Master's Thesis

3D-GRAPHICS RENDERING ON A MULTI-PROCESSOR ARCHITECTURE.

F. van Heesch

Coach: Ir. E. Jaspers (Philips Research Laboratories)
Dr. E. van der Tol (Philips Research Laboratories)
Supervisor: prof.dr.ir. G. de Haan
Date: March 2003
Abstract

Real-time 3D-graphics rendering is a highly computationally intensive task, with a high memory bandwidth requirement. A multiprocessor architecture called SpaceCAKE, developed at Philips Research in the Processor Oriented Architectures department, is used as a high performance multimedia platform to run a software implementation of the 3D-graphics standard OpenGL. The Wasabi chip is the instance of the SpaceCAKE architecture that is currently under development and serves as a target architecture for this project. To achieve an optimal gain for this multiprocessor architecture parallelism in OpenGL needs to be found and exploited.

Two adapted versions of Mesa - a free software implementation of OpenGL - have been implemented and ported to the SpaceCAKE architecture. Both versions exploit data partitioning to achieve parallelism. The first implementation, called Sort-Middle, divides the 2D output in rectangular segments and assigns individual segments to each processing unit of the multiprocessing platform. The second implementation, called Sort-Last, distributes groups of polygons amongst processing units.

The efficiency and speedup of both implementations have been measured using an adapted version of the game “Tuxracer” as a test application. The performance and speedup have been measured with and without taking rasterization into account. The latter case is of importance when rasterization is performed in dedicated hardware.

Both implementations show that the performance gain of parallelism is limited due to overhead and load imbalance. Load imbalance is caused by the highly data-dependent execution times of the rasterization of polygons. Overhead is caused by performing part of the functionality on each processing unit.

We conclude that implementing parallelism for highly data dependent applications that have been optimized for a single processor is difficult, likely resulting in poor scalability. Specifically, the use of a global state machine for 3D-graphics rendering causes significant overhead when implementing data partitioning. A combination of functional partitioning and data partitioning is recommended for further research.
Preface

For hundreds of years, artists have known tricks that can make a flat, 2D (2-dimensional) painting look like a window into the real, 3D world. Objects appear smaller when they are farther away; when objects close to the camera are in focus, objects farther away are fuzzy; colors tend to be less vibrant as they move farther away. Most of these features comply to mathematical rules, or can be approximated by mathematical equations. This implies that computers can be used to “paint” a 2D image of a synthesized 3D world.

Since the early sixties, people in the field of computer graphics have developed means to achieve an efficient solution to apply these mathematical equations to create 3D colored images. The number of computations involved in this process inhibited 3D graphics to be widely used for industry and entertainment purposes until the late eighties.

Since then, the advancements in silicon technologies have provided an exponentially increasing transistor count, raising the question how to put all this hardware to effective use. The exploitation of parallelism seems to be the answer, as it provides a cost-effective way to solve computationally intensive problems which often have built-in parallelism at different levels. It is this built-in parallelism that needs to be elicited to achieve an even higher performance in 3D-graphics rendering.

This report describes the efforts of a nine month graduation project, performed at Philips Research Eindhoven in the group Video Processing and Visual Perception, under the dedicated supervision of Egbert Jaspers, Erik van der Tol, and Prof. Gerard de Haan. These nine months have been an indulging and educational concluding step towards the Masters of Science degree in Electrical Engineering at the Technical University of Eindhoven.
Contents

1 Introduction ................................................................. 1
  1.1 Problem statement .................................................. 1
  1.2 Outline of this thesis ............................................... 2

2 Real-time 3D-graphics rendering .................................... 3
  2.1 The history of 3D-graphics rendering .......................... 4
  2.2 Synthesizing 3D worlds ........................................... 5
  2.3 Generating 2D images ............................................. 8
  2.4 OpenGL ........................................................... 11
    2.4.1 The OpenGL pipeline ......................................... 11
    2.4.2 The OpenGL interface ......................................... 12
  2.5 Mesa ............................................................. 13

3 Multi-processing architectures ...................................... 17
  3.1 Parallelism ....................................................... 18
    3.1.1 Instruction-level parallelism ................................ 19
    3.1.2 Thread-level parallelism .................................... 20
    3.1.3 Implementing parallelism ................................... 20
  3.2 Performance analysis ............................................. 21
  3.3 Architectures for parallelism .................................... 23
  3.4 SpaceCAKE ...................................................... 28
    3.4.1 Architecture ................................................ 28
    3.4.2 Development environment ................................... 30
  3.5 Pthreads ......................................................... 30

4 Parallelization methods ............................................. 32
4.1 Data partitioning versus functional partitioning ........................................... 32
4.2 Sorting methods ......................................................................................... 34
  4.2.1 Sort-First ......................................................................................... 35
  4.2.2 Sort-Middle ...................................................................................... 35
  4.2.3 Sort-Last ......................................................................................... 36

5 Implementation of Mesa on SpaceCAKE ......................................................... 38
  5.1 Porting Mesa to the TriMedia platform .................................................... 38
  5.2 Creating an application for Mesa .............................................................. 39
    5.2.1 Direct approach ............................................................................. 39
    5.2.2 Indirect approach ......................................................................... 40
  5.3 Parallelizing Mesa .................................................................................. 42
    5.3.1 Implementing Sort-Middle ............................................................. 42
    5.3.2 Implementing Sort-Last ................................................................. 43

6 Performance measurements ........................................................................... 45
  6.1 Measuring the performance of Mesa ......................................................... 46
  6.2 Performance of Sort-Middle .................................................................... 50
    6.2.1 Load imbalance ............................................................................. 50
    6.2.2 Parallel overhead ......................................................................... 51
    6.2.3 Efficiency and Speedup .................................................................. 52
  6.3 Performance of Sort-Last ........................................................................ 52
    6.3.1 Load imbalance ............................................................................. 54
    6.3.2 Parallel overhead ......................................................................... 55
    6.3.3 Efficiency and speedup .................................................................. 56
  6.4 Performance without rasterization ............................................................ 56
    6.4.1 Load imbalance ............................................................................. 57
    6.4.2 Parallel overhead ......................................................................... 57
    6.4.3 Efficiency and speedup .................................................................. 59

7 Results and conclusions .................................................................................. 62
  7.1 Results .................................................................................................... 62
    7.1.1 Sort-Middle .................................................................................. 62
    7.1.2 Sort-Last ...................................................................................... 63
## CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.1.3 Tools resulting for this research project</td>
<td>63</td>
</tr>
<tr>
<td>7.2 Conclusions</td>
<td>64</td>
</tr>
<tr>
<td>7.3 Recommendations for future work</td>
<td>66</td>
</tr>
<tr>
<td>A User Manual</td>
<td>70</td>
</tr>
<tr>
<td>A.1 The OpenGL recorder</td>
<td>70</td>
</tr>
<tr>
<td>A.2 The SpaceCAKE simulator</td>
<td>71</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In the field of consumer electronics, the advent of new features such as Internet, games, video conferencing and mobile communication has triggered the convergence of television and computer technologies. This trend is expected to enhance TV applications with infotainment that integrate video, graphics, and interaction. Consequently, 3D-graphics rendering, as familiar in the PC and game console domain, is expected to be required in TV systems or set-top boxes. In this graduation project 3D-graphics rendering for TV-type of systems will be investigated.

Since it is highly uncertain what the penetration speed of 3D graphics within the TV consumer domain will be, it is believed that, for the near future, solutions in software that run on programmable media processors could offer the optimal balance between flexibility and performance. On a longer term the increasing throughput requirement for 3D may be satisfied by the increasing performance of media processors. As it is plausible that the increasing computational power of a single media processor cannot keep up with Moore's law [MOOR65], it is likely that the increasing number of transistors on a chip will be exploited by multi-processor implementations.

From this point of view the question arises, how to map the 3D rendering functionality on such an architecture, providing scalability over the technology road-map. Moreover, how can the performance be scaled to easily enable a tradeoff with hardware costs at a later stage?

1.1 Problem statement

The mapping of 3D-graphics rendering on a multiprocessor platform requires parallelism to be exploited. For parallelism an application needs to be sub-divided in several tasks. The concurrency of these tasks needs to be identified, to be able to orchestrate them amongst processing units.

3D-graphics rendering can be divided into multiple tasks by splitting up its functionality or by splitting up the data. For this project, the solution of data partitioning is explored. For example, the display screen can be partitioned into tiles or slices, and each processing unit can be made responsible for processing a single tile or slice.

Although this solution seems straightforward, it is quite challenging to allow this kind of partitioning in the implementation. Particularly if the inter-processor dependencies are complex and
CHAPTER I. INTRODUCTION

the processing is data dependent. This will require communication overhead that will negatively influence the potential speedup of multiprocessor systems.

The content of this graduation project can be characterized as 'Parallelizing a real-time 3D-graphics renderer'. It is part of a larger project called PADME (Parallel Applications Developed for Multi-processor Environments) that investigates the mapping of applications onto a scalable multiprocessor architecture. Only real-time rendering, as applied in 3D games, VRML-viewers, and other interactive 3D applications is investigated, where each image is generated on-the-fly. This report presents the results of this nine-month effort after introducing the topics 3D-graphics rendering and parallelism in computer architectures.

1.2 Outline of this thesis

The main problem discussed in this graduation project is how to partition a 3D-graphics rendering engine, efficiently over multiple processors. Before presenting possible solutions to this problem an overview on both 3D-graphics rendering and multiprocessor architectures will be given. More specifically, a general introduction to the art of real-time 3D-graphics modeling and rendering will be presented in Chapter 2 to get insight on the computations involved to correctly convert 3D-objects to 2D-images. Section 2.4 discusses a 3D-graphics rendering API (Application Programmers Interface) called OpenGL (an abbreviation of Open Graphics Library), which is a well-known and widely spread standard. Section 2.5 discusses Mesa, a software implementation of the OpenGL standard that has been adopted for this project.

In Chapter 3, a classification of existing multiprocessing architectures will be presented, followed by an introduction to the SpaceCAKE architecture. This specific architecture and its design philosophy will be described in Section 3.4. The chapter concludes with an overview of how application software can be developed for this architecture using available tools.

Parallelization methods for 3D-graphics rendering, known from the literature, are presented in Chapter 4. Advantages and disadvantages of these implementation methods will be briefly discussed.

Subsequently, Chapter 5 explains the steps to implement a parallel 3D-graphics renderer on a multiprocessor architecture. Two different approaches will be presented that allow the computations of the benchmark application to be distributed among processors. One approach partitions the output images in several screen segments, whereas the second approach distributes 3D geometry amongst processors. For both of these implementations, the performance gain is measured relative to the performance of sequential rendering. The results of these experiments are presented in Chapter 6. A detailed performance analysis of both methods will be given in Section 6.2 and Section 6.3, assuming that all rendering functionality is performed in software. This analysis is repeated in Section 6.4, disregarding the computations required for rasterization. This is particularly useful for an implementation in which the rasterization is performed in hardware, whereas the remaining tasks are executed in software.

Finally, Chapter 7 discusses the deliverables of the graduation project, followed by some conclusions. Recommendations on improvements for the SpaceCAKE development environment and the work presented in this report are also outlined in this chapter. The Appendix contains a user manual, with a step by step tutorial on how to repeat the experiments presented in this report.
Chapter 2

Real-time 3D-graphics rendering

3D graphics is the field of computer graphics, concerned with generating and displaying 3D objects in a 2D space, e.g. the display screen. Whereas pixels in a 2D graphic have the properties of position, color, and brightness, 3D-pixels add a depth property that indicates where the pixel lies on an imaginary z-axis. Converting 2D or 3D shapes into a bitmap that can be displayed is known as rendering.

3D graphics are nowadays most commonly encountered in games and movies. The level of realism of 3D graphics in movies is often much higher than that in games, as for movies off-line, i.e. non-real-time, rendering techniques can be used. For games however, the 2D projection of 3D graphics has to be generated on-the-fly, due to their interactive nature. This often imposes a trade-off between performance and realism on the game designer. For both types of rendering techniques the following three process steps are performed:

- Creating a virtual 3D world.
- Determining which part of the world will be shown on the screen.
- Determining the color of every pixel on the screen to make the appearance of the whole image as realistic as possible.

For this graduation report, we are only interested in the techniques that are used for real-time rendering. Therefore, rendering as referred to in this report implies generation of real-time 3D graphics. The purpose of this chapter is to give a general overview of the algorithms that are essential to rendering 3D graphics in a 2D space.

First, a brief history of 3D-graphics rendering is presented. Second, we describe the different aspects that are required to synthesize a 3D world, followed by an overview of the complexity of 3D rendering. This is necessary to determine how data partitioning can be exploited best in a multi-processing environment. Furthermore, Sections 2.4 and 2.5 elaborate on a 3D-graphics standard API and one of its implementations.
2.1 The history of 3D-graphics rendering

In this section, a brief overview is presented of real-time 3D-graphics solutions throughout the years, showing the general shift in implementation methods.

The first serious use of 3D computer graphics was flight simulation in the late sixties, as illustrated in Figure 2.1. At the same time, the vector display was introduced. It enabled the creation of images by lines, which made them perfect for applications like CAD/CAM (computer-aided design/computer-aided manufacturing) since they could easily create and manipulate wireframe objects. In the sixties and seventies, large mainframes were required that performed all computations in software to achieve an acceptable performance.

In the early eighties, the first accelerated output buffers, or frame buffers, had some hardware support for primitive 3D-graphics operations, and implemented a z-buffer, i.e. a depth buffer, for hidden surface calculations. 3D-graphics rendering still required dedicated super-computers (often multi-processor systems) to achieve a frame rate of 15 to 30 Hz like the IRIS 1400 from Silicon Graphics [BAUM98]. These systems were able to render up to several tens of thousands flat shaded textured triangles.

In the late eighties, advancements in semiconductor technologies made it possible to include increasingly more graphics functionality in hardware. Second generation systems raised the realism level by rendering polygons that were Gouraud shaded, Phong-lit, and depth-buffered [FOLE90], which will be discussed in the following section. Additionally, raw transformation and scan conversion performance increased considerably. The first example of a second generation system was the Hewlett-Packard SRX [SWAN86], followed by the Silicon Graphics GT [AKEL88], which was the first system to break the 100,000 polygons/second mark. The improved shading capabilities made these workstations suitable for applications like object modeling, animations, and molecular modeling, which extended the application area of 3D graphics considerably.

Furthermore, hardware graphics acceleration made its introduction in the PC consumer market. Inspired mostly by the gaming industry, hardware manufacturers started mass production of graphics cards for PCs with fast decreasing prices, incorporating increasingly more of the 3D-graphics pipeline in hardware.

