EINDHOVEN UNIVERSITY OF TECHNOLOGY

Eindhoven University of Technology

MASTER

# Tile-based rendering

Burgers, W.F.P.W.

*Award date:*
2005

Link to publication

**Disclaimer**
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

**General rights**
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computing Science

MASTER'S THESIS

# Tile-Based Rendering

by
W.F.P.W. Burgers

Supervisor: dr. ir. A.J.F. Kok

Eindhoven, January 2005

Technical note TN-2004/00983

Issued: 01/2005

# Tile-Based Rendering

W.F.P.W. Burgers
Philips Research Eindhoven

Unclassified

**© Koninklijke Philips Electronics N.V. 2004**

**Concerns:** Master Thesis

**Period of Work:** March 2004 - December 2004

**Notebooks:**

Authors' address W.F.P.W. Burgers burgersw@natlab.research.philips.com

© KONINKLIJKE PHILIPS ELECTRONICS N.V. 2004
All rights reserved. Reproduction or dissemination in whole or in part is prohibited without the prior written consent of the copyright holder.

---

**Title:** Tile-Based Rendering

**Author(s):** W.F.P.W. Burgers

**Reviewer(s):** Drs. Koen Meinds and Dr.ir. Arjan Kok

**Technical Note:** TN-2004/00983

**Additional Numbers:**

**Subcategory:**

**Project:**

**Customer:** Philips Research

---

**Keywords:** Tile-Based Rendering, OpenGL, VLIW, 3D graphics, Silicon Hive

**Abstract:** Today, even the most high-end graphics cards face one problem: limited bandwidth to off-chip memory. We present our research on a technique called Tile-Based Rendering, which decreases off-chip memory traffic. The technique is incorporated in a standard 3D graphics pipeline. We present a formula that estimates the savings of our Tile-Based Rendering system compared to traditional Frame-Based Rendering, followed by several benchmarks that measure the actual savings. We then map part of our system onto a VLIW core, aimed for use in the low-end mobile phone market, to get an indication of the expected triangle-throughput of our system when running on such architectures. Our results show that Tile-Based Rendering generates considerably less off-chip memory traffic than Frame-Based Rendering and is a suitable technique to render 3D scenes on low-end mobile phones.

---

© Koninklijke Philips Electronics N.V. 2004 

**Conclusions:** Our research resulted in the following conclusions:

- Tile-Based Rendering generates considerably less off-chip memory traffic than Frame-Based Rendering. For our test scenes, the savings of Tile-Based Rendering with respect to Frame-Based Rendering can be estimated at around 80%.
- The optimal tile size appears to be 32x32 pixels.
- Tile-Based Rendering is beneficial compared to Frame-Based Rendering for scenes with up to 157K vertices.
- When running on a VLIW core, aimed for usage in low-end mobile phones, our system can handle at least 66000 triangles per second.

As further research, the system could be extended with support for Exact Hidden Surface Removal, which decreases texture traffic by not retrieving texels for parts of polygons that eventually won't be visible. This would decrease off-chip memory traffic even further.

 © Koninklijke Philips Electronics N.V. 2004

# Contents


**1 Introduction** **1**

**2 3D graphics** **3**

2.1 OpenGL and the Mesa 3-D graphics library . . . . . . . . . . . . . . 6

**3 Tile-Based Rendering** **7**

3.1 Previous work . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 8

3.2 Options for enhancements . . . . . . . . . . . . . . . . . . . . . . . 11

3.2.1 Overview . . . . . . . . . . . . . . . . . . . . . . . . . . . 11

3.2.2 Insertion point . . . . . . . . . . . . . . . . . . . . . . . . 14

3.2.3 Vertex format . . . . . . . . . . . . . . . . . . . . . . . . . 14

3.2.4 Vertex array structure . . . . . . . . . . . . . . . . . . . . . 15

3.2.5 Index Array Structure . . . . . . . . . . . . . . . . . . . . . 17

3.2.6 Display list . . . . . . . . . . . . . . . . . . . . . . . . . . 19

3.2.7 Vertex cache . . . . . . . . . . . . . . . . . . . . . . . . . 22

3.2.8 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . 23

**4 Performance analysis and measurements** **24**

4.1 Expected performance . . . . . . . . . . . . . . . . . . . . . . . . . 24

4.1.1 Estimation for data traffic over interconnect C . . . . . . . . . 26

4.1.2 Estimation for data traffic over interconnect D . . . . . . . . . 31

4.1.3 Estimation for data traffic over interconnect E . . . . . . . . . 32

4.1.4 Estimation for data traffic over interconnect A . . . . . . . . . 33

4.1.5 Estimation for data traffic over interconnect B . . . . . . . . . 34

4.2 Measured performance . . . . . . . . . . . . . . . . . . . . . . . . . 34

4.2.1 Test environment . . . . . . . . . . . . . . . . . . . . . . . 34

4.2.2 Test resolution and tile sizes . . . . . . . . . . . . . . . . . . 35

4.2.3 Test scenes . . . . . . . . . . . . . . . . . . . . . . . . . . 35


© Koninklijke Philips Electronics N.V. 2004 

4.2.4 Measured results . . . . . . . . . . . . . . . . . . . . . . . . 36
4.3 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 40
4.3.1 Formula versus test measurements . . . . . . . . . . . . . . . 40
4.3.2 Optimal tile size . . . . . . . . . . . . . . . . . . . . . . . . 41
4.3.3 TBR versus FBR . . . . . . . . . . . . . . . . . . . . . . . . 42

**5 Mapping the Tile-Based Rendering algorithm onto a VLIW core 48**
5.1 Very Long Instruction Word . . . . . . . . . . . . . . . . . . . . . . 48
5.2 Silicon Hive . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 49
5.3 Mosca core . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 51
5.4 Mapping the Tile-Based Rendering algorithm onto the Mosca core . . 51
5.5 Evaluation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 55

**6 Conclusion 58**
6.1 Further research . . . . . . . . . . . . . . . . . . . . . . . . . . . 59

**A Definitions 63**

**B Variable definitions 64**

**C OpenGL ES properties 65**

**D Fractional precision required for Edge Anti-Aliasing 66**

**E Display list entries 67**

**F Test Scenes 70**
F.1 "Subversive Tendencies" scene . . . . . . . . . . . . . . . . . . . . 70
F.2 "Roll on Down the Line" scene . . . . . . . . . . . . . . . . . . . . 71
F.3 "Temple of Retribution" scene . . . . . . . . . . . . . . . . . . . . 71

**G Silicon Hive core 73**

**H Source code 75**
H.1 Common definitions . . . . . . . . . . . . . . . . . . . . . . . . . . 75
H.2 Initial code . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 76
H.3 Code after the ’fixing’ stage . . . . . . . . . . . . . . . . . . . . . 80
H.4 Code after the ’optimization’ stage . . . . . . . . . . . . . . . . . . 84

**I Semantics of Mosca instructions 91**

 © Koninklijke Philips Electronics N.V. 2004

# Section 1

# Introduction

Today, even the most high-end graphics cards face one problem: limited bandwidth to off-chip memory. Therefore it's important to utilize the available bandwidth efficiently. When developing a 2D/3D graphics core for the low-end mobile phone market, the need for efficient use of the available bandwidth is increased by the need to minimize power dissipation.

This thesis, conducted in the Information Technology (IT) group at Philips Research Eindhoven, documents our research on a known technique to decrease off-chip memory bandwidth and the implementation of that technique in a 3D graphics pipeline aimed for integration in a low-end mobile phone.

This technique is Tile-Based Rendering (TBR). It can decrease the data traffic to off-chip memory compared to traditional Frame-Based Rendering (FBR) by dividing the framebuffer in rectangular regions, called tiles, that are rendered independently. This significantly reduces the required number of accesses to off-chip memory because the tile size can be chosen such that it's small enough for the color buffer and depth buffer of a tile to be kept on-chip. The decrease in off-chip memory traffic improves performance whilst decreasing power dissipation.

The objectives of the research on this technique are described as follows:

**Improve an existing Tile-Based Rendering algorithm and provide a formula that predicts its performance. Implement the algorithm into a 3D pipeline and compare the measured data traffic results to both Frame-Based Rendering and the formula.**

The results from this research will give us an indication of the expected performance increase of TBR compared to FBR. When mapping the algorithm onto a Very Long Instruction Word (VLIW) architecture, targeted for use in the low-end mobile phone market, an indication of the expected performance of our TBR algorithm on its target platform can be obtained and this is therefore investigated as well:

**Map a representative part of the TBR algorithm onto a VLIW architecture and modify both the code and the core such that the TBR algorithm runs efficiently on it and an indication of the triangle-throughput can be obtained.**

The research resulted in this thesis. Chapter 2 provides a description of a general 3D pipeline. Chapter 3 discusses TBR, previous work in the field of TBR and describes

© Koninklijke Philips Electronics N.V. 2004 

the design of our algorithm. In chapter 4, a formula will be presented that predicts the performance of our algorithm given some input parameters and then evaluates the accurateness of that formula using several test scenes. In chapter 5, a part of our algorithm will be mapped onto a VLIW architecture intended for usage in low-end mobile phones. Finally chapter 6 covers the conclusions.

 © Koninklijke Philips Electronics N.V. 2004

# Section 2

# 3D graphics

Computer graphics concerns the science and technology required for converting geometry data to an image. A 3D pipeline describes the steps necessary to do so. The input of a 3D pipeline, the geometry, is first specified in a three-dimensional representation in the form of geometric primitives, which are a number of connected vertices that form basic shapes. Examples of those primitives are: triangles, quads, triangle strips or polygons. See Figure 2.1.

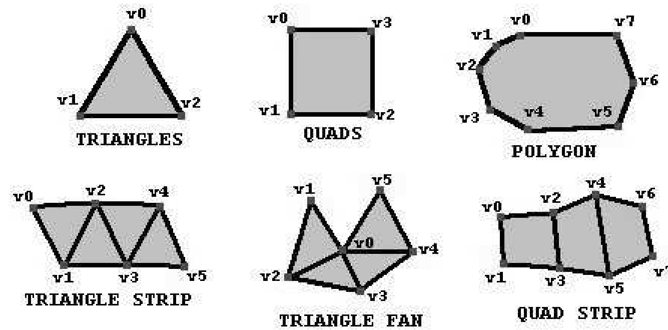



Figure 2.1: Primitive types. Triangles, quads and polygons are examples of independent primitives. Triangle strips, triangle fans and quad strips are examples of connected primitives.

The steps necessary to convert the provided geometry to an output image are shown in Figure 2.2.

**Model & view transformation** A scene is usually composed of several 3D models, which are build up from a number of connected vertices in the form of geometric primitives. Each of these models is described in its own model space with its own coordinates. The model transformation maps each model of the scene into a common reference model, called world space. The view transformation then transforms the models from world space, using the position of the camera in the world, into view space.

© Koninklijke Philips Electronics N.V. 2004

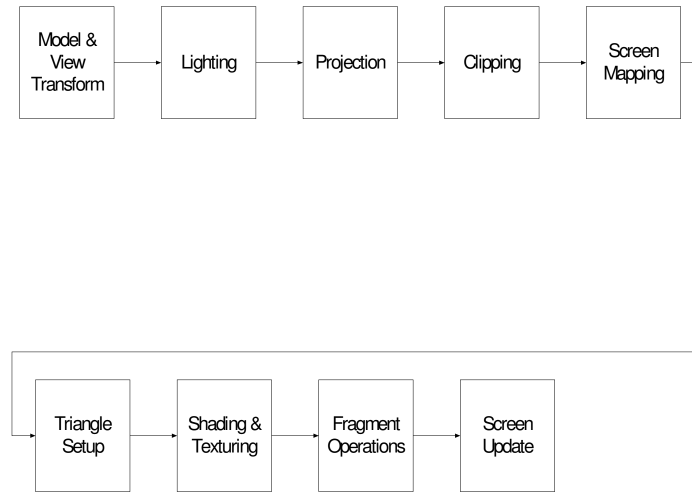



Figure 2.2: Geometric pipeline.

**Lighting** Lighting is applied to the models in the scene by performing a lighting calculation, based upon the number of lights added to the scene, for each of its vertices. Although the lighting is based upon very simplified lighting models, which often have little to do with how lights behave in the real world, the net effect increases the realism of the rendering significantly. How the models are lit is based on the interaction of its material properties and the light sources.

**Projection** Projection transforms the 3-dimensional view space onto the near plane of the camera. A common projection type is the perspective projection that makes objects in the distance appear smaller. The result of this operation are a number of transformed models that can be clipped against the projection volume that, in case of a perspective projection, has the shape of a truncated pyramid and is called the viewing frustrum.

**Clipping** To reduce the processing time spent on objects that won't be visible in the final image, the parts of a model that lie completely or partly outside the viewing frustrum are discarded. This is called clipping. Figure 2.3 shows an example. The result of this operation is a set of clipped primitives.

**Screen mapping** The resulting primitives are mapped to screen space, which determines where each primitive will be displayed on the screen.

**Triangle setup** Triangle setup first performs a step called backface culling. Here primitives that are not facing the camera are discarded, since they won't be visible in

 © Koninklijke Philips Electronics N.V. 2004

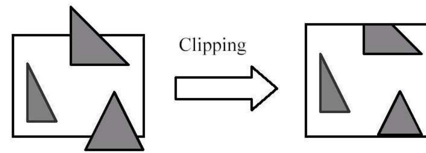



Figure 2.3: Clipping.

the final image. Just as with clipping, this saves processing time. Then it's determined which pixels of the screen are occupied by each primitive. Each primitive is then broken down into those pixels, which at this stage are called fragments. Those fragments are sent further down the pipeline where their final color is determined and whether they should be written to the color buffer or not.

**Shading & Texturing** Using a shading model (common choices are flat shading, gouraud shading [1] or phong shading [2]) and pixels from one or more textures, referred to as texels, the colors of the fragments are calculated.

**Fragment operations** Depending on the pipeline options and the rasterizer state, several more fragment operations may be performed that may alter or even throw out fragments. An example is the depth test, which is usually performed using a Z-buffer. Here the depth value of an incoming fragment is compared to the depth value of a previously stored fragment. This makes it possible to discard fragments that are occluded.

**Screen update** When a fragment arrives at this stage, its color is written to the specified position in the framebuffer. When all primitives have been rasterized, the framebuffer is presented to the screen which displays it to the user.

The following steps combined, which operate on the vertices of a primitive, are known as the front-end of the 3D pipeline:

- Model & View Transform
- Lighting
- Projection
- Clipping
- Screen Mapping

The following steps combined, which operate on fragments, are known as the back-end of the 3D pipeline, which is also known as the rasterizer:

- Triangle Setup
- Shading & Texturing
- Fragment Operations

- Screen update

## 2.1 OpenGL and the Mesa 3-D graphics library

Our target API is OpenGL ES [3]. OpenGL ES is an open standard, lightweight API for advanced 2D/3D graphics capabilities on mobile & handheld devices, appliances and embedded displays. It is based on a well-defined subset of OpenGL and enables an interface between software and hardware acceleration. Several properties of OpenGL ES that may be relevant for the design choices we make are listed in Appendix C.

Our development pipeline will be the open-source Mesa 3-D graphics library [4], that implements the OpenGL API (which is a superset of our target API: OpenGL ES). We will be using Mesa version 2.6, which implements OpenGL 1.1. Those parts of Mesa that are of no concern to OpenGL ES are ignored during our research.

 © Koninklijke Philips Electronics N.V. 2004

# Section 3

# Tile-Based Rendering

Even the most high-end graphic cards face one problem: limited bandwidth to off-chip memory. The higher the screen resolution, colour depth or number of polygons in the scene, the more memory bandwidth is required. We will try to tackle this problem using Tile-Based Rendering. TBR reduces the data traffic to off-chip memory by keeping bandwidth-intensive pixel processing operations, such as color buffer and Z-buffer accesses, on-chip. To do so, an on-chip color buffer and an on-chip Z-buffer are required. Given today's hardware, it's too expensive to put resolution-sized buffers on-chip. It is however viable to put smaller buffers on-chip. Usual sizes vary between 16x16 pixels to 64x64 pixels. However, because these on-chip buffers are not big enough to hold the entire scene at once, the scene now has to be rendered in parts. We therefore divide the framebuffer in rectangular regions, called tiles, that can be rendered independently in the on-chip buffers. Once a tile has been rendered, the on-chip color buffer is copied to the framebuffer in off-chip memory, which holds a view of the entire scene after all tiles have had their turn.

To render a tile, only the geometry visible in that part of the screen needs to be sent to the rasterizer. However, to send all geometry that is visible from a certain tile to the rasterizer all geometry must be known. Therefore, TBR requires a modification to the traditional 3D pipeline described in chapter 2. The traditional approach processes all incoming geometry immediately, while the TBR approach needs to buffer all incoming geometry and divide it over the different tiles. Once the entire scene has been captured, each tile can be rendered. Thus, in TBR, we distinguish two phases:

- Capture phase:
  Incoming geometry gets captured until all geometry in the scene is known. During capturing 2 operations are performed:
  - Sorting:
    Determining for all incoming geometry which parts of that geometry are visible in which tiles (we refer to this as geometry impacting a tile).
  - Buffering:
    The incoming geometry is stored by adding it to the virtual bucket of each tile that it impacts.

  Besides geometry, rasterizer related state changes need to be captured as well.

State changes alter the behavior of the 3D pipeline and the way that geometry is drawn on the screen. For captured geometry to be rendered correctly the rasterizer needs to be in the same state as it was when the geometry was offered to the pipeline.

- Render phase:
  When all geometry is known, rendering of the tiles can start. The following two-stage process is performed for each tile:
  - Rendering:
    A tile is rendered by sending all the geometry (and the state changes corresponding to that geometry) present in its virtual bucket to the rasterizer, which performs its rasterizing operations in the on-chip buffers.
  - Copying:
    When the final colors for a tile have been computed and stored in the on-chip color buffer, the buffer is copied to the framebuffer in off-chip memory.

