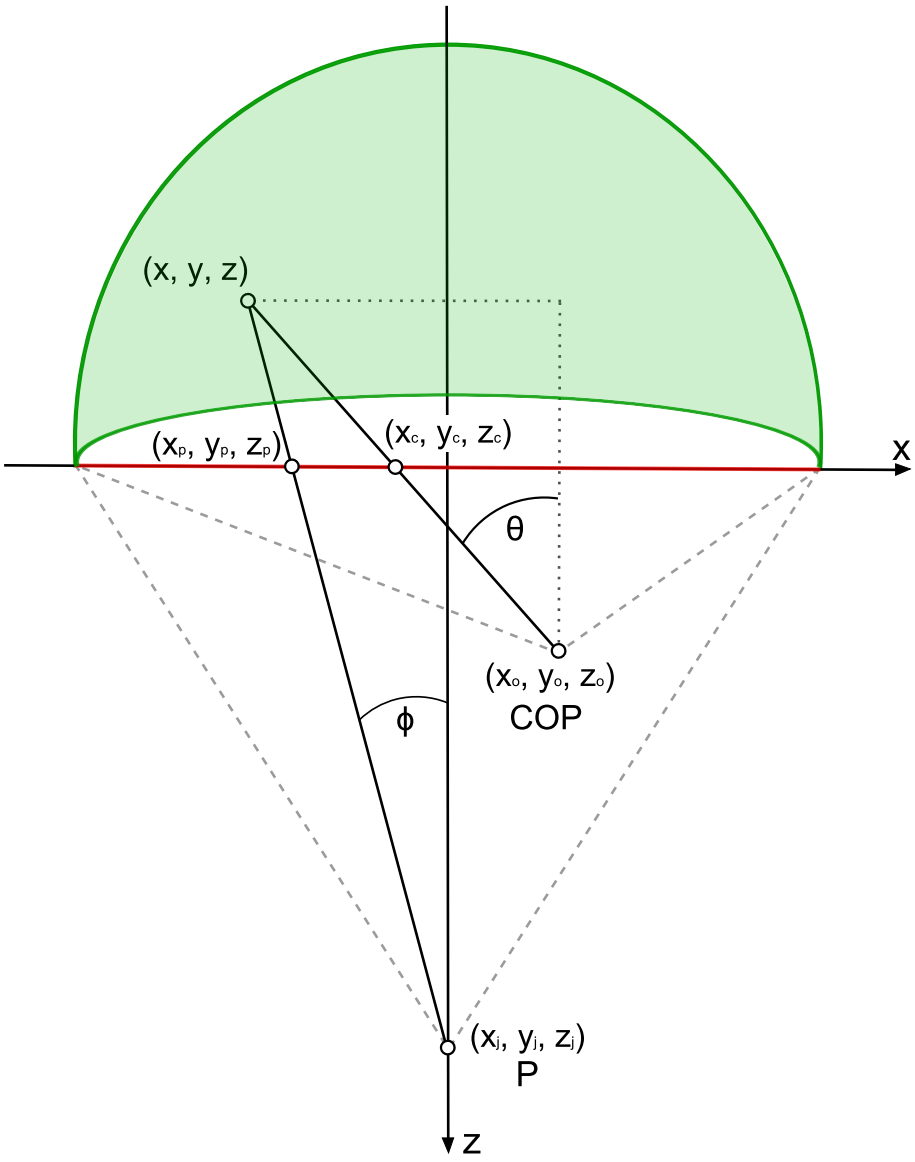

Figure 4.5: Top view of the projection on the spherical part of conCave, for the pixel transformation.

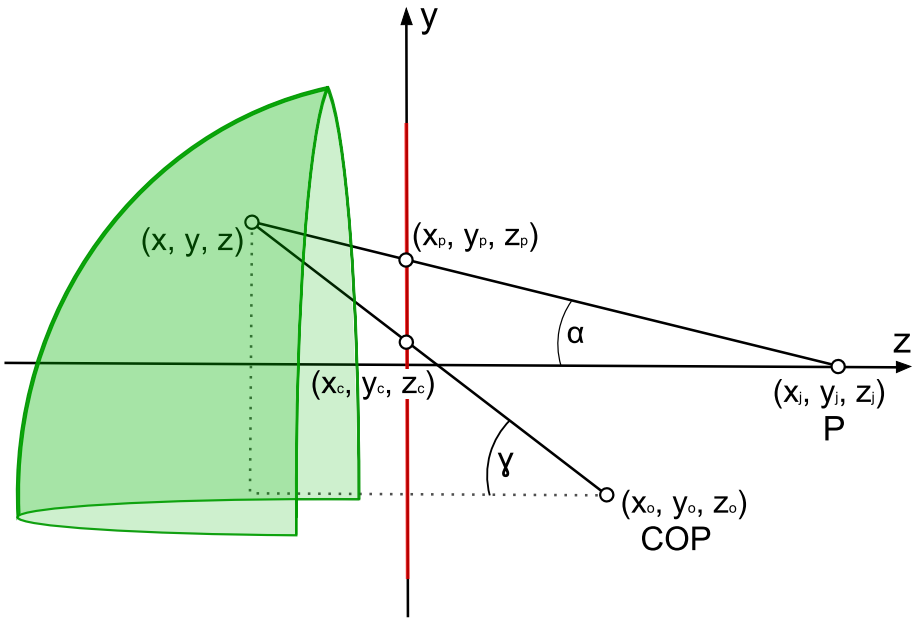

Figure 4.6: Side view of the projection on the spherical part of conCave, for the pixel transformation.

Finding x, y and z

The expressions for x and y are similar to the ones we derived in Section 4.1.2. Once again, the only unknown element is the z-coordinate. As we are working on a hemisphere-shaped surface, we can make use of the following equation:

$$x^2 + y^2 + z^2 = R^2 \tag{4.12}$$

This equation describes a sphere centered at the origin with a radius R. We insert the expressions for x and y derived in Equations 4.3 and 4.6 into Equation 4.12, and obtain the following equation:

$$\begin{aligned}
0 &= x^2 + y^2 + z^2 - R^2 \\
&= \left(\frac{x_p z_j - x_j z_p + z(x_j - x_p)}{z_j - z_p}\right)^2 + \left((y_p - y_j)\sqrt{\frac{(z_j - z)^2 + \left(\frac{x_p z_j - x_j z_p + z(x_j - x_p)}{z_j - z_p} - x_j\right)^2}{(z_j - z_p)^2 + (x_p - x_j)^2}} + y_j\right)^2 \\
&\quad + z^2 - R^2 \\
&= \frac{(x_j(z_p - z) - x_p(z_j - z))^2}{(z_j - z_p)^2} + \left((y_p - y_j)\sqrt{\frac{(z_j - z)^2 + \frac{(x_j - x_p)^2 (z - z_j)^2}{(z_j - z_p)^2}}{(z_j - z_p)^2 + (x_p - x_j)^2}} + y_j\right)^2 + z^2 - R^2
\end{aligned}$$

This results in a fourth degree polynomial, containing four solutions for z. As this is a very long equation, it was not solved by hand. The math tool Maple [16], was used to solve this equation, and also to help us find a much simpler form for the four solutions. Here we only show the simplified formulas for z, which we created

by studying the patterns of the actual solutions:

$$z = \frac{b \pm \sqrt{d}}{a} \ , \ \frac{c \pm \sqrt{e}}{a}$$

where:

$$a = (x_j - x_p)^2 + (y_j - y_p)^2 + (z_j - z_p)^2$$

$$b = z_j(x_p^2 - x_p x_j - y_p(y_j - y_p)) + z_p(x_j^2 - x_p x_j + y_j(y_j - y_p))$$

$$c = z_j(x_p(x_p - x_j) + y_p(y_p - 3y_j) + 2y_j^2) + z_p(x_j(x_j - x_p) + y_j(y_p - y_j))$$

$$\begin{aligned} d = (z_j - z_p)^2(z_j^2(R^2 - x_p^2 - y_p^2) - 2z_p z_j(R^2 - x_p x_j - y_p y_j) - (x_j y_p - x_p y_j)^2 \\ + z_p^2(R^2 - x_j^2 - y_j^2) + R^2((x_j - x_p)^2 + (y_j - y_p)^2)) \end{aligned}$$

$$\begin{aligned} e = (z_j - z_p)^2(z_j^2(R^2 - x_p^2 - y_p^2 + 4y_j(y_p - y_j)) - 2z_p z_j(R^2 + y_p y_j - 2y_j^2 - x_p x_j) \\ + z_p^2(R^2 - x_j^2 - y_j^2) + y_j^2(R + 2x_j - x_p)(R - 2x_j + x_p) \\ - 2y_p y_j(R^2 - 2x_j^2 + x_p x_j) + R^2((x_j - x_p)^2 + y_p^2) - x_j^2 y_p^2) \end{aligned}$$

Similarly to the cylinder case, we are only interested in a negative value for z. We can see that a is always positive, since each part of its corresponding expression is a square. Knowing that d and e must also always be positive means that we can already rule out two of the solutions for z. This is the same procedure we followed to eliminate the positive solution for the cylinder case in Section 4.1.2. The two remaining equations, 4.13 and 4.14, can both be negative.

$$z = \frac{b - \sqrt{d}}{a} \tag{4.13}$$

$$z = \frac{c - \sqrt{e}}{a} \tag{4.14}$$

When the position of the projector is within acceptable ranges in the x and y directions, both solutions seem to always provide the same value for z. This was investigated by creating several 3D plots in Maple, trying to cover as many different cases as possible. Only in cases that are practically irrelevant, such as moving the projector a very long distance in the x or y direction do the solutions differ from each other. Since we know that the projector is always within the range of being able to project the entire image onto the inside surface, we only need to solve one of these equations. We choose to solve for z using Equation 4.13, because it requires fewer computations than solving Equation 4.14.

Once we have a solution for z, we insert this into Equations 4.3 and 4.6 to find the corresponding values of x and y.

Finding x_c and y_c

The expressions for x_c and y_c can be derived in the same way as we did for the cylinder case, from Equations 4.2 and 4.5, and the same as the ones found in Equations 4.10 and 4.11. We solve them in the same way as we did earlier, using the obtained solutions for x, y and z.

Pseudocode for the Sphere Transformation

The following pseudocode describes the order in which the different computations of the sphere transformation are performed.

```
1  // Input: P, COP, Input_image, Output_image
2  for(int i = 0; i< Input_image.width; i++)
3  {
4      for(int j = 0; j< Input_image.height/2; j++)
5      {
6          xp = compute_xp(i, Input_image.width);
7          yp = compute_yp(j, Input_image.height);
8
9          // Choose zp based on desired z-position
10         // for planar surface.
11         zp = 0.0;
12
13         // Solve for z using fourth degree equation
14         z = Solve_z(P, xp, zp);
15
16         x = Compute_x(P, z, xp);
17         y = Compute_y(P,  x, z, xp, yp);
18         xc = Compute_xc(COP, x, zp);
19         yc = Compute_yc(COP, x, y, z, xc, zp);
20         u = Compute_u(xc, Input_image.width);
21         v = Compute_v(yc, Input_image.height);
22
23         Output_image[ i ][ j ] = Input_image[ u ][ v ];
24     }
25 }
```

Listing 4.2: Sphere transformation pseudocode

It is very similar to the cylinder transform code with a few small changes. The difference is that we now need to compute a z value for each (x, y)-pair, while in the previous method we only needed to compute a new z for each value of x. Thus the

equation for z is solved inside the innermost **for**-loop in Listings 4.2. As solving for z is the most expensive step in both transformations, the computational intensity of the sphere transformation is much higher than that of the cylinder transformation, since z has to be calculated more often. It should also be noted that we only need to solve a quadratic equation in the cylinder case, while for the sphere it is a fourth degree equation.

4.2 The Vertex Transformation Method

This approach is similar to the pixel transformation method described in Section 4.1 in many ways. The main difference is that instead of transforming an image, we transform the actual way in which the scene is projected. This means that we do not rely on a pre-rendered image to work on, but alter the projection of the original scene by doing calculations based on the information of the scene that is available to us. By doing so, we will not be getting a pixel perfect transformation as we did in the previous method, but we will avoid the overhead of having to render the image twice.

4.2.1 Overview of the Transformation

The following list provides an explanation of the main steps in the transformation procedure.

1. For each vertex in the scene, we want to calculate the direction in which it is to be projected, so that it appears correctly on a curved surface.

2. A Cylinder transformation procedure is performed on all the vertices at the bottom half of the scene. The positions of where these vertices are to be projected are calculated based on the position of the viewer, the projector, and the projection surface.

3. A Sphere transformation procedure is performed on all the vertices on the top half of the scene. The two transformation procedures are independent of each other and do not need to be called in any specific order.

4. Once a projection point has been calculated and given to each vertex in the scene, the transformed image can be rendered and then projected onto the curved surface.

4.2.2 Mathematical Details

Here we describe the mathematical process of transforming the projection for both the cylindrical and spherical parts of the surface. The basic idea here is to calculate a projection position for each vertex. These positions are calculated to be on an imaginary flat surface in front of the curved surface; however, the calculations are based on the fact that the image is to be projected onto a curved surface.

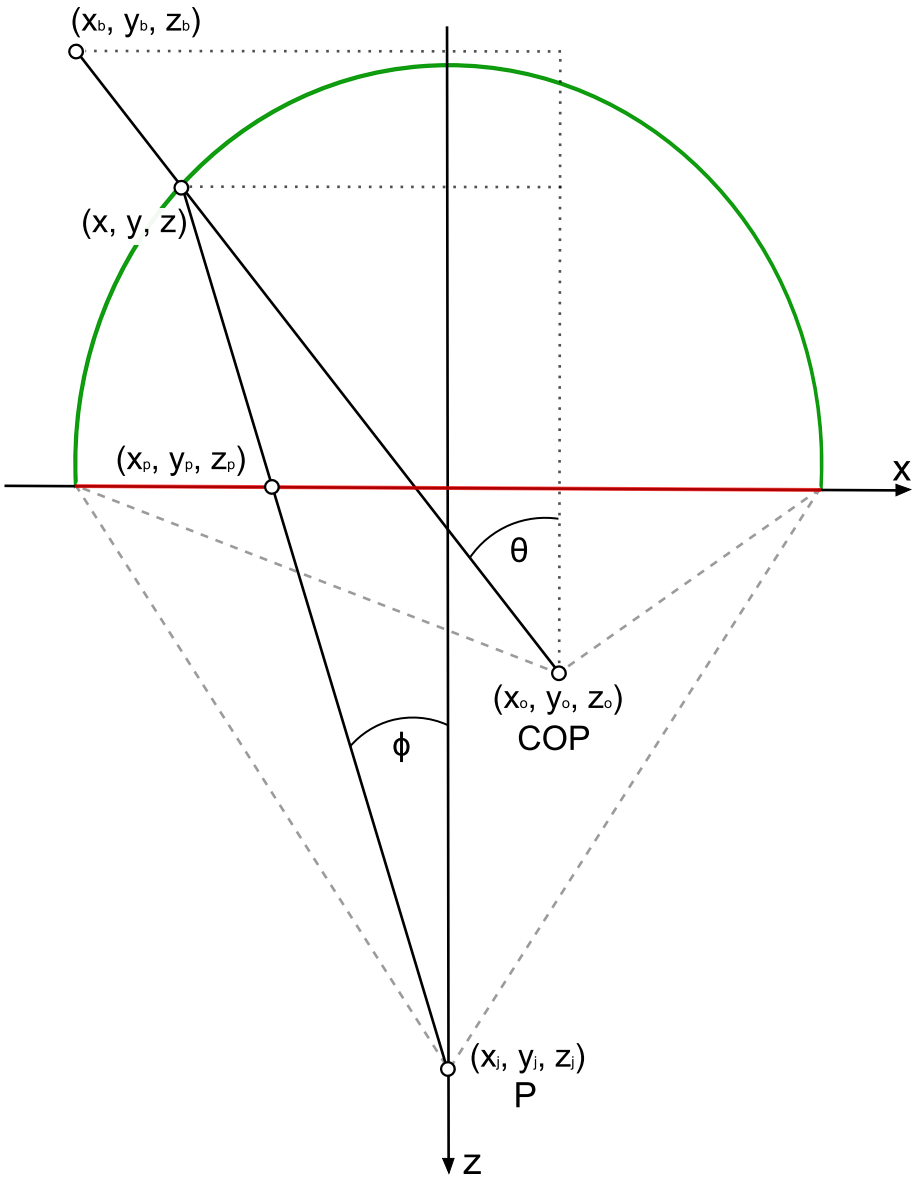

Figure 4.7: Top view of the projection on the cylindrical part of conCave, for the vertex transformation.