Current state-of-the-art graphics accelerators implement all 3D-graphics rendering functionality in hardware in the form of multiple pipelines running in parallel. These accelerators allow ten millions of polygons per second to be displayed on high resolutions. Furthermore, shading models and vertex transformations have become programmable, allowing more flexibility and e.g. object
dynamics to be calculated in hardware as well.

The following summarizes the past trends in the development of 3D-graphics hardware:

- Advancements in the semiconductor technology made parallel supercomputers disappear in favor of dedicated hardware acceleration boards.
- The PC entertainment industry's hunger for improved 3D graphics triggered an explosive development of 3D hardware accelerators for PCs.
- Increasingly more of the 3D-graphics pipeline is being performed in hardware. This is possible because the most popular 3D-graphics APIs (OpenGL and DirectX) have been fixed for several years. Extensions have been made but backwards compatibility has been preserved.
- Software parallelism on supercomputers is replaced by parallel hardware implementations.

These trends have created a shift in the way parallelism is exploited. Where high-level data parallelism on a supercomputer was preferred in the past, nowadays 3D-graphics processing through a pipeline, in which each stage is processed in parallel, is preferred. Chapter 4 discusses advantages and drawbacks of both methods.

2.2 Synthesizing 3D worlds

The description of a 3D world, or scene, consists of explicating the surfaces, visual properties, positions, and orientations of the individual objects that build up the 3D world. Typically, the construction of such descriptions is a cumbersome task. However, this process can be automated with the help of 3D cameras and 3D scanners for instance, but is often performed manually with the help of special software tools that can synthesize and align complex objects. In this section the process of creating a 3D world is explained by discussing each step in the creation process. First let us define some terminology that is often used for 3D rendering.

Geometry: The creation of a virtual 3D world, a process known as modeling, starts by defining 3D shapes. These shapes are approximations of the surface of an object. Often, only triangles and quads are allowed to build up a surface, but polygonal planes are occasionally used.

A surface is approximated by sampling it. Each sample is a point in a 3D coordinate system. These points are called vertices. Sets of these points can be connected, creating edges to generate lines or polygons that approximate the surface. If more samples are used to approximate an object, the approximation of the object becomes more accurate and a higher level of realism can be achieved, as shown in Figure 2.2.

Materials: Additional realism and detail can be obtained by simulating the material of an object. Objects can be solid or transparent, rough or smooth, and can have textures. Textures can greatly increase the level of detail of an object without increasing the amount of vertices. A texture can be assigned to a polygon by projecting a bitmap on it or by applying certain coloring equations. Some examples are depicted in Figure 2.3.
CHAPTER 2. REAL-TIME 3D-GRAPHICS RENDERING

Figure 2.2: Surface approximations of an object with a relative low number of vertices (a) and the same object with a high number of vertices (b).

Figure 2.3: Examples of texture maps on surfaces.

Lighting: Just like in real life lighting is added in a 3D scene to illuminate objects. Several types of lighting can exist simultaneously in a scene like point lights, spots (or cone lights), light beams, and ambient lights. Each light source contributes to the intensity value of a vertex and thereby to the intensity of pixels.

Different mathematical models can be applied to determine the intensity of a vertex or a screen pixel caused by the lights in the scene. These models are called shading models. The three most often used shading models in real-time 3D-graphics rendering are constant shading, Gouraud shading, and Phong shading.

Constant shading (or flat shading) determines a single light intensity value for each polygon. The value of the intensity is determined by several factors like:

- the angle and distance between light source and surface normal;
- the angle between the viewing direction and the surface normal;
- the material assigned to the polygon.

This shading model has the disadvantage of causing (unrealistic) intensity discontinuities at polygon edges.

Gouraud shading is a shading method that eliminates shading discontinuities. The Gouraud shading process calculates the illumination intensity of each vertex followed by linearly interpolating the intensities along and between edges for each pixel.
Phong shading interpolates the normal vector along polygon edges instead of the illumination intensity and re-computes the intensity level at each pixel. This shading model gives better results especially for large polygons with highly specular materials as illustrated in Figure 2.4.

![Figure 2.4](image)

**Figure 2.4:** Examples of an object with (a) flat shading, (b) Gouraud shading, and (c) Phong shading methods. Note that each object has the same amount of polygons.

Other, more advanced lighting and shading methods, like radiosity and global illumination, currently require far too many calculations to be performed in real-time with acceptable frame rates. However, for static scenes, they can be pre-calculated. Their results are then projected on the polygons as an additional texture map. We refer to [FOLE9ü] for a more detailed overview.

**Advanced features:** To satisfy the increasing hunger for realism, lots of additional features have been added to most 3D-rendering systems. These features include applying fog, casting shadows, implementing depth of field, creating mirroring surfaces, using volumetric lights, adding lens flares, etc. Details of any of these features will not be discussed in this report. See [PROU01] and [MURP96] for more information on 3D-graphics functionality extensions. An example of a 3D image using several of these advanced features is illustrated in Figure 2.5.

![Figure 2.5](image)

**Figure 2.5:** Example of 3D image using a combination of advanced features.
CHAPTER 2. REAL-TIME 3D-GRAPHICS RENDERING

2.3 Generating 2D images

In order to get an indication of the amount of calculations for converting a 3D scene to a 2D image, a coarse overview of the required calculations is presented in this section. The conversion process is generally depicted as a pipeline, illustrated in Figure 2.6 [FOLE90].

![Figure 2.6: 3D-graphics pipeline. (WC: World Coordinates, NPC: Normalized Devices Coordinates, and DC: Display Coordinates.)](image)

This pipeline shows geometry data (the description of the 3D surfaces) entering in the order that is dictated by the display traversal, i.e. the order of polygon processing. The geometry data is transformed to several coordinate systems during the traversal through the pipeline, as illustrated in Figure 2.7. This allows efficient operation on the geometry data at each stage.

![Figure 2.7: Transformations to different coordinate systems.](image)

Homogeneous coordinates are used instead of 3D coordinates to utilize a mathematical technique to embed 3D coordinates and transformations into a 4D matrix format. As a result, inversions or combinations of linear transformations are simplified to inversions or multiplications of the corresponding matrices. Therefore, homogeneous coordinates simplify a coordinate transformation and a perspective transformation to a matrix multiplication.

Geometry data enters the pipeline in world coordinates (WC) and is transformed to eye-coordinates. The eye-coordinate system aligns the z-axis with a defined viewing direction. Light-
ing and texture mapping is performed in this coordinate system. From Eye-coordinates the geometry data is transformed to Clip-coordinates, to simplify the determination which part of the geometry data is not visible, i.e. which polygons are clipped.

The data that has not been clipped, is converted back from homogeneous coordinates to 3D coordinates resulting in Normalized Device Coordinates (NDC). The \( x \)- and \( y \)-coordinates of the NDC-system are directly related to the display's \( x \)- and \( y \)-coordinates. The final color values of pixels is calculated in NDC.

A transformation consists of a translation, a scaling, and/or a rotation. Object data is transformed by multiplying its object coordinates with a four-by-four transformation matrix (see Appendix F [WOOM99] and Chapter 3 [FOLE90]), which is a combination of translation, rotation, and scaling matrices, as depicted in Figure 2.8.

\[
\begin{pmatrix}
1 & 0 & 0 & T_x \\
0 & 1 & 0 & T_y \\
0 & 0 & 1 & T_z \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & \cos(A) & -\sin(A) & 0 \\
0 & \sin(A) & \cos(A) & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
\begin{pmatrix}
S_x & 0 & 0 & 0 \\
0 & S_y & 0 & 0 \\
0 & 0 & S_z & 0 \\
0 & 0 & 0 & 1
\end{pmatrix}
\]

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
1/x_v & 1/y_v & 1/z_v & 1
\end{pmatrix}
\]

\( A \): angle of rotation \([0 - 2\pi]\)

\( x_v \): \( x \)-coordinate of viewing point

\( y_v \): \( y \)-coordinate of viewing point

\( z_v \): \( z \)-coordinate of viewing point

**Figure 2.8:** Elementary transformation matrices for translation, rotation, scaling, and perspective distortion.

Along with object data, a viewing point and direction, viewing volume, and field of view is defined, which determines how the scene is projected to 2D. This is illustrated in Figure 2.9. These settings are similar to the settings of a camera in the field of photography. When taking a picture of a model with a camera, a photographer first decides from what position and orientation he will capture the model with his camera. This corresponds to determining the viewing position and direction. The photographers choice for lens and zoom factor, is analogous to determining the shape of the viewing volume. The determination of the size of the photograph is similar to defining the
CHAPTER 2. REAL-TIME 3D-GRAPHICS RENDERING

Figure 2.9: Defining a viewing area.

image resolution.

Typically, only part of the scene is visible from the defined viewing area. Therefore, polygons that are not visible in the output image are usually rejected at the beginning of the pipeline to eliminate unnecessary vertex processing. Polygons that are totally or partially visible are conveyed through the pipeline.

Additional to transforming vertices to NDC, the normals of vertices are transformed as well in order to correctly perform lighting operations. To calculate lighting parameters for all vertices inside the viewing volume a diffuse and specular intensity value can be calculated for each light source, using one of the lighting functions and the material properties of the corresponding vertex.

If textures are used, textures coordinates need to be generated for each vertex. These coordinates need to be transformed to NPC as well. Calculation of texture coordinates depends on the method that is used to project a bitmap on to a polygon. Projections can be defined by the user or is derived from vertex normals and vertex object coordinates.

After vertices are lit and texture-mapped, they are clipped to the viewing volume. Clipping a polygon to the viewing volume can introduce additional vertices, and therefore polygons, at the edges of the viewing volume. Furthermore, at this point in the pipeline the four-by-four matrix is converted back to a 3D-coordinate system.

At the final stage of the 3D rendering process, rasterization is performed, which in general includes visible surface determination, scan conversion, and shading. Visible surface determination involves inspecting the directions of the normal of each polygon (often only triangles remain after
the clipping stage). Scan conversion is the process of converting shapes to pixels. It requires compute intensive filtering, anti-aliasing, and scaling operations. The final shading operations determine the intensity and color of each pixel. It is calculated by interpolating and combining the results from lighting, material and texture interpolations.

Although the rendering process, sketched above, is only a rough description, it should be clear that rendering, and especially the final rasterization, is very computationally intense. Notice that computations for each image can vary significantly, depending on the number of vertices that need to be processed and the use of textures and lights. Furthermore, many variations in the implementation of the graphics pipeline occur, which makes it difficult to estimate the computational load in general.

In the next two sections we will focus on the 3D-graphics standard OpenGL and one of its implementations called Mesa.

2.4 OpenGL

OpenGL is a standardized programming interface to hardware graphics, developed by Silicon Graphics Inc. [SGIN03]. Since its first release in 1992 it has been the most widespread 3D-graphics library available for both Unix and Windows systems. It defines an API that operates on a state machine. The state machine renders 3D graphics by drawing primitives into a frame buffer, controlled by a number of selectable modes.

OpenGL defines several hundreds of commands, which enable creation of objects and operations needed for (real-time) 3D interactive applications. These commands, defined as C-functions, operate on a graphics pipeline similar to the one illustrated in Figure 2.6. This pipeline is implemented as a large state-machine, which is optimized for efficient polygon and texture throughput. However, to maintain its platform independency, OpenGL does not define commands to perform windowing tasks and to obtain user input.

OpenGL allows objects to be constructed from geometric primitives, like points, lines, and polygons. The way these objects are rendered, depends on the current state of the OpenGL state machine. The state variables control things like the current color, the type of viewing transformation, drawing modes, shading methods, and which kind of lights are used.

This standard API and the functions that are performed by the aforementioned state machine is further elaborated on. First, Section 2.4.1 gives an overview of the structure of OpenGL. Second, Section 2.4.2 presents an OpenGL example in to illustrate what functionality an OpenGL implementation needs to perform.

2.4.1 The OpenGL pipeline

The dataflow graph as illustrated in Figure 2.10 illustrates the implementation of the rendering pipeline of OpenGL. It is a specific, and more detailed, implementation of the general pipeline shown in Figure 2.6. It shows two types of resources: object data and pixel data. Both types can be stored in a display list, which enables buffering and easy reuse of object and pixel data.

All object data in OpenGL is described by vertices. Therefore all object data passes through eval-
CHAPTER 2. REAL-TIME 3D-GRAPHICS RENDERING

Figure 2.10: OpenGL dataflow graph. Object data and pixel data are converted to frames in the frame buffer.

...uators to convert surfaces, like polygonal surfaces, to vertices. Then, in the 'per-vertex operations stage' all computations on vertices are performed. This stage consists of two parts. In the first part of this stage, all vertices (and their normals) are transformed to clipping coordinates. Their color and intensity is calculated from material properties and lighting. Furthermore, for all vertices that have textures applied to them the texture coordinates are calculated. In the second part of this stage, polygons are clipped and converted from the four-by-four matrix to 3D coordinates.

As opposed to the general 3D pipeline, pixel data is explicitly present in the OpenGL pipeline. The pixel data, consisting of 2D arrays of color values, are stored in a certain format outside the OpenGL machine. The pixel data needs to be unpacked and transformed (scaled, clamped, biased) before it can be used for rasterization or texture assembly. A Texture memory unit is added as a special resource for texture objects. This makes it, for instance, possible to access texture objects fast and efficient.

The rasterizer of OpenGL produces a series of color values using a 2D description of a point, line segment, or polygon, called fragments. Each fragment represents a portion of an object that corresponds to pixels in the frame buffer.

Each fragment can be modified by texture mapping, after which it is fed to the next stage that performs operations on individual fragments before they finally alter the frame buffer. Per-fragment operations include updates into the frame buffer conditional to incoming and previously stored depth values (to apply depth buffering), blending of incoming fragment colors with stored colors, as well as masking and other logical operations on fragment values.

The final result is written to the frame buffer, which is the 2D image of the 3D scene, viewed from a user defined point. The frame buffer can be displayed, transferred to system memory, or re-used by the rasterizer as pixel data.