  Once this process has been completed for each tile, the framebuffer in off-chip memory can be sent to the screen which displays the full scene to the user.

This approach renders the scene correctly but with a significantly reduced number of read/write accesses to off-chip memory, because during rendering off-chip memory only needs to be accessed for reading texture data and writing the color buffer of a tile to the framebuffer. Because of the reduced number of read/write accesses to off-chip memory, the required bandwidth to that memory is reduced. This makes bandwidth intensive features such as full-screen super super-sampling [1] or A-buffering [5] more viable, because these don't severely degrade performance anymore. Moreover, TBR introduces screen space parallelism to the rendering pipeline and therefore allows independent tiles to be rendered simultaneously by multiple rasterizers [6].

In section 3.1 we will discuss the current implementation of a TBR system. Section 3.2 contains a description of an improved TBR system design.

## 3.1 Previous work

A TBR support module, called the Triangle-to-Tile Sorting/Buffering (TTSB) module, that captures and stores all geometry and state changes was developed and integrated into the Mesa 3-D graphics pipeline at Philips by Sergey Trofimov in 1999 [7]. We now summarize his work.

The module works as a filter, inserted between the front-end and back-end of the pipeline, having the same input and output formats, so that other parts of the pipeline can function properly with little or no changes and the module can easily be integrated in another pipeline.

The module contains the following components:

[1] Rendering to a higher resolution and then downscaling

 © Koninklijke Philips Electronics N.V. 2004

- TTSB software:
  The TTSB software processes incoming primitives and incoming state changes during the capture phase. During the render phase it reads primitives and state changes from the display lists and sends those to the back-end of the pipeline. Several algorithms aid in this cause. Two algorithms were implemented for tile impact determination:
  - Bounding box primitive-to-tile-sorting [8]
    Calculates the screen-aligned bounding box for each primitive and marks any tile that is overlapped by that bounding box as impacted by the primitive.
  - Exact primitive-to-tile-sorting [9]
    Clips the primitive to the tile-grid. This way only actually overlapped tiles are marked as impacted.

  An algorithm to efficiently track state changes, called "smart state change tracking" [7], was developed and implemented.
- Vertex pool:
  The vertex pool contains vertex data (position coordinates, color values and texture coordinates) and consists of a number of memory blocks that are allocated whenever a set of vertices by the front-end is passed to the module. Instead of adding geometry to each tile, the geometry is only added once to the vertex pool to ensure that no geometry is stored twice. The tiles can then index that geometry. This reduces memory space. The vertex pool has the following properties:
  - It supports four different vertex types:
    * VTX_XYZ_RGBA
      No texture layer, one-sided lighting.
    * VTX_XYZ_RGBA_UV
      One texture layer, one-sided lighting.
    * VTX_XYZ_RGBA2
      No texture layer, two-sided lighting.
    * VTX_XYZ_RGBA2_UV
      One texture layer, two-sided lighting.
  - A vertex requires 16-28 bytes of memory space to store.
  - Each allocated memory block can contain up to 65536 vertices at a time.
- Display list(s):
  There is a display list, in [7] referred to as tile buffer chain, associated with every tile. It lists primitives and state changes that impact a given tile. It has the following properties:
  - A display list does not store actual geometry data. It stores pointers to allocated memory blocks and offsets and indices into those blocks. This information can be used to retrieve the actual geometry data.
  - The geometry and state changes are stored in the form of individual display list items, which consist of a 2-byte identification code followed by a variable number of entry type specific parameters. A display list item is usually 2-8 bytes.

© Koninklijke Philips Electronics N.V. 2004 

  - A display list is built of 32-128 byte blocks that are organized into a single linked list. A new block is allocated whenever the current block becomes full.

During the capture phase, the TTSB module operates as follows:

- The module receives a number of incoming primitives.
  Note that state changes are captured using "smart state change tracking" [7].
- The module determines for each single primitive if it will be visible in the final image. If it won't be visible, it shouldn't be stored, since this will waste memory bandwidth and memory space. The module performs the following tests for each primitive:
  - Does the primitive lie, completely or partly, in the viewing frustrum?
  - Is the primitive front facing?

  In the original Mesa implementation, these tests aren't performed until the very end of the pipeline.
- A new memory block is allocated and the vertices of those primitives that will be visible in the final image are stored in it. This memory block is considered part of the vertex pool.
- Then the following steps are performed for each primitive:
  - Determine which tiles the primitive impacts.
  - The display list of those tiles are updated with indices to the vertices of the primitive. These indices are preceded by a pointer to the memory block those vertices reside in and a number of state changes that correspond to the primitive.

The module determines the end of the capture phase using the `EXT_scene_marker` OpenGL extension [10], which informs the pipeline when scene creation is complete. At this point all possibly visible geometry has been captured and all display lists have been generated. Now the render phase can start, in which the module operates as follows:

- Tiles are traversed left-to-right, then bottom-to-top and for each tile the display list is traversed until the end is reached:
  - When encountering a state change, it is sent to the Mesa 3-D back-end, which changes the rasterizer state accordingly.
  - When encountering primitives, the required vertices are loaded from the specified memory block and sent to the Mesa 3-D back-end. The Mesa 3-D back-end then rasterizes those primitives.

  After this the framebuffer is sent to the screen which displays the full scene to the user.

We've identified several problems in the above design:

- Vertex attributes are stored in full precision (For example: the XYZ coordinates are stored using 4-byte floats). This makes the vertex size unnecessary large.

 © Koninklijke Philips Electronics N.V. 2004

- Off-chip memory bandwidth is still high. Redundant vertex data is written to and read from off-chip memory.
- The individual items stored in the display list occupy unnecessary space and therefore generate unnecessary traffic.
- The design of the vertex pool is insufficient. Its uncoherant nature and size make indexing vertices in the vertex pool unnecessary costly.

## 3.2 Options for enhancements

The work of [7] focused more on overcoming the technical difficulties of integrating the module in the Mesa 3-D pipeline than on the design of the module itself. The resulting module can therefore best be described as a proof of concept and it's therefore expected that considerably gains in bandwidth savings are possible. Our research is purely focused on the design of the TTSB module and how the current design can be adapted so that off-chip memory bandwidth requirements are minimized, since that is of key importance for our target platform: low-end mobile phones.

### 3.2.1 Overview

In comparison to [7], our module contains several new components and data structures, which are required to overcome the problems presented above. The entire list of components from our module is given below:

- TTSB software:
  The same software as described in section 3.1, but adapted to fit our data structures.
- Display list(s):
  Stores state changes and primitives. See section 3.2.6.
- Vertex Array Structure (VAS):
  Stores all vertex data for a scene. See section 3.2.4.
- Index Array Structure (IAS):
  Stores indices to vertices stored in the VAS. See section 3.2.5.
- Vertex Cache (VC):
  An on-chip software controlled vertex cache, which keeps a number of vertices on-chip. All requests for a vertex from the VAS are done via this component. See section 3.2.7.

The interaction between the components in the system slightly depends on whether the module is currently in the capture phase or in the render phase.

In Figure 3.1 the interaction is visualized while the module is in the capture phase. During the capture phase, our module operates as follows:

- The module receives a number of incoming primitives. Note that state changes are still captured using "smart state change tracking" [7].

© Koninklijke Philips Electronics N.V. 2004 

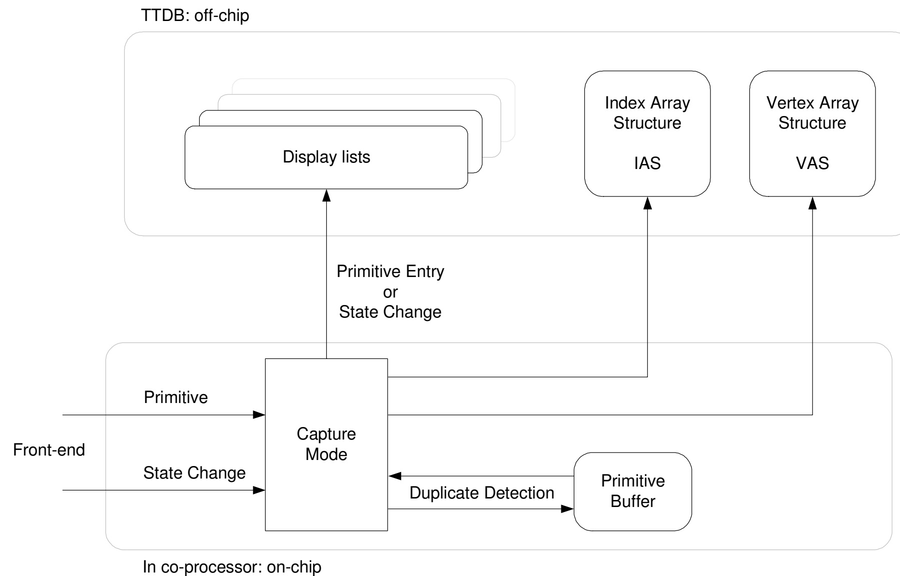



Figure 3.1: Data traffic during scene capturing.

- The module determines for each single primitive if it will be visible in the final image. If it won't be visible, it shouldn't be stored, since this will waste memory bandwidth and memory space. The module performs the following tests for each primitive:
  - Does the primitive lie, completely or partly, in the viewing frustrum?
  - Is the primitive front facing?

  In the original Mesa implementation, these tests aren't performed until the very end of the pipeline.
- Then the following steps are performed for each primitive:
  - Determine which tiles the primitive impacts.
  - Decrease the vertex size of the vertices by reducing the precision of their vertex attributes.
  - Store the vertices of the primitive in the on-chip primitive buffer. When inserting a vertex in the on-chip primitive buffer, a check is performed to see if there already exists an identical vertex in that buffer. In that case the vertex is not added.
- When the primitive buffer is full or a state change is captured the following steps are performed:

 © Koninklijke Philips Electronics N.V. 2004

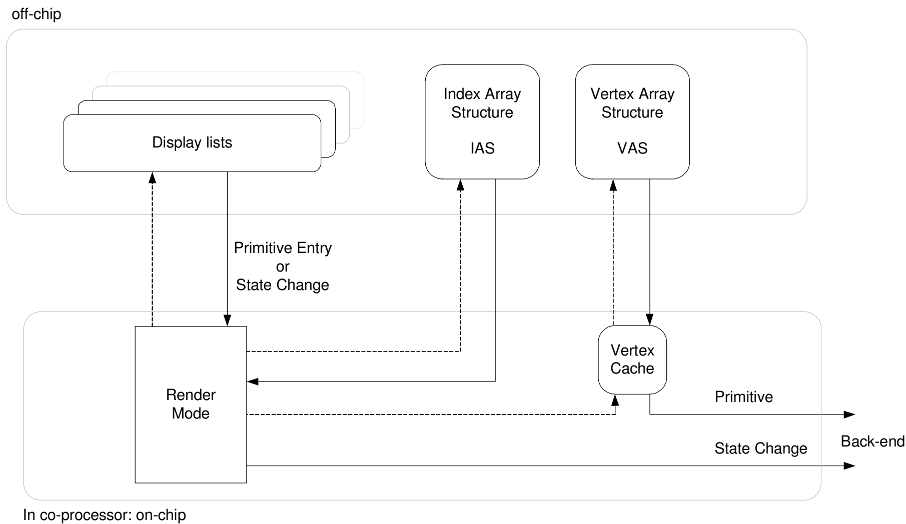



Figure 3.2: Data traffic during scene rendering.

  - The vertices in the on-chip primitive buffer are written to the VAS in off-chip memory, which creates space for those vertices if required.
  - If needed, indices to those vertices are written to the IAS in off-chip memory, which creates space for those indices if required. Indices for a primitive need to be stored if one of the vertices of the primitive was detected as a duplicate by the primitive buffer.
  - Primitive entries are written to the display lists of all tiles that were impacted by one or more of the primitives stored in the primitive buffer. If needed, those primitive entries are preceded by a number of state changes.
  - The primitive buffer is cleared to create space for new primitives.

The end of scene creation is still detected using the `EXT_scene_marker` OpenGL extension.

When scene creation is complete, the render phase can start. The interaction during that phase is visualized in Figure 3.2. During the render phase, our module operates as follows:

- Tiles are traversed left-to-right, then bottom-to-top and for each tile the display list is traversed until the end is reached:
  - When encountering a state change, it is sent to the Mesa 3-D back-end, which changes the rasterizer state accordingly.

© Koninklijke Philips Electronics N.V. 2004 

  - When encountering primitives, the required vertices are requested from the VC. The VC then requests those vertices from the VAS if they are not already present in the VC. The vertices are then sent to the Mesa 3-D back-end. The mesa 3-D back-end then rasterizes those primitives.

  After this the framebuffer is sent to the screen which displays the full scene to the user.

We will now discuss the various components of our module in more detail.

### 3.2.2 Insertion point

In [7] the TTSB module is inserted in the pipeline after vertex coordinates have been mapped to screen coordinates. However, there are several other places in the pipeline that are suitable as the insertion point for our module.

We want to minimize off-chip memory traffic. We thus want to choose the insertion point such that vertex coordinates can be specified using the minimum number of bits. Until the mapping to screen space, vertex coordinates -generally- take up 12 bytes (three 32 bit floats). After the vertices are mapped to screen space they can be stored using significantly fewer bits, because of the reduced range and reduced required precision. Therefore we keep the current insertion point, where vertex coordinates are stored as screen coordinates, in tact.

**Design Decision 1**
*The TTSB module is inserted in the pipeline after the Screen Mapping stage.*

### 3.2.3 Vertex format

In section 3.2.2, we chose the insertion point such that that vertex coordinates in our TTSB module are specified in screen space, because this minimized off-chip memory traffic. We need to store several other attributes for each vertex as well.

Vertices have, besides XYZ coordinates, also a RGBA color value associated with them. This value is calculated before the data is received by the module. Whereas in OpenGL front face and back face properties of a vertex can differ, in OpenGL ES they cannot. This means we only have to deal with a single color value. OpenGL ES supports multi-texturing. So every vertex also has zero, one or more texture coordinates. We assume that support for vertices with 0, 1, 2, 3 or 4 texture layers will suffice for the low-end mobile phone market.

Thus, there are the following vertex types with the following vertex attributes $i \in \mathbb{N} : (0 \leq i \leq 4)$:

- `VTX_TEXTURE`$i$:
  - XYZ coordinates.
  - RGBA color values.
  - $i$ * UV texture coordinates.

 © Koninklijke Philips Electronics N.V. 2004

As an example, consider a vertex as described in [7] with one pair of texture coordinates. That vertex takes up 24 bytes (3 floats for XYZ, 4 bytes for RGBA and 2 floats for UV).

To decrease memory space and off-chip memory bandwidth, we wish to use the minimum required number of bits for each of the vertex attributes. We decided that the minimum vertex requirements for the low-end mobile phone market are:

- 20 bits or 26 bits: XY coordinates. Each coordinate is specified in 10.0 fixed-point or 10.3 fixed-point depending on whether Edge Anti-Aliasing (EAA) is enabled or disabled. See Appendix D for more information.
- 16 bits: Z coordinate.
- 32 bits: RGBA color values.
- 26 bits: UV texture coordinates. Each coordinate is specified in 10.3 fixed-point.

Now again consider a vertex with one pair of texture coordinates. That vertex now takes up 13 bytes when EEA is enabled and 12 bytes when EEA is disabled (assuming that data structures are aligned to byte boundaries). This results in a decrease of up to 50% compared to the system in [7].

**Design Decision 2**
*We limit the precision of vertex attributes to a minimum to save memory space and decrease memory traffic. We use fixed-point instead of floating-point numbers.*

### 3.2.4 Vertex array structure

As seen in section 3.2.3, we have to deal with several different vertex types. They have to be stored in such a way that the required memory bandwidth is minimized. There are a number of options.

A simple approach would be to store all vertices, each preceded by a vertex type identifier, in a single buffer. To access vertex $n$, one has to begin at the start of the buffer and, using the vertex type identifiers, jump over the first $n$ vertices until the required data is reached. This is both computationally expensive and requires a lot of memory bandwidth. To improve that scenario, we could accompany the buffer with a vertex table that stores an offset or pointer into that buffer for each vertex. This would make vertices more easily accessible, but increases memory usage.

To overcome the drawbacks of the above propositions, we use multiple buffers instead of one to store the vertices. Each of these buffers is dedicated to storing vertices of a single type. This has several advantages. We don't have to precede vertex data with vertex identifiers anymore. And because all elements in each buffer are of constant size, each buffer is indexable as an array (which has a positive impact on memory traffic). Because of these advantages, this is the method we adopt. This structure of buffers is called the Vertex Array Structure (VAS).

**Design Decision 3**
*Vertices of different vertex types are stored in separate vertex arrays.*

© Koninklijke Philips Electronics N.V. 2004

We decided that a 3D application with about 3000 triangles per frame within the view volume and front facing, will be representative for a mobile phone application. The VAS should be of sufficient size to store the vertices of those triangles.

When confronted with a scene that consists of vertices of a single type, the vertex array of that type should be large enough to hold all of them. Since we don't know the scene characteristics in advance this would mean all of the vertex arrays should be able to hold the maximum number of vertices, which would waste a lot of memory space.

Instead we opt for an approach where each vertex array is subdivided into one or more sub vertex arrays (SVA). More SVAs are dynamically allocated when needed. This deals with memory size in a much more friendly matter. A new SVA could be allocated for each new primitive individually. However, since dynamically allocating memory is a computationally expensive operation we choose to allocate SVAs in larger fixed-size blocks. Because we want to minimize off-chip memory bandwidth we do not want an index into a SVA to occupy more than 1 byte. We therefore choose the size of these blocks such that each SVA can hold up to 256 vertices.

**Design Decision 4**
*Each vertex array is subdivided into sub vertex arrays. More sub arrays are allocated whenever necessary. Each of these sub arrays can hold up to 256 vertices.*

The derived structure of the VAS can be seen in Figure 3.3.