Projection Setup

An overview of the setup is provided in Figures 4.7 and 4.8. They are very similar to the ones introduced in Figures 4.2 and 4.3 with some minor differences. As we are transforming the actual projection in this approach, our goal is to find a projection position (x_p, y_p, z_p) for each vertex (x_b, y_b, z_b). We do this by first figuring out where the viewer's line of sight intersects the curved projection surface when looking at the vertex. This is the straight line from the point (x_o, y_o, z_o) to the point (x_b, y_b, z_b). The coordinates (x, y, z) describe where this line intersects the curved surface.

After the (x, y, z) coordinates are calculated we can proceed to finding (x_p, y_p, z_p), which are the projection coordinates of the current vertex.

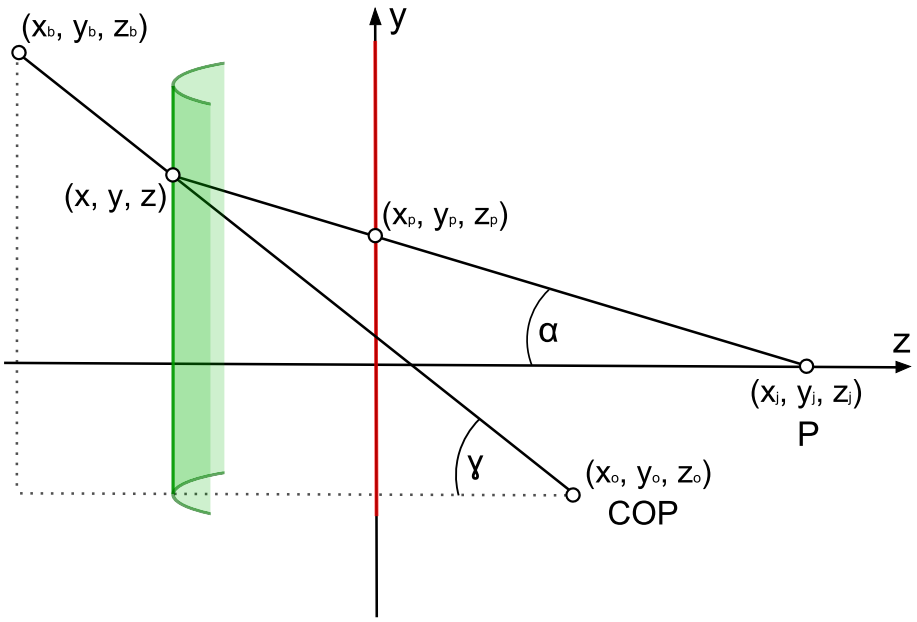

Figure 4.8: Side view of the projection on the cylindrical part of conCave, for the vertex transformation.

The red line corresponds to an imaginary flat surface on which the projection coordinates of all the vertices are to be placed. We see that the point (x_p, y_p, z_p) is the correct projection coordinates for vertex (x_b, y_b, z_b), because the line from the projector (x_j, y_j, z_j), to the projection coordinates intersects the curved surface at the same point in which the viewer observes the vertex on the curved surface.

For further general information on the setup we refer to Section 4.1.2, as all remaining details are very similar to the ones in our previous approach.

Finding x, y and z

The first step is calculating the (x, y, z) coordinates. From the details in Figures 4.7 and 4.8, we obtain the same equation for ϕ and α as we do in Section 4.1.2. The equations for θ and γ are now slightly different, but are still obtained in the same manner as before:

$$\tan\theta = \frac{x_b - x_o}{z_o - z_b} = \frac{x - x_o}{z_o - z} \tag{4.15}$$

$$\tan\gamma = \frac{y_b - y_o}{\sqrt{(z_o - z_b)^2 + (x_b - x_o)^2}} = \frac{y - y_o}{\sqrt{(z_o - z)^2 + (x_o - x)^2}} \tag{4.16}$$

From Equations 4.15 and 4.16 we then derive expressions for x and y:

$$\begin{aligned} x &= x_o + \frac{(z_o - z)(x_b - x_o)}{z_o - z_b} \\ &= \frac{z_o(x_b - x_o) - z(x_b - x_o) + x_o(z_o - z_b)}{z_o - z_b} \\ &= \frac{x_b z_o - x_o z_o + x_o z_o - x_o z_b - x_b z + x_o z}{z_o - z_b} \\ &= \frac{x_b z_o - x_o z_b + z(x_o - x_b)}{z_o - z_b} \end{aligned} \tag{4.17}$$

$$\begin{aligned} y &= \frac{(y_b - y_o)\sqrt{(z_o - z)^2 + (x - x_o)^2}}{\sqrt{(z_o - z_b)^2 + (x_b - x_o)^2}} + y_o \\ &= (y_b - y_o)\sqrt{\frac{(z_o - z)^2 + (x - x_o)^2}{(z_o - z_b)^2 + (x_b - x_o)^2}} + y_o \end{aligned}$$

We substitute the x in this equation with the expression we obtain in Equation 4.17 to get the following expression for y:

$$y = (y_b - y_o)\sqrt{\frac{(z_o - z)^2 + (\frac{x_b z_o - x_o z_b + z(x_o - x_b)}{z_o - z_b} - x_o)^2}{(z_o - z_b)^2 + (x_b - x_o)^2}} + y_o \tag{4.18}$$

The vertex position and the position of the COP is already known, which leaves z as the only unknown variable in Equations 4.17 and 4.18. For the cylindrical part of the surface, we make use of Equation 4.7, and by also including the expression for x derived in Equation 4.17, we obtain the following quadratic equation:

$$\begin{aligned} 0 &= x^2 + z^2 - R^2 \\ &= (\frac{x_b z_o - x_o z_b + z(x_o - x_b)}{z_o - z_b})^2 + z^2 - R^2 \\ &= \frac{x_b^2 z_o^2 - 2x_b z_o x_o z_b + 2z(x_o - x_b)x_b z_o + x_o^2 z_b^2 - 2z(x_o - x_b)x_o z_b + z^2(x_o - x_b)^2}{(z_o - z_b)^2} \\ &\quad + z^2 - R^2 \end{aligned}$$

Multiplying by $(z_o - z_b)^2$ on both sides gives:

$$\begin{aligned} 0 &= x_b^2 z_o^2 - 2x_b z_o x_o z_b + 2z(x_o - x_b)x_b z_o + x_o^2 z_b^2 - 2z(x_o - x_b)x_o z_b + z^2(x_o - x_b)^2 \\ &\quad + (z_o - z_b)^2(z^2 - R^2) \\ &= z^2((x_o - x_b)^2 + (z_o - z_b)^2) + z(2(x_o - x_b)(x_b z_o - x_o z_b)) + x_b^2 z_o^2 - 2x_b z_o x_o z_b \\ &\quad + x_o^2 z_b^2 - R^2(z_o - z_b)^2 \end{aligned} \tag{4.19}$$

We can then solve for z using the quadratic formula:

$$z = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

where:

$$a = (x_o - x_b)^2 + (z_o - z_b)^2$$
$$b = 2(x_o - x_b)(x_b z_o - x_o z_b)$$
$$c = x_b^2 z_o^2 - 2x_b z_o x_o z_b + x_o^2 z_b^2 - R^2(z_o - z_b)^2$$

For the spherical surface, we use Equation 4.12 together with the x and y we derived in Equations 4.17 and 4.18 to obtain the following fourth degree equation:

$$\begin{aligned}
0 &= x^2 + y^2 + z^2 - R^2 \\
&= \left(\frac{x_b z_o - x_o z_b + z(x_o - x_b)}{z_o - z_b}\right)^2 + \left((y_b - y_o)\sqrt{\frac{(z_o - z)^2 + \left(\frac{x_b z_o - x_o z_b + z(x_o - x_b)}{z_o - z_b} - x_o\right)^2}{(z_o - z_b)^2 + (x_b - x_o)^2}} + y_o\right)^2 \\
&\quad + z^2 - R^2 \\
&= \frac{(x_o(z_b - z) - x_b(z_o - z))^2}{(z_o - z_b)^2} + \left((y_b - y_o)\sqrt{\frac{(z_o - z)^2 + \frac{(x_o - x_b)^2(z - z_o)^2}{(z_o - z_b)^2}}{(z_o - z_b)^2 + (x_b - x_o)^2}} + y_o\right)^2 + z^2 - R^2
\end{aligned}$$

Using Maple [16], we solve this equation in the same way we do in Section 4.1.2, and obtain the following solutions:

$$z = \frac{b \pm \sqrt{d}}{a} \ , \ \frac{c \pm \sqrt{e}}{a}$$

where:

$$a = (x_o - x_b)^2 + (y_o - y_b)^2 + (z_o - z_b)^2$$

$$b = z_o(x_b^2 - x_b x_o - y_b(y_o - y_b)) + z_b(x_o^2 - x_b x_o + y_o(y_o - y_b))$$

$$c = z_o(x_b(x_b - x_o) + y_b(y_b - 3y_o) + 2y_o^2) + z_b(x_o(x_o - x_b) + y_o(y_b - y_o))$$

$$\begin{aligned}
d = {} & (z_o - z_b)^2(z_o^2(R^2 - x_b^2 - y_b^2) - 2z_b z_o(R^2 - x_b x_o - y_b y_o) - (x_o y_b - x_b y_o)^2 \\
& + z_b^2(R^2 - x_o^2 - y_o^2) + R^2((x_o - x_b)^2 + (y_o - y_b)^2))
\end{aligned}$$

$$\begin{aligned}
e = {} & (z_o - z_b)^2(z_o^2(R^2 - x_b^2 - y_b^2 + 4y_o(y_b - y_o)) - 2z_b z_o(R^2 + y_b y_o - 2y_o^2 - x_b x_o) \\
& + z_b^2(R^2 - x_o^2 - y_o^2) + y_o^2(R + 2x_o - x_b)(R - 2x_o + x_b) \\
& - 2y_b y_o(R^2 - 2x_o^2 + x_b x_o) + R^2((x_o - x_b)^2 + y_b^2) - x_o^2 y_b^2)
\end{aligned}$$

For the quadratic equation, we use the solution described in Equation 4.9 and for the fourth degree equation, we use the one mentioned in Equation 4.13. The logic behind choosing the proper solutions for both equations, is the same as the one derived in our previous method, and we refer to Sections 4.1.2 and 4.1.3 for further details on this matter. Once we have solved for z, we insert it into Equations 4.17 and 4.18 to find the corresponding values of x and y.

Finding x_p, y_p and z_p

The value of z_p determines the position of the imaginary planar surface on the z-axis, and is already chosen prior to doing these calculations. This means that we just have to deal with finding x_p and y_p to decide the projection position for the vertex. From Equations 4.1 and 4.4, we derive the following expressions for x_p and y_p:

$$\begin{aligned} x_p &= \frac{(z_j - z_p)(x - x_j)}{z_j - z} + x_j \\ &= \frac{(z_j - z_p)(x - x_j)}{z_j - z} + \frac{x_j(z_j - z)}{z_j - z} \\ &= \frac{xz_j - xz_p - x_jz_j + x_jz_p + x_jz_j - x_jz}{z_j - z} \\ &= \frac{x(z_j - z_p) + x_j(z_p - z)}{z_j - z} \end{aligned} \tag{4.20}$$

$$\begin{aligned} y_p &= \frac{(y - y_j)\sqrt{(z_j - z_p)^2 + (x_j - x_p)^2}}{\sqrt{(z_j - z)^2 + (x_j - x)^2}} + y_j \\ &= (y - y_j)\sqrt{\frac{(z_j - z_p)^2 + (x_j - x_p)^2}{(z_j - z)^2 + (x_j - x)^2}} + y_j \end{aligned} \tag{4.21}$$

We then solve for x_p and y_p using the solutions for x and y obtained from Equations 4.17 and 4.18. For z, we use the solution obtained from Equation 4.9 or 4.13, depending on which part of the surface we are currently working on.

Pseudocode for the Transformation

The pseudocode for both the cylinder transformation part and the sphere transformation part in this approach are identical. The only difference is that they are performed on different parts of the model and that the value of z is calculated using

different equations.

```
// Input: P, COP, and all the vertices in the scene
for(int i = 0; i<numVertices; i++)
{
   V = Vertices[i];

   // Solve for z using quadratic equation for
   // cylinder transformation or fourth-
   // degree equation for sphere transformation
   z = Solve_z(COP, V);

   x = Compute_x(COP, z);
   y = Compute_y(COP,  z, x);

   // Choose zp based on desired z-position
   // for planar surface.
   zp = 0.0;

   xp = Compute_xp(P, x, zp);
   yp = Compute_yp(P, x, y, xp, zp);

   Projection_position[i] = (xp, yp, zp);
}
```

Listing 4.3: Cylinder, or sphere transformation pseudocode, for the vertex transformation.

The operations here are performed on each vertex instead of per pixel, which means that the total number of operations is now dependent on the number of vertices in the scene, and not the size of the image that is to be rendered.

CHAPTER 5

Implementation

This chapter covers the implementation of both transformation methods, and describes how we provide stereoscopic rendering of the transformed images.

Section 5.1 describes the working environment in which the implementation of the different methods take place, and which tools, languages and libraries are used throughout the development the code. Then, Section 5.2 provides a brief overview of the execution of the program that handles the transformation methods. Section 5.3 describes the implementation of the pixel transformation method, and covers both the sequential, and a massively parallel version implemented in CUDA. Then, Section 5.4 looks at the vertex transformation method, which is implemented using shaders. Finally Section 5.5 covers the stereoscopic rendering of the transformed images.

5.1 Working Environment

The code developed during the course of this thesis is mainly written in $C++$ using the Visual $C++$ IDE [17] on a Windows 7 platform. The choice of programming language and platform is based on previous experience around writing and testing code, when developing applications.

Certain parts of the code are written in CUDA C, which is the language used to write code for the CUDA framework described in Appendix A, and some parts of the code are written in OpenGL Shader Language (GLSL). The CUDA and GLSL code are also written using the Visual C++ IDE with the aid of some add-ons to enable support for these languages in the IDE. We use OpenGL for handling the visualization, such as positioning the models, lighting and shading, and rendering the different images for projection. Several open source libraries are used, including GLEW and GLFW, and we also use some CUDA based libraries provided by the CUDA SDK.

5.2 Program Overview

This section provides an overview of the program that handles the initialization and execution of the transformation methods, and displays the transformed images. Figure 5.1 shows the most important steps in the execution of the program, which we explain in more detail in the list below.