2.4.2 The OpenGL interface

To control the OpenGL system described above, a set of commands have been defined that treat the pipeline as a state machine. Therefore, only changes in consecutive polygons have to be
2.5. MESA

communicated prior to sending polygons through the pipeline. To illustrate what this means, a
chunk of OpenGL code is presented here adopted from [WOOM99] (Chapter 1):

```c
#include <whateverYouNeed.h>

main () {
    InitializeAWindowPlease();
    glClearColor(0.0, 0.0, 0.0, 0.0);
    glClear(GL_COLOR_BUFFER_BIT);
    glColor3f(1.0, 1.0, 1.0);
    glOrtho(0.0, 1.0, 0.0, 1.0, -1.0, 1.0);
    glBegin(GL_POLYGON)
        glVertex3f(0.25, 0.25, 0);
        glVertex3f(0.75, 0.25, 0);
        glVertex3f(0.75, 0.75, 0);
        glVertex3f(0.25, 0.75, 0);
    glEnd(GL_POLYGON)
    glFlush ();
    UpdateTheWindowAndCheckForEvents();
}
```

All OpenGL functions contain a prefix gl. This piece of code draws one white, square-shaped
polygon in a window. The polygon is defined between the glBegin and glEnd commands.
Its color is defined by setting the state of the current color to white with the OpenGL function
`glColor3f` before defining the polygon. The function `glClear` fills the frame buffer with
the color defined by `glClearColor`.

Furthermore, the function `glOrtho` specifies the coordinate system OpenGL assumes and
`glFlush` makes sure the object data is sent through the rendering pipeline, instead of being
buffered. The non-OpenGL functions `InitializeAWindowPlease` and `UpdateTheWindowAndCheckForEvents` are added to illustrate that platform specific window functions and
functions to handle user-interaction are not included in the OpenGL standard.

A full specification of all OpenGL commands can be found in [SEGA99].

2.5 Mesa

Multiple implementations of the OpenGL standard are available: Windows and MacOS use their
own OpenGL library. Silicon Graphics has created an OpenGL implementation called SGI-
OpenGL [SGIO03] and is involved in the development of GLX, an OpenGL implementation for
the X-Windows system [GILES03]. Furthermore, a partial implementation of the OpenGL stan-
dard, called MiniGL, is currently under development for the Palm Computing platform [SHER03].

Mesa is an OpenGL implementation, that is freely available for download [PAUL03]. Although
its authors do not make any claim being OpenGL compliant, due to licensing issues, it passes
OpenGL's conformance tests. Mesa is used for this project for three main reasons:
1. Mesa supports multi-threading,
2. Mesa contains a full software implementation,
3. Mesa is easily portable to new platforms.

In this section, Mesa's implementation details, which are relevant for this project, will be discussed. First, the implementation of multi-threading is described, followed by the way vertices are passed through the graphics pipeline. Furthermore, some general design concepts of Mesa are discussed that make it portable and flexible.

The state machine defined by OpenGL is implemented in Mesa in the form of a Context. This Context is a large structure containing all state variables, pointers to textures and buffers. OpenGL commands effectively operate on the state machine by setting the state variables of the Context. Because the OpenGL commands make no reference to a context or state machine, Mesa's context is defined globally. When using multiple threads this causes a problem, because several threads would access and change the context simultaneously, leading to undesired results. To solve this a context for each thread is to be created by Mesa. The Contexts are kept local for each thread, meaning that each thread can only access one Context.

To improve the performance of the geometry traversal through the graphics pipeline, Mesa uses a buffer to temporarily store vertices. This vertex buffer reduces the overhead of polygons for traversal between individual stages of the pipeline. The vertex buffer is only sent through the pipeline, if the state of the state machine needs to be changed, or if the vertex buffer is full. Sending a vertex buffer through the pipeline is called flushing.

The vertex buffer contains all vertex properties required for rasterization, as illustrated in Figure 2.11. Some of these properties are set before the buffer is flushed, like the ObjPtr and NormalPtr, which contain respectively the vertex position and normal, both expressed in World-coordinates. Other vertex properties are calculated when the buffer content traverses through each stage of Mesa's graphics pipeline. The traversing of the vertex buffer content through the pipeline is illustrated in Figure 2.12. Furthermore for each pipeline stage an approximation of the computational complexity is sketched in Table 2.5. This table gives some insight in the amount of mathematical operations that are required for 3D-graphics rendering.

Mesa is written such that a custom pipeline can "easily" be created, making it possible to merge stages that can be performed in hardware or to add extra stages like the support for vertex programs (customized operations on vertices) and custom shaders. Furthermore, Mesa can be modified by adding a device driver for hardware accelerators.

![Figure 2.11: The vertex buffer contents of Mesa.](image-url)
### Pipeline stage | Description of tasks | Calculations
--- | --- | ---
Vertex Transformation | A multiplication of a 4D-vector with a 4x4 transformation matrix for each vertex. Determine for all vertices if it is visible on screen. (requires calculation of an inverse matrix) | $16m + 12a$  
$78m + 6d + 54a$
Normal Transformation | A multiplication of a 4D-vector with a 4x4 transformation matrix for each vertex and a normalization of its length. Normalization of each vertex' normal pointer. | $16m + 12a$  
$6m + 1d + 2a$
Vertex Lighting | For each vertex, for each light, calculation of specular and diffuse vertex intensity for both front and back color. | $34m + 4d + 33a$
Fog | For each vertex a fog coordinate is calculated | $3m + 2a$
Texture Generation | For each vertex texture coordinates for all texture units are calculated from normal vector, vertex coordinated and camera angel. Sphere map calculation is assumed here. | $6m + 6a$
Texture Transformation | A multiplication of a 4D-vector with a 4x4 transformation matrix for each vertex. | $16m + 12a$
Point Attenuation | Calculation of the point size for vertex points. | $2m + 2d + 1a$
Clipping and Rasterization | Calculation of step sizes (related to pixel size) for all calculated coordinates. For all polygons visible surface determination, clipping, and rasterization needs to be performed. The number of operations and calculations varies significantly based with parameters listed above. | Entirely dependent on the results calculated above. Approximation of average: $200m + 60d + 300a$

Table 2.1: Approximation of the computations involved with 3D-graphics rendering. Note that depending on the current state, the amount of textures and their size, the number of lights used, the type of shading, etc. the computations for each stage can vary significantly. Furthermore, this table should be seen as the minimum number of computations required. $m$: multiplication, $d$: division, $a$: addition or subtraction.
The application designer can easily reach the computational limit of current hardware implementations, by increasing the level of realism, using more polygons, larger and more textures, or more advanced features. This also implies that the performance of 3D-graphics rendering is highly application specific.
Chapter 3

Multi-processing architectures

A main conclusion that can be drawn from Chapter 2 is that 3D-graphics rendering requires a high computational performance. To increase the performance of a computer, two basic techniques can be used: implementing the machines in faster technology (which allows running the processor at a higher clock frequency) and performing more operations in parallel. These techniques are not mutually exclusive, but as semiconductor technology has matured, it has become apparent that more parallelism must be exploited in order to keep increasing the system performance [LILJ92]. A wide variety of architectures have been proposed that attempt to exploit the parallelism available in an application at different granularities. But in general two levels of parallelism are discriminated: Instruction-Level Parallelism (ILP) and Thread-Level Parallelism (also called Task-Level Parallelism) (TLP). For example vector and multiple instruction issuing processors, such as the superscalar and VLIW (Very Large Instruction Word) machines, exploiting the fine-grained parallelism available at the machine's instruction level, correspond to ILP. In contrast, shared memory multiprocessors, which typically exploit coarse-grained parallelism by distributing entire loop iterations or entire tasks to different processors, correspond to TLP.

Multiprocessing is the coordinated processing of programs by more than one computer or processor. Multiprocessing is a general term that means the assignment of a program to multiple processors working simultaneously on a single computer, or involves multiple computers working on the same program at the same time. Therefore, multiprocessing exploits possible parallelism of applications to increase their performance. This can either be an increase in speed, or a decrease in power consumption.

Every multiprocessing system consists of processing units, memory and some form of interconnect that enables communication between the processing units and their memories. The terms “platform” and “architecture” are overloaded terms often used in hardware and software contexts. In this report, an architecture is a generic collection of hardware elements, such as processors, memories and storage devices etc. assembled in a specific order. A specific instance of an architecture is called a hardware platform in this report. In order to run a program on multiple processing units (i.e. processors or computers) a program has to be split in parts that can be executed independently. In this report, these parts will be referred to as tasks or jobs.

The speedup of a multiprocessing architecture can in theory be equal to the number of processors available. In practice, however, the speedup will be lower, as will be shown in this chapter.
Furthermore, several processor architectures that exploit parallelism will be discussed. The use of ILP is discussed only very briefly. Most attention will be paid to maximizing the exploitation of coarse-grained parallelism. First, aspects of parallelism will be discussed in Section 3.1, followed by an overview of classifications of multiprocessor architectures. Each of these parallel architectures have significant differences in synchronization overhead, instruction scheduling constraints, memory latencies, and implementation details, as will be discussed in Section 3.3.

These differences in architectures, result in different performance gains that are obtained from the parallelism. In Section 3.2, the factors that limit the performance gain and ways to calculate this gain will be described.

In Section 3.4, the multiprocessor architecture adopted for this project is introduced. Finally, Section 3.5 will describe briefly the API used to model parallelism, called Pthreads.

3.1 Parallelism

Parallelism in computer systems is the execution of program instructions by distributing them among multiple processing units with the objective of running a program in less time or by using less power. Although this seems straightforward, it is quite challenging to partition an application such that the performance gain is maximized, particularly if the processing is very complex and data dependent. Choices have to be made at what granularity parallelism is to be exploited, and who is responsible for detecting possible parallelism: automatically by the compiler, or should the programmer explicitly model parallelism? In this section, different aspects of parallelism will be discussed.

In the parallelization process four tasks can be discriminated as depicted in Figure 3.1[THOM00]. These four steps form an iterative process, which re-adjusts the partitioning to an optimal parallel program.

Decomposition
The first step in creating a parallel application is breaking up the computations in tasks or units of work. Tasks can be statically created at configuration-time, or can become available dynamically at run-time. The concurrency of the tasks has to be identified at this point, which depends on the level of granularity the parallelism is to be exploited. If many small tasks are created, this often results in a large number of interdependencies, but good load balancing. If a few large tasks are created load imbalance becomes an issue, but interdependencies becomes less. Furthermore, at this stage, the method for partitioning the application needs to be chosen: functional partitioning, data partitioning, or a combination of both. Advantages and disadvantages of these methods are discussed below.

Assignment
A mechanism is required for assigning tasks to processes, to try to balance workload and minimize synchronization and communication costs. The tasks need to be clustered into processes thereby optimizing the communication overhead between tasks and the load balancing. Communication (and synchronization) between processes should be minimal and each process should finish in approximately the same number of cycles. These aspects depend on both the tasks that are clustered as processes, and the underlying system architecture.
3.1. PARALLELISM

Orchestration deals with the implementation details of the programming models and system architecture. Its goal is to implement the communication and coordination between processes as efficiently as possible for all levels of parallelism and to provide means for shared data access. For example, some form of scheduling of jobs and communication between jobs has to be implemented.

Mapping

Mapping is the assignment of processes to processors. Often each process is assigned to a single processor. However, it might be beneficial to fold a process on multiple processors, for instance if there are more processors available than processes. In this case, the process of Decomposition, Assignment, and Orchestration is repeated.

3.1.1 Instruction-level parallelism

The parallelization process described above is applicable for parallelism on instruction level as well as parallelism on a thread level. Processors that exploit Instruction-Level Parallelism (ILP) allow independent instructions in an instruction stream to be executed in parallel. The easiest way to execute instructions in parallel is the usage of pipelining. Pipelining allows the different stages of an instruction's execution to be overlapped with those of other instructions. At best, pipelining results in the execution of one instruction each cycle on each processor. To improve
the performance further, multiple pipelines executing in parallel can be used. This provides an opportunity for mutually independent instructions to be grouped and executed in parallel. The maximum number of instructions that are being executed in parallel is limited by the number of parallel pipelines and the maximum number independent instructions that can be grouped. Determining the interdependencies of instructions and scheduling them over the available execution units, can be done by the compiler, on a VLIW (Very Large Instruction Word) architecture, or at run-time by the hardware architecture itself, on a superscalar processors.

A VLIW architecture executes a sequence of long instruction words, and issues them to a set of execution units based on the position of each instruction in the long instruction word as illustrated in Figure 3.2. The superscalar processor executes a sequential list of instructions, dynamically selecting instructions for issue to the different execution units. Unlike the VLIW processor, it dynamically determines dependencies between the instructions, and schedules them at run-time.

### 3.1.2 Thread-level parallelism

Because the exploitation of ILP on general-purpose architectures is limited to a few instructions, due to their interdependencies, a complementary technique is often used that exploits Thread Level Parallelism (TLP). Instead of trying to exploit parallelism in a single instruction stream, TLP allows the execution of multiple instruction streams (i.e. threads) in parallel, making parallelism at a higher abstraction level possible. Multiple streams of instructions in parallel are suitable for multiprocessing architectures.

TLP can not be automatically determined in general. Therefore, it needs to be modeled explicitly by the programmer.

### 3.1.3 Implementing parallelism

Independent of the type of parallelism, it is important to know in advance what type of parallelism should be exploited and at what granularity. From a programmers point of view it is important to know whether parallelism has to be explicitly modeled at algorithm design-time, or if it is
3.2. PERFORMANCE ANALYSIS

Implicit parallelism

Programmer
Sequential source code
Parallelizing compiler
Parallel object code
Execution by run-time system

Explicit parallelism

Programmer
Explicitly modeled parallelism
Parallel preserving compiler
Parallel object code
Execution by run-time system

Figure 3.3: Implicitly versus explicitly modeled parallelism.

implicitly determined at compile-time and/or run-time. The differences in design paths are shown in Figure 3.3.

Implicit parallelism allows the programmer to write conventional or sequential code, burdening the compiler with creating independent parallel tasks. Contrarily, explicit parallelism burdens the programmer with defining processes, their intercommunication, and synchronization. For explicit parallelism, the compiler has to preserve these interdependencies to be able to generate concurrent object code. In general, explicit parallelism results in parallelism on a higher abstraction level of the software implementation compared to implicit parallelism, creating a higher potential performance gain. For this project, explicit parallelism is used, with the help of an API called Pthreads as discussed in more detail in Section 3.5.

3.2 Performance analysis

In this section, the common performance measures of a parallel program are described. Definitions are presented which make it possible to compare different parallel implementations of an algorithm and to calculate the performance gain of parallel programs. Performance of parallel programs is generally not measured in MIPS (Million Instructions Per Second) or MFLOPS (Million Floating-point Operations Per Second). Instead the metrics speedup and efficiency are used. Speedup describes how much performance gain is achieved by parallelizing a program. Efficiency indicates the fraction of time the processors are doing useful work. How these measures can be
determined shall be illustrated below.