In [7], indices to vertex data are stored in the display lists and a single index is 2 bytes. In our system a vertex within a SVA is uniquely identified using a 1 byte index. Therefore our system improves on the space occupied by the index data, because all vertices of a primitive can be accessed using 1 byte SVA indices and a single 1 byte number of the SVA in which all its vertices reside. The only restriction is that all vertices of a primitive should always lie in the same SVA. Now, for a triangle strip primitive, consisting of $n$ vertices, the index data will occupy $1 + n$ bytes instead of the $2n$ bytes required by the simple approach used in [7].

**Design Decision 5**
*Vertices of a primitive are always stored in the same SVA, so we can use an efficient approach for indexing them.*

In a scene, it's common that a vertex is shared among multiple primitives. Because inserting all those duplicate vertices multiple times requires a lot of unnecessary bandwidth and memory space, measures are taken to detect those duplicates. Obviously, we want an efficient method to check if a vertex that's being inserted is a duplicate. A naive approach would be to check every inserted vertex against every vertex already in the VAS. However this approach requires too much off-chip memory traffic. We think it's likely that when a vertex gets duplicated (e.g. it's a shared vertex used by multiple primitives), it's duplicated within an short interval (locality of reference). This leads us to the approach of using a small (like 1KB to 4KB, we haven chosen 2KB) on-chip primitive buffer that buffers vertices of accepted primitives whilst checking for duplicates. When the buffer is full, its contents are written to the VAS in off-chip memory and the on-chip buffer is emptied. This reduces computation costs significantly while still detecting most of the duplicates.

 © Koninklijke Philips Electronics N.V. 2004

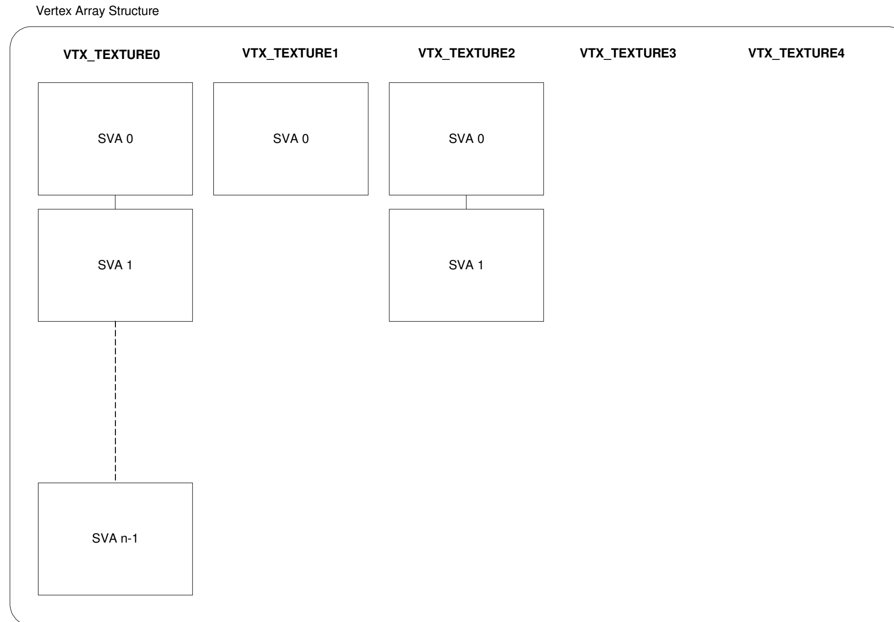



Figure 3.3: The Vertex Array Structure. Shown is a VAS with a number of allocated SVAs for different vertex types. Each SVA contains up to 256 vertices.

**Design Decision 6**
*Before being written to the VAS, the on-chip primitive buffer is used to detect duplicates in a set of vertices.*

The size (in bytes) of a vertex array of type $n$ is equal to: $size[n] = vts(n) * nsa(n) * 256$, where $vts$ is the size in bytes of a vertex of the given type and $nsa$ is the number of SVAs allocated for the given type. The total size occupied by all vertex arrays is then $\sum_{i=0}^{4} size[i]$.

### 3.2.5 Index Array Structure

Because of design decision 6 to use duplicate detection, it's not guaranteed anymore that the vertices of a primitive are stored sequential in the VAS. We have to deal with two types of primitives. Those primitives whose vertices are stored sequentially in the VAS (none of its vertices were marked as a duplicate) and those primitives whose vertices aren't stored sequentially in the VAS (one or more of its vertices were marked as a duplicate).

A primitive whose vertices are stored sequentially can be accessed using the following information:

- The vertex type.

© Koninklijke Philips Electronics N.V. 2004 

- An index to the SVA in which the vertices of the primitive are stored.
- An index pointing into that SVA to the start vertex of the primitive.
- The length of the primitive.

**Design Decision 7**
*Primitives whose vertices are stored sequentially in the VAS, from now on referred to as sequential primitives, are accessed directly.*

We want a similar mechanism to access primitives whose vertices aren't stored sequentially. For this we introduce the Index Array Structure (IAS).

**Design Decision 8**
*Primitives whose vertices aren't stored sequentially in the VAS, from now on referred to as indexed primitives, are accessed using an extra layer of indirection, called the Index Array Structure (IAS).*

This extra layer of indirection is an array of indices. For a primitive of $n$ vertices which aren't or partly aren't stored sequentially in the VAS, there exist $n$ sequentially stored indices in the IAS. These indices point to the vertices of the primitive in the VAS, in the correct order.

Because we don't know in advance how many memory is required by the IAS and we don't want to allocate more memory space then necessary, we subdivide the IAS in 256 entry sub index arrays (SIA). An approach similar to what we used for the VAS, as described in section 3.2.4.

**Design Decision 9**
*Just as the VAS, the IAS is subdivided into sub index arrays. More sub arrays are allocated whenever necessary. Each of these sub arrays can contain at most 256 indices.*

The derived structure of the IAS can be seen in Figure 3.4.

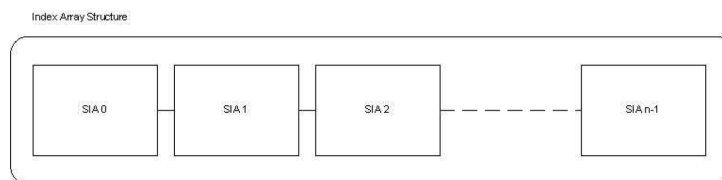



Figure 3.4: The Index Array Structure. Shown is an IAS with a $n$ allocated SIAs. Each SIA contains up to 256 indices.

The indices stored in the IAS point into a SVA of the VAS. Since a SVA contains no more than 256 vertices, the indices in the IAS will be 1 byte in size. Note that the indices in the IAS don't hold information on which SVA they are pointing into since this information is stored in the display list.

Now, using this layer of indirection, an indexed primitive can be accessed using the following information:

- The vertex type.

 © Koninklijke Philips Electronics N.V. 2004

- An index to the SVA in which the vertices of the primitive are stored.
- An index to the SIA in which the indices are stored.
- An index pointing into that SIA to the start index.
- The length of the primitive.

The size (in bytes) for the IAS is equal to: $size = nsa * 256$, where $nsa$ is the current number of allocated sub arrays. How many of those sub arrays will be allocated is hard to predict. It depends among others on the number of indexed primitives (which depends on the number of duplicated vertices) and the length of those primitives.

### 3.2.6 Display list

There is a display list associated with every tile and it holds a number of display list items that represent geometry or state changes. Since there can be a large number of tiles, and thus a large number of display lists, it's important to store the display list of each tile as compact and efficient as possible to reduce off-chip memory traffic.

| | |
|---|---|
| 1 | Clear screen |
| 2 | Sequential Primitive |
| 3 | Sub Vertex Array: #1 |
| 4 | A state change |
| 5 | Sequential Primitive |
| 6 | Indexed Primitive |
| 7 | Sub Index Array: #1 |
| 8 | Sub Vertex Array: #2 |
| 9 | Enabling texturing |
| 10 | Setting the active texture |
| 11 | Indexed Primitive |

Table 3.1: An example display list.

Before we discuss the exact design of the display list, we present an example of its usage. Consider a tile with the display list of Table 3.1. The meaning of the different entries is described below.

1. This state change clears the on-chip color buffer. The background color is assigned to all pixels in this buffer.
2. A sequential primitive entry. It contains the following information:
   - Start vertex of the primitive.
   - Length of the primitive.

   The vertices are accessed using this information and the following data derived from the display list:

   - Vertex type: VTX_TEXTURE0
     No state change enabled texturing and no textures is the default.

- Sub vertex array: #0
  No sub vertex array change occured before and 0 is the default.

3. A sub vertex array change setting the active SVA to 1.
4. A state change. Examples are enabling blending, disabling fog or changing the shade model. All primitives in the display list appearing after this state change are affected by this entry.
5. A sequential primitive entry. It contains the following information:
   - Start vertex of the primitive.
   - Length of the primitive.

   The vertices are accessed using this information and the following data derived from the display list:
   - Vertex type: VTX_TEXTURE0
     No state change enabled texturing and no textures is the default.
   - Sub vertex array: #1
6. An indexed primitive entry. It contains the following information:
   - Start vertex of the primitive.
   - Length of the primitive.

   The vertices are accessed using this information and the following data derived from the display list:
   - Vertex type: VTX_TEXTURE0
     No state change enabled texturing and no textures is the default.
   - Sub vertex array: #1
   - Sub index array: #0
     No sub index array change occured before and 0 is the default.
7. A sub index array change setting the active SIA to 1.
8. A sub vertex array change setting the active SVA to 2.
9. A state change that enables texturing.
10. A state change that sets the active texture.
11. An indexed primitive entry. It contains the following information:
    - Start vertex of the primitive.
    - Length of the primitive.

    The vertices are accessed using this information and the following data derived from the display list:
    - Vertex type: VTX_TEXTURE1
    - Sub vertex array: #2
    - Sub index array: #1

 © Koninklijke Philips Electronics N.V. 2004

The different kinds of display list items will vary in size (to minimize traffic), making it impossible to store the display list as an array. It could be stored as a linked list, but that would require an additional `next` pointer per item (which costs 4 bytes per pointer). Due to this costly overhead and the sequential nature of a display list (it needs to be traversed from start to end in sequential order to render a tile) the display list items are instead stored sequential in a buffer (a continuous block of memory).

Every tile requires a display list, so every tile also needs its own display list buffer. Again we are confronted with a situation in which we need to decide how much memory to allocate per buffer. We think the design in [7] in which extra buffer space is allocated whenever a buffer of a tile gets full (and chaining those buffers together as a linked list) is the correct way and therefore keep that intact in our design.

**Design Decision 10**
*Every tile has its own display list. This display list is represented by several buffers of display list items. These buffers themselves are chained together as a single linked list. The items in the display list can be of variable sizes.*

There are two sorts of items that need to be stored in the display list. These are state changes and primitives. There are two different types of primitives (indexed and sequential). We propose that the vertices of those primitives are retrieved using the following entries in the display list:

- `DLI_INDEXED_PRIMITIVE`:
  - 8 bits: An index into a SIA of the IAS, that indicates the start index of the primitive. There are 256 indexable locations in a SIA, so 8 bits are required for the start index.
  - 5 bits: Length of the primitive.

    We think the average length of a stored primitive will be well below 32, so 5 bits for the length of a primitive will suffice. In the unexpected case a stored primitive has a length larger than 31 it will be split in separate primitives of length equal to or smaller than 31.
- `DLI_SEQUENTIAL_PRIMITIVE`:
  - 8 bits: An index into a SVA of the VAS, that indicates the start vertex of the primitive. There are also 256 indexable locations in a SVA, so 8 bits are required here as well.
  - 5 bits: Length of the primitive. See `DLI_INDEXED_PRIMITIVE`.

From section 3.2.5 we know that to access the vertices of a primitive we also need to know the vertex type, the SVA in which the vertices are stored and possibly the SIA in which the indices are stored. This information isn't present in the display list entry of a primitive. Because consecutive primitive entries in a display list will often reference the same SVA and SIA, this information is stored as separate state change entries in the display list that set the currently active SVA and SIA. The vertex type isn't stored anywhere, because this information follows implicitly from the OpenGL (ES) state.

Besides primitives we also need to store rasterizer related state changes. To decrease bandwidth requirements, state changes need to be stored in a format as compact as

possible. We roughly differentiate between two different types of state changes. Those that toggle between various pre-defined OpenGL states (shade mode, front face vertex order, texture wrapping mode, etc...) and those that take parameters (clear color, line width, etc...). The first kind can be stored compactly, when we assign a unique code to every possible pre-definable state change. State changes of the second type take up more space because there are one or more parameters that have to be stored as well. To save memory space, we could store the parameters in a separate place and index them in each display list, but this would increase off-chip memory traffic. Therefore state changes of the second type are stored as variable sized entries in the display list consisting of an identifier followed by a number of parameters. The possible entries are listed in Appendix E.

### 3.2.7 Vertex cache

The IAS introduced in section 3.2.5 reduces the space required to store the vertices since vertices that are shared among primitives are no longer stored multiple times but only once. This also reduces the cost for writing vertices to the VAS in off-chip memory. Reading of vertices however still generates redundant traffic. Consider the following two cases:

- A vertex that is used by multiple primitives is fetched from off-chip memory each time one of those primitives is accessed.
- A vertex of a primitive that impacts multiple tiles is fetched from off-chip memory for each tile.

In both cases, all traffic resulting from reading such a vertex is redundant after the initial read if somehow the vertex data could be kept on-chip until it is requested again. For this, we introduce an on-chip software controlled vertex cache.

All requests for a vertex, identified by its vertex type, an index to a SVA and an index into that SVA, are passed through the vertex cache. Using the vertex identifiers it can determine if the requested vertex is already on-chip (a so called cache hit) and if so, it can be passed back immediately, sparing out the off-chip memory traffic that would be generated if the vertex would have to be fetched from the VAS. If the requested vertex is not on-chip (a so called cache miss), it's fetched from the VAS and stored in the cache. If no room is available in the cache to store the vertex, space is created by invalidating another vertex.

**Design Decision 11**
*All requests for a vertex are passed through the vertex cache, so off-chip memory traffic resulting from requesting a vertex multiple times is reduced.*

The performance of the vertex cache is measured as the number of times a requested vertex is already found in the cache compared to the total number of requested vertices. This so called hit-ratio is influenced by a number of factors, such as the number of shared vertices, the size of the vertex cache and also the cache strategy.

 © Koninklijke Philips Electronics N.V. 2004

Optimal values for those factors as well as an optimal cache strategy might be a topic for further research. We have chosen a size of 1024Kb for our on-chip vertex cache and have it act as a regular FIFO (First-In-First-Out) buffer for our cache strategy.

### 3.2.8 Evaluation

We have just presented a series of modifications to an existing TBR system [7]. The consequences of those modifications are listed below:

- Replacing the vertex pool by a combination of the VAS and the IAS:
  - Efficient usage of available memory as the VAS and IAS grow in size if needed.
  - Supports more texture layers, which is required by modern applications.
  - The VAS provides a single consistent point in which vertices are stored, instead of the non-coherent memory blocks that required display lists to contain expensive 4-byte pointers.
  - Maximum number of vertices that can be stored in the VAS is 65536.
- Detection of duplicate vertices among the input data:
  - Space required to store vertices is reduced.
  - Memory traffic to write vertices to off-chip memory is decreased.
  - Marginal increase in memory traffic because indices might need to be written to and read from the IAS.
- Reducing precision of vertex attributes:
  - Space required to store a vertex is reduced up to 50%.
  - Memory traffic to write a vertex to off-chip memory is decreased.
  - Memory traffic to read a vertex from off-chip memory is decreased.
- Optimizing the way items are stored in the display list:
  - Space required to store display lists is reduced.
  - Memory traffic to write display lists to off-chip memory is decreased.
  - Memory traffic to read display lists from off-chip memory is decreased.
- The introduction of a vertex cache into the system:
  - Memory traffic to read vertices from off-chip memory is decreased.
  - The VC provides a single consistent point from which vertices are requested.

In the next chapter we will evaluate the performance of our TBR system compared to FBR.

© Koninklijke Philips Electronics N.V. 2004

# Section 4

# Performance analysis and measurements

In chapter 3 we introduced our improved version of the TBR algorithm. It's interesting to see how this algorithm performs in practice. In this section we analyze the performance of our TBR algorithm. The performance figure we are interested in is the amount of off-chip memory traffic of Tile-Based Rendering compared to Frame-Based Rendering.

We will first attempt to establish a formula that estimates the off-chip memory traffic of the algorithm based on a number of input parameters. The purpose of this formula is to determine, given some scene statistics, whether a scene can better be rendered using TBR or FBR. We will then measure the off-chip memory traffic of our algorithm on several test scenes.

## 4.1 Expected performance

To estimate the off-chip memory traffic of our algorithm we first analyse the module as it is placed in its environment. Figure 4.1 shows the different interconnects between the on-chip and off-chip components.

TBR introduces extra traffic to off-chip memory (interconnects C, D and E). However, it saves traffic by keeping data on-chip (interconnects A and B). We are interested to see if and under which circumstances (screen resolution, tile size, scene characteristics) TBR is preferred over FBR, e.g. how does the combined traffic over interconnects C, D and E compare to the combined traffic over interconnects A and B?

We can ignore the texture traffic over interconnect F, since this will be equal for both TBR and FBR. A textured primitive can overlap multiple tiles, but texels aren't retrieved for pixels that fall outside tile boundaries and therefore the traffic over interconnect F for TBR is the same as for FBR (although there might be some neglectable overhead on tile edges for TBR).