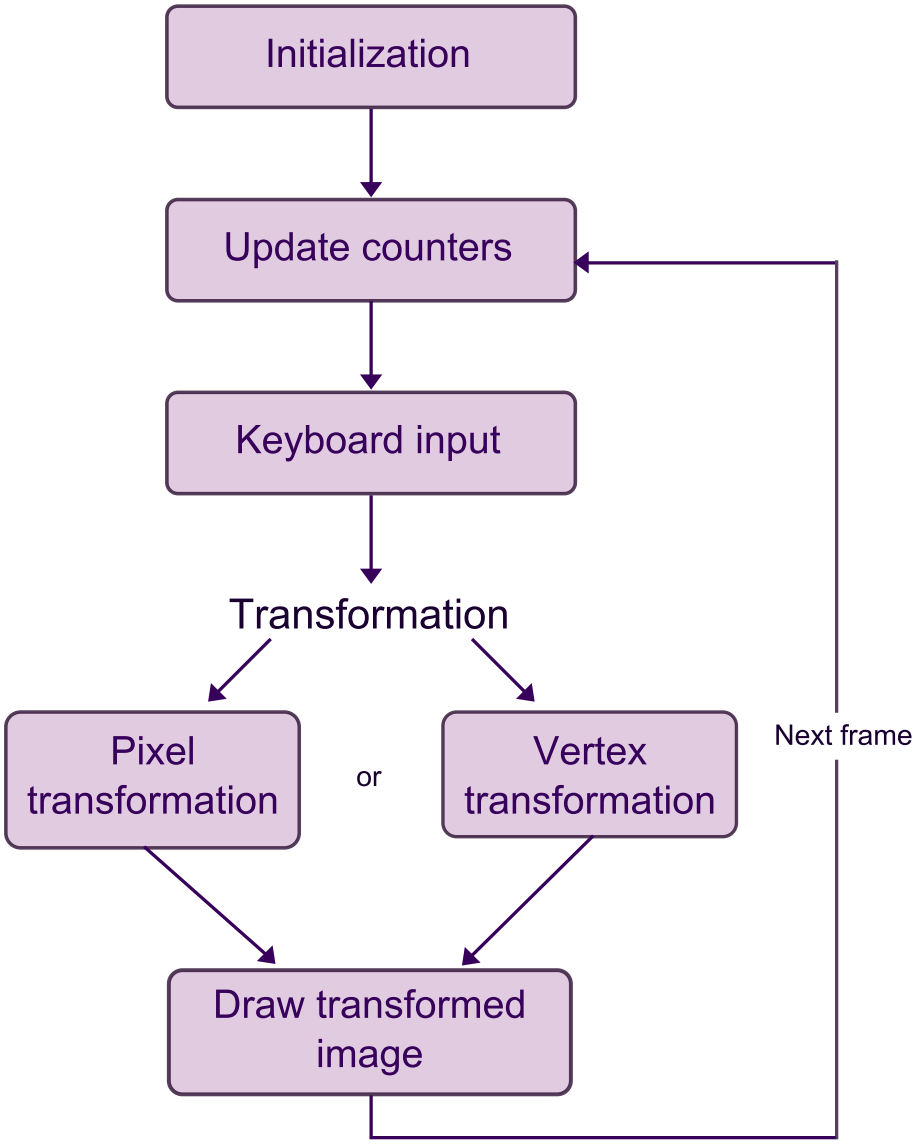

Figure 5.1: Overview of the transformation program execution.

1. The first step is the initialization. This covers the initialization of every element in the program, such as setting the window parameters, key handling, initializing OpenGL and setting the projection matrix to be fit for a perspective projection with given parameters. If the program will be using the pixel transformation method, we must allocate space on device memory to receive the positions of the projector and viewer. If the program will be using the vertex transformation method, we must load and initialize the shader files that will be used for the transformation.

2. We enter the main loop of the program. The first step here is to update the counters for the time and the frame count so that we can calculate the amount of time and frames that have passed in total, and since the previous frame was rendered.

3. Any keyboard commands that are given are checked at this point, and corresponding actions are carried out. Commands for exiting the program, toggling various features such as transformation and stereo rendering, and also navigating within the projection are all handled at this point.

4. The transformation phase depends on what method of transformation has been set prior to starting the program. The program branches off into different function calls based on whether it is the sequential pixel transformation method, parallel pixel transformation method, or the vertex transformation method that has been chosen. If the program is set to run the sequential pixel transformation method, then it first renders and image using the perspective projection matrix, and then calls two sequential functions for doing the transformation of the cylindrical part and the spherical part of the image. If the parallel pixel transformation method is chosen, then the program calls functions that communicate with the device, transfers the image data and all other the necessary information over to device memory, and has all the computation done on the device through kernel functions. The transformed image is then returned to the host memory. If the vertex transformation method is chosen, then the program calls a function that runs the shader programs which transform the positions of all the vertices in the model.

5. Finally, the finished image is rendered and drawn to the buffer, so that it can be displayed on screen. If stereo rendering is enabled, then it will create a stereo pair of the transformed scene. This concludes a single iteration of the main loop, and the program proceeds to create the next frame.

5.3 Pixel Transformation Method

This section describes the implementation of the pixel transformation method described in Section 4.1. Two different versions of this method are developed, one sequential version coded in C++, and one massively parallel version using C++ and CUDA. The implementations (especially the sequential one) are very straightforward and follow the pseudocodes presented in Section 4.1. They can be found in Listings 4.1 and 4.2. Although very similar to the pseudocode, some intermediate steps were added to simplify common expressions, such that the calculations performed are much more readable and closer to the form in which they were described in Section 4.1. The sequential code will not be described in any further detail, but we will be looking more closely at the parallel version and describing key parts of the code.

5.3.1 Computing in Parallel

The parallel version utilizes the CUDA framework for doing all the heavy computing on the GPU, and all parts of the code that communicates with, and relies on the graphics processor, is written in CUDA C. The actual code is still very similar to the sequential version; however, instead of being run sequentially through a doubly nested **for**-loop, the operations are run in parallel across thousands of threads at a time. This means that each thread is responsible for calculating and outputting the results for a handful of pixels. To accomplish this, we must create a thread assignment pattern for the kernel function (a function that is run on the GPU device) that splits the amount of work among several threads. When we then run the kernel, we provide it with the number of threads it requires to perform its operation on the given input. One example of how different threads are handled in the kernel can be seen in Listing 5.1.

```
1  // Call with n threadblocks,
2  // each containing WIDTH/n threads.
3
4  __global__ void TransformationKernel(...)
5  {
6      int i = (blockIdx.x * blockDim.x) + threadIdx.x;
7
8      // Transformation code, where each thread, i,
9      // is responsible for row i of the image
10     // ...
11 }
```

Listing 5.1: Threads in a kernel

The variables *blockIdx, threadIdx* and *blockDim* are built-in variables that are meant to help the kernel obtain the id of the current thread, the id of its thread block, and the dimensions of the threadblock. In line 6 of the code in Listings 5.1, we use these built-in variables to find the global id of current thread within the 3-level thread hierarchy. We can then proceed with giving commands to each thread based on their global id. The thread hierarchy is explained in more detail in Appendix A.

Row per Thread

If our transformation kernel follows the thread assigning pattern displayed in Listing 5.1, we can remove the outermost **for**-loop present in the pseudo code in Listings 4.1 and 4.2, and give each thread the task of computing an entire iteration of that loop. This means that if the width of the image is 512 pixels, instead of having a **for**-loop going from 0 to 511, we run our kernel on 512 separate threads, each having a unique global id between 0 and 511. The thread whose global id corresponds to the index i is responsible for everything that is done during iteration i in the sequential version. If the dimensions of the image are 512x512, we have made the transformation go a lot faster by diving the work of calculating 512x512 pixels over 512 threads.

Pixel per Thread

But there is no reason to stop there. As the GPU is capable of having hundreds of thousands of threads working in parallel, we can code the kernel in such a way that the calculations for each pixel are done by a separate thread. An example of this is shown in Listing 5.2.

```
1  // Call with WIDTH*HEIGHT/(a*a) number of two-dimensional
2  // threadblocks, each containing a * a threads.
3
4  __global__ TransformationKernel(...)
5  {
6      int global_block_id = blockIdx.x * blockDim.x * blockDim.y;
7
8      int current_thread = global_block_id
9                          + (threadId.x + blockDim.x * threadIdx.y);
10
11     int i = t modulo WIDTH;
12     int j = (t - i) / WIDTH;
13
14     // Transformation code, where each thread, (i, j),
15     // is responsible for pixel [i, j] of the image
16     // ...
17 }
```

Listing 5.2: One threads per pixel

In this example the kernel is called with $\frac{b \cdot b}{a \cdot a}$ thread blocks, each containing a two dimensional array of threads. Each block has axa threads, and bxb defines the resolution of the image. In total this makes $\frac{b \cdot b}{a \cdot a} \cdot (a \cdot a)$ threads, which is also the number of pixels in a bxb image. The reason we cannot simply call the kernel with just n thread blocks, each containing $b \cdot b/n$ threads, is due to the limitations of the thread hierarchy. The allowed grid size is usually a lot larger than the allowed block size, which means that we must resort to clever ways of working around this problem so that we can still assign a similar assignment of pixels to a large number of threads. The thread assignment code we present in Listings 5.2 first calculates the thread's global id in lines 6-8, which is a number between 0 and $b \cdot b - 1$. Then it finds the corresponding pixel position that number would have on a two dimensional array of size bxb in lines 10-11. Each thread is then responsible for doing calculations concerning the pixel located at that position.

Although using one thread per pixel does parallelize the code a lot more, it does not necessarily mean that it is always the best choice. The code used for assigning one thread per pixel requires a lot more operations than the code used for assigning a thread for each row, and this needs to be taken into account. It depends on how much the actual computation benefits from such parallelism and whether

this performance gain is enough to hide the overhead cost of the pixel-per-thread assignment.

5.3.2 Transformation Kernels

Several different kernel functions are implemented to experimentally find the combination of kernels that give the highest performance:

- CylinderTransformKernel$\langle\langle\langle$ ... $\rangle\rangle\rangle$(...)
- CylinderTransformKernel2$\langle\langle\langle$ ... $\rangle\rangle\rangle$(...)
- SphereTransformKernel$\langle\langle\langle$ ... $\rangle\rangle\rangle$(...)
- SphereTransformKernel2$\langle\langle\langle$ ... $\rangle\rangle\rangle$(...)
- CombinedTransformKernel$\langle\langle\langle$ ... $\rangle\rangle\rangle$(...)
- CombinedTransformKernel2$\langle\langle\langle$ ... $\rangle\rangle\rangle$(...)

Two versions of the cylinder transformation function and the sphere transformation function are made. In the first version ("number 1" kernels), each thread is assigned to calculate the values of an entire row of pixels, while in the second version ("number 2" kernels), each thread is assigned to a single pixel in the image. These thread assignment patterns are further described in Section 5.3.1 and Listings 5.1 and 5.2. Two versions of a combined kernel were also created. The combined kernel does both the sphere and cylinder transformations in one go, and the two different versions correspond to our two different pixel-thread assignment patterns. All six kernels will be tested under different conditions to find the best performing kernel function, or combination of kernel functions.

Each kernel receives the same input. These include a list of variables needed for the transformation, such as the positions of the projector and the viewer, an array of pixels representing the input image, and an array of pixels corresponding to the output image. These values must be located in device memory for the kernels to be able to use them as input and output variables. The position of the viewer and projector are located in host memory, and must be copied over to device memory prior to each kernel call. This is done by first allocating enough space for the variables in device memory by calling the *cudaMalloc(...)* function, and then calling the *cudaMemcpy(...)* function to copy data from host memory to device memory. It is not necessary to do the same for the input image, since the image is generated by the graphic card, thus already located in device memory. The input and output images only need to be placed in special buffer objects that can be accessed by both CUDA and OpenGL. The kernels and OpenGL functions can then freely exchange data through these buffer objects without having to do any memory transfers to or from host memory.

The number of threads per thread block and the number of thread blocks per grid also need to be specified when calling a kernel. This is done within the three angle brackets located next to the name of the kernel function. For the kernel functions

that follow the *one row of pixels per thread* pattern, we specify the grid dimension to be $(n, 1, 1)$, making it a one dimensional array containing n thread blocks, and the block dimension to be $(w/n, 1, 1)$, where w is the number of pixels corresponding to the width of the image. We set the value of n based on the width of the input image and how many threads per thread block the GPU can handle. When a kernel that is using the thread assignment pattern displayed in Listing 5.1 is called with these specifications, each thread is given the task of computing the values of w pixels.

For the kernel functions that follow the *one pixel per thread* pattern, the specifications for the dimensions are dependent on how many threads per thread block the GPU can handle and the maximum acceptable number of thread blocks per grid. We also need to specify the block dimension to be a two dimensional array. This means that if the maximum allowed dimension for the thread block is $(t, t, 1)$ and the resolution of the image is *b*x*b* pixels, the dimension of the grid (which is the number of thread blocks necessary) is given by $(\frac{b \cdot b}{t \cdot t}, 1, 1)$. For example if the size of the image is 512x512 pixels and the maximum thread block dimension the device can handle is $(16, 16, 1)$, we need to set the grid dimension to $(1024, 1, 1)$ to be able to have one thread per pixel. No changes need to be made to the code inside the kernel; as we can see in Listings 5.2 it will automatically assign each thread to its corresponding pixel based on the dimensions of the image and the thread block size. The kernels that only deal with one transformation method are naturally called with only half as many threads in the y dimension of the thread block. If the kernel is only to be working on a 512x256 array then we can specify the thread block dimension to be $(16, 8, 1)$ and grid dimension to be $(1024, 1, 1)$.

The full source code for the *CylinderTransformKernel2* and *SphereTransformKernel2*, and how they are called from the host can be seen in Appendix B.1.

5.4 Vertex Transformation Method

Here we describe the implementation of the vertex transformation method described in Section 4.2. We write the code in OpenGL Shader Language (GLSL), since we are working directly on the programmable shaders in the graphics pipeline. The implementation very closely follows the pseudo code presented in Listing 4.3, except for the fact that the operations on each of the vertices are run in parallel. Unlike the implementation of the pixel transformation method described in Section 5.3, this implementation has no control over the number of threads that are used to perform the operations. All decisions regarding the number of threads, as well as any type of hierarchy, is handled by the device and cannot be altered by the program.

5.4.1 Defining the Shaders

The two programmable parts of the graphics pipeline that must be defined for this implementation are the vertex shader and the fragment shader. The vertex shader is responsible for doing all calculations on the vertices in the scene, while the fragment shader (also referred to as the pixel shader), is responsible for doing all calculations on a pixel by pixel level.

Since this method revolves around transforming the projection by finding new projection positions for each vertex in the scene, all of the code for the transformation is written in the vertex shader. The fragment shader does some pixel-by-pixel shading and lighting of the scene based on the information it receives from the vertex shader, but does not play any part in the actual transformation process.

To be able to use these shaders for vertex and pixel manipulation, the program needs to be able to load and read the shader code, and then bind the vertex and fragment shaders to the process of rendering the image. This will let the shader programs take over the responsibility of calculating the projection of the vertices and values the pixels. Input arguments, such as the positions of the viewer and the projector, need to be passed to the shader programs through special function calls before the rendering takes place. This works in somewhat the same way as how input arguments are passed for the CUDA transform kernels in Section 5.3.2. The full vertex and fragment shader code used in this transformation are displayed in Appendix B.2.

5.4.2 Finding The Projection Coordinates

The final output of the vertex shader defines the projection of each vertex in the scene, and provides the coordinates that they should have when displayed on the screen or a projection surface. This is usually done using a projection matrix, a model view matrix, and the positions of the vertices in the scene. For each vertex its projection point is calculated in the following way:

```
1 void main(void)
2 {
3     gl_Position = gl_ProjectionMatrix
4                 * gl_ModelViewMatrix * gl_Vertex;
5 }
```

Listing 5.3: Finding the projection coordinates.

The variable *gl_Vertex* is a vector holding the position of the current vertex, and the vector *gl_Position* is the output of the vertex shader and defines the screen/surface coordinates of this vertex. These are the coordinates needed for projection. Both the model view matrix and the projection matrix are defined prior to running the shader, and they are received and stored in the variables *gl_ModelViewMatrix* and *gl_ProjectionMatrix*. The model view matrix defines the coordinate system that is used to place and orient the objects in the scene, and also to combine the geometric transformations of the objects with the transformation to this coordinate system. The projection matrix decides how the scene is to be projected onto the screen. For example, if it should be an orthogonal projection or a type of perspective projection.