Consider a platform with \( n \) processing units executing a task with a workload of \( W \). If the task is performed in \( T_n \) seconds, the speedup \( S_n \) is said to be

\[
S_n = \frac{T_s}{T_n},
\]

with \( T_s \) the time it takes for the application to run if it would be executed on one processor. Note that \( T_s \) is not the same as the time for the parallel algorithm to be executed on one processor, because parallel implementations often contain additional functionality to accommodate the parallelism.

Due to the fact that a program generally contains some inherent sequential parts, parallelism rarely scales linearly with the number of processing units. Amdahl [AMDA67] showed that the maximum obtainable speedup is limited by these sequential parts. This is defined by Amdahl's law

\[
S_n = \frac{T_s}{T_n} = \frac{s + p}{s + \frac{p}{n}},
\]

where \( s \) is the sequential portion of the program and \( p \) is the parallel portion (identified on one processor). Amdahl's law is used as an estimate of the maximum obtainable speedup of an algorithm. To illustrate this, assume workload \( W \) consists of an inherent sequential part \( \alpha \) (with \( \alpha \) between 0 and 1) and thus a parallel part of \( 1 - \alpha \). This results in a maximum obtainable speedup of

\[
S_n = \frac{1}{\alpha + \frac{1-\alpha}{n}}.
\]

From Equation 3.3 can be concluded that even for an infinite number of processing units \( (n = \infty) \) the maximum obtainable speedup is limited to \( S_n = \frac{1}{\alpha} \). Note that Amdahl's law is an estimate; it does not take issues such as increased problem size and cache performance into account. Speedup theoretically lies between 0 and \( n \), with \( n \) being the ideal speedup. However, in ideal cases even larger speedups can occur due to improved cache and register performance [TOLE03].

Another measure of parallel performance is efficiency. Efficiency \( E_n \) is closely related to speedup. It is defined by

\[
E_n = \frac{S_n}{n}.
\]

The efficiency is a number between 0 and 1, illustrating what fraction of time a processing unit is spending on \( W \). The rest of the time is called overhead. Overhead can amongst other things be caused by scheduling, memory latency, inter-processor communication, overhead due to adaptation of the algorithm for parallel execution, load imbalance, memory congestion, and synchronization.

For this research project the following types of overhead are particularly important:
3.3. ARCHITECTURES FOR PARALLELISM

Parallel overhead
Execution time of additional functionality to facilitate parallel execution. This includes the time spent on communication between processors.

Memory congestion
Extra clock cycles needed for retrieving data from memory due to increased traffic on the memory bus. The penalty of increased memory traffic is often difficult to measure, especially if the compiler takes latencies into account during the scheduling of instructions.

Load imbalance
Idle cycles of processing units due to synchronization are counted as load imbalance. The execution time of processing units often differs because of different tasks with different cycle requirements, e.g. data dependent execution times, and differences in memory latency. If synchronization is required during program execution, processing units wait on each other to finish.

Often trade-offs between different types of overhead have to be made during the development of a parallel application. For instance, communication overhead can be minimized by running the application on one processing unit. However, this results in poor load balance. Adversely, load balance can be optimized by randomly assigning very small tasks to processing units. However, this leads to an increased communication overhead and possibly memory congestion. For an optimal parallelization, enough concurrency needs to be identified at the adequate level of granularity. The "adequate" level of granularity greatly depends on the hardware architecture. A classification of hardware architectures will therefore be presented in the next section.

3.3 Architectures for parallelism

This section presents an overview of different hardware systems that exploit parallelism in different ways using the four main computer architectures, as discriminated by Flynn [FLYN66]. Flynn based his classification on two taxonomies, the amount of independent instruction paths and the amount of independent data paths. The four classes are: SISD Single Instruction Single Data, SIMD Single Instruction Multiple Data, MISD Multiple Instruction Single Data, and MIMD Multiple Instruction Multiple Data. Each will be described in more detail here.

![Figure 3.4: SISD architecture.](image)

SISD
A SISD architecture, as depicted in Figure 3.4, consists of a single processing unit receiving a single instruction stream and which operates on a single stream of data. At each step, the control unit emits one instruction to the processing units which operates on data obtained
CHAPTER 3. MULTI-PROCESSING ARCHITECTURES

from the memory unit. Almost all non-supercomputers in use today adhere to this model invented by John von Neumann in the late fourties. An algorithm that runs on a SISD computer is said to be sequential (or serial). The “single” in single instruction does not mean that there is only one instruction unit, but rather that there is only one instruction stream. Instructions can be executed in parallel within the single processing unit by multiple execution units to try to obtain the maximum throughput of one instruction every clock cycle. In other words: A SISD architecture can exploit ILP. Because each instruction has a unique place in the execution stream, and thus a unique time at which it is executed, the entire execution is said to be deterministic. The SISD architecture is limited by the number of instructions that can be issued in a given unit of time.

Figure 3.5: MISD architecture (see Figure 3.4 for abbreviations).

Figure 3.6: SIMD architecture (see Figure 3.4 for abbreviations).

MISD

To obtain a higher system performance, the architecture can be extended by adding instruction streams or data streams, which results in the MISD or SIMD architecture respectively, illustrated in Figures 3.5 and 3.6. The MISD architecture can perform multiple operations on a single stream of data in parallel. This type of parallelism is very difficult to exploit
3.3. ARCHITECTURES FOR PARALLELISM

to general purpose processing and has therefore, been rarely implemented. Hence this architecture will not be discussed any further.

SIMD

SIMD architectures are machines capable of applying the exact same instruction stream to multiple streams of data simultaneously. For certain classes of problems, such as data-parallel problems, this type of architecture is perfectly suited to achieve very high processing throughput. It can be split into many independent parallel parts. Consequently multiple instruction units can process these parts in parallel. Often the processors on this platform run synchronously, meaning that the processors receive the same instruction at the same time, and thus all will potentially be able to execute the same operation simultaneously. SIMD architectures are deterministic because at any one point in time, there is only one instruction being executed, even though multiple units may be executing it. So, if the same program is running every time, with the same number of execution units, exactly the same result is guaranteed at every step in the process. In general the SIMD architecture is well-suited for both instruction- and task-level parallelism. Examples of such architectures are Intel Pentium processor family [KAGA97], The DEC Alpha processor [DECA03], and the MasPar [NICK90].

MIMD

Most of the existing multiprocessing architectures fall into the MIMD category. Each processor executes its own instruction stream in its unique data stream. In order to coordinate tasks of multiple processors working on a different part of the same problem in MIMD architectures, some form of inter-processor communication is required to convey information and data between processors and to synchronize activities. Two types of inter-processor communication are discriminated: communication via shared memory and message passing. If shared memory is used for inter-processor communication, the term multiprocessor architecture is used to denote the architecture, if message passing is used for communication and the system does not contain a shared memory, but rather contains a distributed memory, the term multi-computer architecture is used. In multi-computer architectures, multiple processors operate independently and each has its own private memory. Data is shared across

Figure 3.7: MIMD architecture (see Figure 3.4 for abbreviations).
a communication network using message passing. Memory and bandwidth are scalable with the number of nodes but data sharing among tasks is slow due to the latency of the interconnection network. In a multiprocessor architecture, multiple processors operate independently but share the same memory resources at the highest level. Only one processor can access a shared memory location at the same time. Data sharing among tasks depends on the speed of memory access and can be fast if e.g. the number of processors is small, and if the memory bandwidth is sufficient. However, an increase of processors without increasing the memory bandwidth can create a bottleneck. Multi-processor architectures can be sub-divided further, dependent upon their shared memory model as illustrated in Figure 3.8. Basically, three memory models, can be distinguished:

Uniform memory-access (UMA) model
The UMA model consists of multiple processors, which can all have their own cache, connected to a main memory as depicted in Figure 3.9.
this architecture is the increasing bandwidth to the shared memory for an increasing number of processors. This makes the UMA model suitable for only a limited number of processors. The term SMP (Symmetrical Multi-Processing) is used if all memory latencies are equal for each processor.

**Non-uniform-memory-access (NUMA) model**

The NUMA model is a multiprocessing network with a more complex memory model. The access time to the memory for each processor depends on the memory address. As depicted in Figure 3.10, two types of NUMA models exist; the local shared memory model and the hierarchical model. In the local shared memory model each processor can access its own local memory or a remote memory with some extra latency due to the interconnect. The remote memory of one processor is a local memory of another processor. The hierarchical model consists of multiple UMA models connected with each other through a global interconnect network. Each processor can access a shared memory in its own cluster and a shared global memory, connected to the global interconnect network. Both NUMA models allow for better scalability compared to the UMA model.

![Figure 3.10: The NUMA models.](image)

**Cache-only memory (COMA) model**

The COMA model, as illustrated in Figure 3.11, is a special case of the NUMA model. It uses a cache instead of "normal" memory. To localize data in the cache additional hardware is required in the form of directories, or a broadcasting system that tells each processor the intentions of a processor when it operates on data. By using caches, the impact of frequent long-latency accesses is automatically reduced by replicating and migrating data across memory modules. However, maintaining the coherency of the replications is a complex problem.

Also other models of multi-processor systems exist, based on a combination of the aforementioned models. However, each multiprocessing architecture has its typical communication speed and memory latency, which determines the optimal granularity of parallelism to be used. In general,
the more loosely coupled the processor architecture is, the coarser the granularity of parallelism should be. In the next section the hardware architecture, used for this project, will be elaborated on.

3.4 SpaceCAKE

SpaceCAKE [STRAV00] is a multiprocessor architecture that addresses two main trends in the design of SOCs (Systems On Chips). Firstly, the design complexity of systems doubles roughly every 18 months according to Moore’s Law. Secondly, the silicon area used for memory increases relative to the area used for computational structures. The first trend implies that the number of transistors on a chip grows exponentially. To utilize these transistors effectively parallelism should be exploited on all levels; i.e. pipelining, ILP and TLP. The SpaceCAKE architectures allows all these levels of parallelism to be exploited simultaneously.

The growing dominance of memory on a chip allows many types of computational units close to memory banks. This naturally leads to the clustering of a set of computational units, accessing the same memory module. The SpaceCAKE architecture complies to this layout where the clusters are referred to as tiles. Multiple interconnected tiles can be defined. All tiles are identical, which provides scalability and a higher production yield at the cost of hardware redundancy.

The SpaceCAKE architecture will be described in Section 3.4.1 focusing on the aspects that are of importance for this project. Its development environment will be discussed in Section 3.4.2.

3.4.1 Architecture

A SpaceCAKE tile consists of CPUs, caches, SPUs, memory banks, and additional hardware to enable communication with other tiles, connected to each other via a local interconnect as illustrated in Figure 3.12. A tile consists of different types of processors: CPUs and SPUs. Such a
3.4. SPACECAKE

Figure 3.12: SpaceCAKE tile architecture.

The focus for this project is targeted at the configuration of the Wasabi chip [WASA03], an instance of the SpaceCAKE architecture. The Wasabi chip consists of one tile with one MIPS core assisted by eight TriMedia VLIW cores [TRIM03], sharing 12 Mbytes of L2 cache. The Wasabi
is an instance of a COMA.

Both MIPS and TriMedia processor-cores exploit pipelining for intra-instruction parallelism. ILP
is exploited using the superscalar strategy on the MIPS and the VLIW strategy on the TriMedia
processors. Inter-tile TLP can be exploited by modeling parallelism using one of several multi­
threading APIs. For this project only the TriMedia processor-cores are used (not the available
MIPS-core). Furthermore, all effort was put in exploiting TLP, exploitation of ILP is left to the
compiler. Exploiting application level parallelism by static mapping on multiple tiles has not been
investigated, because only one tile has been used.

3.4.2 Development environment

As mentioned in Section 3.1.3, TLP needs to be modeled by the programmer. SpaceCAKE allows
several APIs to be used such as YAPI [KOCKO1] and Pthreads [NIC96]. The programmer has
to create processes that are to be executed independently. Processes should be designed such that
the optimum trade-off between inter-process communication, load balance, and synchronization
overhead is realized as described earlier in Section 3.1. Pthreads are used to create our multi­
threaded application. Details of Pthreads are described in the next section.

To execute a multi-threaded application the SpaceCAKE simulator, named cakesim, is used.
The number of MIPS processors and TriMedia processors can be defined at start-up. Object files
for both type of processors are loaded and run. Additionally, options can be set to generate a
ProView statistics file [TOLE00] at the end of a simulation run. ProView is a tool for visualizing
information on instruction and data cache miss-rates, and execution efficiency on a function level
for each TriMedia processor together with information on the memory bandwidth within each tile.

Debugging with SpaceCAKE is possible though cumbersome. Debugging can be enabled with a
start-up option, which results in the logging of each instruction cycle for each processor to a file.
To find errors in these log files can be tedious.

3.5 Pthreads

The Pthreads API is defined in the ANSI/IEEE POSIX 1003.1 standard. The standard is defined
in the C language as a set of functions for managing threads, mutexes\(^1\), conditional variables and
thread attributes. In this section the Pthread functions, relevant for this project, are enumerated.
For a detailed description of Pthread see [NIC96].

\[
\text{pthreads.create(thread, attributes, start_routine, arguments)} \text{ This routine creates a new thread and makes it executable. It runs the function defined by} \\
\text{start_routine. The arguments of the start routine are stored in arguments. Thread} \\
\text{attributes, such as priority and stack size, are contained in attributes.}
\]

\[
\text{pthreads.exit(status)} \text{ This function terminates the thread that calls this function. Threads} \\
\text{can also terminate if the function returns. A termination status may be specified, which is} \\
\text{stored as a void pointer, for any thread that may join the calling thread.}
\]

\(^{1}\)"Mutex" is a shortened form of the words "mutual exclusion".
3.5. PTHREADS

pthread_join(threadid, status) Joining of threads is a way to accomplish synchronization. The routine blocks the calling thread until the thread specified by threadid returns.

pthread_mutex_lock(mutex) and pthread_mutex_unlock(mutex) The first routine is used by threads to acquire a lock on the specified mutex variable. The basic concept of a mutex, as used in Pthreads, is that only one thread can lock (or own) a mutex at any given time. Thus, even if several threads try to lock a mutex only one thread will be successful. No other threads can own that mutex, until the owning thread unlocks that mutex with pthread_mutex_unlock(), and therefore block, effectively preventing simultaneous access to shared data.

pthread_cond_wait(condition, mutex) and pthread_cond_signal(condition) These routines use conditional variables. A conditional variable is used in conjunction with a mutex lock and a predicate (typically a Boolean variable) to allow one thread to signal (or wake up) other threads which are waiting on that condition variable. This is useful for often encountered situations where a thread needs to wait on a certain condition to be met before it can proceed. A typical situation would be where one thread requires data from another thread before it can proceed.