Ideally, we want to make an estimation based on the input parameters listed in Table 4.1. When we cannot express a part of our estimation in terms of those input values, we

 © Koninklijke Philips Electronics N.V. 2004

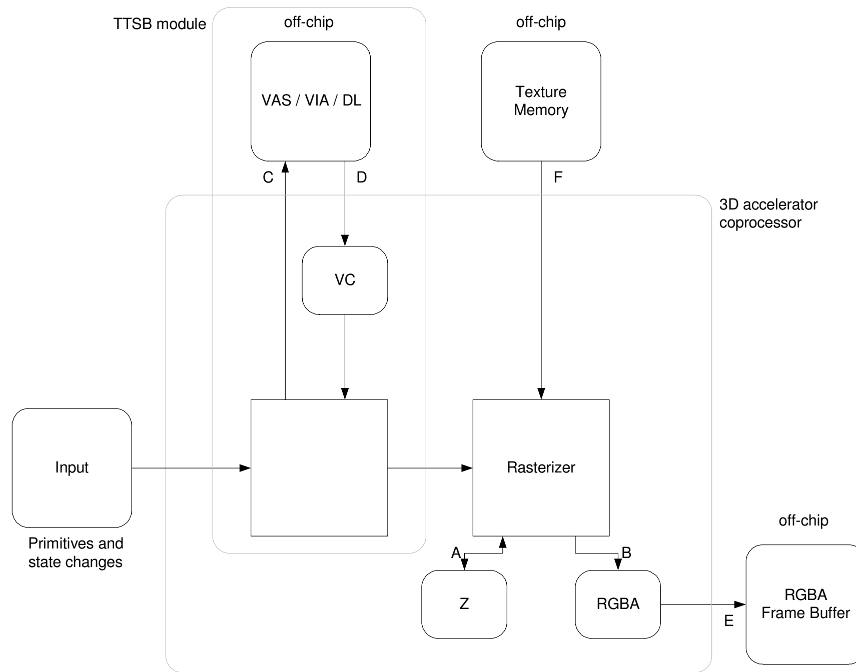



Figure 4.1: TTSB module placed in its environment.

| Parameter | |
|---|---|
| Screen resolution width (horizontal) | $D_{sx}$ |
| Screen resolution height (vertical) | $D_{sy}$ |
| Tile width (horizontal) | $D_{tx}$ |
| Tile height (horizontal) | $D_{ty}$ |
| Depth complexity | $DC$ |
| Number of triangles | $N_p$ |
| Number of vertices | $N_v$ |
| Number of tiles (horizontal) | $N_{tx}$ |
| Number of tiles (vertical) | $N_{ty}$ |

Table 4.1: Input parameters.

will use measured values from our test scenes for that part of the formula, because this is the fairest way to compare FBR to TBR. If statistical information was gathered on a large enough number of test scenes, averaged results from those tests could be used as input for our formula instead. This was not part of our research. Measurements were done by our module itself, except for the depth complexity. The depth complexity is provided by our test environment.

The amount of off-chip data traffic that TBR saves with respect to FBR equals the traffic that can now stay on-chip (interconnects A and B) minus the off-chip traffic

© Koninklijke Philips Electronics N.V. 2004 

introduced by TBR (interconnects C, D and E):

$$A + B - (C + D + E) \tag{4.1}$$

If the above equation is positive, TBR saves off-chip memory bandwidth. Otherwise, it does not.

Because the formula is tested in an environment similar to that of the intended use of the module we can make some assumptions about how the scene is provided to the pipeline by the application. We assume that optimal models are provided. It should be trivial to see that if an application provides the scene as a series of independent primitives (e.g. triangles), the number of duplicated vertices is significantly bigger than when the application provides the scene as a series of connected primitives (e.g. triangle strips). Since duplicated vertices cost unnecessary computations, a good application will keep those to a minimum and provide the scene as a series of connected primitives. Also, a good application minimizes the number of state changes since these are in general computationally expensive operations. The test scenes, shown in Appendix F, satisfy these conditions.

### 4.1.1 Estimation for data traffic over interconnect C

We estimate that the data traffic over interconnect C (Writing to off-chip memory during the capture phase) can be expressed as:

$$\begin{aligned} C &= (c1) \text{ Writing the display lists} \\ &+ (c2) \text{ Writing vertices (to the VAS)} \\ &+ (c3) \text{ Writing indices (to the IAS)} \end{aligned} \tag{4.2}$$

**c1: Writing the display lists** We estimate that the data traffic for updating the display lists can be expressed as:

$$\begin{aligned} (c1) \text{ Writing the display lists} = & \\ & (dl1) \text{ Writing primitive entries} \\ + & (dl2) \text{ Writing texture changes} \\ + & (dl3) \text{ Writing primitive changes} \\ + & (dl4) \text{ Writing clear changes} \\ + & (dl5) \text{ Writing end markers} \\ + & (dl6) \text{ Writing sub array changes} \end{aligned} \tag{4.3}$$

During scene capturing, several primitives and state changes are encountered. For our data traffic estimate, we ignore all state changes except texture state changes and the clear state change. The reason for this is that we assume the number of (other) state

 © Koninklijke Philips Electronics N.V. 2004

changes would be small and thus neglectable. Also, the texture state change and the clear state change are the most expensive to write to the display list (See Appendix E for details) and thus impact state change traffic the most.

To determine the amount of traffic required to update the display lists, we have to know the average overlap ratio. This is the number of tiles that a primitive covers. A primitive covering $n$ tiles, is sent to the rasterizer $n$ times. This also holds for texture changes impacting those primitives. For primitives we call the overlap ratio $O_p$ and for texture state changes we call it $O_t$. These values are dependant, among others, on the depth complexity of the scene, the screen resolution and the tile resolution.

Not every primitive has a separate primitive entry associated with it, since single primitives are grouped together as one primitive entry whenever possible. If the average length of a primitive entry is given by $A_{pl}$ and the number of primitives accepted by the TTSB module by $N_p$, the number of primitive entries equals:

$$N_{pe} = \frac{N_p * O_p}{A_{pl}} \tag{4.4}$$

A factor $f_{seq}$ ($0 \leq f_{seq} \leq 1$) of the primitive entries will be sequential primitive entries ($N_{pes}$) and a factor will be indexed primitive entries ($N_{pei}$). Therefore:

$$N_{pes} = f_{seq} * N_{pe} \tag{4.5}$$

$$N_{pei} = (1 - f_{seq}) * N_{pe} \tag{4.6}$$

Recall from section 3.2.5 that a primitive entry will be written as an indexed primitive entry if one of its vertices duplicated a previously stored vertex (of the current primitive or from another primitive). Therefore, in a scene with few duplicated vertices, many sequential primitive entries will be written and vice versa. Since our target application provides the geometry as connected primitives that do not duplicate many vertices[1], we estimate $f_{seq} \approx 0,8$ for now.

The size of an indexed primitive entry ($S_{pei}$) and the size of a sequential primitive entry ($S_{pes}$) need not be equal. The data traffic for writing all primitive entries can then be expressed as:

$$(dl1) \text{ Writing primitive entries} = (N_{pes} * S_{pes}) + (N_{pei} * S_{pei}) \tag{4.7}$$

The overlap ratio of a primitive $O_p$, is one of the central factors in evaluating the performance of TBR. There has been research evaluating the overlap ratio compared to different tile sizes and on how to predict the overlap ratio [11, 12, 8, 13]. In [13], Molnar predicts the overlap ratio as follows, given a set of input primitives with average width $A_w$ and average height $A_h$:

$$O_p = 1 + \frac{A_w}{D_{tx}} + \frac{A_h}{D_{ty}} + \frac{A_w * A_h}{D_{tx} * D_{ty}} \tag{4.8}$$

[1] A vertex that is shared by multiple connected primitives is currently not detected as a duplicate due to implementation issues. Only duplicates among independent primitives are found.

There is a close fit of expected overlap to observed overlap by assuming that the bounding boxes of the input primitives are square [8]. This means $A_w = A_h$. In this case, $A_w$ and $A_h$ can be given as follows [12]:

$$A_w = A_h = \sqrt{\frac{(D_{sx} * D_{sy}) * DC}{N_p}} \quad (4.9)$$

The measured overlap ratio from our test scenes and the overlap ratio predicted by equation 4.8 are given in Table 4.2.

| Tile size | 16x16 | 32x16 | 32x32 | 64x64 |
|---|---|---|---|---|
| Tequila scene | | | | |
| Measured primitive overlap ratio | 5.13 | 3.64 | 2.75 | 1.80 |
| Predicted primitive overlap ratio | 3.83 | 2.89 | 2.19 | 1.54 |
| Rally scene | | | | |
| Measured primitive overlap ratio | 5.59 | 3.76 | 2.83 | 1.83 |
| Predicted primitive overlap ratio | 3.91 | 2.95 | 2.22 | 1.55 |
| Temple scene | | | | |
| Measured primitive overlap ratio | 5.16 | 3.71 | 2.84 | 1.81 |
| Predicted primitive overlap ratio | 3.74 | 2.84 | 2.15 | 1.52 |

Table 4.2: Measured and predicted overlap ratio's ($O_p$).

Because there is some unexpected difference between the measured overlap ratio and the predicted overlap ratio, especially for smaller tile sizes, we use the measured overlap ratio $O_p$ as input for our formula instead of predicting it.

Because of the lack of good estimators for the average primitive length $A_{pl}$ and the texture state change overlap ratio $O_t$ we use measured values for those as well. These can be found in Table 4.3.

| Tile size | 16x16 | 32x16 | 32x32 | 64x64 |
|---|---|---|---|---|
| Tequila scene | | | | |
| Measured average primitive length | 2.56 | 2.80 | 2.99 | 3.32 |
| Measured texture overlap ratio | 94.68 | 53.43 | 31.95 | 12.78 |
| Rally scene | | | | |
| Measured average primitive length | 1.67 | 1.83 | 1.93 | 2.14 |
| Measured texture overlap ratio | 92.40 | 49.75 | 28.37 | 10.25 |
| Temple scene | | | | |
| Measured average primitive length | 2.53 | 2.75 | 2.89 | 3.15 |
| Measured texture overlap ratio | 58.94 | 33.13 | 19.85 | 7.63 |

Table 4.3: Measured average primitive length ($A_{pl}$) and measured texture state change overlap ratio ($O_t$).

Using the texture state change overlap ratio, the data traffic generated by writing tex-

 © Koninklijke Philips Electronics N.V. 2004

ture entries to the display lists can be expressed as follows:

$$(dl2) \text{ Writing texture changes} = N_t * O_t * S_t \tag{4.10}$$

Here $N_t$ is the number of texture changes encountered during scene recording and $S_t$ is the size of a texture entry in the display list.

Each display list starts with a clear state change (in general). This state change usually occurs just once per frame. It is responsible for clearing a set of buffers (Z-buffer and/or the RGBA-buffer). In a typical application the scene covers the entire screen and thus the clear state change impacts each tile. If the size of a clear entry in the display list is given by $S_{clear}$, we estimate the data traffic as follows:

$$(dl4) \text{ Writing clear changes} = N_{tx} * N_{ty} * S_{clear} \tag{4.11}$$

Since the module has no knowledge of the number of display list entries per display list, each display list ends with an end marker to signal the module to stop retrieving entries. An end marker being sized $S_{end}$, the data traffic the end markers generate can be expressed similar to that of the clear state changes:

$$(dl5) \text{ Writing end markers} = N_{tx} * N_{ty} * S_{end} \tag{4.12}$$

There are a number of primitive changes per display list. Since our target application provides all primitives as triangle strips, one might think there's only a single primitive change per tile. However, triangle parts, for example, of a triangle strip that are clipped to screen boundaries result in polygons that can not be considered part of any triangle strip anymore. These polygons are rendered as a series of independent triangles[2]. Tiles in the center of the screen are not likely to be impacted by clipped primitives. They contain few or probably even just one primitive change. For tiles near the edges of the screen, there will be several primitive changes per display list that flips the primitive type back and forth between triangle strips and independent triangles. The number of primitive changes ($N_{pc}$) will be given by measured values:

| Tile size | 16x16 | 32x16 | 32x32 | 64x64 |
|---|---|---|---|---|
| Tequila scene | | | | |
| Measured #primitive changes | 2471 | 1391 | 875 | 401 |
| Rally scene | | | | |
| Measured #primitive changes | 3375 | 1844 | 1096 | 457 |
| Temple scene | | | | |
| Measured #primitive changes | 2793 | 1632 | 1073 | 543 |

Table 4.4: Measured number of primitive changes ($N_{pc}$).

[2] The way it was implemented in [7]

Using $S_{pc}$, the size of a primitive change entry in the display list, the data traffic is then estimated:

$$(dl3) \text{ Writing primitive changes} = N_{pc} * S_{pc} \tag{4.13}$$

Remaining is the number of sub array changes ($N_{sa}$). These can be sub vertex array changes ($N_{sva}$) as well as sub index array changes ($N_{sia}$). The measured input values for those are listed in Table 4.5.

| Tile size | 16x16 | 32x16 | 32x32 | 64x64 |
|---|---|---|---|---|
| Tequila scene | | | | |
| Measured #sub array changes | 4912 | 2788 | 1670 | 654 |
| Measured #sub index array changes | 1564 | 879 | 534 | 211 |
| Measured #sub vertex array changes | 3348 | 1909 | 1136 | 443 |
| Rally scene | | | | |
| Measured #sub array changes | 7273 | 3841 | 2154 | 748 |
| Measured #sub index array changes | 1833 | 975 | 524 | 178 |
| Measured #sub vertex array changes | 5440 | 2866 | 1630 | 570 |
| Temple scene | | | | |
| Measured #sub array changes | 4470 | 2579 | 1615 | 638 |
| Measured #sub index array changes | 1214 | 705 | 457 | 193 |
| Measured #sub vertex array changes | 3256 | 1874 | 1158 | 445 |

Table 4.5: Measured number of sub array changes ($N_{sa}$).

Given that a sub index array change entry equals the size of a SVA change entry and that that size is given as $S_{sa}$, we estimate the data traffic as:

$$(dl6) \text{ Writing sub array changes} = N_{sa} * S_{sa} \tag{4.14}$$

**c2: Writing vertices (to the VAS)** Initially we assume the scene does not contain any shared vertices. Then every vertex in the scene is unique and is written to the VAS. Given an average vertex size of $S_v$, the data traffic for this operation can be expressed as:

$$(c2) \text{ Writing vertices} = N_v * S_v \tag{4.15}$$

In general the above assumption isn't true. A scene contains shared vertices and those vertices will be detected, whenever possible, by our module and only unique vertices are written to the VAS. The number of unique vertices in the scene is a factor ($f_{uniq}$; $0 < f_{uniq} \leq 1$) of the total number of vertices in the scene. Taking shared vertices in mind, formula 4.15 is restated as:

$$(c2) \text{ Writing vertices} = (f_{uniq} * N_v) * S_v \tag{4.16}$$

Since we expect a strong coherence between $f_{seq}$ and $f_{uniq}$ (many unique vertices means many sequential entries) we estimate $f_{uniq} \approx 0,8$ as well for now.

 © Koninklijke Philips Electronics N.V. 2004

**c3: Writing indices** The cost for writing indices to accommodate for indexed primitives, needs to be calculated as well. Recall that our target application provides all primitives as connected primitives. Connected primitives are rarely stored as indexed primitives because every vertex in the connected primitive is in general unique. However the independent triangles near the screen boundaries that result from clipping primitives will almost always be stored as indexed primitives because they always share vertices among them. Since an independent triangle requires three indices, we approximate the number of indices as:

$$\#\text{Indices} = \#\text{Triangles resulting from clipping} * 3 \tag{4.17}$$

The data traffic generated by writing indices is expressed as;

$$(c3)\text{ Writing indices} = \#\text{Indices} * S_i \tag{4.18}$$

Where $S_i$ is the size of a single index.

### 4.1.2 Estimation for data traffic over interconnect D

We estimate that the data traffic over interconnect D (Reading from off-chip memory during the render phase) can be expressed as:

$$\begin{aligned} D \;=\; & \\ & (d1)\text{ Reading the display lists} \\ + \; & (d2)\text{ Reading vertices (from the VAS)} \\ + \; & (d3)\text{ Reading indices (from the IAS)} \end{aligned} \tag{4.19}$$

**d1: Reading the display lists** Since each display list is written once and read once, reading the display lists costs as much as it took to write them.

$$(d1)\text{ Reading the display lists} = \text{Writing the display lists} \tag{4.20}$$

**d2: Reading vertices (from the VAS)** The traffic generated by reading vertices is heavily influenced by our on-chip vertex cache behaviour. For now, we assume that the cache never hits (e.g. there is no vertex cache). Then, each time a primitive is rasterized, all of its vertices are requested from off-chip memory. Therefore:

$$(d2)\text{ Reading vertices} = N_v * S_v * O_p \tag{4.21}$$

Where $S_v$ is the average vertex size of a vertex stored in the VAS.

Depending on the cache strategy, a vertex that is requested from the on-chip vertex cache is already found in that vertex cache and thus need not be transferred from off-chip memory again. This decreases the vertex traffic by a factor $f_{hit}$ ($0 \leq f_{hit} \leq 1$):

$$(d2)\text{ Reading vertices} = (1 - f_{hit}) * (N_v * S_v * O_p) \quad (4.22)$$

When rendering of a tile is finished, the on-chip vertex cache will be filled with vertices of primitives that impacted that tile. If those primitives also impact the next tile to be rendered, the vertex cache will hit often since many of those vertices are already in the vertex cache. So $f_{hit}$ will increase when $O_p$ increases. Given a scene with a fixed number of primitives, a fixed resolution and a FIFO vertex cache strategy, for $O_p$ to increase, the tile size needs to decrease. So the smaller the tile size, the bigger we expect $f_{hit}$ to be. The measured values show this behavior as seen in Table 4.6.

| Tile size | 16x16 | 32x16 | 32x32 | 64x64 |
|---|---|---|---|---|
| Tequila scene | | | | |
| Measured cache hit ratio | 0.67 | 0.55 | 0.51 | 0.39 |
| Rally scene | | | | |
| Measured cache hit ratio | 0.57 | 0.47 | 0.39 | 0.28 |
| Temple scene | | | | |
| Measured cache hit ratio | 0.65 | 0.53 | 0.48 | 0.37 |

Table 4.6: Measured vertex cache hit ratio's ($f_{hit}$).