Since matrices can only describe linear transformations, it is not possible to define a projection matrix that can create an accurate and mathematically correct transformation of the scene intended for a curved surface. This is why we need to write specific vertex shader code that can define the projection of the vertices based on where they are to be displayed on the curved surface. One possibility is to first do the transformation described in Listings 5.3, and then make changes to the *gl_Position* vector based on how they will be projected onto the curved surface. But the method we describe in Section 4.2 is able to still calculate the correct projection for each vertex without doing this transformation first. We are able to avoid using the projection matrix altogether, thus avoiding the overhead of the matrix-matrix multiplication operation between the model view and projection matrix, as we can see below:

```
1  uniform vec3 P;
2  uniform vec3 COP;
3  void main(void)
4  {
5      vec4 vertPos = gl_ModelViewMatrix * gl_Vertex;
6      float xb = vertPos.x;
7      float yb = vertPos.y;
8      float zb = vertPos.z;
9
10     // Transformation code to find (xp, yp zp)
11     // based on P, COP and (xb, yb, zb)
12     //...
13
14     gl_Position.x = xp;
15     gl_Position.y = yp;
16     gl_Position.z = zp;
17 }
```

Listing 5.4: Finding the projection coordinates without the projection matrix.

The point *vertPos*, which we get from this equation, is the position of the vertex in the coordinate system defined by the model view matrix. This point also corresponds to the vertex position (x_b, y_b, z_b) described in Figures 4.7 and 4.8. From this point on, the implementation closely follows the pseudo code in Listings 4.3 to find the projection coordinates (x_p, y_p, z_p) for each vertex.

5.4.3 Adding New Vertices

As mentioned earlier in Section 4.2, the vertex transformation does not provide a pixel perfect transformation of the scene. The reason for this is that we are only calculating the projection points of the vertices. But during the rasterization stage of the graphics pipeline, where the vertex representation is converted to a pixel representation, the lines between the vertices are drawn by interpolating between

their positions, meaning that only straight lines can be drawn between each pair of vertices. The outcome of this limitation is that this method will give a very poor transformation of scenes where the vertices are far apart from each other.

One possible solution to this problem is to insert new vertices into the scene where they seem appropriate. We do this by inserting n number of new vertices between two vertices that would originally form a line, thus making it n consecutive lines, each $1/n$ the size as the original. This will improve the accuracy of where the pixels are placed during the rasterization, since there are now a lot more vertices representing the same area as before. The number of new vertices we need to add between a pair of two old vertices should be dependent on three things:

1. The original distance between the two vertices.
2. The distance the vertices have to the COP.
3. The resolution at which the scene will be rendered in.

The idea is to prevent the perceived distances between two vertices from becoming so large that it will affect the visual outcome of the transformation. If we move the object closer to the COP it will appear larger, which increases the perceived distance between its vertices. This also happens when the scene is rendered to a much bigger resolution, making the object appear larger and thus pushing the vertices within the object further apart. For these reasons, it might be necessary to add additional vertices to the object to give the transformed image a much smoother appearance. Naturally, when the object is further away from the viewer, or the image resolutions is very small, it may not be necessary to add as many new vertices to get the same effect. We implement the following method to determine how many new vertices we should add between a pair of vertices in an object:

```
1  void constrct_object(...)
2  {
3      V_max = 16; //maximum number of new vertices we
4                  //want to add between two old vertices
5
6      for(int i=0; i<num_vertices; i++)
7      {
8          // draw a vertex ...
9
10         // for each pair of vertices we do the following:
11             if(distance_to_COP <= minimum_distance)
12             {
13                 num_new_vertices = V_max;
14             }
15
16             else
17             {
18                 num_new_vertices = floor(scaleDetail * vert_distance
19                                          * image_width  / distance);
```

```
20              }
21
22              // draw the new vertices...
23          }
24      }
```

Listing 5.5: Inserting additional vertices.

The number of new vertices we add between a pair of vertices is given by: The original distance between the two vertices multiplied by the resolution of the image, and then divided by the average distance to the COP. It is also multiplied by a constant value that has the purpose of scaling the number, and thus controlling the level of detail and smoothness in the deformation of the image.

Although this is a very simple method, it can soon become very costly in terms of performance. The new vertices cannot be added from the vertex shader, so they must be added before starting the transformation. This means that the process of adding new vertices is handled purely on the CPU, so for huge models consisting of a large number of vertices this may cause a performance issue. However, highly detailed models that already consist of a large number of vertices will not necessarily need new additional vertices for the transformed image to appear correctly.

5.5 Stereoscopic Rendering

Although projecting the transformed image on a curved surface already provides a certain amount of depth perception to the viewer, we can enhance the feeling of depth by rendering the transformed image for stereoscopic viewing. We do this by creating a stereo pair of the scene, as described in Section 2.2.2.

When only rendering a single image for regular monoscopic viewing, the position of the COP, as seen in Figures 4.1 - 4.3, corresponds to the reference point in Figure 2.4. When creating a stereo pair, we need to render two images with their respective COPs corresponding to the two eyes in Figure 2.4. We render the first image where we displace the COP a distance d to the left of the reference point, and we render the second image with the COP displaced a distance d to the right. As the viewing direction of the COP is always parallel to the z-axis in our implementation, we only shift the position of the COP in the x direction, as we can see in Listings 5.6 and 5.7. The viewing directions of the displaced COPs are still parallel to the z-axis of the coordinate system.

We implement two types of stereoscopic rendering of the projection, known as interlaced stereo rendering, and quad-buffered stereo rendering. They both fall under the classification of active stereo, which we discussed in Section 2.2.3, and require special shutter glasses to be viewed. The main difference between these two types is that interlaced stereo rendering combines the two images of the stereo pair into one single image for each frame, while quad-buffered stereo rendering shows both

images one after another for each frame.

5.5.1 Interlaced Stereo Rendering

Interlaced stereo rendering is done by rendering the first image of the stereo pair by only drawing every odd numbered line of the image, then rendering the second image of the stereo pair by only drawing the even numbered lines of the image, and finally interweaving the two images into one image.

We implement this in our code by creating a stencil mask where every other row has the value '1' and the remaining rows are '0'. Then, by performing simple boolean operations between the stencil mask and the stereo pair images, we render the two images as shown in Figure 5.2. The final image consists of all the even lines from one image and all the odd lines from the other image. Below we provide some pseudocode of the rendering procedure.

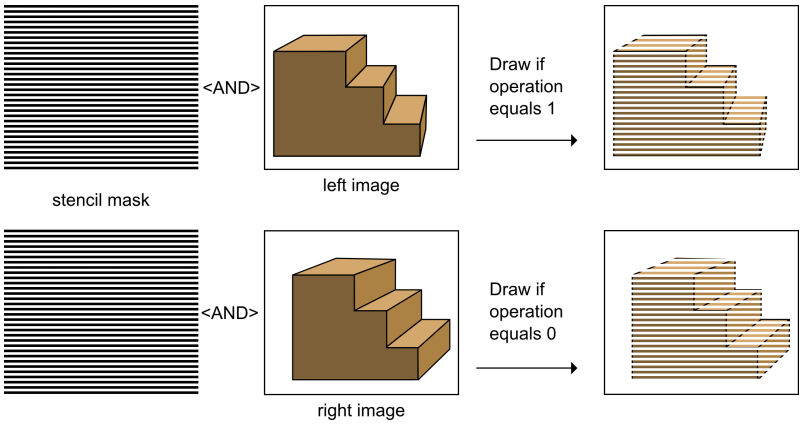

Figure 5.2: Rendering the stereo pair using a stencil mask.

```
1  void Render_Stereo()
2  {
3      float eye_seperation = 0.05;
4
5      move_COP(-eye_seperation, 0, 0);
6      draw(); // (left image)
7      transform();
8
9      move_COP(eye_seperation, 0, 0);
10     draw(); // (right image)
11     transform();
12
13     glEnable(GL_STENCIL_TEST);
14     glStencilFunc(GL_NOTEQUAL, 1, 1);
15         draw_left_image();
16     glStencilFunc(GL_EQUAL, 1, 1);
17         draw_right_image();
```

```
18      glDisable(GL_STENCIL_TEST);
19  }
```

Listing 5.6: Interlaced stereo rendering.

The stereo rendering procedure is slightly different for both our transformation methods. The pseudocode in Listings 5.6 is similar to how we render the images when performing the pixel transformation method. For this method, the draw function is called a total of four times when rendering a single frame. First we draw the scene twice (once for each eye), then perform transformations on both images before finally rendering each of them again while doing a stencil test. For the vertex transformation method, the draw function is only performed twice since we are already rending the transformed image for each eye on the first go.

5.5.2 Quad-buffered Stereo Rendering

The usual way a scene is rendered in OpenGL is through double buffering. This means that the rendered frames are being drawn consecutively on two different buffers, such that while the first frame is being displayed by the first buffer, the second frame is being rendered to the second buffer. Then the third frame is rendered to the first buffer while the second buffer is being displayed, and so on. Quad-buffered rendering utilizes four buffers, providing double buffering for each eye. The two buffers corresponding to one eye are swapped in sync with the two buffers corresponding to the other eye. In the pseudocode below we see that a separate buffer is provided for each eye. For the next frame two new buffers will be used to render the images while the current buffers are displayed on screen.

```
1  void Render_Stereo()
2  {
3      float eye_seperation = 0.05;
4
5      move_COP(-eye_seperation, 0, 0);
6      glDrawBuffer(GL_BACK_LEFT);
7          draw(); //left image
8          transform_image();
9          draw();
10
11     move_COP(eye_seperation, 0, 0);
12     glDrawBuffer(GL_BACK_RIGHT);
13         draw(); //right image
14         transform_image();
15         draw();
16 }
```

Listing 5.7: Quad-buffered stereo rendering.

It should also be noted here that the pseudo code in Listings 5.7 corresponds to the pixel transformation method. Even though there is no stencil test here, the pixel transformation method still require four draw calls when rendering in stereo, since the transformation and drawing procedures are separated. The vertex transformation can still be rendered with just two.

This rendering method is much more effective than the previous one, as there is no need for doing any stencil testing. However, quad-buffering requires special support by the graphic card, and is not available on most cards. To be able to display a stereoscopic image correctly through quad-buffered rendering, we require special displays or projectors that can display the image at a much higher frequency (120Hz) than standard displays, in addition to shutter glasses that can synchronize with this frequency.

CHAPTER 6

Benchmarking and Results

This chapter describes the benchmarking stage of both transformation methods. The different testing routines, as well as the obtained results, will be presented and discussed thoroughly in the following sections.

Firstly, in Section 6.1 the testing environment which we use to measure the performance and various other results is described. Secondly, Section 6.2 presents the results and discussion regarding the benchmarking of the pixel transformation method. Finally, in Section 6.3 we display the results obtained from testing vertex transformation method, and compare the outcome of both transformation methods for various problem areas.

6.1 Testing Environment

As the two transformation methods we developed during the course of this thesis are both massively parallel methods, it was important that we tested them on several different graphics cards. However, since the pixel transformation code was written in CUDA, we were limited to only testing on NVIDIA GPUs. These include the GEFORCE GT 240M, the GEFORCE GTX 280, the QUADRO 5800, and the TESLA C2050, which was also the main graphics card present in our benchmarking system. The system housing the GEFORCE GT240M was used mostly during the stages of implementation and debugging. The QUADRO card was used for testing quad-buffered stereo rendering, as it was the only type of NVIDIA card that supported this rendering mode. Detailed specifications of our main benchmarking system can be seen in Table 6.1.

Hardware	
CPU	Intel Core 2 Quad
CPU clockspeed	2.83 GHz
Memory size	4 GB
Graphics card #1	NVIDIA TESLA C2050
Graphics card #1 memory	4 GB
Graphics card #2	NVIDIA GEFORCE GTX280
Graphics card #2 memory	1 GB

Software	
OS	Windows 7
Visual Studio ver.	2008, with SP1
NVIDIA graphics driver ver.	263.06
CUDA toolkit ver.	3.2

Table 6.1: Specifications of the benchmarking system.

All benchmarking results which we present and discuss in the following section are from testing the code on this system, using the TESLA C2050 GPU.

6.2 Pixel Transformation Results

Here we present and discuss the performance and visual results of testing the pixel transformation method. We tested both the sequential and parallel versions, including all the kernels described in Section 5.3.2.

6.2.1 Deformation of The Image

When we transform the image, we are deforming it in a way that is exactly the opposite to the geometry of the curved surface. Basically, we are bending the image in the reverse shape of the projection surface, so that when we project it, the shape of the surface and the shape of the image cancel each other out, and the scene appears correctly. In Figure 6.1, we show how a scene looks before and after the transformation has taken place. When rendering this scene, the position of P was set to be at $(0, 0, 2.5)$, and the position of the COP at $(0, 0, 1)$. In other words, this scene is meant to be projected from a position right in front of the projection surface. Furthermore, the viewer should be standing right between the projector and the surface. If the viewer moves to the left or right, we would need to project an image with different COP coordinates. Similarly, repositioning the projector also means that we will need to change the P coordinates and render a new image for the projection to appear correctly. The grey area at the top half of the image corresponds to the part of the image that will never fall on the curved projection surface.

The top half of the image has been deformed into a circular shape to fit the spherical part of the projection surface. When viewing on a regular display or a flat surface,

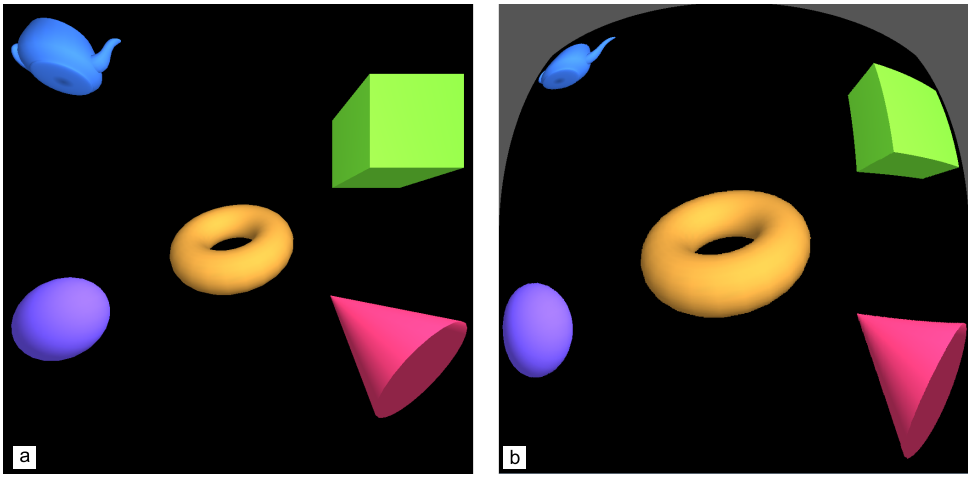

Figure 6.1: (a) Original image before transformation, (b) transformed image.

it will appear as if this part of the image forms a convex quarter-sphere, as we can see in Figure 6.1b. Since the intended projection surface has the shape of a concave quarter-sphere, it will cause this part of the image in Figure 6.1b to appear as it does in Figure 6.1a when projected onto it. The same applies for the lower half of the image, which appears as if it is being displayed on top of a convex cylinder in Figure 6.1b. The deformation of the lower half is not as clear in this image, but we can see from the object close to the lower edges that there are several changes there as well. The torus in the middle of the image remains mostly the same. The only visible difference there is that it appears larger, which is also explained by the convex deformation of the image.

6.2.2 Transformation Kernels

In Section 5.3.2 we presented six different transformation kernels we developed to find the most efficient way of performing the transformation of the scene. We tested these kernels by running each of them 1000 times for different dimensions of the input image. Then we calculated the average time each kernel took for each of the image sizes. The NVIDIA Compute Visual Profiler was used to record the run time of the kernels functions. Since the kernels perform the same calculation on each pixel regardless of their content, it does not matter what is actually on the input image we send to the kernels. As far as the execution time is concerned, the only deciding factor is the number of pixels in the image, as this affects the total number of operations that must be performed. The actual pixel values in the input image have no bearing on the run time of the kernels.

Individual Kernel Results

The results of the individual kernels are displayed in Figure 6.2 and Table 6.2. Note that the kernel functions post-fixed by the number 1 follow the row-per-thread pattern, while the ones post-fixed by the number 2 compute one pixel per thread, as described in Section 5.3.2.