The pthread_cond_wait() blocks a thread until the specified condition is signaled. This routine should be called while mutex is locked, and it will automatically release the mutex while it waits. The pthread_cond_signal() routine is used to signal (or wake up) another thread which is waiting on the condition variable. It should be called after mutex is locked. The pthread_cond_broadcast() routine should be used instead of pthread_cond_signal() if more than one thread is in a blocking wait state.

pthread_getspecific(key) and pthread_setspecific(key, value) With these functions thread specific data can be stored and resolved. Data stored in value can be associated with key by pthread_setspecific() for each thread and retrieved with pthread_getspecific(). Thread specific data is useful when threads want to associate their specific data to a global variable.
Chapter 4

Parallelization methods

Before discussing how Mesa has been customized to enable its 3D-graphics functionality to run in parallel on multiple TriMedia processors, an overview is given on previous efforts on parallel 3D-graphics rendering. The research presented in this chapter is the result of a short literature survey in this area.

Section 4.1 discusses the comparison between data partitioning and functional partitioning. Advantages and disadvantages of both methods will be discussed. A classification of data partitioning solutions for 3D-graphics rendering will be presented in Section 4.2.

4.1 Data partitioning versus functional partitioning

Previous research on running 3D-graphics in parallel can be categorized in functional partitioning and data partitioning. As illustrated in Figure 4.1, both partitioning schemes can be subdivided further.

Functional partitioning can be subdivided further into two sub-categories: pipeline partitioning and custom partitioning. With pipeline partitioning, tasks directly correspond to the different pipeline stages of the 3D-graphics pipeline. Custom partitioning groups functionality to form tasks that minimize the overall parallel overhead and load imbalance. However, if the number of pipeline stages is adjusted to match the number of parallel processing units, this classification becomes less clear. In general, functional pipeline partitioning is applied for heterogeneous multi-processor systems, where each processing unit is optimized for its assigned pipeline stage [KIRK90].

Two classifications of data partitioning methods are illustrated in Figure 4.1, partitioning of the object-space, and partitioning of the image-space. Object-space partitioning distributes polygons based on their processing order, which is dictated by the display traversal, or on the location of objects in 3D space. Examples of these algorithms can be found in [FRAN90], [CLEA83], [DIPP84], and [CASP89].

The image-space partitioning methods are further divided by discriminating the method of pixel grouping. Pixels can be grouped by display area, e.g. rows, columns or rectangular areas, or they can be grouped by polygon spans, i.e. arrays of pixels between polygon edges. Examples of sev-
Data Partitioning versus Functional Partitioning

Figure 4.1: A classification of partitioning methods for 3D-graphics rendering.

3D-graphics rendering is often implemented as a multi-staged pipeline, because many operations on geometry and pixels are performed consecutively. Therefore, custom functional partitioning based on minimizing overhead, seems to be a good solution. A pipeline based functional partitioning seems ideal for a hardware implementation. However, two disadvantages of functional partitioning with respect to data partitioning are the lack of scalability, and a limit on the throughput.

To illustrate the scalability issue, consider a 3D-graphics pipeline that consists of eight equally computationally complex pipeline stages. For functional partitioning the pipeline is partitioned amongst processing units, whereas for data partitioning each processing unit contains all functionality and operates on only part of the data. This is illustrated in Figure 4.2 for four processing units.

If the performance of this platform is increased by adding a fifth processing unit, functional partitioning requires a re-partitioning of the 3D-graphics application. Since the eight pipeline stages do not fit optimally on five processing units, load imbalance will occur. Contrarily, data partitioning requires only the data to be re-partitioned to five instead of four processing units.

Data partitioning has the disadvantage that additional functionality is needed to distribute data amongst processing units and correctly combine data after processing. Furthermore, possible data dependencies can be present between jobs, resulting in additional communication. However, for data partitioning all communication between functional tasks is omitted compared to functional partitioning.

Since scalability is an important criteria for this project, data partitioning is preferred. In the next section, data partitioning methods for 3D-graphics rendering will be discussed in more detail by presenting another classification method. This classification groups partitioning methods based...
CHAPTER 4. PARALLELIZATION METHODS

4.2 Sorting methods

In the literature (e.g. [MOLN94], [SAMA99], [MITR98]), three classes of data partitioning methods are discriminated. To get a clear view of these different data partitioning schemes, the general three-staged pipeline is illustrated in Figure 4.3.

For all three schemes one rendering pipeline for each processing unit is present, which processes a subset of the geometry data. Somewhere in this rendering pipeline, data has to be distributed on the location in the pipeline, where data is partitioned. This classification is more applicable for pipelined implementations of 3D-graphics rendering.
4.2. SORTING METHODS

amongst processors, and the output of all processing units has to be combined to form the final image. Furthermore, the geometry data needs to be sorted to coordinate the distribution and combination of the subsets. The position in the pipeline, where geometry data is sorted, categorizes the three schemes. Sorting can be done at three places in the pipeline: After the display traversal (Sort First), after the transformation stage (Sort Middle) or at the end of the pipeline (Sort Last). These sorting methods will be described in Sections 4.2.1, 4.2.2, and 4.2.3, respectively.

4.2.1 Sort-First

The Sort-First method sorts geometry data, i.e. polygons, based on their coordinates in world space. An example shown in Figure 4.4 clarifies this. The geometry is sorted early in the pipeline before geometry data is transformed and rasterized. The first challenge for using this method is to divide the world, possibly dynamically, in at least $n$ segments, in such a way that all $n$ processors complete their tasks at approximately the same time. Furthermore, the resulting image fragments from each processing unit, have to be combined somehow to construct the final 2D image.

This method has the advantage that frame to frame coherence can be exploited. This requires an adaptive algorithm to divide the world-space and allows only a limited change in geometry data at each consecutive frame. Because this condition is generally not met, especially not in many 3D games, this method is not considered to be an acceptable approach, often resulting in extreme load imbalance.

4.2.2 Sort-Middle

The Sort-Middle method sorts geometry data based on their position on the screen as depicted in Figure 4.5. Because the screen coordinates of geometry data is not known in advance, the geometry transformation has to be done before data partitioning can be applied. Combining the resulting image fragments afterwards is straightforwardly achieved by assigning the output frame buffer space of each screen segment to one processing unit.
Figure 4.4: Subdivision of geometry using the Sort-First method. Colors indicate different data partitions.

Figure 4.5: Subdivision of geometry using the Sort-Middle method. Colors indicate different data partitions.

Although the geometry data is processed sequentially until after the transformation stage, this method is considered to be a natural way to partition the workload. Furthermore, the transformation stage typically requires only a relatively small part of the amount of computations involved. Load imbalance is still a problem when using this method, because the rasterization of equally sized screen segments can differ considerably in computational complexity.

4.2.3 Sort-Last

The Sort-Last method distributes geometry data equally amongst rendering pipelines, without sorting them first. Often, schemes are used that create small groups of polygons (belonging to the same object) in the order they are presented. Figure 4.6 illustrates the Sort Last method. Each
4.2. SORTING METHODS

A group of polygons is sent to a processing unit, which is selected according to a round-robin or first-come-first-serve scheme. When all geometry of a frame has been processed, each processing unit has created a number of image fragments that have to be combined to create the final image.

Figure 4.6: Subdivision of geometry using the Sort-Last method. Colors indicate different partitions.

Molnar et al. [MOLN94] have shown that the Sort-Last method provides the best load balance, despite the fact that the time required to rasterize a single polygon can vary. A disadvantage of this method is the quadratic growth of the number of computations required for merging image segments for increasing screen resolutions. The Sort-Middle and Sort-Last method are investigated for the implementation of Mesa on the SpaceCAKE platform. The next chapter discusses this implementation in more detail.

Load imbalance for all three sorting methods can be reduced, by creating many small jobs (relative to the number of processing units) at the cost of additional communication. For instance, for the Sort-First method many small 3D-world segments can be defined that are scheduled amongst processing units, at the cost of additional computations for merging frame segments. Similarly, the Sort-Middle method can sub-divide the output frame into a large number of screen segments, which are subsequently distributed adaptive amongst processing units, at the cost of additional clipping at screen segment borders and an increase in memory usage. For the Sort-Last method the granularity of the polygon distribution can easily be adjusted. The optimal choice for this granularity, i.e. grouping polygons such that minimum overhead is achieved, is often highly dependent on the 3D-graphics implementation.
Chapter 5

Implementation of Mesa on SpaceCAKE

In the previous chapter, a general overview of three data partitioning methods has been presented. This chapter discusses the implementation of two of these methods, Sort-Middle and Sort-Last, by combining the research presented in the previous chapters. This chapter elaborates on how a customized, i.e. parallelized, implementation of Mesa for the SpaceCAKE architecture has been developed. Furthermore, it explains the design steps required to run a test application, exploiting the implemented parallelism.

Development of the test application, to extract performance data, together with a parallel implementation of Mesa, can be separated into three different design problems:

- Porting Mesa to the TriMedia architecture,
- Creating a test application, and
- Implementing a sorting method.

The porting of Mesa to the TriMedia architecture is elaborated in Section 5.1. Section 5.2 describes the creation process of an application to test custom Mesa implementations on the SpaceCAKE platform. Section 5.3 discusses the implementation of the partitioning in Mesa using the Sort-Middle and Sort-Last methods.

5.1 Porting Mesa to the TriMedia platform.

The Mesa source code already runs on a wide range of hardware platforms for various operating systems including Unix, Linux, BeOS, AmigaOS and Windows. Furthermore, Mesa includes drivers for graphic cards, which use the 3Dfx chip-sets and optimizations for processor architectures, such as the Intel's x86. Support for all these platforms shows that Mesa is easily ported towards an architecture.

The porting of Mesa entails compiling a sub-set of all Mesa source files, such that only the software implementation is compiled. The lack of support for double-precision floating-point num-
5.2. CREATING AN APPLICATION FOR MESA.

bers on a TriMedia is the only limitation of the TriMedia implementation of Mesa. Although this puts a restraint on the application programmer, this limitation is beyond the scope of this report.

As mentioned in Section 2.4, OpenGL, and therefore Mesa, does not include any functions for addressing displays or input devices to avoid platform dependencies. For Mesa, an additional platform-specific library called “glut” exists to provide user-interaction and display functionality. Because porting of glut to the SpaceCAKE platform is time-consuming and comprises an implementation on top of the OpenGL API, it is not implemented on the SpaceCAKE architecture for the time being.

Mesa’s multi-threading option can be enabled with compilation option _REENTRANT_. The PTHREADS flag can be enabled to support Pthreads, and USE_IEEE can be set for the correct floating-point conventions used by the TriMedia.

5.2 Creating an application for Mesa.

To test the performance of the 3D-graphics library, a relevant test application that can be used as a benchmark is needed. Since the throughput requirement for 3D graphics on SpaceCAKE is not yet determined a 3D game or a VRML-browser seems a good choice. Since these type of applications use a 3D-graphics library the most extensive. Porting such an application to the SpaceCAKE platform is difficult (assuming the source code is available). Typically, an application does not depend on the 3D-graphics library only. All other libraries, and their dependencies, need to be ported to the SpaceCAKE platform as well. Because this can be a cumbersome and sometimes impossible task, and because it is not relevant for the aim of the project, an indirect approach to create a test application has been pursued.

To avoid the need to port an entire application to SpaceCAKE the same application is executed on a host platform and the OpenGL commands are redirected to SpaceCAKE. Two methods to redirect the OpenGL commands are:

1. A direct approach, where OpenGL commands are sent at run-time to an application running on SpaceCAKE via FIFO-queues (First In First Out).

2. An indirect approach, where OpenGL commands are recorded to a file, which are then used to create a SpaceCAKE application.

Both methods will be elaborated on in more detail in Sections 5.2.1 and 5.2.2.

5.2.1 Direct approach

Figure 5.1 illustrates the concept of the direct method. The OpenGL library on the host platform is replaced by a proxy that marshals the issued OpenGL commands and sends them via a FIFO-queue to the SpaceCAKE platform. On the SpaceCAKE platform a stub reads the OpenGL commands from the queue and returns the results through another FIFO-queue after execution. FIFO-queues are commonly used as a means to communicate between processes or platforms. Khambatti M. [KHAM01] discusses an efficient means to use FIFO-queues for inter process communication.

This method needs the following issues to be resolved:
If an OpenGL command contains references to local memory, this memory needs to be sent (through FIFO-queues) to the SpaceCAKE platform. Similarly, memory might need to be transferred from the SpaceCAKE platform to the host.

Because the SpaceCAKE platform runs on a simulator, there is an enormous speed difference between the host and the simulated SpaceCAKE environment. Interaction with the application becomes difficult due to the latency of the feedback. A single frame takes minutes to hours for rendering and getting feedback from user-interaction takes even longer.

When a FIFO-queue blocks on a read or write operation, not only the thread that performs this action on the FIFO-queue blocks, but the entire process, i.e. all threads, block. This complicates the modeling of parallelism using threads.

5.2.2 Indirect approach

Due to the complications of the direct method a second (indirect) method, as illustrated in Figure 5.2 has been implemented.

The indirect method runs an application on a host with a custom OpenGL library pre-loaded in memory. This custom library executes each issued OpenGL command using the hardware accelerated OpenGL library and writes the OpenGL commands with the correct ANSI-C syntax into a trace-file. Additionally, if OpenGL commands contain references to memory, i.e. one of the function arguments is a pointer, this memory is written to a data-file in the form of an ANSI-C structure. The size of the memory block that needs to be written to file can be derived from the arguments of the OpenGL function containing the memory reference. This holds for all OpenGL functions, except for the ones that reference vertex arrays outside Mesa’s state machine. OpenGL

---

1 Local memory in this context is memory on the host platform.
2 Recall that the OpenGL API is defined as a set of C functions.
5.2. CREATING AN APPLICATION FOR MESA.

Host Platform  SpaceCAKE platform

![Diagram](image)

**Figure 5.2:** Recording OpenGL commands to file, for playback on the SpaceCAKE platform.

supports this kind of memory addressing outside its state machine. Vertex arrays can be created by an application which are accessed with an index number by OpenGL. To reconstruct the contents of the memory containing the vertex arrays for our OpenGL recorder, the maximum index number of the vertex arrays is determined in a first pass, while in a second pass the vertex arrays are stored to a file. Note that a second pass is normally not possible, if the redirected OpenGL commands are executed on-the-fly.