Note that the above is true because in our implementation tiles are rendered in sequential, adjacent order. When tiles are rendered in a random order or multiple rasterizers render several tiles in parallel, the cache hit ratio will most likely be smaller.

**d3: Reading indices (from the IAS)** The overlap ratio of a primitive influences the number of times an index from the IAS is read. As explained in section 3, the number of indices written is directly influenced by the number of clipped primitives. Because clipped primitives lie on the edges of the screen and generally occupy an area that is larger than average they also have an above average primitive overlap ratio. The estimation below which is using the average primitive overlap ratio is expected to be smaller than the actual traffic.

$$(d3)\text{ Reading indices} = (\#\text{Indices} * S_i) * O_p \quad (4.23)$$

### 4.1.3 Estimation for data traffic over interconnect E

We estimate that the data traffic over interconnect E (Writing the on-chip color buffer to the off-chip framebuffer) can be expressed as:

$$E = \quad (4.24)$$
$$(e1)\text{ Pixel data copied (to off-chip memory)}$$

 © Koninklijke Philips Electronics N.V. 2004

**e1: Pixel data copied** After rendering of a tile is finished, the on-chip tile framebuffer must be written to the framebuffer in off-chip memory over interconnect E. This can be done in fast block-based transfers. The data traffic this generates depends on the screen resolution (since the number of pixels of all tiles combined equals the number of pixels of the screen resolution) and color precision ($S_{rgba}$) and can be expressed as:

$$(e1)\text{ Pixel data copied} = D_{sx} * D_{sy} * S_{rgba} \tag{4.25}$$

### 4.1.4 Estimation for data traffic over interconnect A

We estimate that the data traffic over interconnect A (On-chip Z-buffer traffic) can be expressed as:

$$\begin{aligned} A \quad = & \\ & (a1)\text{ Z-buffer clear} \\ + & \ (a2)\text{ Z-buffer tests (reads)} \\ + & \ (a3)\text{ Z-buffer updates (writes)} \end{aligned} \tag{4.26}$$

**a1: Z-buffer clear** Clearing the Z-buffer is in general done once every frame at the very beginning. Clearing the Z-buffer is done by writing a value (usually the depth of the far plane) to every location in the Z-buffer. If the depth precision is given by $S_z$ clearing the Z-buffer can be expressed as:

$$(a1)\text{ Z-buffer clear} = D_{sx} * D_{sy} * S_z \tag{4.27}$$

If the hardware supports resetting blocks of memory to a certain value, the bandwidth for clearing the Z-buffer can be saved.

**a2: Z-buffer tests (reads)** All other accesses involving the Z-buffer are tests against the current Z-buffer value for a pixel (reading) and updating the Z-buffer value for a pixel (writing). The number of times these accesses take place per pixel depends on the average depth complexity ($DC$), also called overdraw factor. This is the average number of times that for a pixel the Z-value is evaluated. The memory traffic for read accesses into the Z-buffer can be expressed as:

$$(a2)\text{ Z-buffer reads} = (D_{sx} * D_{sy} * S_z) * DC \tag{4.28}$$

**a3: Z-buffer updates (writes)** The number of Z writes is not equal to the number of Z reads, since the Z-value of a pixel is only updated if it passed the Z-test. It can be

© Koninklijke Philips Electronics N.V. 2004 

shown that the number of pixels that pass the Z-test for a scene with depth complexity $DC$ is given by (assuming no attempts at Z sorting the polygons are made):

$$f_{zpass} = \sum_{n=1}^{DC} \left(\frac{1}{n * DC}\right) \tag{4.29}$$

The memory traffic for write accesses into the Z-buffer can then be expressed as:

$$(a3)\text{ Z-buffer writes} = (D_{sx} * D_{sy} * S_z) * DC * f_{zpass} \tag{4.30}$$

### 4.1.5 Estimation for data traffic over interconnect B

We estimate that the data traffic over interconnect B (On-chip color buffer traffic) can be expressed as:

$$B = \tag{4.31}$$
$$(b1)\text{ RGBA-buffer updates (writes)}$$

**b1: RGBA-buffer updates (writes)** Whenever a pixel passes the Z-test, its Z-buffer value is updated, but also the RGBA value in the RGBA buffer is updated to reflect the RGBA value of the pixel that just passed the Z-test. Therefore memory traffic for RGBA-buffer write accesses can be expressed as:

$$(b1)\text{ RGBA-buffer writes} = (D_{sx} * D_{sy} * S_{rgba}) * DC * f_{zpass} \tag{4.32}$$

A common scenario in a typical application today is that the 3D scene covers the entire screen, which means the RGBA buffer need not be cleared.

## 4.2 Measured performance

### 4.2.1 Test environment

We wish to test Formula 4.1 on conditions similar to those encountered in its intended use. As said, its intended use is 3D games on a low-end mobile phone platform. A common benchmark on (PC) accelerators is the game "Quake III Arena", from now on referred to as Q3. We expect that games with a scene complexity similar to Q3, will appear on our target platform in the near future and thus Q3 makes an excellent test environment for our module.

The algorithm implementation from [7] in the Mesa 3-D pipeline was modified by implementing all our design decisions. Because of legacy reasons every vertex includes not one but two color attributes (front and back), even though OpenGL ES does not support this. This increases the data traffic for reading and writing vertices as this increases the vertex size.

 © Koninklijke Philips Electronics N.V. 2004

Also because of legacy reasons, not every OpenGL call that changes the rasterizer state (and thus should be captured) is supported. The reason for this is that these calls are not needed to be supported to render the VRML test scenes used in [7] and therefore were never implemented. Every call that changes the rasterizer state and isn't listed in Appendix E is unsupported. For commercial use, an implementation that conforms to all OpenGL standards is desired, but for our evaluation purposes this is not required.

All tests were performed with Edge Anti-Aliasing enabled.

All tests were performed on a system running Microsoft Windows 2000. The Mesa 3-D graphics pipeline including our TBR modifications was compiled to a dynamic link library (`opengl32.dll`), which was placed in the Q3 directory. This way Q3 loads our graphics library instead of the one provided by the operating system.

### 4.2.2 Test resolution and tile sizes

We've established that our test resolution should be standard VGA (640x480). For this resolution we've evaluated the module performance for the following tile sizes: 16x16, 32x32, 64x64 and 32x16. This results in the statistics of Table 4.7.

| Resolution | Tile size | Tiles (along X) | Tiles (along Y) | Tiles (Total) |
|---|---|---|---|---|
| 640x480 | 16x16 | 40 | 30 | 1200 |
| 640x480 | 32x16 | 20 | 30 | 600 |
| 640x480 | 32x32 | 20 | 15 | 300 |
| 640x480 | 64x64 | 10 | 8 (7,5) | 80 |

Table 4.7: Evaluated tile sizes.

### 4.2.3 Test scenes

We have chosen several test scenes of different variety representative for the target application. These are described in Appendix F. Geometry statistics of the test scenes can be found in Table 4.8. Scene statistics of the test scenes can be found in Table 4.9. From Table 4.9 follows that in all cases the number of state changes (other than texture changes) is small, both absolute and compared to the number of texture changes. This justifies our decision to ignore them in our bandwidth calculations since they will be of small impact.

| | | Tequila | Rally | Temple |
|---|---|---|---|---|
| Number of single primitives (TTSB input) | | 3558 | 5089 | 4376 |
| Accepted (in view volume & front facing) | | 2525 | 3136 | 2766 |
| Accepted as the result of clipping | | 747 | 686 | 1082 |
| Rejected by clipping | | 62 | 145 | 86 |
| Rejected by back-face culling | | 720 | 1562 | 1176 |
| Number of vertices: (TTSB output) | $N_v$ | 6264 | 8248 | 7988 |
| Number of single primitives (TTSB output) | $N_{pr}$ | 3272 | 3822 | 3848 |

Table 4.8: Geometry statistics.

| | | Tequila | Rally | Temple |
|---|---|---|---|---|
| Depth complexity | $DC$ | 2.50 | 3.05 | 2.80 |
| Number of texture changes | $N_t$ | 37 | 60 | 62 |
| Number of other state changes | | 8 | 14 | 26 |

Table 4.9: Scene statistics.

### 4.2.4 Measured results

We now present the results of our measurements and estimations. For our estimations several parameters to our formula that are fixed for every test scene are required. These are given in Table 4.10.

| Parameter | | Value |
|---|---|---|
| Screen width | $D_{sx}$ | 640 |
| Screen height | $D_{sy}$ | 480 |
| Size indexed primitive entry | $S_{pei}$ | 16 bits |
| Size sequential primitive entry | $S_{pes}$ | 16 bits |
| Size texture change entry | $S_{tc}$ | 19 bits |
| Size predefined state changes | $S_{sc}$ | 11 bits |
| Size compressed vertex | $S_v$ | 20 bytes |
| Size clear change entry | $S_{clear}$ | 38 bits |
| Size end marker entry | $S_{end}$ | 3 bits |
| Size sub vertex array entry | $S_{sa}$ | 16 bits |
| Size sub index array entry | $S_{sa}$ | 16 bits |
| Size primitive type entry | $S_{pc}$ | 11 bits |
| Precision framebuffer | $S_{rgba}$ | 32 bits |
| Precision z-buffer | $S_z$ | 32 bits |

Table 4.10: Fixed input parameters.

 © Koninklijke Philips Electronics N.V. 2004

Measured parameters and memory traffic on the scenes for different tile sizes are given in Tables 4.11 to 4.16.

| Tequila scene | | | | | |
|---|---|---|---|---|---|
| Factor | | 16x16 | 32x16 | 32x32 | 64x64 |
| Unique vertices | $f_{uniq}$ | 0.744 | 0.744 | 0.744 | 0.744 |
| Sequential entries | $f_{seq}$ | 0.691 | 0.715 | 0.735 | 0.853 |
| Total primitive entries | $N_{pe}$ | 6563 | 4250 | 3015 | 1778 |
| Indexed primitive entries | $N_{pei}$ | 2029 | 1211 | 800 | 390 |
| Sequential primitive entries | $N_{pes}$ | 4534 | 3039 | 2215 | 1388 |
| Sub vertex array entries | $N_{sva}$ | 3348 | 1909 | 1136 | 443 |
| Sub index array entries | $N_{sia}$ | 1564 | 879 | 534 | 211 |
| Primitive change entries | $N_{pc}$ | 2471 | 1319 | 875 | 401 |
| Vertex cache hit ratio | $f_{hit}$ | 0.670 | 0.547 | 0.509 | 0.393 |
| Primitive overlap ratio | $O_p$ | 5.129 | 3.637 | 2.753 | 1.804 |
| Texture changes overlap ratio | $O_t$ | 94.676 | 53.432 | 31.946 | 12.784 |
| Average primitive entry length | $A_{pl}$ | 2.557 | 2.80 | 2.988 | 3.319 |

Table 4.11: Scene dependant parameters for different tile sizes for the Tequila scene.

| Tequila scene | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 16x16 | | 32x16 | | 32x32 | | 64x64 | |
| | Formula | Test | Formula | Test | Formula | Test | Formula | Test |
| $c1$ | 40805 | 41752 | 23695 | 24238 | 14907 | 15177 | 6950 | 7080 |
| $c2$ | 100224 | 93200 | 100224 | 93200 | 100224 | 93200 | 100224 | 93200 |
| $c3$ | 2241 | 2202 | 2241 | 2202 | 2241 | 2202 | 2241 | 2202 |
| $C$ | 143270 | 137154 | 126160 | 119640 | 117372 | 110579 | 109415 | 102482 |
| $d1$ | 40805 | 41752 | 23695 | 24238 | 14907 | 15177 | 6950 | 7080 |
| $d2$ | 212045 | 275200 | 206406 | 256640 | 169344 | 206360 | 137185 | 156980 |
| $d3$ | 11494 | 23841 | 8151 | 15555 | 6169 | 11409 | 4043 | 6393 |
| $D$ | 264344 | 340793 | 238252 | 296433 | 190420 | 232946 | 148178 | 170453 |
| $E$ | 1228800 | | | | | | | |
| $A$ | 6365642 | | | | | | | |
| $B$ | 2064842 | | | | | | | |

Table 4.12: Memory traffic (in bytes) for the Tequila scene.

© Koninklijke Philips Electronics N.V. 2004 

| Rally scene | | | | | |
|---|---|---|---|---|---|
| Factor | | 16x16 | 32x16 | 32x32 | 64x64 |
| Unique vertices | $f_{uniq}$ | 0.830 | 0.830 | 0.830 | 0.830 |
| Sequential entries | $f_{seq}$ | 0.694 | 0.723 | 0.774 | 0.841 |
| Total primitive entries | $N_{pe}$ | 12779 | 7845 | 5615 | 3265 |
| Indexed primitive entries | $N_{pei}$ | 3912 | 2143 | 1271 | 518 |
| Sequential primitive entries | $N_{pes}$ | 8867 | 5702 | 4344 | 2747 |
| Sub vertex array entries | $N_{sva}$ | 5440 | 2866 | 1630 | 570 |
| Sub index array entries | $N_{sia}$ | 1833 | 975 | 524 | 178 |
| Primitive change entries | $N_{pc}$ | 3375 | 1844 | 1096 | 457 |
| Vertex cache hit ratio | $f_{hit}$ | 0.574 | 0.474 | 0.391 | 0.279 |
| Primitive overlap ratio | $O_p$ | 5.591 | 3.763 | 2.837 | 1.826 |
| Texture changes overlap ratio | $O_t$ | 92.400 | 49.750 | 28.366 | 10.250 |
| Average primitive entry length | $A_{pl}$ | 1.672 | 1.834 | 1.931 | 2.137 |

Table 4.13: Scene dependant parameters for different tile sizes for the Rally scene.

| Rally scene | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 16x16 | | 32x16 | | 32x32 | | 64x64 | |
| | Formula | Test | Formula | Test | Formula | Test | Formula | Test |
| $c1$ | 64090 | 68084 | 36088 | 38207 | 22603 | 23808 | 10532 | 11014 |
| $c2$ | 131968 | 136880 | 131968 | 136880 | 131968 | 136880 | 131968 | 136880 |
| $c3$ | 2058 | 2032 | 2058 | 2032 | 2058 | 2032 | 2058 | 2032 |
| $C$ | 198116 | 206996 | 170114 | 177119 | 156629 | 162720 | 144558 | 149926 |
| $d1$ | 64090 | 68084 | 36088 | 38207 | 22603 | 23808 | 10532 | 11014 |
| $d2$ | 392896 | 481880 | 326512 | 387340 | 285007 | 329500 | 217177 | 237500 |
| $d3$ | 11506 | 26153 | 7744 | 16637 | 5839 | 11351 | 3758 | 6031 |
| $D$ | 468492 | 576117 | 370344 | 442184 | 313449 | 364659 | 231467 | 254545 |
| $E$ | 1228800 | | | | | | | |
| $A$ | 7246755 | | | | | | | |
| $B$ | 2270115 | | | | | | | |

Table 4.14: Memory traffic (in bytes) for the Rally scene.

 © Koninklijke Philips Electronics N.V. 2004

| Temple scene | | | | | |
|---|---|---|---|---|---|
| Factor | | 16x16 | 32x16 | 32x32 | 64x64 |
| Unique vertices | $f_{uniq}$ | 0.696 | 0.696 | 0.696 | 0.696 |
| Sequential entries | $f_{seq}$ | 0.771 | 0.792 | 0.803 | 0.832 |
| Total primitive entries | $N_{pe}$ | 7832 | 5201 | 3782 | 2220 |
| Indexed primitive entries | $N_{pei}$ | 1789 | 1081 | 745 | 373 |
| Sequential primitive entries | $N_{pes}$ | 6043 | 4120 | 3037 | 1847 |
| Sub vertex array entries | $N_{sva}$ | 3256 | 1874 | 1158 | 445 |
| Sub index array entries | $N_{sia}$ | 1214 | 705 | 457 | 193 |
| Primitive change entries | $N_{pc}$ | 2793 | 1632 | 1073 | 543 |
| Vertex cache hit ratio | $f_{hit}$ | 0.647 | 0.525 | 0.478 | 0.369 |
| Primitive overlap ratio | $O_{pr}$ | 5.155 | 3.713 | 2.843 | 1.817 |
| Texture changes overlap ratio | $O_{tc}$ | 58.935 | 33.129 | 19.855 | 7.629 |
| Average primitive entry length | $A_{plen}$ | 2.533 | 2.747 | 2.893 | 3.150 |

Table 4.15: Scene dependant parameters for different tile sizes for the Temple scene.

| Temple scene | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 16x16 | | 32x16 | | 32x32 | | 64x64 | |
| | Formula | Test | Formula | Test | Formula | Test | Formula | Test |
| $c1$ | 43305 | 44558 | 25740 | 26443 | 16729 | 17090 | 7978 | 8199 |
| $c2$ | 127808 | 111220 | 127808 | 111220 | 127808 | 111220 | 127808 | 111220 |
| $c3$ | 3246 | 3207 | 3246 | 3207 | 3246 | 3207 | 3246 | 3207 |
| $C$ | 174359 | 158985 | 156794 | 140870 | 147783 | 131517 | 139032 | 122626 |
| $d1$ | 43305 | 44558 | 25740 | 26443 | 16729 | 17090 | 7978 | 8199 |
| $d2$ | 290718 | 343560 | 281765 | 321600 | 237091 | 267180 | 183169 | 197540 |
| $d3$ | 16733 | 25020 | 12052 | 17019 | 9228 | 12837 | 5898 | 7440 |
| $D$ | 350756 | 413138 | 319557 | 365062 | 263048 | 297107 | 197045 | 213179 |
| $E$ | 1228800 | | | | | | | |
| $A$ | 6850444 | | | | | | | |
| $B$ | 2181004 | | | | | | | |

Table 4.16: Memory traffic (in bytes) for the Temple scene.

© Koninklijke Philips Electronics N.V. 2004 

## 4.3 Evaluation

### 4.3.1 Formula versus test measurements

In this section we will compare the predicted figures of our formula against the measured figures and how well they relate.