What we can immediately notice from the results, is that the pixel-per-thread assigning pattern seems to work far better than the row-per-thread pattern for all problem sizes. If we had tested on lower image dimensions than 128x128, the overhead of assigning one thread per pixel as we showed in Listings 5.2, might have been large enough to hinder the performance of the "number 2" kernels.

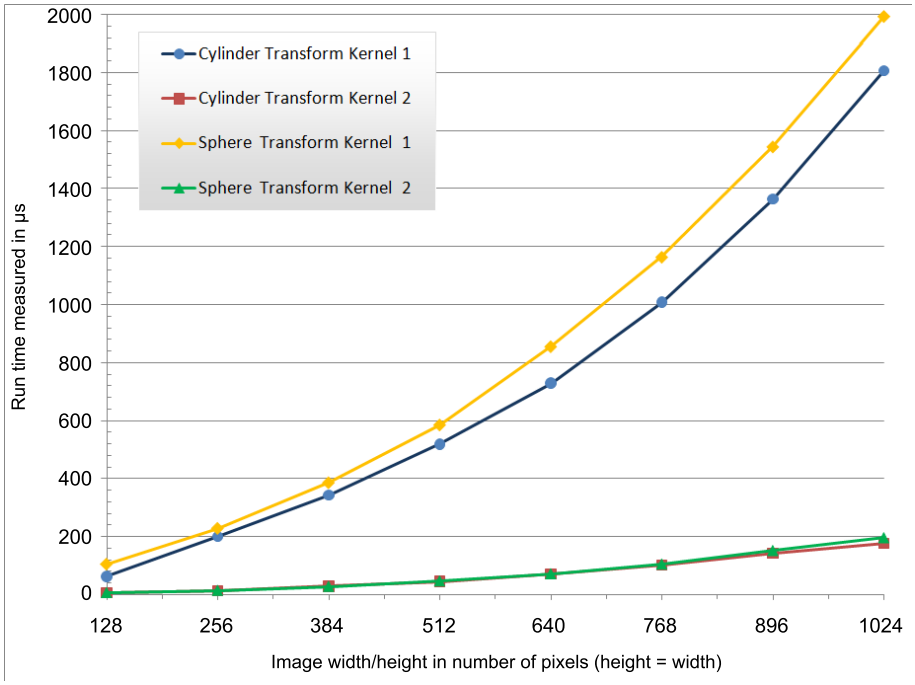

Figure 6.2: Execution time of the single transformation kernels, for different image sizes.

	CylinderT..1	**CylinderT..2**	**SphereT..1**	**SphereT..2**
128x128	63.07μs	6.3μs	105.81μs	6.81μs
256x256	199.93μs	14.28μs	229.24μs	14.65μs
384x384	343.92μs	28.97μs	386.81μs	28.35μs
512x512	518.92μs	45.3μs	586.24μs	46.91μs
640x640	729.45μs	71.49μs	856.37μs	72.05μs
768x768	1008.15μs	101.71μs	1165.27μs	105.47μs
896x896	1363.94μs	142.19μs	1546.23μs	152.187μs
1024x1024	1805.74μs	176.35μs	1993.06μs	196.15μs

Table 6.2: Execution time of the single transformation kernels, for different image sizes.

As we increase the problem size, the kernels that are using the pixel-per-thread pattern seem to gain a tremendous speed advantage over their respective counterparts.

For the sphere transformation kernel, the reason for this is mainly behind calculating the value of z. It consists of many operations, and needs to be recalculated

for each pixel. Since it is by far the most expensive step in the algorithm, we gain a very large speedup in parallelizing this process. For the *CylinderTransformKernel2*, the speedup over its corresponding row-per-thread equivalent is not quite as big, but still large enough to make a big difference. This is because recalculating the value of z, which is also the most expensive step, only needs to be done once per column of pixels in the image, and not per pixel. Using the pixel-per-thread pattern, this step is performed on a per-pixel basis instead, so this alone really should not provide much of a speedup. We believe that the speed advantage gained by the *CylinderTransformKernel2* is due to parallelizing the computations of all other variables such as x, y, x_c and y_c. The combined cost of these computational steps is large enough such that, when parallelized, it hides the overhead of assigning threads in the pixel-per-thread pattern. In general, the "number 2" kernels seemed to be at least 10 times faster than their equivalent "number 1" kernels.

Combined Kernel Results

In Table 6.3 we see the results of testing the combined kernels. By combining the cylinder and sphere transformation kernels, it is possible to combine similar steps from both functions into a single step, and thereby reduce the number of operations that are performed. The combined kernels used close to only half the number of registers that their respective individual kernels used together, in addition to having a much higher level of warp occupancy. However, there still did not seem to be much of a speedup when comparing the run times. For both types of thread assigning patterns, the individual transformation functions together seemed to perform a little bit better than their combined version. This may be due to the branching in the code introduced by fusing together both transformation procedures. Since the function needs to perform different operations, and relies on different variables depending on which part of the image is being transformed, **if** and **else** statements are used to guide the execution of the code. This may hamper the control flow of the kernel and force parts of the code into serial execution.

	CombinedTransformKer..1	**CombinedTransformKer..2**
128x128	214.78μs	10.09μs
256x256	521.53μs	26.99μs
384x384	855.38μs	56.07μs
512x512	1296.33μs	93.13μs
640x640	1814.79μs	147.28μs
768x768	2481.54μs	212.25μs
896x896	3333.01μs	299.36μs
1024x1024	4262.59μs	378.91μs

Table 6.3: Execution time of the combined transformation kernels, for different image sizes.

Based on these results, we conclude that the *CylinderTransformKernel2* and *SphereTransformKernel2* are definitely the best choices for transforming an image that is

within relevant dimensions. These kernels will fail for very large problem sizes, where the number of required threads is larger than the number of threads the GPU is capable of providing. With the Tesla C2050, we were able to transform a 4096x4096 image using these kernels, but for all other graphics cards the maximum possible image size was much smaller.

6.2.3 Sequential vs Parallel

We tested the parallel implementation of the transformation method using the individual "number 2" kernels against the sequential implementation of the same method to see how much of a speedup the program as a whole gains due to parallelization. We executed the entire code, which includes drawing an animated scene, lighting and shading of the models in the scene, and stereoscopic rendering of the transformed image. The animated scene consisted of an orange teapot bouncing around within a cubic domain. The shading was handled by OpenGL calls and covered basic ambient and diffuse shading of the object with regards to a single light source. For stereoscopic projection of the scene, we used the interlaced stereo rendering method described in Section 5.5.1. We tested for the same problem sizes as we did when testing the kernel functions in Section 6.2.2. For each image dimension, we ran the sequential and parallel versions of the program for two minutes and then calculated the average number of frames that were rendered per second. The results can be seen in Figure 6.3 and Table 6.4.

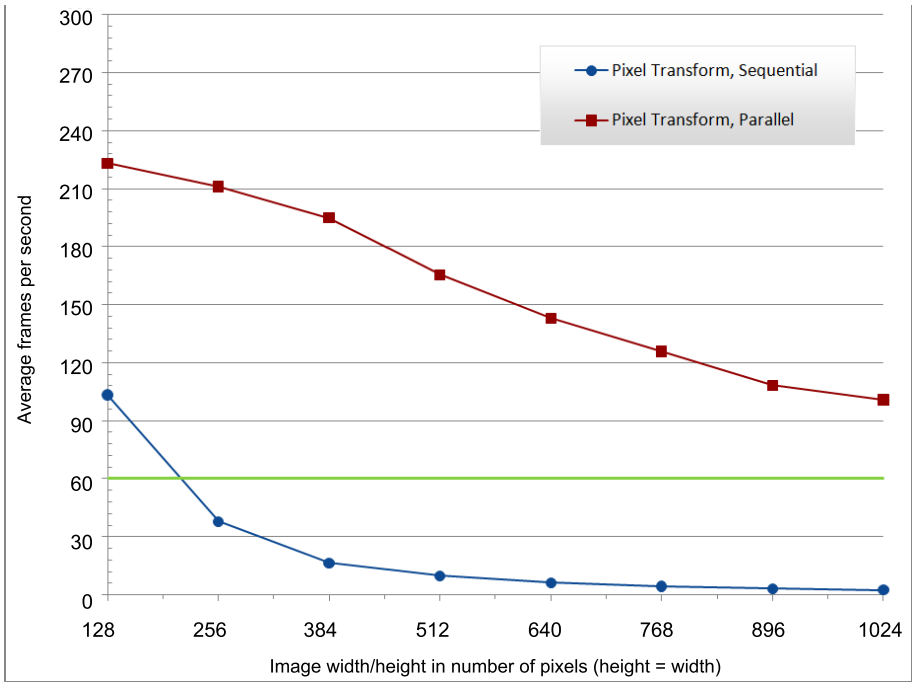

Figure 6.3: Average number of frames per second for the sequential and parallel pixel transformation code, for different image sizes.

	Sequential Transform	**Parallel Transform**	**Speedup**
128x128	103.27	270.63	2.62
256x256	37.94	154.3	4.07
384x384	16.41	98.86	6.02
512x512	9.73	66.7	6.86
640x640	6.21	43.25	6.96
768x768	4.42	31.99	7.24
896x896	3.22	23.95	7.44
1024x1024	2.49	21.8	8.76

Table 6.4: Average number of frames per second for the sequential and parallel pixel transformation code, and the speedup of the parallel code over the sequential one, for different image sizes.

The green horizontal line at 60fps marks the maximum frame refresh rate that most displays and projectors have. Any animation above the rate of 60fps gets clamped to this frequency when actually viewing the projection on the display. We disabled this functionality to be able to see the true number of frames rendered per second.

Comparing Average Frames per Second

The average frame rate of the sequential code immediately fell below 60fps, even for small problem sizes, while for the parallel code it was able to stay above this rate, even for dimensions as large as 1024x1024 pixels. As we increased the resolution further past 1024x1024, the parallel code also eventually fell below this line, but not until we reached a fairly large image size. However, the frame rate of 60fps is only the refresh rate of the display, and going below does not always imply loss of fluidity in the animation. It also depends on the amount of detail in the scene, the variations in the lighting, and how fast things are moving around. A frame rate of 24 is the de facto standard for animated motion pictures; for animations as simple as the ones we have used in our testing, the required frame rate for the human eye to perceive fluid motion is far less. Even for the problem size of 2048x2048, the parallel version was able to render at an average rate of 36,82 frames per second, which was enough to provide a visually pleasing and fluid 3D animation of our test scene. The sequential code on the other hand, was unable to obtain the same result even for much smaller images.

When we disabled stereoscopic rendering, both the parallel and sequential versions of the program performed almost twice as fast for all dimensions of the image. But even at this rate, the sequential code was just barely able to produce a fluid animation for images of size 512x512. The parallel version, however, was able to provide a frame rate of 20 frames per second for the image size of 4096x4096. The part of the rendered animation that was visible on our display (which only had a resolution of 1920x1200) seemed to have an acceptable level of smoothness and speed to still be visually pleasing for the human eye.

Speedup of The Transformation Functions

For the problem size of 1024x1024 pixels, the parallel version ran approximately 40 times as fast as the sequential version. This may not seem that impressive when considering the number of pure computational operations within the transformation that is parallelized, but this "low" speedup is actually due to all the parts of the program that are forced to run sequentially for both versions. This includes drawing the models according to their updated coordinates, updating the model view matrix, updating counters and calculating statistics, performing a stencil test for stereo rendering, etc. To determine the speedup purely from parallelizing the transformations, we measured the combined runtime of both sequential transformation functions, and compared them to the combined runtime of both parallel transformation functions from Table 6.2. For the image size of 512x512 the cylinder and sphere transformation kernels together spent a total average of 92,21μs. Compared to the run time of 45,28ms measured for the sequential transformation, the speedup is 491,05.

6.3 Vertex Transformation Results

Here we present and discuss the performance and visual results obtained from testing the vertex transformation method. We also compared it to the parallel pixel transformation test results presented in Section 6.2.

6.3.1 Visual results

As we covered earlier in Section 5.4.3, the vertex transformation method does not provide a pixel perfect transformation, as it only transforms the projection of the vertices in the scene. The pixels values of the image are determined by interpolating between the transformed vertices. This resulted in the transformed image always consisting of straight lines. The overall impression it provided through such a result, was that the objects in the image were being skewed in certain directions, but not deformed in ways that correspond to semi-cylindrical or hemispherical shapes. This was especially easy to notice when performing the transformation on models consisting of few vertices that were all far apart from each other. However, as the number of vertices in a model increased, or the distances between the vertices decreased, the quality of the transformed image increased accordingly.

The question is, how many extra vertices need to be added to a model so that once it is transformed, the resulting image will be identical to the one obtained from the pixel perfect pixel transformation method? The answer to this, as we have discussed in Section 5.4.3, depends on many different factors. We performed testing for a simple case, where we placed a large cube in the part of the scene where it is subject to the most deformation due to the transformation. The cube originally consisted of 8 vertices, which were all of equal length apart from each other. By gradually adding additional vertices to the cube we increased the quality of the transformed image. In Figure 6.4 we display the visual results of the transformation for different number of additional vertices.

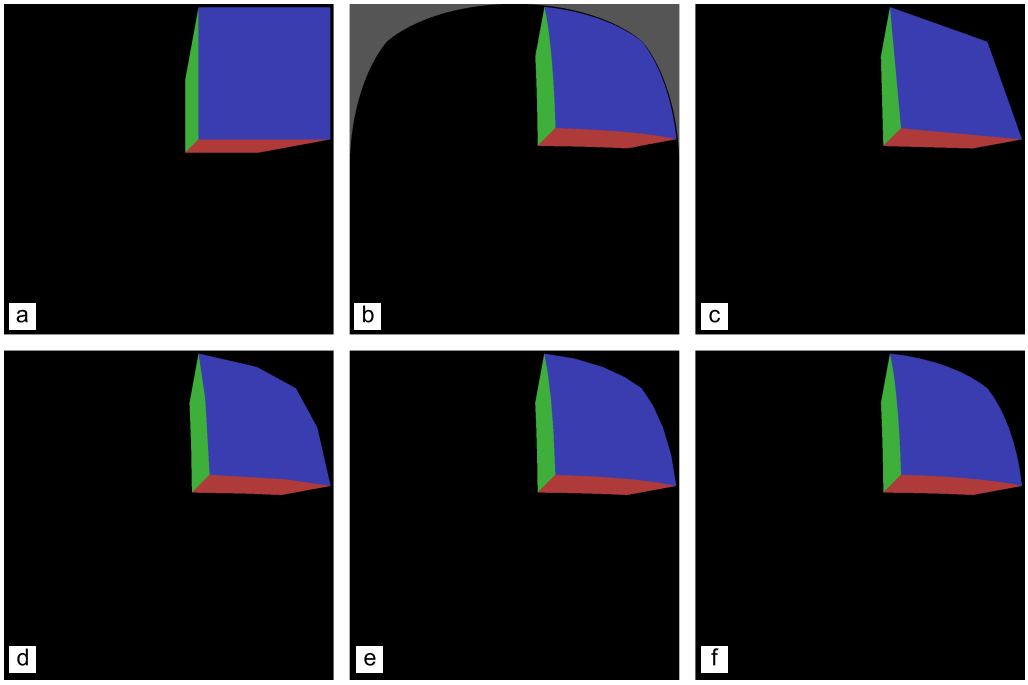

Figure 6.4: (a) Original image before transformation, (b) pixel transformation, (c)-(f) vertex transformation with 0, 1, 2, 4 and 8 additional vertices between each pair of vertices that form a line.

In Figure 6.4a, we see the original image before any transformation has been performed, Figure 6.4b shows the same scene transformed using the pixel transformation method, and Figure 6.4c shows the result of the vertex transformation. Here we see that all the lines are straight when instead they should be curved in accordance to the formation of the hemispherical surface, as they are in Figure 6.4b. Figures 6.4d - 6.4f show how the result gradually improved as we drew additional vertices between each pair of vertices that form a line. In the last picture, the result looks identical to the pixel transformation result, but it required a total of 96 additional vertices to be inserted into the cube. This is not such a bad trade-off considering the number of pixels that needed to be transformed by the pixel transformation method. The resolution of the images rendered in Figure 6.4 is 1024x1024, meaning that the pixel transformation method needed to perform transformation operations on a total of 1048576 pixels. The vertex transformation method only needs to transform 104 vertices to get the same result.