Using this indirect method has the advantage that a hardware accelerated driver can be used while recording the OpenGL information. This is useful because most applications require a 3D-accelerated implementation of OpenGL. A second advantage is the ability to edit and re-use the trace- and data-file, which makes comparing different runs straightforward. Note that the recording of OpenGL commands only has to be done once. Its result can be re-used as often as required. Another advantage of this indirect method is that user-interaction during playback is not required, because the playback of OpenGL calls is per definition non-interactive. This is an advantage because the latency of feedback is high on the SpaceCAKE simulator.

A disadvantages of this indirect method are the large application size that results from the trace- and data-file and the inability to measure the performance degradation that might result from adding feedback.

For this research project, a test application has been created using the indirect method by running the game Tuxracer. Tuxracer is an OpenGL game created by Sunspire Studios Inc. (see Figure 5.3 for a screenshot), with average complexity, meaning that it does not require the latest state-of-the-art graphics accelerator to achieve an acceptable frame rate. The game was executed on Linux (Red Hat 7.2), using an OpenGL library that exploits the available 3D-graphics hardware acceleration based on an NVIDIA TNT2. The recording of the first one hundred frames of OpenGL calls and their associated data have been used for the experiments.

The recorded OpenGL calls have been grouped per frame to be able to easily skip duplicated frames (viewing the menu) at the beginning of the game and to facilitate the measurement of the frame rate.
5.3 Parallelizing Mesa

Recall from Section 2.5 that Mesa's multi-threading option enables parallel execution of the OpenGL commands, by duplicating the state machines of OpenGL for each thread, respectively. This is necessary for an efficient parallel execution, but not sufficient, because the distribution of the workload is not trivial. In this section, two implementations, based on the Sort-Middle and Sort-Last methods will be described, respectively.

5.3.1 Implementing Sort-Middle

The Sort-Middle method creates a thread for each screen segment. The display screen of \( W \) by \( H \) pixels is divided in \( N \) by \( M \) rectangular segments, resulting in \( M \times N \) threads. For each thread an entire Mesa state machine is allocated. A pseudo algorithm of Sort-Middle is illustrated in Figure 5.4. It shows how the function \( \text{render}() \) is being called for each thread. The \( \text{start\_thread}() \) function lets thread \( \text{thr} \), start the function \( \text{render}() \). \( \text{render}() \) waits for a signal from the \( \text{signal\_thread}() \) function to execute all OpenGL calls of frame \( \text{framenr} \) and rasterizes the screen segment defined by \( x, y, w, \) and \( h \). When all threads have finished the processing on their part of a frame, the \( \text{write\_frame}() \) function writes the result to a file. This method requires almost no adjustments to Mesa, since OpenGL commands are available to limit rasterization to a certain screen segment.

The number of screen segments has been chosen equal to the number of processors to minimize the number of states machines that have to be created simultaneously. Because each state machine maintains a local copy of all texture data, creating many threads would exceed the available memory. Sharing texture data amongst processing units requires additional communication to prevent simultaneous write accesses.
5.3. PARALLELIZING MESA

5.3.2 Implementing Sort-Last

The Sort-Last method partitions polygons amongst threads. The amount of threads have been chosen equal to the number of processors and for each thread a Mesa state machine is allocated. For the distribution of polygons, Mesa’s vertex buffer (as described in Section 2.5) is used to control the granularity of polygon distribution. This granularity seems a logical choice, because the vertex buffer of Mesa is flushed whenever it is full (by default it can contain 216 vertices [PAUL03]) or is flushed for a state change.

Again, all OpenGL calls are sent to each thread to make sure all state changes are conveyed to every state machine. All polygons in the vertex buffer are rendered by only one state machine. The content of a vertex buffer is assigned to a render thread according to a round-robin fashion (recall Figure 2.12).

When all OpenGL commands of a single frame have been processed, each thread has produced a frame buffer containing image fragments. These fragments need to be merged to create the frame. This merging requires depth-buffer sorting of the pixel fragments. A pseudo algorithm of the Sort-Last implementation is illustrated in Figure 5.5.

The `render()` function executes all OpenGL commands of a frame and writes its output to its specific frame buffer `fb`. What vertex buffer’s contents are processed by each thread is determined with the help of the thread’s identification number.

The next chapter discusses the extraction of the performance data for the Sort-Middle and Sort-Last methods, followed by a presentation of the results.
s := Number of threads;
framenr := 0;
foreach s do
create_thread(thr_s);
create_framebuffer(fb_s);
start_thread(thr_s, render(fb_s, thr_id));
for all frames do
foreach s do
call render(fb_s, thr_id);
wait_for_signal();
... 
do actual rendering
of VB: thr_id mod thr_s
to frame fb_s;
...
signal ready(thr_s);

Figure 5.5: Pseudo algorithm of Sort-Last method.
Chapter 6

Performance measurements

This chapter presents the results on the efficiency of the Sort-Last and Sort-Middle partitioning methods implemented for Mesa, described in Section 5.2, using the benchmark application. Drawing general conclusions from the quantitative results presented in this chapter should be done with great care, because 3D-graphics rendering is extremely data dependent. Therefore the results in this chapter should not be interpreted as general performance measures for the Mesa implementation. Instead, the results show the main causes for a limited performance gain for an increasing thread-count of the game “Tuxracer”.

First, Section 6.1 describes how the performance of the implemented partitioning methods can be measured. The following sections present the results of the efficiency, as described in Section 3.2 and the speedup measurements for the game “Tuxracer”. Only one frame, as depicted in Figure 6.1: The frame of Tuxracer used for performance measurements.
Figure 6.1, is used for the measurements. Additionally, the most significant causes for non-optimal speedup are determined for each experiment. In Section 6.2, the Sort-Middle method is used, whereas Section 6.3 presents the results of the Sort-Last method. In Section 6.4 both experiments are repeated without taking rasterization into account. By disabling the rasterization, the performance of the 3D-graphics software can be measured for the case in which rasterization is performed in dedicated hardware.

6.1 Measuring the performance of Mesa

Currently, there are no tools available for SpaceCAKE that automate the extraction of performance data related to parallelism such as speedup, memory congestion, load imbalance, and communication overhead. Fortunately, these parameters can be partly derived from the Pro View [TUL00] output of SpaceCAKE and from logging the CPU cycle-count at the begin and end of each frame. In this section, we will describe how performance statistics are extracted. At the end of this section, we present details on the number of clock-cycles that are required to render a frame without parallelism, which serves as a reference. First we shall discuss the definition of the metrics that are used to evaluate our experiments.

Framerate

Although the frame rate is not directly related to parallelism, it allows the calculation of the overall speedup. The performance of 3D games is often expressed by the average number of frames per second that can be generated. The framerate is determined by counting the number of clock-cycles to process one frame by all threads. The slowest thread, i.e. the one that uses the most clock-cycles to process its part of a specific frame, determines the framerate. The internal function tct.cycle(), which returns the current clock-cycle count of a processor, is used to measure the number of clock-cycles of a frame for each thread. The number of cycles required to process each frame at each processor is logged in a file.

Speedup and efficiency

As described in Section 3.1, speedup and efficiency are closely related. If one of the two parameters is known, the other can be calculated if the number of processing units is known using equation 3.4. The overall speedup, and therefore the overall efficiency, can be obtained by investigating the increase in framerate.

However, for obtaining more detailed measurements of the speedup, e.g. the speedup of individual pipeline stages, the ProView statistics files of SpaceCAKE needs to be examined. ProView statistics files list the number of clock-cycles spent on each decision-tree of each function by all threads. The ProView statistics files detail the clock-cycles into several categories: instruction calls, instruction-cache stalls, data-cache stalls, copy backs, cache conflicts, issued operations, and executed operations, as shown in Figure 6.2 (a). With the help of the ProView tool, this information can be visualized, as illustrated in Figure 6.2 (b). See [TUL00] for further details of ProView.

Although ProView statistics files only give information on decision-tree and function level for all threads after a simulation has finished, several simulations with a different number

\[^1\text{A decision-tree is a part of a function containing instructions that can be executed sequentially, i.e. it contains no branch instructions.}\]
6.1. MEASURING THE PERFORMANCE OF MESA

---

Figure 6.2: An example of a ProView statistics file (a) and its visualization with ProView (b).
of frames and number of threads can be compared to collect information on the speedup of individual pipeline stages. A statistics file of the single frame \textit{framenr} is obtained by running two simulations, one with \textit{framenr} frames and one with \textit{framenr-1} frames. Next, the difference of both statistics files is calculated, by subtracting individual data fields. An example is shown in figure 6.3.

### Figure 6.3: Two ProView statistics files are subtracted to obtain the statistics per frame.

A statistics file of frame \textit{framenr} can be calculated for a different number of threads running in parallel, resulting in statistics files \textit{threads} \textsubscript{1} till \textit{threads} \textsubscript{8}. Recall from Section 3.4.1 that the Wasabi architecture contains a maximum of 8 TriMedias, therefore a maximum of 8 threads in parallel are used. Although more than one thread can be created for each processing unit, this does not increase the execution speed of Mesa, because each thread has to administer its own state machine. Furthermore it increases the required memory usage.

The speedup and efficiency can be determined for each function of each frame. However, since Mesa consists of more than 900 functions, only those functions that significantly contribute to the total number of clock-cycles are investigated. Furthermore, there is no precise correlation between the C-functions of Mesa and the 3D-graphics functionality. For example a C-function that multiplies two vectors is used throughout the whole rendering process, making it difficult to take its efficiency into account for a specific pipeline stage. Therefore, the results shown in this chapter are estimates, rather than exact figures. Because the functions that were taken into account comprises over 85% of the total clock-cycle count, the expected accuracy is approximately 15%.

The three types of overhead, that are of particular interest are parallel overhead, memory congestion, and load imbalance. How to determine these types of overhead is discussed below.
Parallel overhead and memory congestion

Two types of parallel overhead are distinguished, external parallel overhead and internal parallel overhead. External parallel overhead consists of clock-cycles spent on creating threads, thread administration, inter-thread communication, and additional functionality to implement parallelism. Contrarily, internal parallel overhead is caused by the parallel state machines of Mesa, which, for instance, all have to perform the state changes and all have to load textures into memory.

Memory congestion occurs if multiple processors want to access the memory at a higher bandwidth than available, or if multiple processors simultaneously access the same memory bank. Determining the memory congestion of SpaceCAKE is difficult with the currently available tools. A qualitative estimate has been attempted, from two sources. First, the data report of the SpaceCAKE simulator gives information on the average number of clock-cycles per instruction for each processor when a simulation has finished. Second, from the ProView statistics files, functions such as `memcpy()` can indicate if memory congestion occurs, because they will show an increase in the cycles/execution ratio, for increasing numbers of threads in parallel.

The total average number of clock-cycles per execution of `memcpy()` for increasing numbers of threads in parallel is presented in Table 6.1 for the Sort-Middle implementation.

<table>
<thead>
<tr>
<th># Threads</th>
<th>Cycles/execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>161</td>
</tr>
<tr>
<td>2</td>
<td>161</td>
</tr>
<tr>
<td>3</td>
<td>174</td>
</tr>
<tr>
<td>4</td>
<td>173</td>
</tr>
<tr>
<td>5</td>
<td>183</td>
</tr>
<tr>
<td>6</td>
<td>162</td>
</tr>
<tr>
<td>7</td>
<td>191</td>
</tr>
<tr>
<td>8</td>
<td>171</td>
</tr>
</tbody>
</table>

Table 6.1: Clock-cycles spent on `memcpy()` per execution for the Sort-Middle partitioning method.

This table shows that the number of clock-cycles per execution increases for an increasing number of threads. To what extend this increase is caused by memory congestion cannot be determined with the currently available tools. Because of this and the observation that the global number of clock-cycles per instruction has remained constant throughout the experiments, memory congestion is considered to be negligible.

Load imbalance The execution time of threads often differs. This difference is caused by an imperfect distribution of the workload due to data-dependent execution times. Each time threads are synchronized, load imbalance has a negative impact on the speedup and efficiency. Load imbalance is determined for our application by comparing the number of cycles required per frame on each thread. The number of idle clock-cycles per frame caused by load imbalance is calculated with Equation 6.1 for each frame.
\[ \#\text{Idle cycles} = n \times \text{MAX} \left( \text{cycles}_n \right) - \left( \sum_{n=0}^{\#\text{threads}} \text{cycles}_n \right), \quad (6.1) \]

with \( n \) the number of threads in parallel and \( \text{cycles}_n \) the number of cycles required for thread \( n \) to compute a frame. Without any partitioning, the frame illustrated in Figure 6.1 requires approximately 422 Mcycles to compute on a single TriMedia. Approximately 88\% of these cycles are spent on rasterization, the other 12\% are mainly spent on geometry transformations, and performing state changes. The following sections show how the number of clock-cycles change when parallelism is implemented.

## 6.2 Performance of Sort-Middle

To get an indication of the scalability of the Sort-Middle implementation and the efficiency of parallelism, eight experiments have been conducted, as described in the previous section. Threads are assigned to rectangular screen areas as illustrated in Figure 6.4.

![Diagram showing eight subdivisions](image)

**Figure 6.4**: Eight subdivisions used for the Sort-Middle experiments.

In the following sections the results of the experiments on load imbalance, parallel overhead, and memory congestion are presented for each of these eight experiments. Section 6.2.3, illustrates the impact of these factors on the efficiency and speedup of the Sort-Middle method.

### 6.2.1 Load imbalance

At the start of each frame, all threads are started almost simultaneously. However, their finishing times differ significantly. For all eight experiments the cycle counts of the test frame shown in Figure 6.1 are illustrated in Figure 6.5 for each thread. The execution times are normalized to the experiment that runs on one thread.

Figure 6.5 shows a general decrease in the execution time for an increasing thread-count. Furthermore, the figure shows that using 5 threads in parallel results in a longer execution time compared
6.2. PERFORMANCE OF SORT-MIDDLE

Figure 6.5: Thread execution times of the Sort-Middle experiments for one frame, normalized to the experiment that uses one thread.

to using only 4 threads. Similarly, the use of 7 threads results in a longer execution time compared to using only 6 threads. Figure 6.5 also shows that the use of 6 threads in parallel results in the fastest execution of the test frame, instead of using 8 threads in parallel.

Since all threads are synchronized at the end end of each frame, each thread waits until all threads have finished their part of the frame. Consequently, the load imbalance has a negative effect on the overall performance. The load imbalance for the Sort-Middle implementation is caused by the rasterization, since the Sort-Middle implementation performs the geometry transformation of the whole frame for every thread.