The cost for writing indices, $c3$, is predicted by equation 4.18 almost perfectly. Given equation 4.17 this means that indexed primitive entries solely consist of series of independent triangles that resulted from clipping. The polygon that results from clipping a triangle to screen boundaries could have been rendered as a triangle fan instead of a series of independent triangles. A triangle fan requires fewer vertices compared to a series of independent triangles and does not duplicate vertices, since it's a connected primitive. Then all geometry is provided as connected primitives that are generally stored as sequential primitives, possibly making support for indexed primitives and duplicate detection superfluous compared to the complexity it adds to the system. An argument for keeping support for indexed primitives in the system is that it's an efficient way to speed up rendering of scenes where the geometry is not provided as connected primitives but as independent primitives. In this scenario it tries to compensate the programmers' decision of not using connected primitives by detecting the many shared vertices as duplicates among the primitive data provided.

The traffic for writing and reading the display lists, $c1$ and $d1$ respectively, are also estimated very good (See Tables 4.12, 4.14 and 4.16). It should be noted though that in those calculations we've made extensive use of measured input values (Examples are: $A_{pl}$, $O_t$, $N_{pc}$ and $N_{sa}$), which significantly improves the correctness of our estimation. The predicted traffic is in all cases lower then the measured traffic. This might be because of the exclusion of the other state changes in the calculation as well as the overhead of `DLOP_CONTINUE` (See Appendix E) display list entries that link blocks of display list entries together.

Our estimation of $c2$, the cost for writing vertices, is greatly influenced by the factor of unique vertices ($f_{uniq}$). Recall that we did not use measured input values for $f_{uniq}$, but estimated it to be approximately 0.8. Tables 4.11, 4.13 and 4.15 show this was not a bad estimate, even though analysis of more test scenes could improve the approximation. Because of this good estimation, the estimation for $c2$ is good as well. The Temple scene, for which $f_{uniq}$ differs the most from 0.8 of all test scenes, has the greatest error in $c2$ and this error is around 13%.

Reading indices, $d3$, is not approximated very well, especially for small tile sizes where the error can be over 50%. Recall from equation 4.23 that our estimation uses $O_p$, which is the average primitive overlap ratio. We already established that indexed primitives solely consist of independent primitives that resulted from clipping. Those primitives lie near the boundaries of the screen, where primitives in our scenes are generally bigger (for example: walls built from large polygons). Bigger primitives have a larger than average primitive overlap ratio $O_p$. This explains the error in our estimation of $d3$. We expect the estimation to improve severely if the overlap ratio of independent primitives that result from clipping was measured separately and used in the calculation of $d3$ instead of $O_p$.

 © Koninklijke Philips Electronics N.V. 2004

Traffic for reading vertices, $d2$, impacts the traffic over both interconnects C and D the most, as is immediately obvious from Tables 4.12, 4.14 and 4.16 and it's therefore important to predict it as accurate as possible. The relative error of the estimated traffic compared to the measured traffic becomes smaller, the bigger the tile size. For our test scenes the maximum error occurs for the Tequila scene and is smaller than 30%.

Given the above, we think the formula can effectively be used to approximate the off-chip memory bandwidth of our TBR system. Some parts of the formula are more usable than others though. Especially the parts that have a big impact on the total off-chip memory traffic are interesting. These are writing vertices ($c2$), reading vertices ($d2$) and copying pixel data to the off-chip framebuffer ($E$). The sum of these parts can be used as a rough-and-ready estimation for the total traffic of our TBR system.

### 4.3.2 Optimal tile size

We've evaluated 4 different tile dimensions. Each of these tile dimensions covers a certain area of pixels, see Table 4.17.

| Tile dimensions | Number of pixels |
|---|---|
| 16x16 | 256 |
| 32x16 | 512 |
| 32x32 | 1024 |
| 64x64 | 4096 |

Table 4.17: Number of pixels covered by a tile for different dimensions.

Larger tile sizes mean less traffic over interconnects C, D and E, mainly because the overlap ratios decrease. This can be clearly seen from Figure 4.2. This might give the impression that the optimal tile size is equal to the screen resolution (in this case 640x480). That tile resolution would generate the least traffic, but it's not viable, since it's too expensive given today's hardware to put a color buffer of that size on-chip.

Figure 4.2 also shows that increasing the tile area from 256 (16x16) to 512 (32x16) and from 512 (32x16) to 1024 (32x32) both substantially decrease traffic over interconnects C, D and E. However, the increase in tile area from 1024 (32x32) to 4096 (64x64) only results in marginal gains when one compares it to the increase in tile size. This leads us to believe that, in view of the increase ratio in tile size and the traffic over interconnects C, D and E for that tile size, the optimum for the tile size of the tile sizes we've considered, is 32x32. This believe is strengthened by the results of [12], which investigates the optimal tile size more in depth and in the end draws the same conclusion.

The behavior of the traffic over interconnects C, D and E looks very much alike for all test scenes. Differences seem to be solely influenced by scene characteristics such as the number of vertices and the number of primitives.

© Koninklijke Philips Electronics N.V. 2004 

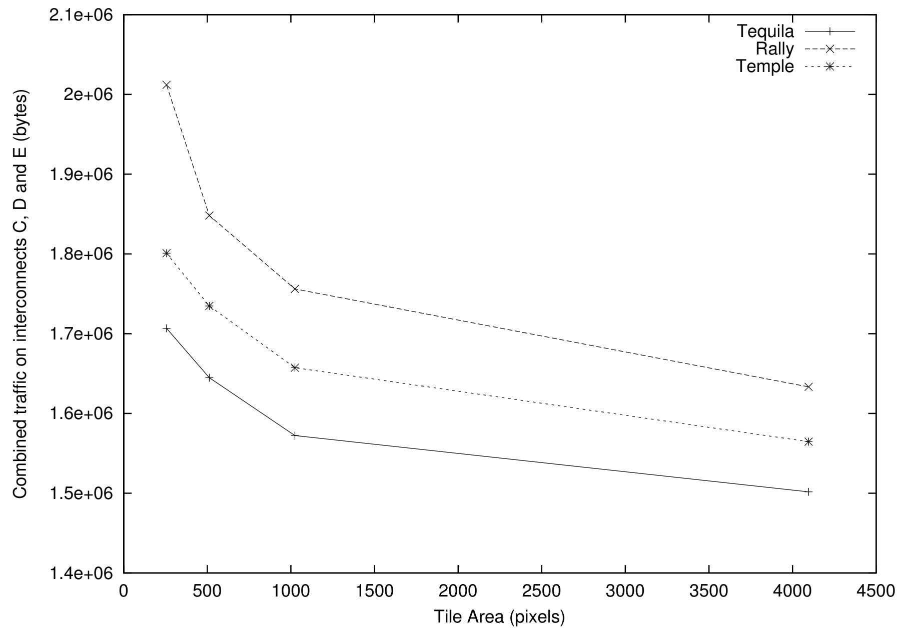



Figure 4.2: Traffic over interconnects C, D and E relative to tile area.

### 4.3.3 TBR versus FBR

Tables 4.18 and 4.19 show the measured savings of TBR with respect to FBR. The result is positive in all cases, meaning TBR is beneficial for all evaluated tile sizes. The savings are mostly due to the fact that all Z-buffer traffic is kept on-chip. Z-buffer related traffic is a known bottleneck in today's graphics accelerators and this bottleneck is one of the reasons why TBR is considered in the first place.

| | A+B-(C+D+E) | | | |
|---|---|---|---|---|
| | 16x16 | 32x16 | 32x32 | 64x64 |
| Tequila | 6723737 | 6785611 | 6858159 | 6938031 |
| Rally | 7504957 | 7668767 | 7760691 | 7883599 |
| Temple | 7230525 | 7296716 | 7374024 | 7466843 |

Table 4.18: Savings of TBR with respect to FBR for different tile sizes in bytes.

 © Koninklijke Philips Electronics N.V. 2004

| | A+B-(C+D+E) | | | |
|---|---|---|---|---|
| | 16x16 | 32x16 | 32x32 | 64x64 |
| Tequila | -79.8% | -80.5% | -81.3% | -82.2% |
| Rally | -78,9% | -80.6% | -81.5% | -82.8% |
| Temple | -80.1% | -80.8% | -81.6% | -82.7% |

Table 4.19: Savings of TBR with respect to FBR for different tile sizes in terms of percentage.

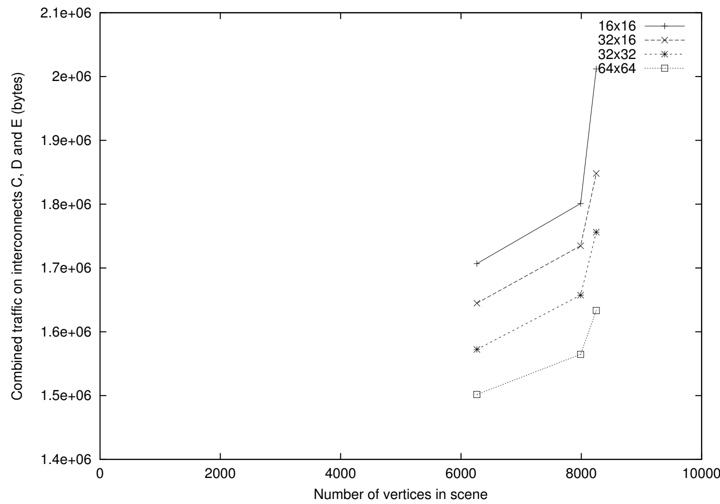



Figure 4.3: Traffic over interconnects C, D and E relative to the number of vertices in the scene.

Table 4.19 gives the impression that the tile size has a negligible impact on the savings of TBR with respect to FBR. However, looking at Tables 4.12, 4.14 and 4.16 we see that the traffic over interconnect C and interconnect D combined can differ up to almost 50% between a tile size of 16x16 and one of 64x64. This difference however has little impact on the total traffic, because for our test resolution of 640x480 pixels the traffic over interconnect A and interconnect B is significantly bigger.

Since vertex traffic influences the combined traffic over interconnects C and D the most, it is expected that scenes with higher vertex count will also have higher off-chip memory traffic. This is correct, as can be seen in Figure 4.3. Figures 4.3 and 4.4 show, based on only 3 test scenes though, that when the vertex count or primitive count increases, the traffic over interconnects C, D and E increases more rapidly for smaller tile sizes than for bigger tile sizes.

Vital information would be to know up to what vertex count TBR stays beneficial over FBR. We will first evaluate a worst-case scenario. We will evaluate that scenario using our derived formula and several parameters that maximize the traffic for TBR

© Koninklijke Philips Electronics N.V. 2004 

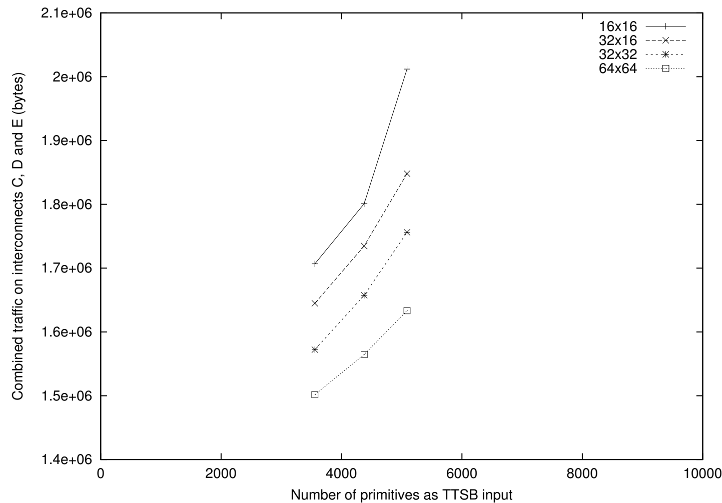



Figure 4.4: Traffic over interconnects C, D and E relative to the number of primitives as TTSB input.

and minimize the traffic for FBR:

- A single rendered frame.
- A resolution of 640x480 pixels.
  - $D_{sx} = 640$
  - $D_{sy} = 480$
- A tile size of 16x16 pixels. From Tables 4.11, 4.13 and 4.15 we see that for a tile size of 16x16 pixels, the overlap ratio $O_p$ is never above 6.
  - $D_{tx} = 16$
  - $D_{ty} = 16$
  - $O_p = 6$
- Every vertex in the scene is unique. In this case the traffic generated by TBR for writing vertices is at a maximum.
  - $f_{uniq} = 1$
- The cache never hits. In this case the traffic for reading vertices generated by TBR is at a maximum.
  - $f_{hit} = 0$
- A minimal depth complexity, since this means minimal traffic for FBR. Assuming a scene covers the entire screen we have a minimum depth complexity of 1.
  - $DC = 1$

 © Koninklijke Philips Electronics N.V. 2004

Note that in reality, both $DC$ and $O_p$ are dependent on the scene complexity (primitive count and/or vertex count), but for the sake of this evaluation we assume them constant. Estimating $DC$ and $O_p$ given the number of vertices in the scene ($N_v$) is a topic for further research.

The traffic for FBR (interconnect A and interconnect B) in bytes is given, using Formulas 4.26 and 4.31, by:

$$\begin{aligned}
\text{Traffic for FBR} &= A + B \qquad (4.33)\\
&= ((D_{sx} * D_{sy} * S_z) * (1 + DC + DC * f_{zpass})) + ((D_{sx} * D_{sy} * S_{rgba}) * DC * f_{zpass})\\
&= ((640 * 480 * 4) * (1 + 1 + 1 * 1) + ((640 * 480 * 4) * 1 * 1)\\
&= 3686400 + 1228800\\
&= 4915200
\end{aligned}$$

Looking at Tables 4.12, 4.14 and 4.16, we see that the traffic for TBR (interconnect C, interconnect D and interconnect E) is influenced the most by the vertex traffic (writing them over interconnect C and reading them over interconnect D) and the traffic over interconnect E (equation 4.24). We know that the vertex traffic is determined by writing vertices (equation 4.16) and reading vertices (4.22). The traffic for TBR in bytes is thus approximated using our rough-and-ready rule:

$$\begin{aligned}
\text{Traffic for TBR} &= \text{Vertex traffic} + E \qquad (4.34)\\
&= c2 + d2 + E\\
&= (f_{uniq} * S_v * N_v) + (1 - f_{hit}) * (S_v * N_v * O_p) + (D_{sx} * D_{sy} * S_{rgba})\\
&= (1 * 20 * N_v) + (1 * (20 * N_v * 6)) + (640 * 480 * 4)\\
&= 140 * N_v + 1228800
\end{aligned}$$

Solving this equation against the traffic for FBR gives:

$$\begin{aligned}
4915200 &= 140 * N_v + 1228800 \qquad (4.35)\\
3686400 &= 140 * N_v\\
26331 &= N_v
\end{aligned}$$

This tells us that for scenes that contain up to 26K vertices, our TBR system will use less off-chip memory data traffic than the classic FBR approach. Remember that this is a worst-case scenario. With more realistic numbers for the depth complexity, number of unique vertices and vertex cache hit ratio that vertex count will increase. We evaluate that using the the following, more realistic, parameters:

- A single rendered frame.
- A resolution of 640x480 pixels.
  - $D_{sx} = 640$

© Koninklijke Philips Electronics N.V. 2004 

  - $D_{sy} = 480$

- A tile size of 32x32 pixels. From Tables 4.11, 4.13 and 4.15 we see that for a tile size of 32x32 pixels, the overlap ratio $O_p$ is never above 3.
  - $D_{tx} = 32$
  - $D_{ty} = 32$
  - $O_p = 3$
- A unique vertices factor of 0.8. This is equal to the value used in all previous calculations.
  - $f_{uniq} = 0.8$
- A vertex cache hit ratio of 0.4. Looking at Tables 4.11, 4.13 and 4.15 this can be seen as a realistic value.
  - $f_{hit} = 0.4$
- A depth complexity of 3. Looking at Table 4.9 this can be seen as a realistic value.
  - $DC = 3$

The traffic for FBR (interconnect A and interconnect B) in bytes is given, using Formulas 4.26 and 4.31, by:

$$\begin{aligned}
\text{Traffic for FBR} &= A + B \qquad (4.36)\\
&= ((D_{sx} * D_{sy} * S_z) * (1 + DC + DC * f_{zpass})) + ((D_{sx} * D_{sy} * S_{rgba}) * DC * f_{zpass})\\
&= ((640 * 480 * 4) * (1 + 3 + 3 * \frac{11}{18})) + ((640 * 480 * 4) * 3 * \frac{11}{18})\\
&= 7168000 + 2252800\\
&= 9420800
\end{aligned}$$

The traffic for TBR in bytes is then approximated using our rough-and-ready rule:

$$\begin{aligned}
\text{Traffic for TBR} &= \text{Vertex traffic} + E \qquad (4.37)\\
&= c2 + d2 + E\\
&= (f_{uniq} * S_v * N_v) + (1 - f_{hit}) * (S_v * N_v * O_p) + (D_{sx} * D_{sy} * S_{rgba})\\
&= (0.8 * 20 * N_v) + (0.6 * (20 * N_v * 3)) + (640 * 480 * 4)\\
&= 52 * N_v + 1228800
\end{aligned}$$

Solving this equation against the traffic for FBR gives:

$$\begin{aligned}
9420800 &= 52 * N_v + 1228800 \qquad (4.38)\\
8192000 &= 52 * N_v\\
157538 &= N_v
\end{aligned}$$

 © Koninklijke Philips Electronics N.V. 2004

This tells us that for scenes that contain up to 157K vertices, our TBR system will use less off-chip memory data traffic than the classic FBR approach. Compared to the number of vertices of our test scenes (Table 4.8), this means that the complexity of the scenes we can handle can increase by approximately a factor of 16. Although these are rough approximations, these numbers do give confidence that our system should be preferred over FBR for the not so near future.