6.3.2 Image vs Vertex Transform

The best possible way to compare the results of both methods was not very straightforward. The pixel transformation method relies heavily on the number of pixels in the image, while the vertex transformation is heavily dependent on the number of vertices in the scene. For this reason, it was necessary to do several different tests to be able to more accurately compare the performance of both transformation methods.

Fixed Number of Vertices

We performed the same test as the one described in Section 6.2 using the vertex transformation method. The test was run for 2 minutes for each of the different resolutions, with both interlaced stereo rendering, and lighting enabled. For now we let the number of vertices remain the same for all measurements. Since this transformation method relies on using shaders to produce the actual image, the lighting and shading calculations also needed to be done through shaders. For this, we implemented a simple pixel-by-pixel ambient and diffuse lighting method in the fragment shader, similar to the one performed by OpenGL for the pixel transformation. The test scene was an animation of a teapot bouncing around within a cubic domain. The results can be in Figure 6.5.

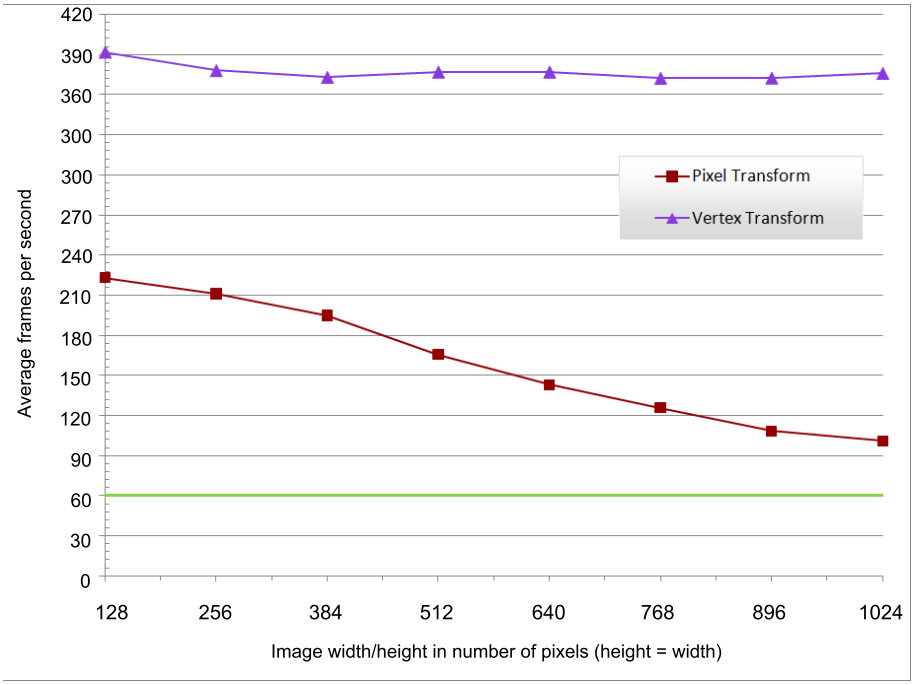

Figure 6.5: Average number of frames per second for the pixel transformation and vertex transformation code for different image sizes.

These results are not surprising at all, considering that the number of vertices remained the same throughout the entire test. For the vertex transformation method, the number of calculations performed is the same for the image size of 128x128 and the image size of 1024x1024. The visual results, on the other hand, can vary a lot depending on the resolution of the image. The reason for this is that as the size of the image increases, so does the perceived distance between the vertices projected onto the image. Due to the vertex transformation method always drawing straight lines between each pair of vertices, increased distance between these vertices can affect the smoothness of the rendered image. However, in Figure 6.6 we see that this is not the case for the teapot scene.

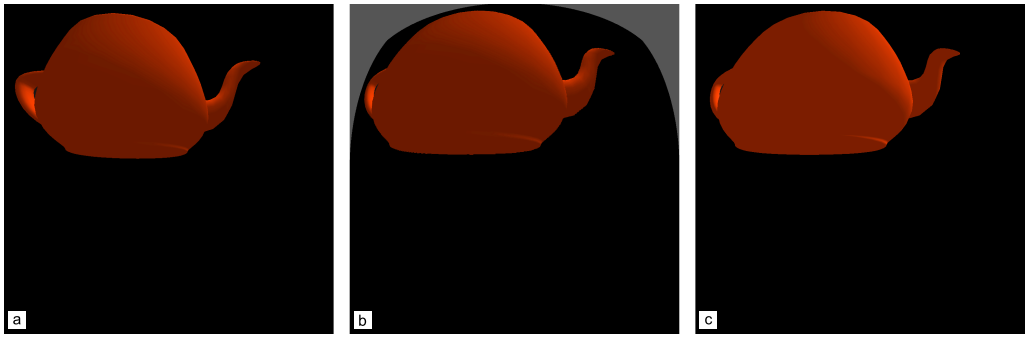

Figure 6.6: (a) Original image before transformation, (b) pixel transformation, (c) vertex transformation.

In Figure 6.6a we see the original image, Figure 6.6b shows the result of the pixel transformation, and the vertex transformation result is displayed in Figure 6.6c. The teapot has been moved very close to the viewer, and the scene is rendered to a resolution of 1024x1024. Both these factors affect the perceived distance between the vertices in the teapot, but we see that both transformations still give exactly the same results. Disregard the difference in the shading of the teapot, which is due to small differences in the way the lighting and shading is performed for both methods. The size and shape of both transformed images are exactly the same. This is due to the teapot consisting of a large number of vertices positioned very closely together, so even after increasing the resolution and scaling the teapot, the vertices were still not far enough apart to cause loss of correctness in the transformation. If we had increased the resolution by a lot more, then we might have noticed a difference, but for relevant problem sizes there did not seem to be any changes.

The fact that the vertex transformation method, in general, requires fewer calculations per vertex than the pixel transformation method does per pixel, should also be taken into consideration. Another important point is that the pixel transformation method needs to pre-render the original image before performing its calculations, while the vertex transformation method can perform its calculations based on the vertex data of the models in the scene. This might explain why the vertex transformation method performed much better even for small problem sizes like 128x128.

Increasing Number of Vertices

We tested the scalability of the vertex and pixel transformation methods for increasing number of vertices, with a fixed image resolution of 1024x1024 for both methods. The test scene consisted of a number of cubes with lighting and stereoscopic rendering enabled. Each cube consisted of 8 vertices, and as we were testing, we increased the number of vertices by adding more and more cubes into the scene. We increased the number of vertices by 4000 at a time by adding new cubes into the scene, and the tests were run for 2 minutes for each group of additional vertices. We added the cubes sequentially, so it should be taken into consideration that this might have affected the overall frame rate of both methods during this particular test. The results are displayed in Figure 6.7 and Table 6.5.

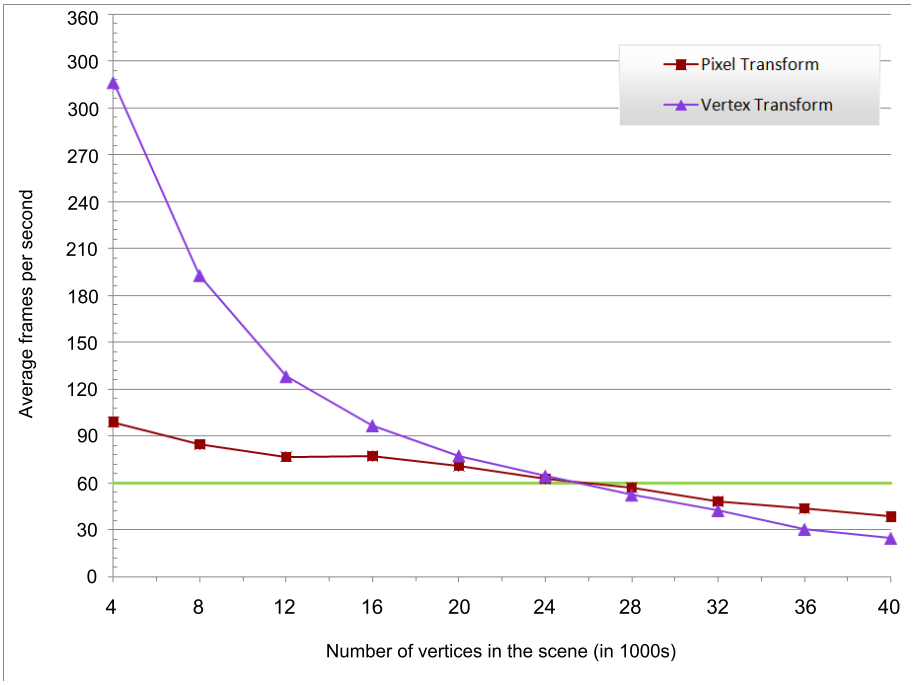

Figure 6.7: Average number of frames per second for the vertex transformation and pixel transformation code for different number of vertices.

	Vertex Transf...	**Pixel Transf...**	**Speedup**
4000	316.49	98.84	3.20
8000	192.88	84.59	2.28
12000	128.17	76.64	1.67
16000	96.7	77.19	1.25
20000	77.36	70.76	1.09
24000	64.48	62.43	1.03
28000	52.25	57.01	0.91
32000	42.34	48.02	0.88
36000	30.27	43.5	0.69
40000	24.5	38.37	0.63

Table 6.5: Average number of frames per second for the vertex transformation and pixel transformation code, and the speedup of the vertex transformation over the pixel transformation, for different number of vertices.

For a low number of vertices, the vertex transformation naturally performed a lot better than its counterpart, but as we went past 25000 vertices in the scene, the pixel transformation gradually gained the advantage. As we increased the number of vertices, the pixel transformation method lost speed very slowly. This is because,

the only time it is dependent on the number of vertices is for pre-rendering the original image, which is not very expensive. The vertex transformation's execution time, on the other hand, is highly dependent on the number of vertices, which explains the steep drop in performance. From these results, we can gain a certain notion of when it may be beneficial to use the different transformation methods, but of course the ultimate decision also depends on the visual aspect.

Dynamically Updating Number of Vertices

Finally, we tested the method we implemented for adding new vertices dynamically to the model based on different factors, as described in Section 5.4.3. We render a simple scene consisting of a number of floating cubes moving around in a cubic domain. For each frame, the code does calculations based on how far the cubes are from the viewer, the distance between the vertices of each cube, and the resolution of the image. Based on these calculations, it determines how many additional vertices to place between each pair of vertices that form a line on each of the cubes. Using this method for adding vertices, we compared the image and vertex transformation methods for different image resolutions once again. The difference this time was that increasing the image resolution also affected the execution time of the vertex transformation method, because of the responsive growth in the number of vertices. The results can be seen in Figure 6.8 and Table 6.6. The average frame rate is somewhat higher for both methods, since the scene we are rendering is a lot simpler than the teapot scene rendered for Figure 6.5.

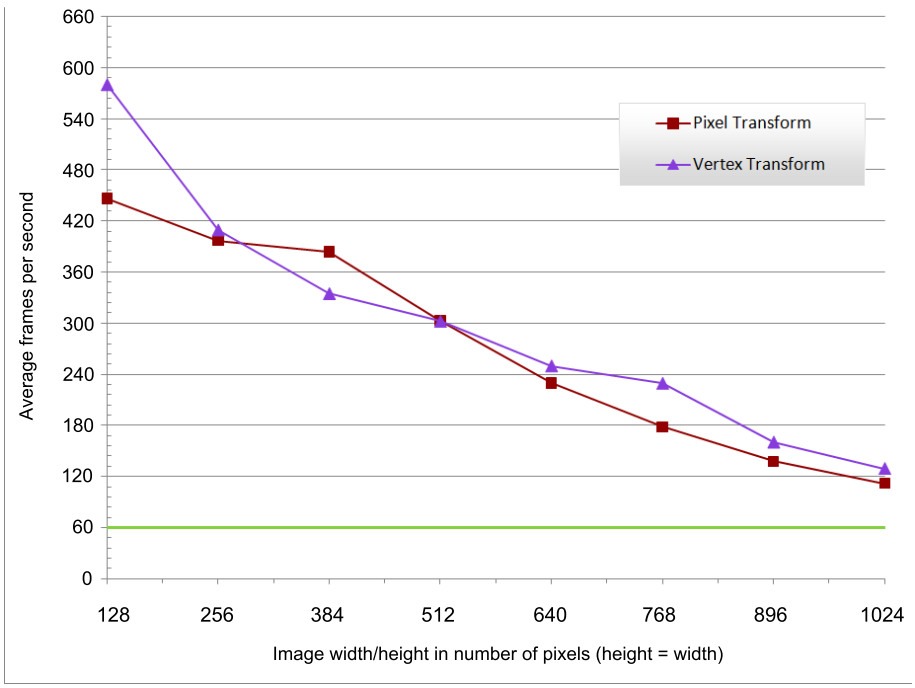

Figure 6.8: Average number of frames per second for the vertex transformation and pixel transformation code for image sizes, and with dynamic addition of vertices.

	Vertex Transf...	**Pixel Transf...**	**Speedup**
128x128	580.54	446.15	1.30
256x256	409.6	396.96	1.03
384x384	334.88	383.72	0.87
512x512	302.34	303.16	0.99
640x640	249.94	229.65	1.08
768x768	229.45	178.25	1.28
896x896	160.35	137.83	1.16
1024x1024	129.39	111.88	1.15

Table 6.6: Average number of frames per second for the vertex transformation and pixel transformation code for image sizes, and with dynamic addition of vertices.

Here, the average frame rates of both transformation methods seemed to have a more similar declining pattern as we increased the image resolution. When rendering at a resolution of 1024x1024, the vertex transformation method added up to 29 new vertices between each pair of vertices, for cubes that came really close to the viewer. Furthermore, it gave a very smooth and visually pleasing transformation.

Based on these results, the method for adding vertices dynamically seems to work quite well; however, there is a lot of overhead involved, as this operation must be done sequentially on the CPU. If a large number of objects are present in the scene, adding new vertices to all of them will most likely slow down the program a lot. Another issue is that it is not a simple task to add extra vertices to just any model. For cubes it is a pretty simple procedure, since we know that all the vertices are equally far apart; nonetheless for more complicated models, like the teapot, it may be too expensive to perform such an operation. As we showed in Figure 6.6, a model does not necessarily need additional vertices for the transformation to appear correctly, so the question of whether vertices should be added dynamically or not, relies on the amount of detail in the model itself.

In general, we can conclude from all the results presented here that, whether or not to use the pixel transformation method depends first and foremost on the number of vertices present in the scene. As we showed in Figure 6.7, after the total number of vertices passes a certain point, the pixel transformation method will always be the best choice. As for the vertex transformation, the choice depends on several factors. For scenes with a small to average number of vertices, the vertex transformation may be the best choice, but this is not a given. It depends on the resolution of the image that is being rendered, the distance the models in the scene have to the viewer, and details of each model. It also depends on whether it is possible to add extra vertices to the model, and the cost of doing so. Scenes consisting of a very large number of simple models would mostly benefit from the pixel transformation, since the scene may consist of an exceptionally high vertex count, or each model may be too simple to be transformed correctly by the vertex transformation method. An example of such a scene would be a scene filled with thousands of cubes such as the one showed

in Figure 6.4. For scenes consisting of few, but detailed models, such as the teapot, it would be more beneficial to use the vertex transformation. This will make the rendering a lot faster, as we can see in Figure 6.5, and the image resolution can still be scaled to a certain point without loss of quality from the transformation, as we show in Figure 6.6.

CHAPTER 7

Conclusions and Future Work

In this chapter we provide a brief summary of our work, the achieved results, and the experiences gained throughout this thesis. We also present the conclusions we have formed, and provide some interesting ideas for future work.