6.2.2 Parallel overhead

The parallel overhead of the Sort-Middle method has been investigated by examining four groups of functions: geometry transformation functions, rasterization functions, clipping functions, and thread related functions (thread administration). The increase in the number of clock-cycles, summed over all threads, compared to the non-parallel implementation, is illustrated in Table 6.2.

Table 6.2 shows that the overhead caused by thread-specific functions and the overhead of performing extra clipping is small compared to the overhead of rasterization functions. Furthermore, Table 6.2 shows that the geometry transformation, clipping, and (thread) administration increase almost linearly with the number of threads. The amount of parallel overhead for rasterization, seems to depend on the way the screen is partitioned, since partitioning the 2D output frame in 5 and 7 segments results in a considerable larger parallel overhead.
### Table 6.2: Parallel overhead of the Sort-Middle implementation, in percentage of the single-threaded frame-time.

<table>
<thead>
<tr>
<th># Threads</th>
<th>Geometry transform</th>
<th>Rasterization</th>
<th>Clipping</th>
<th>Administration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2.41</td>
<td>20.0</td>
<td>0.09</td>
<td>0.17</td>
</tr>
<tr>
<td>3</td>
<td>4.74</td>
<td>3.98</td>
<td>0.17</td>
<td>0.25</td>
</tr>
<tr>
<td>4</td>
<td>4.79</td>
<td>20.3</td>
<td>0.27</td>
<td>0.34</td>
</tr>
<tr>
<td>5</td>
<td>9.38</td>
<td>75.4</td>
<td>0.35</td>
<td>0.42</td>
</tr>
<tr>
<td>6</td>
<td>9.05</td>
<td>24.2</td>
<td>0.43</td>
<td>0.51</td>
</tr>
<tr>
<td>7</td>
<td>14.3</td>
<td>119</td>
<td>0.53</td>
<td>0.60</td>
</tr>
<tr>
<td>8</td>
<td>13.8</td>
<td>63.3</td>
<td>0.59</td>
<td>0.68</td>
</tr>
</tbody>
</table>

#### 6.2.3 Efficiency and Speedup

The efficiency of the eight Sort-Middle experiments are illustrated in Figure 6.6. This figure shows that load imbalance and parallel overhead are both significantly limiting the efficiency.

The efficiency has been determined with Equation 3.1 and 3.4, with \( n \) equal to the number of threads, \( T_n \) the maximum cycle-count of the thread executions, illustrated in Equation 6.2, and \( T_s \) the total number of cycles required to render the frame with a single-threaded implementation. The reduction of efficiency, caused by overhead, is calculated with Equation 6.3. The percentage of load imbalance is calculated with a similar equation as shown in Equation 6.4.

\[
T_n = \text{MAX}(\text{cycles}_n). \tag{6.2}
\]

\[
\text{Overhead} = \frac{\# \text{Cycles overhead}}{T_n \cdot n}. \tag{6.3}
\]

\[
\text{Load imbalance} = \frac{\# \text{Idle cycles}}{T_n \cdot n}. \tag{6.4}
\]

Figure 6.7 shows the speedup, which is calculated using Equation 3.4. It shows an increasing in speedup for increasing thread-count. Furthermore, it shows the maximum speedup is not achieved for 8 but 6 threads in parallel.

#### 6.3 Performance of Sort-Last

Similar to the procedure used to evaluate the Sort-Middle method, the efficiency and speedup of Sort-Last is presented in this section. First, the parallel overhead and the load imbalance are discussed in detail.
6.3. PERFORMANCE OF SORT-LAST

Figure 6.6: Efficiency of the Sort-Middle implementation.

Figure 6.7: Speedup of the Sort-Middle implementation.

Again, eight experiments have been conducted with a different number of threads in parallel, ranging from one to eight. Recall that each thread renders a sub-set of all polygons, resulting in a frame of image fragments for each thread. These image fragments need to be merged to construct the final image. However, since the merging of image fragments has not yet been implemented, the amount of required clock-cycles for this operation cannot be determined. Therefore, an estimate of the required number of cycles is presented below.

If eight threads run in parallel, eight frame buffers need to be merged. Merging frame buffers entails the determination of the minimum depth-value (32-bit unsigned integers) of each pixel.
in the eight frame buffers. A TriMedia has an IMIN operation that returns the minimum of two integers if they are loaded in registers, in two clock-cycles. The minimum of eight values can be determined with seven IMIN instructions. To load and store depth-values, we assume cache lines of 64 bytes are read and written in 3 clock-cycles.

The number of clock-cycles to merge 8 frame buffers for a frame of 640x480 pixels is now calculated as:

**Computing the minimum depth-value**

\[ 640 \times 480 \times 7 \times 2 \text{ cycles} = 4.3 \text{ Mcycles} \]

**Reading and writing to memory**

\[ 640 \times 480 / 16 = 19.2k \text{ cache lines for each frame buffer} \]

Reading cache lines: \[ 19.2k \times 8 \times 3 \text{ cycles} = 460 \text{ kcycles} \]

Writing cache lines: \[ 19.2k \times 8 \times 3 \text{ cycles} = 460 \text{ kcycles} \]

**Total**

5.22 Mcycles

Similarly, the total number of clock-cycles for merging frame buffers can be calculated for different thread-counts, as illustrated in Table 6.3. Note, that transparency and data-cache stalls have not been taken into account, which is rather optimistic.

<table>
<thead>
<tr>
<th># Threads</th>
<th># Cycles</th>
<th>% of frame time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>1.53M</td>
<td>0.36</td>
</tr>
<tr>
<td>3</td>
<td>2.15M</td>
<td>0.51</td>
</tr>
<tr>
<td>4</td>
<td>2.76M</td>
<td>0.65</td>
</tr>
<tr>
<td>5</td>
<td>3.38M</td>
<td>0.80</td>
</tr>
<tr>
<td>6</td>
<td>3.99M</td>
<td>0.95</td>
</tr>
<tr>
<td>7</td>
<td>4.16M</td>
<td>1.09</td>
</tr>
<tr>
<td>8</td>
<td>5.22M</td>
<td>1.24</td>
</tr>
</tbody>
</table>

**Table 6.3:** The number of clock-cycles required for merging frame buffers, presented in number of clock-cycles and as a percentage of the single-threaded frame-time.

In the following sections, the results of the experiments on load imbalance, parallel overhead, and memory congestion are presented. In section 6.2.3 we illustrate the efficiency and speedup of the Sort-Last method.

### 6.3.1 Load imbalance

Figure 6.8 shows the number of cycles spent on the frame, illustrated in Figure 6.1, for each thread and for all eight experiments. The cycle-count is again normalized to the experiment using only one thread.

As opposed to the results from [MOLN94], load imbalance is still significantly present, mainly due to one thread in each experiment that requires significantly more computation time than the
6.3. PERFORMANCE OF SART-LAST

others. This is probably caused by the background texture, which is entirely rasterized by this thread. Therefore, this characteristic is specific for this frame.

### 6.3.2 Parallel overhead

The external and internal parallel overhead of the Sort-Last implementation are listed in Table 6.4. Again the internal parallel overhead is caused by processing the state changes on each thread and performing certain functions for the geometry transformation and rasterization on all threads. External parallel overhead is caused by thread administration and merging image fragments.

<table>
<thead>
<tr>
<th># Threads</th>
<th>Internal overhead</th>
<th>External overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Geometry transform</td>
<td>Rasterization</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2.89</td>
<td>2.20</td>
</tr>
<tr>
<td>3</td>
<td>0.50</td>
<td>3.84</td>
</tr>
<tr>
<td>4</td>
<td>7.25</td>
<td>5.55</td>
</tr>
<tr>
<td>5</td>
<td>8.36</td>
<td>6.02</td>
</tr>
<tr>
<td>6</td>
<td>10.5</td>
<td>7.49</td>
</tr>
<tr>
<td>7</td>
<td>11.4</td>
<td>8.20</td>
</tr>
<tr>
<td>8</td>
<td>13.4</td>
<td>9.41</td>
</tr>
</tbody>
</table>

Table 6.4: The parallel overhead of the Sort-Last implementation as a percentage of the single-threaded frame time.
Table 6.4 shows internal parallel overhead is approximately 10 times as large as external parallel overhead. Note that the parallel overhead increases almost linearly with the number of threads and is significantly less compared to the Sort-Middle implementation. To what extent the presented parallel overhead is caused by memory congestion is difficult to determine for the same reasons as mentioned in Section 6.1. Again, its influence is considered negligible.

### 6.3.3 Efficiency and speedup

Similar to Section 6.2.3, the efficiency and speedup of the eight Sort-Last experiments are calculated below. Figure 6.9 shows for each experiment the percentage that is spent on rendering the frame and what percentage is spent on parallel overhead and load imbalance. Figure 6.10 shows the speedup, which is calculated using Equation 3.4. It shows a maximum speedup of approximately 2 when 8 threads are running in parallel.

![Efficiency of the Sort-Last implementation.](image)

### 6.4 Performance without rasterization

In this section, the experiments described in Sections 6.2 and 6.3 are repeated without enabling Mesa's rasterization. This illustrates the speedup and efficiency of the software for both methods if rasterization would be performed in hardware. The number of clock-cycles required to calculate the frame presented in Figure 6.1 when rasterization is disabled equals 10 Mcycles, without rasterization. Approximately 95% is spent on the geometry transformation and 5% is spent on performing state changes. This single-threaded case is used as reference for the other experiments.

Section 6.4.1 compares the results of the load balance of the Sort-Middle and Sort-Last implementations, while Section 6.4.2 shows the amount of parallel overhead for both implementations.
6.4. PERFORMANCE WITHOUT RASTERIZATION

The results of these two sections are combined in Section 6.4.3 to show the efficiency and speedup of these methods.

6.4.1 Load imbalance

Figures 6.11 (a) and 6.11 (b) illustrate the number of cycles spent for each thread using the Sort-Middle and Sort-Last implementation, respectively. The cycle-counts are normalized to the single-threaded experiment.

Figures 6.11 (a) and 6.11 (b) illustrate an almost perfect load balance. Furthermore, the figures show no performance gain is achieved for the Sort-Middle implementation. The Sort-Last algorithm shows a speed increase of approximately 40%, when using 8 threads in parallel. The lack of speedup for the Sort-Middle implementation is expected, because Sort-Middle distributes the workload after the geometry transformation stage. If rasterization is disabled, the geometry transformation is the only stage that is performed. Therefore no workload is distributed and each thread executes all functionality.

6.4.2 Parallel overhead

Parallel overhead is the main cause of a limited speedup and efficiency for both partitioning implementations when rasterization is disabled. Especially the Sort-Middle implementation has extreme overhead. This was expected, because the Sort-Middle method partitions geometry data after the geometry transformation stage. If the rasterization stage is disabled, the geometry transformation is the only stage that is executed. The Sort-Last implementation performs all state

![Figure 6.10: Speedup of the Sort-Last implementation.](image-url)
changes on each processor, but distributes the polygons amongst threads. Thread administration is a third cause of overhead. Table 6.5 quantifies these results, showing an almost linear increase for all types of parallel overhead.
6.4. PERFORMANCE WITHOUT RASTERIZATION

6.4.3 Efficiency and speedup

Sections 6.4.1 and 6.4.2 show that parallel overhead is the main cause for a low speedup and efficiency. Contrarily, load imbalance has little influence on the speedup and efficiency of both partitioning implementations. Figures 6.12 and 6.13 illustrate the efficiency and speedup of both implementations respectively.

Figure 6.13 (a) shows again that the Sort-Middle implementation does not result in any performance increase, i.e. the speedup is 1 for each number of threads in parallel. Figure 6.13 (b) shows that the Sort-Last implementation results in a performance increase of approximately 1.5 if more than 4 threads are used.

The speedup, displayed in Figure 6.13 (b), can easily be verified with Amdahl's law, defined by Equation 3.2 together with the results presented in Figure 6.11 (b). Figure 6.11 shows a saturation of the normalized execution time at 60%. According to Amdahl's law the speedup using eight threads in parallel will therefore be:

\[
\text{Speedup} = \frac{s + p}{s + \frac{p}{n}} = \frac{0.60 + 0.40}{0.60 + \frac{0.40}{8}} = 1.53,
\]

which corresponds with the result shown in Figure 6.13 (b).

This chapter has shown a limited performance increase for the presented partitioning methods. The Sort-Middle implementation, with rasterization, suffers from both severe load imbalance and parallel overhead. The presented Sort-Last implementation, with rasterization, shows severe load imbalance mainly due to one thread that requires more than three times as many cycles compared to the other threads. The Sort-Last implementation shows better results on parallel overhead compared to the Sort-Middle implementation.

Without rasterization, the parallel overhead, which is caused by the geometry transformation, hampers a speedup of more than 50%. Because the Sort-Middle implementation does no data partition at all, this implementation can never achieve any performance gain compared to the single-threaded implementation. The Sort-Last implementation without rasterization shows that

<table>
<thead>
<tr>
<th># Threads</th>
<th>Geometry transform</th>
<th>State changes</th>
<th>Thread Administration</th>
<th>Geometry transform</th>
<th>State changes</th>
<th>Thread Administration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>89</td>
<td>5.10</td>
<td>7.10</td>
<td>38</td>
<td>4.5</td>
<td>6.3</td>
</tr>
<tr>
<td>3</td>
<td>176</td>
<td>8.76</td>
<td>10.7</td>
<td>71</td>
<td>6.8</td>
<td>7.10</td>
</tr>
<tr>
<td>4</td>
<td>265</td>
<td>12.3</td>
<td>14.6</td>
<td>104</td>
<td>10.9</td>
<td>13.4</td>
</tr>
<tr>
<td>5</td>
<td>356</td>
<td>16.2</td>
<td>18.0</td>
<td>146</td>
<td>14.3</td>
<td>17.0</td>
</tr>
<tr>
<td>6</td>
<td>452</td>
<td>20.1</td>
<td>21.4</td>
<td>179</td>
<td>17.5</td>
<td>20.3</td>
</tr>
<tr>
<td>7</td>
<td>549</td>
<td>24.0</td>
<td>25.2</td>
<td>215</td>
<td>20.7</td>
<td>23.7</td>
</tr>
<tr>
<td>8</td>
<td>648</td>
<td>28.3</td>
<td>28.9</td>
<td>251</td>
<td>24.4</td>
<td>26.8</td>
</tr>
</tbody>
</table>

Table 6.5: The parallel overhead of both partitioning implementation without rasterization as a percentage of the single-threaded frame-time.
some gain can be obtained from the implemented partitioning. However, parallel overhead is still present. In the next chapter, we draw conclusions from these results and presents recommendations for future work.
6.4. PERFORMANCE WITHOUT RASTERIZATION

Figure 6.13: The speedup of the Sort-Middle (a) and Sort-Last (b) implementations without rasterization.
Chapter 7

Results and conclusions

In Chapter 4 three methods have been presented to data partition a real-time 3D-graphics renderer, as defined by OpenGL. Two of the data partitioning methods, called Sort-Middle and Sort-Last, have been implemented for Mesa, the chosen implementation of the OpenGL standard. We have shown in Section 6 that linear speedup could not be achieved for the Sort-Middle and Sort-Last implementations. The two causes for the limited performance gain, namely parallel overhead and load imbalance, have been quantified. In this chapter we will elaborate on the results of the previous chapters, and draw conclusions from both implementations. In section 7.1 we shall discuss the results of this graduation project. In Section 7.2 conclusions about implementing Mesa on the SpaceCAKE architecture are drawn. Finally, in Section 7.3 we focus on future work and present some recommendations.