© Koninklijke Philips Electronics N.V. 2004 

# Section 5

# Mapping the Tile-Based Rendering algorithm onto a VLIW core

As second part of our research we mapped a part of our Tile-Based Rendering algorithm onto a Very Long Instruction Word (VLIW) Digital Signal Processing (DSP) core. This chapter will first introduce the general concept of VLIW architectures. Next, it describes Silicon Hive and its VLIW architectures. Finally the concept of mapping is introduced along with our results of the mapping.

## 5.1 Very Long Instruction Word

While current mainstream processors are able to execute a small number of operations in parallel, the most important aspect of a VLIW processor is that it can execute many operations in parallel. It does so by including many issue slots. Current mainstream processors use hardware to determine at run time which operations can run in parallel, while VLIW processors put heavy emphasis on the compiler software to schedule the operations of a program such that they can be performed in parallel. Because scheduling complexity is moved from the hardware to the software, the hardware can be smaller, cheaper and require less power to operate. The scheduled operations are then put into an instruction word, which can be 'very long' because several issue slots need to be fed in a single instruction (hence the name VLIW).

Figure 5.1 shows an example VLIW processor. It contains 3 issue slots, also called execution units. This means that within one cycle, up to 3 operations can be executed in parallel. Each issue slot is made up from one or more functional units. These are hardware units that offer a set of related operations (an example of a functional unit is an ALU or Arithmetic and Logical Unit, which contains the most common integer arithmetic operations). The results of the functional units are presented onto a bus which is connected to the register file, which stores the data so it can be used as input for the functional units again.

 © Koninklijke Philips Electronics N.V. 2004

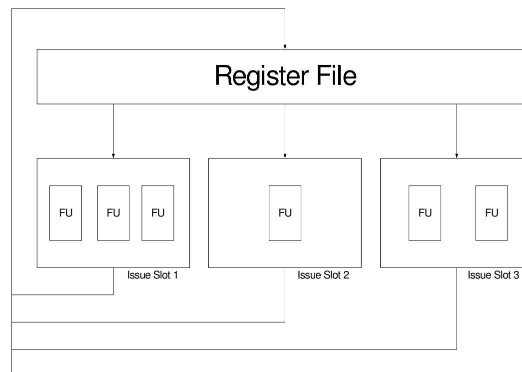



Figure 5.1: An example VLIW processor. The processor is 3 issue slots wide, each of which contain a number of functional units and each of which is connected to the register file.

## 5.2 Silicon Hive

Silicon Hive [14] is a start up business in the Philips Technology Incubator creating application-domain specific VLIW processors. In Silicon Hive terminology a processor can consist of a number of VLIW cores. This way an application (for example a 3D pipeline) can be broken down in pieces and each piece can run on a VLIW core most suited for its needs. For example, wider cores can be used for number crunching and smaller cores for irregular control processing.

Silicon Hive distinguishes itself in the following ways:

- Classic VLIW cores only have a single register file that is shared by all issue slots. Silicon Hive cores can have multiple register files (e.g. a 5 issue slot core may have 10 register files), that are positioned close to the issue slots that require their data. This locality of reference improves performance (short delays) and lowers power dissipation (short interconnects).
- Its configurable design process [15] allows for the generation of new cores within days. So processors can be tuned to the needs of particular application domains and/or customers. Examples of possible modifications to a core include:
  - Changing the number of issue slots.
  - Changing the size and number of register files.
  - Changing the interconnects between issue slots and register files.
  - Changing the instruction set.

  All these changes are performed in the machine description file, which describes a Silicon Hive core. A graphical example of a Silicon Hive core is given in Appendix G. It shows a 6 issue slots core. Some of its properties are:

© Koninklijke Philips Electronics N.V. 2004 

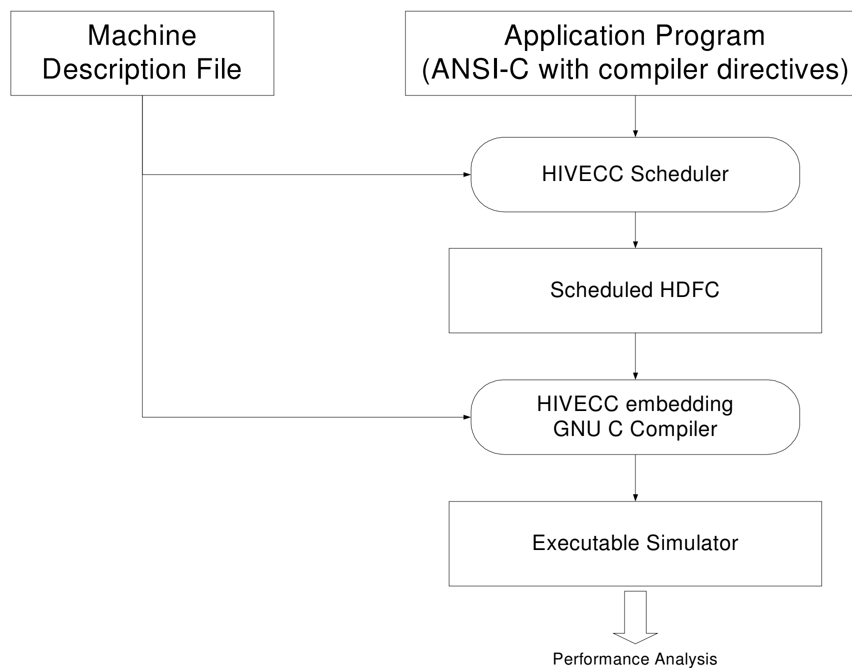



Figure 5.2: Creation of executable simulation and performance analysis.

- 6 issue slots.
- 15 1024x32 bit register files.
- 3 8x64 bit register files.
- Several vectorized functional units, meaning they can work on 1x32 bit, 2x16 bit and 4x8 bit. These functional units are prefixed by a $v$.
- The branch unit, which updates the program counter (PC). These can be found on every Silicon Hive core.
- The status update unit, which updates the status register (SR). These can be found on every Silicon Hive core.

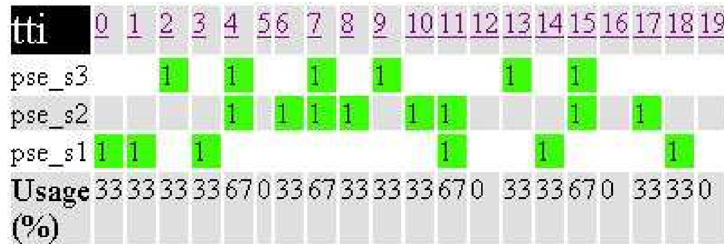
For the configurable design process, Silicon Hive developed a set of tools known as the hivecc compiler suite [16]. This suite allows to simulate an application on a Silicon Hive core described by a machine description file. The compiler flow is illustrated in Figure 5.2. The flow starts with an application written in C. The hivecc scheduler will translate the C code into scheduled machine operations (scheduled HDFC or scheduled Hierarchical Data Flow C, an intermediate language) and will optimize the scheduling. Along with the machine description file (and the accompanying library files for the core), the scheduled HDFC code is compiled using GNU's gcc [17] to produce an executable simulator.

The output of the simulations is the following performance analysis:

 © Koninklijke Philips Electronics N.V. 2004

- Map of cycle count per instruction address, and overall program cycle count.
- A table on the utilization of resources per instruction address.
- Details of the applications execution activity per cycle. This includes a graphical overview of the produced schedule (See Figure 5.3).

The performance analysis can be used during the iterative processor of refining the core and/or application to optimize it to its fullest potential.



| tti | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| pse_s3 | | | 1 | | 1 | | | 1 | | 1 | | | | 1 | | 1 | | | | |
| pse_s2 | | | | | 1 | | 1 | 1 | 1 | | 1 | 1 | | | | 1 | | 1 | | |
| pse_s1 | 1 | 1 | | 1 | | | | | | | | 1 | | | 1 | | | | 1 | |
| Usage (%) | 33 | 33 | 33 | 33 | 67 | 0 | 33 | 67 | 33 | 33 | 33 | 67 | 0 | 33 | 33 | 67 | 0 | 33 | 33 | 0 |

Figure 5.3: Example schedule of an application simulated on a 3 issue slot core.

## 5.3 Mosca core

Silicon Hive is currently implementing a 3D graphics pipeline according to the OpenGL ES specification as an add-on for its video processor. Different parts of this pipeline will run on different cores of this processor. Our algorithm will run on a core known as the Mosca core. We are free the modify this core with the restriction that we are not allowed to break compatibility of the core with the video algorithms developed for it. Some of the core's attributes are:

- 3 issue slots
- 3 32x32-bit register files
- 3 4x64-bit register files
- 2 multipliers (also as SIMD[1] 2-way and 4-way)
- 3 ALUs (also as SIMD 2-way and 4-way)
- 32KB local memory (width: 64 bit)
- 32KB instruction memory cache

## 5.4 Mapping the Tile-Based Rendering algorithm onto the Mosca core

Currently Silicon Hive's hivecc compiler suite can not handle large pieces of code. We will therefore only map a part of our algorithm onto the Mosca core. We will first

[1] Single Instruction Multiple Data

© Koninklijke Philips Electronics N.V. 2004 

introduce the concept of mapping, then describe the part of code that will be used in the mapping process and finally our results of the mapping.

Mapping an application onto a VLIW core is a two stage process:

- Fixing the application
  An existing application will not work out-of-the-box on a VLIW core. Therefore, parts of the application may have to be redesigned to overcome limitations that a specific VLIW core may have. Limitations we have encountered are:
  - Lack of a floating-point unit.
  - Lack of a division operation.
- Optimizing the application
  After the first stage, the application works and runs on the core. However, in general the initial working version does not take full advantage of the VLIW core and optimizations can be made to increase performance. The optimization steps consist of:
  1. Changing the application code to take advantage of specific instructions offered by the VLIW instruction set.
  2. Changing the VLIW core to better suit specific needs presented by the application. For example by adding new instructions or more issue slots to the hardware.
  3. Including compiler directives (for example in the form of pragmas) in the application code that guide the compiler to an efficient end-result.

Since we can only map part of our algorithm onto the Mosca core, care should be taken with the selection of that part. We selected a part that we think gives a good impression for the performance of the entire algorithm on the core. The part we've chosen is the exact primitive-to-tile-sorting algorithm [9], which clips a primitive to the tile grid to determine which tiles it impacts. The algorithm was implemented into the module by [7] as a function, which is called for every single primitive accepted by the module. Profiling shows that a significant amount of time (See Tables 5.1 and 5.2) is spent inside it making it an excellent candidate to map onto the Mosca core. The following steps have been taken to map the initial code (See Appendix H.3) onto the core.

| | Total time | Capture Phase | Render Phase |
|---|---|---|---|
| Tequila | 0.1989916 | 62.3% | 37.7% |
| Rally | 0.3103900 | 59.5% | 40.4% |
| Temple | 0.2468946 | 63.7% | 36.3% |

Table 5.1: Time spent inside the routines of our TTSB module during the rendering of a single frame (Using a resolution of 640x480 pixels and a tile size of 32x32 pixels) and how that time is divided over the different phases.

**Fixing the application** The following problems have been solved:

 © Koninklijke Philips Electronics N.V. 2004

| | Capture Time | Mapped algorithm |
|---|---|---|
| Tequila | 0.123898 | 22.2% |
| Rally | 0.184750 | 19.0% |
| Temple | 0.157150 | 21.3% |

Table 5.2: Time spent inside the capture routines of our TTSB module during the rendering of a single frame (Using a resolution of 640x480 pixels and a tile size of 32x32 pixels) and how much of that time is spent inside the exact primitive-to-tile-sorting algorithm

- Lack of a floating-point unit:
  None of the cores at Silicon Hive features a floating point unit, since it is very costly to include one. Our original code however uses floating-point arithmetic. Extending the hardware with a floating-point unit is not a preferred option, so a modification to the algorithm code was required.

  An alternative to floating-point arithmetic is fixed-point arithmetic. Fixed-point arithmetic uses ordinary integer operations to do arithmetic with real numbers that have have a fixed number of digits after the decimal (or binary) point. For example, a fixed-point number with 4 digits after the decimal point could be used to store numbers such as 2.4758, 357263.2447 and 0.4000, but would round 1.0401789 to 1.0402 and 0.0000654 to 0.0001.

  Our algorithm code was rewritten to use fixed-point variables and fixed-point operations in places where ordinarily floating-point variables and floating-point operations were used.

- Lack of a division operator:
  The Mosca core does not feature a division operator for integer arithmetic, which is used by our code in several places.
  - The code was modified to substitute a different operation for the division operation wherever possible. For example:

    ```
    y = x / 32;
    ```

    was replaced by the following shift right operation on several places:

    ```
    y = x >> 5
    ```

  - However, this approach is only possible if the divider is of the form $2^n$, therefore an integer division operation was added to the Mosca core to accomadate for those places where substitution is not possible.

Once fixed, the execution of the application on the core can be simulated. The code at this point can be seen in Appendix H.3. A number of measures were taken to optimize the application.

**Optimizing the application**

1. The Mosca core offers several instructions that can increase the performance of our algorithm. Looking at our code, we see a lot of statements of the following forms:

```
if ( x > y ) x = y;
if ( x < y ) x = y;
```

These statements are a form of conditional branching. Conditional branching increases the scheduling problems the compiler has to solve. They should be removed whenever possible. The statements above can conveniently be replaced by the dmins (double signed minimum) and dmaxs (double signed maximum) respectively.

There are also conditional branches of the form:

```
if ( condition )
{
  x = a;
  y = b;
}
else
{
  x = b;
  y = a;
}
```

These can be replaced by the dmux (double mux) operator. This is an instruction with three inputs (two variables and a condition) and two outputs. Depending on the condition the two outputs are set to the two input variables. This behavior corresponds to that of the statement above.

There are also conditional branches of the form:

```
if ( condition )
{
  x = 1;
}
else
{
  x = -1;
}
```

These can be rewritten to the following statement (which is still a form of conditional branching):

```
x = (condition) ? 1 : -1;
```

However, the hivecc compiler recognizes a statement of this form as a mux operation (which sets a variable to a value depending on the input condition) and implicitly optimizes it. For readability, the optimization isn't written down explicitly.

The complete semantics of the Mosca instructions discussed here can be found in Appendix I.

2. The only modification made to the Mosca core was the addition of a division instruction.

 © Koninklijke Philips Electronics N.V. 2004

3. Due to time constraints this step was not performed. Because the behavior of the scheduler has not been evaluated, several possible modifications to the core were not considered. For example, studying the behavior of the scheduler might have led to the decision of adding extra issue slots.

   Performing this step could increase the performance by several factors. How many factors exactly was not investigated. If the number of issue slots remains the same we expect that factor to be between 2 and 3.

The code at this point can be seen in Appendix H.4.

## 5.5 Evaluation

Based on the measured performance of the exact primitive-to-tile-sorting algorithm, we make an approximation of the number of triangles that the total TBR system can process on a Mosca core per second.

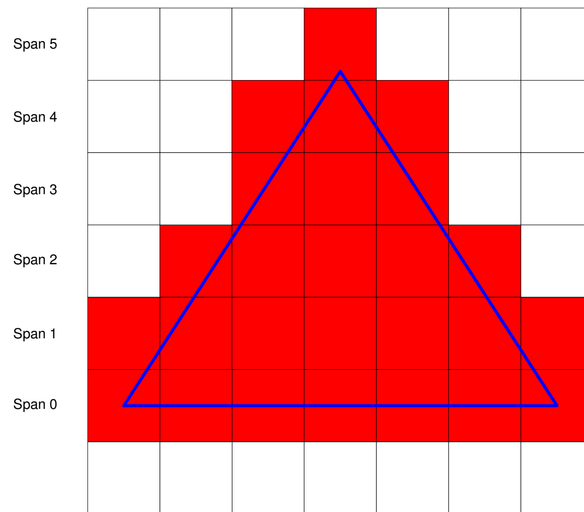



Figure 5.4: Colored tiles are marked as impacted by the exact primitive-to-tile-sorting algorithm.

The exact primitive-to-tile-sorting algorithm determines which tiles are impacted by a given triangle (See Figure 5.4). This is done by walking over the edges of the triangle. Each time an edge crosses over a span boundary a check is performed whether the entered tile is either the leftmost or rightmost impacted tile for that horizontal span of the triangle. The number of steps it takes to walk over a triangle-edge is determined by the tile height of that edge (number of horizontal spans an edge impacts). Therefore, the number of instructions our algorithm requires to determine which tiles are impacted by a certain triangle depends solely on the tile height of the triangle and not on the tile width. For example, the two triangles in Figure 5.5 require the same number

of instructions even though the width of the second triangle is twice that of the first triangle.

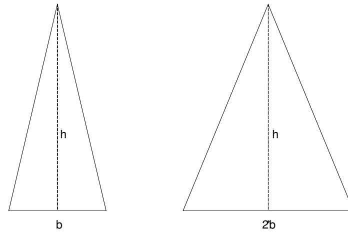



Figure 5.5: Two triangles.

Although probably not representative for a real scene, for the sake of this performance estimation we assume triangles having equal width and height. When the area of such a triangle doubles the height of the triangle is increased by a factor of $\sqrt{2}$. Consequently, the number of instructions increases by a factor of $\sqrt{2}$ as well. Increasing or decreasing the area of a triangle increases or decreases the number of instructions required for that triangle by a lineair factor. Therefore we can estimate the total number of instructions required for all triangles (bigger triangles and smaller triangles) in the scene by substiting each of them for a triangle of average height.

For a tile size of 32x32 pixels, we know that the average triangle overlap ratio is approximately 2.8 tiles (See Tables 4.11, 4.13 and 4.15). The height of such a triangle is 2 tiles. We've measured the number of instructions for a triangle of 2 tiles heigh and found it to be 241. Now we can make an estimation of the total number of triangles our TBR system can handle, assuming the following:

- The entire TBR system is mapped onto the Mosca core.
- The core runs at 200Mhz.
- For a scene, the total time spent in the different phases is divided as (based on Tables 5.1 and 5.2):
  - Render phase: 60%
  - Capture phase: 40%
    - During the capture phase, 20% of the time is spent inside the exact primitive-to-tile-sorting algorithm.