7.1 Summary

The goal of this thesis was to investigate two different methods for transforming an image, such that it can be projected onto a curved surface, and to develop parallel solutions to these methods. Our work included a study of various topics including virtual reality, curved projection surfaces, stereoscopy, and parallel computing on the GPU. This was followed by research into the two transformation methods, setting up equations based on the geometry of the curved surface, and deriving the necessary formulas required to perform the transformations. Massively parallel solutions were developed for both transformation methods, and implemented on the GPU. The pixel transformation method was implemented using the CUDA framework, and the vertex transformation method was implemented on the vertex and fragment shaders of the graphics pipeline. Various experiments were performed on both methods to test their performance and visual appeal, and the results were documented and discussed. The results of both transformations were also compared to each other, to gain a better understanding of the strengths and weaknesses of each method.

7.2 Conclusion

Based on the outcome of the different experiments, we conclude that the pixel and vertex transformation methods both have areas where they excel in, and areas where they perform very poorly. We believe that it is not possible to generally conclude whether one method is better than the other, because the performance and visual

outcome of both transformations are largely dependent on the type of the scene.

Performance wise, the pixel transformation method is most suitable for scenes consisting of an exceptionally large number of vertices, while the vertex transformation performs better on scenes consisting of a small to average number of vertices. The reason for this is that only the pre-rendering phase of the pixel transformation method is affected by the number of vertices in the scene, leading to the execution time declining very slowly as we increase the number of vertices. The runtime of the vertex transformation method, on the other hand, is primarily dependent on the number of vertices, and would therefore not be a very effective choice for rendering such scenes. The pixel transformation is mostly affected by the resolution of the image that is rendered, while the vertex transformation is not affected by this at all. However, the results obtained from our experiments suggest that the pixel transformation performs fairly well, even for large image sizes. It was always able to stay above an acceptable frame rate for resolutions as high as 1024x1024. For even higher resolutions it might be wise to consider the vertex transformation method; however, the number of vertices in the scene also needs to be taken into consideration.

Considering the visual aspect, the vertex transformation seems to fare poorly for scenes consisting of very simple models; especially models that consist of a very small number of vertices spaced far apart from each other. An option is to dynamically add new vertices to these models, but this may degrade the performance, depending on the number of new vertices that need to be added. It is also not an easy task to add new vertices to just any kind of model. The pixel transformation always provides visually pleasing images. As it performs a pixel perfect transformation of the image, we can be certain that the transformed image will always be displayed correctly regardless of what kind of scene it is.

The absolute worst case scenario for the vertex transformation would be a scene consisting of a very large number of simple models, such that each model has a very low vertex count, yet the number of vertices in the scene is very large. This would provide very poor results both performance wise and visually. The pixel transformation method, on the other hand would perform very well on such scenes. Scenes that consist of very few but highly detailed models, would benefit more from the vertex transformation.

7.3 Future Work

The work done during the course of this thesis has brought into light a number of ideas that we feel might be suitable for future theses or projects. These include thoughts on further development of the pixel and vertex transformations, in addition to further work regarding projections on curved surfaces in general.

7.3.1 General Ideas

Now that two different transformation methods have been developed and tested thoroughly, the next step would naturally be to try projecting the transformed im-

ages onto the conCave projection surface. There is a lot we still do not know about the visual outcome of the transformation methods, as we have only judged them by the way they appear on a display. It would be interesting to see how they appear on a curved surface, and if there are any special details or visual flaws that were not noticeable when just viewing them on the display.

Another issue that should be investigated is the position of the imaginary planar surface we set when doing the pixel transformation. During our experiments throughout this thesis, we always fixed the imaginary plane at the front of the curved surface. The position of the imaginary surface is irrelevant when we are projecting or viewing the transformed image on a flat surface. However, when projecting the image on a curved surface, this affects the visual outcome, as it determines which parts of the projection are in focus. This relates to the concept known as circle of confusion.

Another idea worth looking into is, when projecting onto a curved surface, we also need to take into consideration how the positions of the projector and the viewer affect the perceived intensity of different parts of the image. Due to the curvature of the conCave system, the light from the projector is spread a lot thinner across certain parts of the curved surface. When projecting onto a flat plane, the light distribution per surface area is the same across the entire surface. For a curved surface, which would have a much larger surface area due to its curvature, the light distribution per surface area is much less; especially for the edges of the surface. When the viewer is standing very close to the projector, the projection angle and viewing angle are somewhat the same, thus the quality of the perceived scene is not affected to a noticeable extent. However, when the viewer is standing further away from the projector, the low light distribution per surface area near the edges will reduce the intensity of the perceived image projected on those parts of the surface. It is possible to correct this visual flaw by determining the intensity that each pixel in the image should have, based on the positions of the projector and the viewer.

The code already contains functionality for moving the position of the viewer, so it should be easy to integrate the use of position and orientation sensors, such that the transformation is updated as the viewer moves around. The feasibility of these transformation methods within virtual or augmented reality systems with such equipments should be investigated, with regards to their speed as well as their visual results.

7.3.2 Ideas for the Pixel Transformation

Fermi-based optimizations should be considered to improve the performance of the pixel transformation method when running on a Fermi GPU. These include cache and memory configurations, making changes to the way global memory is accessed, running several kernel functions in parallel, and adjusting the level of precision on floating point operations. For more detail on these subjects, we refer to our project report regarding Fermi optimizations [5].

The visual results of the pixel transformation might benefit from some anti-aliasing

of the transformed image. Anti-aliasing cannot be applied to the image prior to transformation, as this gives a very unnatural look once the pixels have been moved around, causing the silhouettes of the transformed objects to end up looking very "rough", as if no anti-aliasing has been performed. A suggestion is to implement a parallel anti-aliasing kernel in CUDA, that can be called right after the transformation of the image has taken place.

7.3.3 Ideas for the Vertex Transformation

Since this method transforms the scene by transforming the vertices, this might need to be taken into consideration when implementing the lighting of the scene. It should be investigated whether, and how, the light source should be repositioned as we transform each vertex position, such that when the shading of the scene is performed, the fragment shader provides correct shading of the objects in the scene according to the transformation. This is not an issue for the pixel transformation method, since the shading is only performed on the pre-rendered image; for the vertex transformation it is done on the transformed image.

Bibliography

[1] John Kåre Akeren. *Stereographic visualization on strongly curved projection surfaces.* Master's thesis, Norwegian University of Science and Technology, 2003.

[2] Sverre Djønne and Rune Solheim. *Stereografisk visualisering på sterkt krummende overflate.* Specialization project, Norwegian University of Science and Technology, 2003.

[3] Sverre Djønne. *Visualisering på sterkt krummende overflater ved bruk av polygon-triangulering.* Master's thesis, Norwegian University of Science and Technology, 2004.

[4] Rune Solheim. *Stereografisk visualisering av polygonmodell på sterkt krummende overflate.* Master's thesis, Norwegian University of Science and Technology, 2004.

[5] Joel Chelliah. *The NTNU HPC snow simulator on the Fermi GPU.* Specialization project, Norwegian University of Science and Technology, 2010.

[6] Jerry Isdale. *What is Virtual Reality?* Web-based introduction, 1998. `http://vr.isdale.com/WhatIsVR/noframes/WhatIsVR4.1.html`
Visited during January - June, 2011.

[7] Frederick P. Brooks, Jr. *What's real about virtual reality.* IEEE: Computer Graphics and Applications, University of North Carolina at Chapel Hill, 2005. `http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=799723`
Visited during January - June, 2011.

[8] *Mechdyne.com* Producer of various immersive display systems. `http://www.mechdyne.com/`
Visited on March 2011.

[9] Ashutosh Saxena, Jamie Schulte and Andrew Y. Ng. *Depth estimation using monocular and stereo cues.* Computer Science Department, Stanford University, 2007. `http://www.aaai.org/Papers/IJCAI/2007/IJCAI07-354.pdf`

[10] Paul Bourke. *Calculating Stereo Pairs.* July 1999. `http://paulbourke.net/miscellaneous/stereographics/stereorender/`
Visited during January - June, 2011.

[11] *Stereoscopy.com.* Collection of various information articles and images concerining stereoscopy. `http://www.stereoscopy.com`
Visited during January - June, 2011.

[12] David Luebke and Greg Humphreys. *How GPUs work.* Overview, University of Virginia, 2007. http://www.cs.virginia.edu/~gfx/papers/pdfs/59_HowThingsWork.pdf

[13] *NVIDIA's next generation Cuda compute architecture: Fermi.* White paper, NVIDIA, 2009. http://www.nvidia.com/content/PDF/fermi_white_papers/NVIDIA_Fermi_Compute_Architecture_Whitepaper.pdf

[14] *NVIDIA Cuda programming guide, version 3.2.* Guide, NVIDIA, 2010. http://developer.download.nvidia.com/compute/cuda/3_2/toolkit/docs/CUDA_C_Programming_Guide.pdf

[15] *NVIDIA Cuda C best practices guide, version 3.1.* Guide, NVIDIA, 2010. http://developer.download.nvidia.com/compute/cuda/3_1/toolkit/docs/NVIDIA_CUDA_C_BestPracticesGuide_3.1.pdf

[16] *Maple, official website.* http://www.maplesoft.com/products/maple/
Visited during January - June, 2011.

[17] *Visual Studio, official website.* http://www.microsoft.com/express/Windows/
Visited during January - June, 2011.

Appendices

APPENDIX A

CUDA Framework

Note: This is an excerpt from out fall specialization project report [5], covering the details of the CUDA framework.

Typically, a CUDA application will consist of some sequential code that is to be run on the CPU, which we call host code, and some code that is to be run in parallel on the GPU. The language used for writing CUDA is called CUDA C, which is an extension of the programming language C. This extension allows the programmer to write code that is run in parallel across a large number of threads on the GPU. During the call to a kernel, which is a function run on the GPU, the programmer can set up a hierarchical ordering of how the kernel should be executed across several groups of threads.

A.1 Kernel Functions

Kernels are data-parallel functions that run in parallel on many threads on the GPU. The kernel is defined using special syntax that denotes the hierarchy of threads it will be running on. The programmer will also need to specify a special classifier that indicates how the kernel is called and where it is supposed to execute, e.g using a __global__ declaration specifier, means that this kernel is run on the GPU and can only be called from host functions. There are also other such classifiers as, __device__ , for code that is run on the GPU and can only be invoked by other kernels, and __host__ defines functions that can only be executed on the host. This is also the default classifier and can be omitted in most cases.

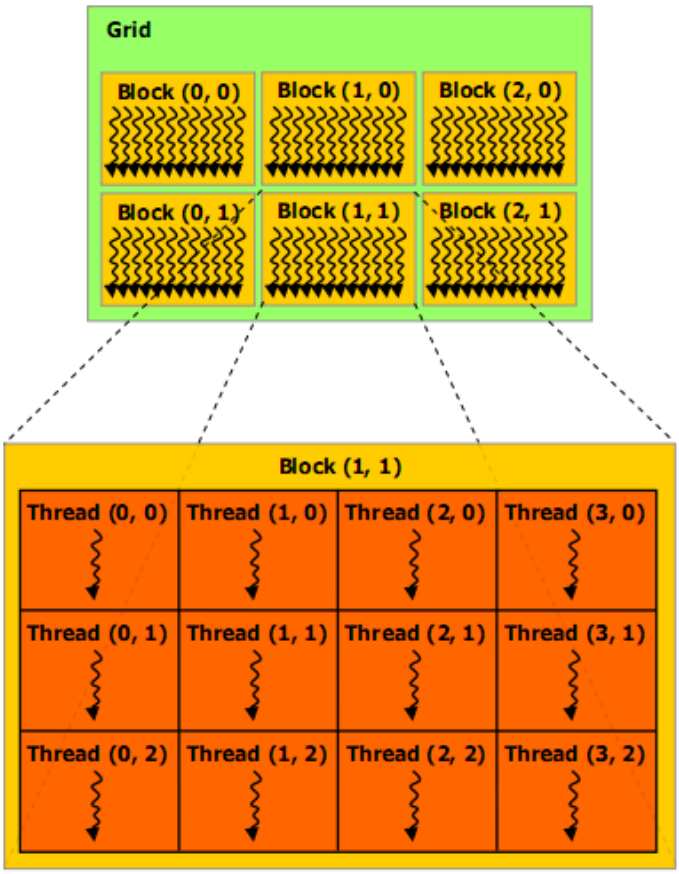

Figure A.1: CUDA thread hierarchy, taken from [13] with permission from NVIDIA.

A.2 Thread Hierarchy

The concept of threads on the GPU is quite similar to threads on the CPU. A thread is the most basic unit that executes on the GPU and it can have its own variables and control flow independent from all other threads. Threads are grouped into a hierarchy of thread blocks which contain several threads, and grids which are arrays of several thread blocks. This three level hierarchy is illustrated in Figure A.1. When a kernel function is called, it executes as a grid of thread blocks, where each grid can be a one- or two dimensional array, and each thread block can have up to three dimensions. The threads within each thread block have several built-in identification variables that can be used to determine their unique locations at both the grid and thread block level. When a kernel is finished executing there is an implicit synchronization of all the threads, however threads within a thread block can synchronize during a kernel execution by calling the __syncthreads() function.

A.3 Memory Hierarchy

As mentioned above, kernels are executed in parallel across multiple threads on the GPU. These threads have several memory spaces to access data from during their execution.

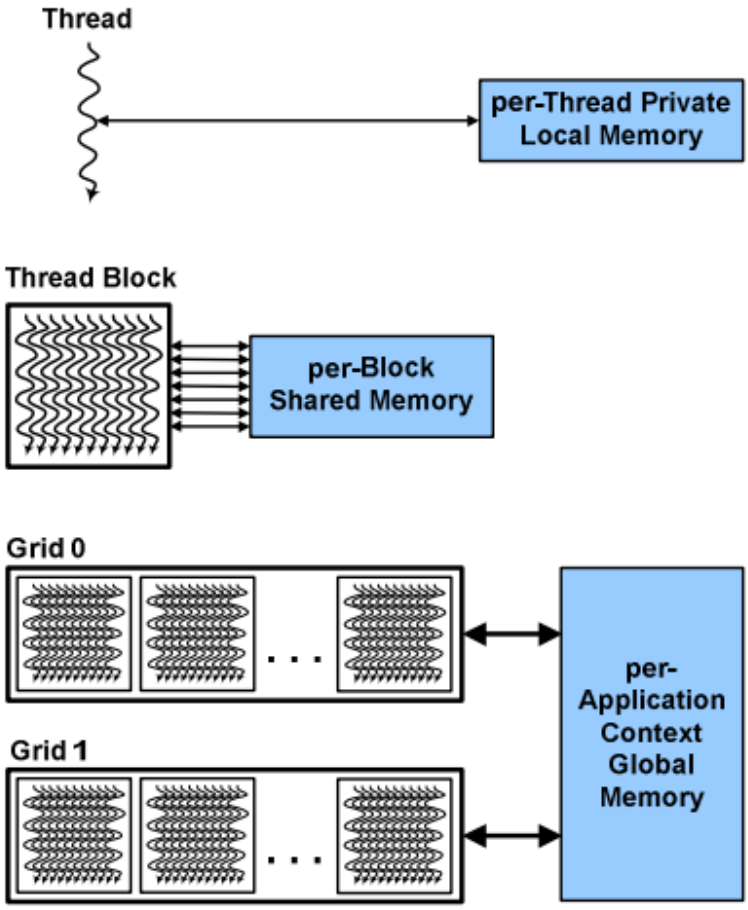

Figure A.2: CUDA memory hierarchy, taken from [13] with permission fron NVIDIA.

A.3.1 Registers and Local Memory

Each thread has its own private local memory and also a set of registers. The registers are naturally the fastest data storage but the total amount of registers available to each thread block is limited. The local memory for each thread is actually stored on the global memory meaning that it has high latency and registers should be used instead whenever possible. If a kernel uses more registers than there are available, register spilling occurs. This means that the data gets put into local memory instead.

A.3.2 Shared Memory

Each thread block has a shared memory portion that all the threads within can use to share data. The lifetime of shared memory variables are the same as the thread block's. Access to shared memory has higher latency than for registers but is still a lot faster than global memory accesses.