7.1 Results

For the Sort-Middle and the Sort-Last implementations, causes for the limited speedup and efficiency are given in Sections 7.1.1 and 7.1.2, respectively. Furthermore, the tools developed for this graduation assignment, are presented in Section 7.1.3, which enable the performance measurements of the parallel 3D-graphics rendering implementations.

7.1.1 Sort-Middle

The Sort-Middle implementation suffers from severe load imbalance and parallel overhead, which prohibits a high speedup when increasing the number of parallel threads. The differences in polygon concentration and polygon complexity at different rectangular screen segments is the main cause of the load imbalance.

The load imbalance can be decreased by increasing the number of screen segments at the cost of more memory allocation for creating the additional state machines. The parallel overhead is more difficult to decrease. Currently, most overhead is caused by performing part of the functionality, such as updating the state machine and loading and converting texture memory formats, on each processing unit, instead of distributing this functionality. It could be argued that a single thread can perform these actions and broadcast the results to other threads to reduce the parallel over-
head. However, broadcasting the results requires synchronization and results in communication overhead. A trade-off between communication between threads and performing operations on multiple threads, has to be determined to find the minimum overhead.

Section 6.4.1 shows that the Sort-Middle implementation does not result in any performance increase if rasterization has not been enabled. This was expected, because the Sort-Middle method partitions geometry data after the geometry transformation stage. If the rasterization stage is disabled, the geometry transformation is the only stage that is executed. Therefore, each thread executes the same operations and no workload is distributed. As we have seen in Section 6.4, this results in an optimal load balance but also in a maximum parallel overhead.

7.1.2 Sort-Last

The results of the Sort-Last implementation show significantly less overhead compared to the Sort-Middle implementation. However, the load imbalance is larger, mainly due to a single thread that in most experiments requires more than three times as many clock-cycles compared to the other threads running in parallel. This extreme load imbalance is probably caused by a few large polygons, such as the background or the sky, which require significantly more computations during the rasterization process. Similar to the Sort-Middle method, load imbalance might be decreased by implementing a different scheme to distribute the workload, such as a first-come-first serve distribution.

If the rasterization is disabled, the load imbalance is significantly reduced. However, the parallel overhead still reduces the performance gain, because part of the functionality of the geometry transformation is still performed on every thread, despite the distribution of polygons.

7.1.3 Tools resulting for this research project

The tools that have been created to perform the measurements described in Chapter 6 are listed here. A detailed usage of these applications is described in Appendix A.

**TriMedia version of the Mesa 5.0 Library**

Version 5.0 of the Mesa library is ported to the TriMedia based SpaceCAKE architecture to serve as the OpenGL library for the single-threaded implementation and the multi-threaded Sort-Middle and Sort-Last implementations. The Sort-Last test application requires a modified version of the Mesa library.

**OpenGL recorder**

An API parser has been created to automate the creation of the OpenGL recorder source code. The API parser reads the q1.h header file, which contains the definition of the API, and generates the source code of the OpenGL recorder. The source code of the OpenGL recorder contains a function for each OpenGL command, which executes the OpenGL command using the available OpenGL library and logs the command and its arguments to a trace-file.

The OpenGL recorder is a library that needs to be pre-loaded into memory, before running an OpenGL application. It creates three files while the OpenGL application runs.
• A file containing a list of OpenGL calls grouped per picture, including its arguments,
• a file containing texture memories in the form of ANSI C-structures (structs), and
• a file containing the vertex arrays.

Test applications
The three files, which have been created with the OpenGL recorder have been combined to create the test applications for each sorting method and for the single threaded implementation.

Test applications have been made for the single-threaded, the Sort-Middle, and the Sort-Last implementations with rasterization enabled and disabled, for different threads in parallel and for a different number of frames to render.

The single-threaded test application and the Sort-Middle test application, with rasterization use the TriMedia port of the original Mesa library. The Sort-Last test applications require a modified version of the TriMedia Mesa library, because polygon distribution amongst threads cannot be implemented with OpenGL commands. To disable the rasterizer, the Mesa library required some additional modifications, which have been realised.

7.2 Conclusions

Two data partitioning methods, called Sort-Middle and Sort-Last, have been implemented for Mesa on the SpaceCAKE architecture to exploit parallelism. Linear speedup could not be achieved due to parallel overhead and load imbalance. Both factors have been quantified and their causes have been determined. In this section we list the conclusions that can be drawn from this research.

• Functional partitioning of 3D-graphics rendering seems a natural way for mapping onto a multiprocessor architecture, because 3D-graphics rendering requires many independent and consecutive operations on its input data. However, if scalability is important, data partitioning becomes favorable as explained in Section 4.1.

• In general, an application that is optimized for a single processor, requires more effort for conversion to a parallel implementations, than a non-optimized version. Often, an algorithm contains some inherent parallelism that can be executed concurrently. However, this parallelism might not be visible anymore if the algorithm is implemented and optimized for a single processor. As a result, algorithms might need to be re-implemented from the start, taking parallel execution into account to achieve a higher speedup from parallelism.

• The performance of 3D-graphics rendering is very data dependent and thus application dependent. Therefore, 3D-graphics rendering is a difficult application to perform efficiently on scalable multiprocessor architectures, and load imbalance is hard to prevent.

• Additional performance gain can be achieved by optimizing the code for the TriMedia VLIW architecture. 3D-graphics rendering consists of many filtering operations and matrix multiplications which can often be implemented efficiently on VLIW architectures.
7.2. CONCLUSIONS

- The Sort-Middle and Sort-Last partitioning implementations suffer from large overhead caused by the Mesa library being optimized for maximum polygon throughput on a single processing pipeline. The OpenGL state machine, and the Mesa implementation in particular, is optimized for minimizing the setup time of individual polygons, by defining polygon attributes (e.g. color, material, and lighting) as global state information. In this way only states changes have to be communicated and configured. Minimizing the configuration and setup time of polygons maximizes their throughput. However, on a multiprocessor architecture, a state machine does not result in optimal performance, because on each processing unit a state machine needs to be implemented or all state information needs to be broadcasted to all processing units. In both cases, the overhead limits the speedup.

- For the Sort-Middle implementation, with rasterization, the optimum of two trade-offs needs to be determined to find the optimum performance. Load imbalance can be reduced at the cost of more memory allocation, and communication overhead can be exchanged for performing computations on each processing unit. The optimum of both trade-offs needs to be found to achieve optimal parallel performance. Currently, the use of 6 threads in parallel, and therefore 6 processing units, results in the highest speedup of approximately 2.6. The use of more threads in parallel results in a lower speedup.

- The Sort-Last implementation suffers from severe load imbalance. A different distribution scheme, for instance a first-come-first-serve distribution of vertex buffer contents, might reduce this load imbalance. This has not been implemented due to the time constraints of this graduation assignment. The performance of the current implementation is limited to a speedup of 2 compared to the single-threaded implementation when eight threads are running in parallel. If load imbalance can be totally reduced a maximum speedup of approximately 4.5 can be obtained.

- If the rasterization is performed in hardware, the Sort-Middle implementation does not result in any performance increase, because no workload is distributed. The Sort-Last method does show some performance increase, but parallel overhead is still present, resulting in a saturation of the speedup, using more than 5 threads in parallel.

- The Sort-Last method currently seems the best method for partitioning 3D-graphics rendering with Mesa. If rasterization is implemented in dedicated hardware, the Sort-Middle method does not result in any performance increase. If rasterization is performed in software the Sort-Last implementation causes less parallel overhead, compared to the Sort-Middle method. Since parallel overhead is more difficult to reduce than load imbalance, the Sort-Last method is preferable.

- The SpaceCAKE development environment is limited in its functionality. Detailed performance measures for speedup and efficiency cannot be extracted easily. Furthermore, memory congestion, which is often a performance bottleneck in multiprocessor architectures, cannot be determined. To debug an application, the simulator logs each instruction of each processing unit, together with the register values, to a file. Finding bugs in these files can be tedious.
7.3 Recommendations for future work

The results of the data partitioning implementations presented in this graduation project have raised additional questions that require further research. Recommendations and suggestions for future research are listed below:

- The target 3D-graphics application for the SpaceCAKE architecture needs to be defined. The complexity of this application determines if hardware acceleration is required or not.

- If rasterization is to be implemented in dedicated hardware, a hardware driver for Mesa needs to be implemented.

- The Mesa source-code needs to be optimized for the TriMedia architecture. Both the geometry and the rasterization stages consist of a large number of vector and matrix multiplications. Furthermore, the rasterization stage requires many filtering operations. Both are well suited for execution on VLIW processors like the TriMedia. A large gain is expected to be obtained from TriMedia optimizations, including the use of custom operations.

- To facilitate the efficient implementation of software on the SpaceCAKE platform, the SpaceCAKE development environment can be extended with a debugger, tools to extract detailed information on speedup, efficiency, and memory bandwidth at run-time.

- The impact of adding user-interaction and control software on parallelism needs to be investigated. Furthermore, if the available MIPS is used for the application software on top of the OpenGL interface, a communication channel between the MIPS and the TriMedia processors to communicate OpenGL commands needs to be implemented. The proxy/stub-mechanism discussed in Section 5.2.1 can be used as a starting point for this implementation.

- Modifying the Sort-Last implementation to use a first-come-first-serve scheme for polygon distribution might reduce the load imbalance. Because this can improve the speedup significantly (from 1.5 to 4.5), further investigation is recommended.

- A combination of data partitioning and functional partitioning, might result in a higher parallel efficiency. For instance, a functional partitioning of pipeline stages can be combined with data partitioning within each stage. A Sort-Last implementation could be used to distribute polygons at the geometry stage, combined with a Sort-Middle implementation for each polygon at the rasterization stage.
Bibliography


[THOM00] Thompson K. Processor Load Balancing.


Appendix A

User Manual

This appendix explains how the experiments discussed in this report have been conducted. Section A.1 discusses how the OpenGL recorder can be used to record OpenGL calls at run-time for an application. Section A.2 describes how to operate the simulator to execute the implemented sorting algorithms.

A.1 The OpenGL recorder

The OpenGL recorder has currently been implemented for Linux. In order to successfully record the OpenGL commands, together with referenced texture memory and vertex arrays, an operational OpenGL library is required. Furthermore, the OpenGL recorder only works if the OpenGL application, which is to be recorded, is dynamically linked to the available OpenGL library.

If both these requirements are met, the OpenGL commands of the application can be recorded with the following command:

```
LD_LIBRARY_PATH=:<OpenGL_recorder_dir> LD_PRELOAD=glrecorder.so \
<application>,
```

where `<OpenGL_recorder_dir>` is the directory where the OpenGL recorder is installed and `<application>` is the OpenGL application that is to be recorded.

At run-time the following files are generated:

- **gltrace.c**
  This file contains a C-function for each recorded frame, called `frame0()`, `frame1()`, `frame2()`, etc. Each C-function contains a list of OpenGL commands in the order they were issued.

- **textures.c**
  This file contains a C-structure for each texture object that is referenced in `gltrace.c`. Depending on the texture format red, green, blue, and alpha values for each pixel are stored in integer or floating point format.
A.2. THE SPACECAKE SIMULATOR

**data.c**

This file contains a list of C-structures for each vertex array referenced in `gltrace.c`. A vertex array contains vertex coordinates, and can also contain vertex colors, texture coordinates and the direction of the vertex normal, depending on the format that is defined by the OpenGL command that references this data. Currently, the number of elements of each vertex array must be determined manually by inspecting `gltrace.c` to find the maximum index number of each vertex array.

These three files have been included in the tests application described in chapter 6, without any manual adjustment.

**A.2 The SpaceCAKE simulator**

The installation and usage of the SpaceCAKE simulator is described in [STRAV02], which also describes how to compile source code for this architecture using `caketmCC`. The creation and execution process of the Sort-Middle and Sort-Last test applications is described in this section.

**Sort-Middle**

The experiments discussed in chapter 6 use the Proview output from frame 41 of `gltrace.c` to extract performance measures. To obtain this output two experiments need to be run: one experiment rendering frames 0, 39, and 40, and one experiment rendering frames 0, 39, 40, and 41. The experiments can be conducted on the SpaceCAKE simulator with the following commands, respectively:

```
cakesim -cO -d<N> -x sm_3fr_rast.out -v -b
```

```
cakesim -cO -d<N> -x sm_4fr_rast.out -v -b
```

what `<N>` equal to the number of threads that are to be executed in parallel. A configuration file, named `benchmark.conf` needs to present in the directory where `cakesim` is started, which contains the number or rows and the number of columns the screen is subdivided in, and the number of threads that are to be created, respectively.

To perform the Sort-Middle experiments without rasterization the executables `sm_3fr_rast.out` and `sm_4fr_rast.out` are to be replaced with `sm_3fr_norast.out` and `sm_4fr_norast.out`, respectively.

**Sort-Last**

The Sort-Last experiments can be performed similar to the Sort-Middle experiments with the following command:

```
cakesim -cO -d<N> -x sl_3fr_rast<N>.out -V -b
```

```
cakesim -cO -d<N> -x sl_4fr_rast<N>.out -V -b
```
with $<N>$ the number of threads that are to executed in parallel. The Sort-Last experiments without rasterization can be performed by replacing `sl_3fr_rast<N>.out` and `sl_4fr_rast<N>.out` with `sl_3fr_norast<N>.out` and `sl_4fr_norast<N>.out`, respectively.

Each experiment generates a Proview statistics file called `proview.T0.stat`. A Proview statistics file for frame 41 can be calculated with a tool called `stat-diff`, which returns the difference of two Proview statistics files. Proview statistics file can be graphically visualized with Proview.