A.3.3 Global Memory

All threads can access the global memory which is very large but has high access latency. When memory transfer is taking place to and from the GPU, memory is moved in and out of global memory using different CUDA API calls. It's important that memory that is read and written from and to global memory should be coalesced. This means that each thread in a warp accesses the corresponding word segment (i.e. thread k accesses the k-th word in a segment). Threads can additionally also access two other memory spaces which also reside in global memory, but can be accessed a lot faster under certain conditions. These are texture memory and constant memory.

A.3.4 Constant and Texture Memory

These are read-only, which means that they cannot be altered from inside the kernels. Any variables that are to be placed in texture or constant memory must be defined in the host and copied over to the GPU before the kernel is called. They are both cached on the SMs and repeated lookups can be much faster than accessing regular global memory. Constant memory is specifically used for storing constants and texture memory is used to bind a section of memory as a cached texture.

APPENDIX B

Kernel and Shader Code

Here we present some of the code produced during the course of this thesis. First, we show the "number 2" cylinder and sphere transformation kernels of the pixel transformation method, written in CUDA C, and how they are called from the host. Then we show the vertex and fragment shader code of the vertex transformation method, written in GLSL.

B.1 Pixel Transformation Kernels

After pre-rendering the scene the following function is called from the host on the pre-rendered image. It is responsible for transfering data between device and host memory and calling the kernels:

```
1  void CUDATransformPixels
2  (int pbo_in, int pbo_out, point3 cop,  point3 p)
3  {
4     // Input parameters
5     tIn[0] = cop.x;    //xo
6     tIn[1] = cop.y;    //yo
7     tIn[2] = cop.z;    //zo
8     tIn[3] = p.x;      //xj
9     tIn[4] = p.y;      //yj
10    tIn[5] = p.z;      //zj
11
12    // Store parameters in device memory
13    cudaMemcpy(transformInput, tIn, 6 * sizeof(float),
14               cudaMemcpyHostToDevice);
15
16    // Map pbo to image
17    cudaGLMapBufferObject( (void**)&in_data, pbo_in);
```

```
18      cudaGLMapBufferObject( (void**)&out_data, pbo_out);
19
20
21      // Block and grid dimensions
22      dim3 block(16,16,1);
23      int numBlocks = (WIDTH * (HEIGHT/2)) / (16 * 16);
24      dim3 grid(numBlocks,1,1);
25
26
27      // Call transform kernels
28      CylinderTransformKernel2<<<grid, block>>>(transformInput,
29                                                in_data, out_data);
30      SphereTransformKernel2<<<grid, block>>>(transformInput,
31                                              in_data, out_data);
32
33      // Unmap pbo from image
34      cudaGLUnmapBufferObject( pbo_in);
35      cudaGLUnmapBufferObject( pbo_out);
36  }
```

Listing B.1: Transformation function

The cylinder transform and sphere transform kernels which are called from the above function are excecuted on the device:

Cylinder Transform Kernel

```
1   __global__ void CylinderTransformKernel2
2   (float *input, unsigned char *in_pixels,
3    unsigned char *out_pixels)
4   {
5       int t = (blockIdx.x  * blockDim.x * blockDim.y)
6               + (threadIdx.x + blockDim.x * threadIdx.y);
7
8       int i = t modulo WIDTH;
9       int j = (t - i) / WIDTH;
10
11      //COP
12      float xo = input[0];
13      float yo = input[1];
14      float zo = input[2];
15      //P
16      float xj = input[3];
17      float yj = input[4];
18      float zj = input[5];
19
20      float R2 = WIDTH_SCALE_FACTOR * WIDTH_SCALE_FACTOR;
```

```
21      float halfHeight = HEIGHT/2.0;
22      float halfWidth = WIDTH/2.0;
23
24      // For stepping in the x and y directions
25      float deltaXP = 2.0 * WIDTH_SCALE_FACTOR / WIDTH;
26      float deltaYP = 2.0 * HEIGHT_SCALE_FACTOR /HEIGHT;
27
28      // Where projection intersects the planar surface
29      float xp = (-halfWidth  * 2.0 * WIDTH_SCALE_FACTOR  /WIDTH )
30                + (i * deltaXP);
31      float yp = (-halfHeight * 2.0 * HEIGHT_SCALE_FACTOR /HEIGHT)
32                + (j * deltaYP);
33      float zp = 0;
34
35      // Simplifying common expressions
36      float xj_xp = xj - xp;
37      float zj_zp = zj - zp;
38      float xj_xp2 = xj_xp * xj_xp;
39      float zj_zp2 = zj_zp * zj_zp;
40      float xpzj = xp * zj;
41      float xjzp = xj * zp;
42
43      // Solve z-equation
44      float a = xj_xp2 + zj_zp2;
45      float b = 2.0 * xj_xp * (xpzj - xjzp);
46      float c = (xpzj - xjzp) * (xpzj - xjzp) - R2 * zj_zp2;
47
48      float z = (-b - sqrt(b * b - 4 * a * c)) / (2 * a);
49
50      // Solve for x and xc, and decide u
51      float x = (xpzj - xjzp + z * xj_xp) / zj_zp;
52      float xc = (x *(zo - zp) + xo * (zp - z)) / (zo - z);
53      int u = halfWidth * (xc / WIDTH_SCALE_FACTOR + 1);
54
55      // Solve for y and yc, and decide v
56      float y = (yp - yj) * sqrt((  (zj - z) * (zj - z)
57                                  + (x - xj) * (x - xj)) / a) + yj;
58      float yc = (y - yo) * sqrt((  (zo - zp) * (zo - zp)
59                                  + (xo - xc) * (xo - xc)
60                                 )/ (  (zo - z) * (zo - z)
61                                     + (xo - x) * (xo - x))) + yo;
62      int v = halfHeight * (yc / HEIGHT_SCALE_FACTOR + 1);
63
64      // Put pixel value in output image
65      out_pixels[(i + j * WIDTH) * NUM_COLOR_FACTOR     ] =
66       in_pixels[(u + v * WIDTH) * NUM_COLOR_FACTOR     ];
67      out_pixels[(i + j * WIDTH) * NUM_COLOR_FACTOR + 1] =
68       in_pixels[(u + v * WIDTH) * NUM_COLOR_FACTOR + 1];
69      out_pixels[(i + j * WIDTH) * NUM_COLOR_FACTOR + 2] =
70       in_pixels[(u + v * WIDTH) * NUM_COLOR_FACTOR + 2];
```

```
}
```

Listing B.2: Cylinder transform kernel

Sphere Transform Kernel

```
__global__ void SphereTransformKernel2
(float *input, unsigned char *in_pixels,
 unsigned char *out_pixels)
{
    int t = (blockIdx.x  * blockDim.x * blockDim.y)
            + (threadIdx.x + blockDim.x * threadIdx.y);

    int i = t modulo WIDTH;
    int j = (t - i) / WIDTH;

    //COP
    float xo = input[0];
    float yo = input[1];
    float zo = input[2];

    //P
    float xj = input[3];
    float yj = input[4];
    float zj = input[5];

    float R2 = WIDTH_SCALE_FACTOR * WIDTH_SCALE_FACTOR;
    float halfHeight = HEIGHT/2.0;
    float halfWidth = WIDTH/2.0;

    // For stepping in the x and y directions
    float deltaXP = 2.0 * WIDTH_SCALE_FACTOR / WIDTH;
    float deltaYP = 2.0 * HEIGHT_SCALE_FACTOR /HEIGHT;

    // Where projection intersects the planar surface
    float xp = (-halfWidth  * 2.0 * WIDTH_SCALE_FACTOR  /WIDTH )
               + (i * deltaXP);
    float yp = (j * deltaYP);
    float zp = 0;

    // Simplifying common expressions
    float xj_xp = xj - xp;
    float zj_zp = zj - zp;
    float yj_yp = yj - yp;
    float xj_xp2 = xj_xp * xj_xp;
    float zj_zp2 = zj_zp * zj_zp;
    float yj_yp2 = yj_yp * yj_yp;
    float xj2 = xj * xj;
```

```
43      float xp2 = xp * xp;
44      float zj2 = zj * zj;
45      float zp2 = zp * zp;
46      float yj2 = yj * yj;
47      float yp2 = yp * yp;
48      float xpxj = xp * xj;
49      float xpzj = xp * zj;
50      float xjzp = xj * zp;
51      float zpzj = zp * zj;
52      float xjyp = xj * yp;
53      float xpyj = xp * yj;
54      float ypyj = yp * yj;
55
56      // Solve z-equation
57      float a = xj_xp2 + yj_yp2 + zj_zp2;
58      float b = zj * (xp2 - xpxj - yp * yj_yp)
59                + zp * (xj2 - xpxj + yj * yj_yp);
60      float d = zj_zp2 * (zj2 * (R2 - xp2 - yp2)
61                - 2 * zpzj * (R2 - xpxj - ypyj)
62                - (xjyp - xpyj) * (xjyp - xpyj)
63                + zp2 * (R2 - xj2 - yj2)
64                + R2 * (xj_xp2 + yj_yp2));
65
66      float z = (b - sqrt(d)) / a;
67
68      // Solve for x and xc, and decide u
69      float x = (xpzj - xjzp + z * xj_xp) / zj_zp;
70      float xc = (x *(zo - zp) + xo * (zp - z)) / (zo - z);
71      int u = halfWidth * (xc / WIDTH_SCALE_FACTOR + 1);
72
73      // Solve for y and yc, and decide v
74      float y = (yp - yj) * sqrt((   (zj - z) * (zj - z)
75                                  + (x - xj) * (x - xj)
76                                 ) / (zj_zp2 + xj_xp2)) + yj;
77      float yc = (y - yo) * sqrt((   (zo - zp) * (zo - zp)
78                                  + (xo - xc) * (xo - xc)
79                                 ) / (   (zo - z) * (zo - z)
80                                       + (xo - x) * (xo - x))) + yo;
81      int v = halfHeight * (yc / HEIGHT_SCALE_FACTOR + 1);
82
83
84      // draw on the top half
85      j += halfHeight;
86
87
88      // Put pixel value in output image
89      if(u < 0 || u >= WIDTH || v < 1 || v >= HEIGHT)
90      {
91          out_pixels[(i + j * WIDTH) * NUM_COLOR_FACTOR     ] = 0x55;
92          out_pixels[(i + j * WIDTH) * NUM_COLOR_FACTOR + 1] = 0x55;
```

```
93          out_pixels[(i + j * WIDTH) * NUM_COLOR_FACTOR + 2] = 0x55;
94      }
95      else
96      {
97          out_pixels[(i + j * WIDTH) * NUM_COLOR_FACTOR     ] =
98           in_pixels[(u + v * WIDTH) * NUM_COLOR_FACTOR     ];
99          out_pixels[(i + j * WIDTH) * NUM_COLOR_FACTOR + 1] =
100          in_pixels[(u + v * WIDTH) * NUM_COLOR_FACTOR + 1];
101         out_pixels[(i + j * WIDTH) * NUM_COLOR_FACTOR + 2] =
102          in_pixels[(u + v * WIDTH) * NUM_COLOR_FACTOR + 2];
103     }
104 }
```

Listing B.3: Sphere transform kernel

B.2 Vertex Transformation Shaders

Before rendering, the vertex and fragment shaders are attached to the program to take responsibility of the projection of the vertices and drawing of the pixels. The vertex transformation procedure is done in the vertex shader alone:

Vertex Shader

```
1  //input
2  uniform vec3 P;
3  uniform vec3 COP;
4
5  //for lighting
6  varying vec4 diffuse,ambient;
7  varying vec3 normal,lightDir,halfVector;
8
9  void main(void)
10 {
11     //COP
12     float xo = COP.x;
13     float yo = COP.y;
14     float zo = COP.z;
15
16     //P
17     float xj = P.x;
18     float yj = P.y;
19     float zj = P.z;
20
21     float R = 1.0;
22     float R2 = R * R;
23
24     float zp = 0.0;
```

```
25
26      // Vertex position
27      vec4 vertPos = gl_ModelViewMatrix * gl_Vertex;
28      float xb = -vertPos.x;
29      float yb = vertPos.y;
30      float zb = -vertPos.z;
31
32      // Simplifying expressions
33      float xo_xb = xo - xb;
34      float yo_yb = yo - yb;
35      float zo_zb = zo - zb;
36      float xo_xb2 = xo_xb * xo_xb;
37      float zo_zb2 = zo_zb * zo_zb;
38      float xo2 = xo * xo;
39      float xb2 = xb * xb;
40      float zo2 = zo * zo;
41      float zb2 = zb * zb;
42      float xbxo = xb * xo;
43      float xbzo = xb * zo;
44      float xozb = xo * zb;
45      float zbzo = zb * zo;
46
47      float a, b, c, d, z;
48
49      if(yb<0.0)
50      {
51          a = xo_xb2 + zo_zb2;
52          b = 2.0 * xo_xb * (xbzo - xozb);
53          c = xb2 * zo2 - 2.0 * xbzo * xozb + xo2 * zb2 - R2 *
                zo_zb2;
54
55          z = (-b - sqrt(b * b - 4.0 * a * c)) / (2.0 * a);
56      }
57      else
58      {
59          float yo_yb2 = yo_yb * yo_yb;
60          float yo2 = yo * yo;
61          float yb2 = yb * yb;
62          float ybyo = yb * yo;
63          float xoyb = xo * yb;
64          float xbyo = xb * yo;
65
66          a = xo_xb2 + yo_yb2 + zo_zb2;
67          b = zo * (xb2 - xbxo - yb * yo_yb)
68              + zb * (xo2 - xbxo + yo * yo_yb);
69          d = zo_zb2 * (zo2 * (R2 - xb2 - yb2)
70              - 2.0 * zbzo * (R2 - xbxo - ybyo)
71              - (xoyb - xbyo) * (xoyb - xbyo)
72              + zb2 * (R2 - xo2 - yo2)
73              + R2 * (xo_xb2 + yo_yb2));
```

```
74
75          z = (b - sqrt(d)) / a;
76      }
77
78      // Find x and xp
79      float x = (xbzo - xozb + z * xo_xb) / zo_zb;
80      float xp = (x *(zj - zp) + xj * (zp - z)) / (zj - z);
81
82      // Find y and yp
83      float y = (-yo_yb) * sqrt((   (zo - z) * (zo - z)
84                                  + (xo - x) * (xo - x)
85                                ) / (zo_zb2 + xo_xb2)) + yo;
86      float yp = (y - yj) * sqrt((   (zj - zp) * (zj - zp)
87                                   + (xp - xj) * (xp - xj)
88                                 ) / (   (zj - z) * (zj - z)
89                                       + (x - xj) * (x - xj))) + yj;
90
91      vec4 projectionPoint;
92      projectionPoint.x = xp;
93      projectionPoint.y = yp;
94      projectionPoint.z = zp;
95      projectionPoint.w = vertPos.w;
96
97      gl_Position =  projectionPoint;
98
99      // lighting calculations
100     normal = normalize(gl_NormalMatrix * gl_Normal);
101     lightDir = normalize(vec3(gl_LightSource[0].position));
102     diffuse = gl_FrontMaterial.diffuse
103               * gl_LightSource[0].diffuse;
104     ambient = gl_FrontMaterial.ambient
105               * gl_LightSource[0].ambient;
106 }
```

Listing B.4: Vertex Shader

The fragment shader is responsible for the pixel-by-pixel lighting. Some of the lighting calculations are done in the vertex shader, and the variables are passed to the fragment shader to complete the calculations and set the pixel values:

Fragment Shader

```
1 //lighting variables from the vertex shader
2 varying vec4 diffuse,ambient;
3 varying vec3 normal,lightDir,halfVector;
4
5 void main (void)
```

```
6  {
7      vec3 n,halfV;
8      float NdotL,NdotHV;
9
10     vec4 color = ambient;
11     n = normalize(normal);
12
13     NdotL = max(dot(n,lightDir),0.0);
14
15     if (NdotL > 0.0)
16     {
17         color += diffuse * NdotL;
18     }
19
20     gl_FragColor = color;
21 }
```

Listing B.5: Fragment Shader