The total number of cycles spent inside the exact primitive-to-tile-sorting algorithm is then calculated as:

$$\text{Cycles spent inside the mapped portion} = 200000000 * 0.4 * 0.2 = 16000000 \quad (5.1)$$

We know that on average a single triangle requires 241 instructions. Therefore the

 © Koninklijke Philips Electronics N.V. 2004

approximated triangle-throughput equals:

$$\text{Approximated triangle-throughput} = 16000000/241 \approx 66000 \text{ triangles per second} \tag{5.2}$$

It should be noted that the average number of 241 instructions required for a single triangle would have been lower if all optimization steps were performed. This would result in a higher triangle-throughput.

© Koninklijke Philips Electronics N.V. 2004

# Section 6

# Conclusion

We have presented a series of modifications to an existing Tile-Based Rendering algorithm that are all aimed at decreasing the off-chip memory bandwidth. This increases performance and decreases power dissipation, which is important for low-end mobile phones, the target platform of our algorithm. We also created a formula that, given various scene characteristics, predicts the savings of our Tile-Based Rendering algorithm with regard to traditional Frame-Based Rendering. We integrated our algorithm into a 3D graphics pipeline and then tested the functional correctness and bandwidth savings of our algorithm using Quake III with various test scenes. Those tests showed the following:

- For a screen resolution of 640x480, TBR generates considerably less off-chip memory traffic than FBR. The savings of TBR with respect to FBR can be estimated at around 80%, mostly due to the fact that Z-buffer traffic remains on-chip.
- The formula was able to predict, within a margin of error, the expected savings of TBR with respect to FBR for the various test scenes. Using the test results we were able to make some suggestions that are likely to decrease the margin of error of the formula.
- Comparing the increase ratio in tile size and the off-chip memory traffic generated for that tile size, the optimal tile size appears to be 32x32 pixels.
- The algorithm appears to be beneficial compared to FBR for scenes with up to 157K vertices.

A part of our TBR algorithm was then mapped onto a Very Long Instruction Word architecture, targeted for use in the low-end mobile phone market to get an indication of the triangle-throughput of our algorithm on the target platform. This showed the following:

- To get our system running on a Silicon Hive VLIW core, a conversion of all floating-point arithmetic to fixed-point arithmetic is required. The core also requires a division operation for integer arithmetic.
- When mapped on a Silicon Hive VLIW core, our system will be able to handle a triangle-throughput of at least 66000 triangles per second.

 © Koninklijke Philips Electronics N.V. 2004

## 6.1 Further research

Given the complexity of our algorithm and TBR in general there remain several areas for further research. For example, while this research focuses on decreasing off-chip memory traffic, others might be more concerned with decreasing the extra storage required for TBR or the computational overhead originating from it.

Issues with regard to our algorithm that remain unanswered and may be good topics to explore are:

- Extending the primitive buffer with support for detecting duplicates among connected primitives and see how this affects performance. This would make the system slightly more complex.
- Instead of extending the primitive buffer, one could also choose to remove it along with duplicate detection and indexed primitives altogether and see how this affects performance. This would significantly increase the simplicity of the system.
- Investigating various cache strategies other than the FIFO strategy used in our implementation.
- Gathering more data from measured test scenes to improve estimated input values to our formula and to improve the comparison between TBR and FBR.
- Extending the algorithm with support for Exact Hidden Surface Removal, which decreases texture traffic by not retrieving texels for parts of polygons that eventually won't be visible, and measure how this affects performance.
- Improving the formula. For example, by determining an estimate for the depth complexity.
- Performing the last optimization step of the mapping process and determining the performance increase factor this results in and the effect this has on the triangle-throughput.
- Mapping the other parts of our TBR system onto the VLIW core.

# References


[1] Henri Gouraud . Continious shading of curved surfaces . *IEEE Transactions on Computers, 20(6): 623-628* , June 1971 .

[2] Bui Tuong Phong . Illumination for computer generated pictures . *Commun. ACM* , 18 ( 6 ): 311–317 , 1975 .

[3] David Blythe . OpenGL ES Common/Common-Lite Profile Specification, Version 1.0 (Annotated), `http://www.khronos.org` . Technical report, Khronos Group , August 2003 .

[4] Brian Paul . Mesa 3-D Graphics Library internet webpage at `http://www.mesa3d.org` , 1993 .

[5] Loren Carpenter . The A-buffer, an antialiased hidden surface method . In *Proceedings of the 11th annual conference on Computer graphics and interactive techniques* , pages 103–108 . ACM Press , 1984 .

[6] Henry Fuchs and John Poulton and John Eyles and Trey Greer and Jack Goldfeather and David Ellsworth and Steve Molnar and Greg Turk and Brice Tebbs and Laura Israel . Pixel-planes 5: a heterogeneous multiprocessor graphics system using processor-enhanced memories . In *Proceedings of the 16th annual conference on Computer graphics and interactive techniques* , pages 79–88 . ACM Press , 1989 .

[7] Sergey Trofimov . Tile Based Rendering in an OpenGL Implementation . Nat. Lab. Technical Note , Philips Research , 1999 .

[8] Michael Cox and Narendra Bhandari . Architectural implications of hardware-accelerated bucket rendering on the PC . In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* , pages 25–34 . ACM Press , 1997 .

[9] P.A. Beerkens . Algorithm for deciding whether triangles effect tiles. PRL Redhill, Interactive Systems Group, Informal Document H95_125 , 1995 .

[10] EXT_scene_marker OpenGL Extension Specification. `http://oss.sgi.com/projects/ogl-sample/registry/EXT/scene\protect\T1\textunderscoremarker.txt` , 1997 .


 © Koninklijke Philips Electronics N.V. 2004

[11] Milton Chen and Gordon Stoll and Homan Igehy and Kekoa Proudfoot and Pat Hanrahan . Models of the impact of overlap in bucket rendering . In *Proceedings of the ACM SIGGRAPH/EUROGRAPHICS workshop on Graphics hardware* , pages 105–112 . ACM Press , 1998 .

[12] I. Antochi and B.H.H. Juurlink and S. Vassiliadis . Selecting the optimal tile size for low-power tile-based rendering . In *Proceedings ProRISC 2002* , pages 1–6 , November 2002 .

[13] Steven Edward Molnar . *Image-composition architectures for real-time image generation* . PhD thesis, 1992 .

[14] Silicon Hive Technology Primer, `http://www.siliconhive.com` .

[15] Jeroen Leijten . A Massively Parallel Reconfigurable ULIW Core. Microprocessor Forum, San Jose . 2003 .

[16] Lex Augusteijn . The HiveCC Compiler for Massively Parallel ULIW Cores. Embedded Processor Forum, San Jose . 2004 .

[17] GNU Compiler Collection, GCC. http://www.gnu.org/software/gcc/gcc.html.

[18] Koen Meinds and Bart Barenbrug . Resample hardware for 3D Graphics . In *Proceedings of Graphics Hardware 2002* , pages 17 – 26 , 2002 .

© Koninklijke Philips Electronics N.V. 2004 

 © Koninklijke Philips Electronics N.V. 2004

# Appendix A

# Definitions

| Term | Definition |
|---|---|
| ALU | Arithmetic and Logical Unit |
| API | Application Programmers Interface |
| DSP | Digital Signal Processing |
| EAA | Edge Anti-Aliasing |
| FBR | Frame-Based Rendering |
| FIFO | First-In-First-Out |
| Framebuffer | A buffer containing the image that is sent to the display device each frame |
| HDFC | Hierarchical Data Flow C |
| IAS | Index Array Structure |
| Primitive | A basic element with which, when combined, more complex objects can be created |
| Primitive (single) | Either an independent primitive or a part of a connected primitive |
| Primitive (independent) | A separate triangle, separated quad or polygon |
| Primitive (connected) | A triangle strip, a triangle fan or quad strip |
| Primitive (indexed) | A primitive (either connected or independent) whose vertices are not stored sequential in the VAS and therefore needs to be indexed so its vertices can be retrieved |
| Primitive (sequential) | A primitive (either connected or independent) whose vertices are stored sequential in the VAS |
| SIA | Sub Index Array |
| SVA | Sub Vertex Array |
| TBR | Tile-Based Rendering |
| TTSB | Triangle-To-Tile Sorting/Buffering |
| VAS | Vertex Array Structure |
| VC | Vertex Cache |
| VLIW | Very Long Instruction Word |
| Z buffer | A buffer containing image depth coordinates |

Table A.1: Definitions on terms that recur throughout this thesis.

© Koninklijke Philips Electronics N.V. 2004 

# Appendix B

# Variable definitions

| Term | Definition |
|---|---|
| $A$ | Z-buffer traffic |
| $B$ | Color-buffer traffic |
| $C$ | Writing data to VAS/IAS/DL |
| $D$ | Reading data from VAS/IAS/DL |
| $E$ | Writing on-chip color buffer to off-chip framebuffer |
| $A_{pl}$ | Average primitive length |
| $DC$ | Depth complexity |
| $D_{sx}$ | Screen resolution width (horizontal) |
| $D_{sy}$ | Screen resolution height (vertical) |
| $D_{tx}$ | Tile width (horizontal) |
| $D_{ty}$ | Tile height (vertical) |
| $f_{hit}$ | Vertex cache hit ratio |
| $f_{seq}$ | Sequential primitive entries factor |
| $f_{uniq}$ | Unique vertices factor |
| $N_p$ | Number of triangles |
| $N_{pc}$ | Number of primitive changes |
| $N_{pe}$ | Number of primitive entries |
| $N_{pei}$ | Number of indexed primitive entries |
| $N_{pes}$ | Number of sequential primitive entries |
| $N_{sa}$ | Number of sub array changes |
| $N_{sia}$ | Number of sub index array changes |
| $N_{sva}$ | Number of sub vertex array changes |
| $N_t$ | Number of texture changes |
| $N_{tx}$ | Number of tiles (horizontal) |
| $N_{ty}$ | Number of tiles (vertical) |
| $N_v$ | Number of vertices |
| $O_p$ | Overlap ratio for primitives |
| $O_t$ | Overlap ratio for texture changes |

Table B.1: Definitions of variables.


© Koninklijke Philips Electronics N.V. 2004

# Appendix C

# OpenGL ES properties

Several properties of OpenGL ES, our target API, are listed here:

- Support for the following primitive types:
  - Point.
  - Line list, line strip.
  - Triangle list, triangle strip, triangle fan.
- Geometric objects are drawn exclusively using OpenGL ES' vertex arrays.
- Texture support is limited to 2D textures.
- The following data types can be used in OpenGL ES for the specification of vertex attributes:
  - byte.
  - short.
  - float.
- In the OpenGL ES Common Lite profile the float data type is replaced by the fixed-point S15.16 data type fixed.
- One cannot change the front and back face properties independently.

© Koninklijke Philips Electronics N.V. 2004 

# Appendix D

# Fractional precision required for Edge Anti-Aliasing

The simplest way to display an edge of a polygon is to simply color certain pixels to show where the edge is. However, because screens are of much lower resolution than human vision, just coloring pixels yes or no makes the 'step' in height from one scan line or column to another very visible to the human eye. These jagged edges are most visible when edges are almost horizontal or vertical and especially annoying during animation.

Edge Anti-Aliasing (EAA) is the process by which pixels on either side of an edge are altered in order to make the edge appear much smoother, such that the aliased patterns are removed or alleviated. See the tones at the anti-aliased polygon's edges making the transition between foreground and background color in Figure D.1.

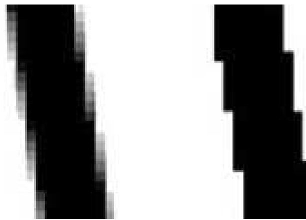


Figure D.1: An anti-aliased polygon (left) and an aliased polygon (right).

At Philips, a Forward Texture Mapping (FTM) rasterizer [18] has been developed. To support EEA, the FTM rasterizer requires screen space coordinate sub pixel accuracy to calculate the transition colors. A sub pixel accuracy of 3 bits allows for 8 different transition colors, which we assume to be enough for the quality required by the low-end mobile phone market.

 © Koninklijke Philips Electronics N.V. 2004

# Appendix E

# Display list entries

All possible display list entries and their description are listed here:

- DLOP_PRIMITIVE_INDEXED (3 bits)
  Represents an indexed primitive. It's followed by two values:
  - index (8 bits): an index into a sub array of the vertex index array, that indicates the start vertex of the primitive.
  - length (5 bits): length of the primitive.
- DLOP_PRIMITIVE_SEQUENTIAL (3 bits)
  Represents a sequential primitive. It's followed by two values:
  - index (5 bits): an index into a sub array of the VAS, that indicates the start vertex of the primitive.
  - length (5 bits): length of the primitive.
- DLOP_STATE_PREDEFINED (3 bits)
  Represents a predefined state change. It does not have any parameters, but consists of a single unique code. They are: (8 bits each)
  - DLS_ENABLE_BLEND: glEnable( GL_BLEND )
  - DLS_ENABLE_CULLFACE: glEnable( GL_CULL_FACE )
  - DLS_ENABLE_DEPTHTEST: glEnable( GL_DEPTH_TEST )
  - DLS_ENABLE_FOG: glEnable( GL_FOG )
  - DLS_ENABLE_TEXTURE2D: glEnable( GL_TEXTURE_2D )

  - DLS_DISABLE_BLEND: glDisable( GL_BLEND )
  - DLS_DISABLE_CULLFACE: glDisable( GL_CULL_FACE )
  - DLS_DISABLE_DEPTHTEST: glDisable( GL_DEPTH_TEST )
  - DLS_DISABLE_FOG: glDisable( GL_FOG )
  - DLS_DISABLE_TEXTURE2D: glDisable( GL_TEXTURE_2D )

  - DLS_SHADEMODEL_FLAT: glShadeModel( GL_FLAT )

© Koninklijke Philips Electronics N.V. 2004

- DLS_SHADEMODEL_SMOOTH: glShadeModel( GL_SMOOTH )

- DLS_FRONTFACE_CCW: glFrontFace( GL_CCW )
- DLS_FRONTFACE_CW: glFrontFace( GL_CW )

- DLS_TEXENV_MODE_MODULATE: glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_MODULATE)
- DLS_TEXENV_MODE_BLEND: glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_BLEND)
- DLS_TEXENV_MODE_DECAL: glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_DECAL)
- DLS_TEXENV_MODE_REPLACE: glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, GL_REPLACE)

- DLS_TEXPAR_WRAPS_REPEAT: glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_REPEAT)
- DLS_TEXPAR_WRAPS_CLAMP: glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP)
- DLS_TEXPAR_WRAPT_REPEAT: glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_REPEAT)
- DLS_TEXPAR_WRAPT_CLAMP: glTexParameterf(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP)

- DLS_MODE_POINTS: GL_POINTS
- DLS_MODE_LINES: GL_LINES
- DLS_MODE_LINE_LOOP: GL_LINE_LOOP
- DLS_MODE_LINE_STRIP: GL_LINE_STRIP
- DLS_MODE_TRIANGLES: GL_TRIANGLES
- DLS_MODE_TRIANGLE_FAN: GL_TRIANGLE_FAN
- DLS_MODE_TRIANGLE_STRIP: GL_TRIANGLE_STRIP
- DLS_MODE_QUADS: GL_QUADS
- DLS_MODE_QUAD_STRIP: GL_QUAD_STRIP
- DLS_MODE_POLYGON: GL_POLYGON

• DLOP_STATE_PARAMETIZED (3 bits)
Represents a parameterized state change. It's followed by a parameter. The parameter is followed by a number of other values. The parameters are:
  - DLS_PAR_CLEAR (3 bits): a state change as a result from glClear. It's followed by the value mask. Mask (32 bits) represents the buffers that need to be cleared.
  - DLS_PAR_SUBVERTEXARRAY (3 bits): a sub array change concerning the VAS. It's followed by the value idx. Idx (8 bits) represents the index of the sub array.

 © Koninklijke Philips Electronics N.V. 2004

  - DLS_PAR_SUBINDEXARRAY (3 bits): a sub array change concerning the IAS. It's followed by the value idx. Idx (8 bits) represents the index of the sub array.

- DLOP_STATE_TEXTURE (3 bits)
  Represents a texture change as a result from glBindTexture. It's followed by the value id. Id (16 bits) is the identifier of the texture.

- DLOP_CONTINUE (3 bits)
  Represents a change of display list block. It's followed by the value ptr. Ptr (32 bits) is a pointer to the next display block.

- DLOP_END_OF_LIST (3 bits)
  Acts as a marker that indicates the end of the display list. It's not followed by any value.

# Appendix F

# Test Scenes

## F.1 "Subversive Tendencies" scene

| | |
|---|---|
| Title | Subversive Tendencies |
| Owner | R. Bettenberg |
| Available via | http://bettenberg.home.mindspring.com |
| Engine | Quake III Arena |
| Filename | teqtrny3 |
| Referred to as | "Tequila" scene |
| Description | Indoor "tournament" map |

Figure F.1: "Subversive Tendencies" test scene

 © Koninklijke Philips Electronics N.V. 2004

## F.2 "Roll on Down the Line" scene

| Title | Roll on Down the Line |
|---|---|
| Owner | Jeff "Stecki" Garstecki |
| Available via | http://www.quakerally.com |
| Engine | Quake III Arena |
| Filename | q3r_country01 |
| Referred to as | "Rally" scene |
| Description | Outdoor "racetrack" map for the Q3 Rally modification<br>Test scene features a car model near the front |

Figure F.2: "Roll on Down the Line" test scene

## F.3 "Temple of Retribution" scene

| Title | Temple of Retribution |
|---|---|
| Owner | id Software |
| Available via | http://www.idsoftware.com |
| Engine | Quake III Arena |
| Filename | q3dm7 |
| Referred to as | "Temple" scene |
| Description | Indoor/Outdoor "deathmatch" map included with Q3 |

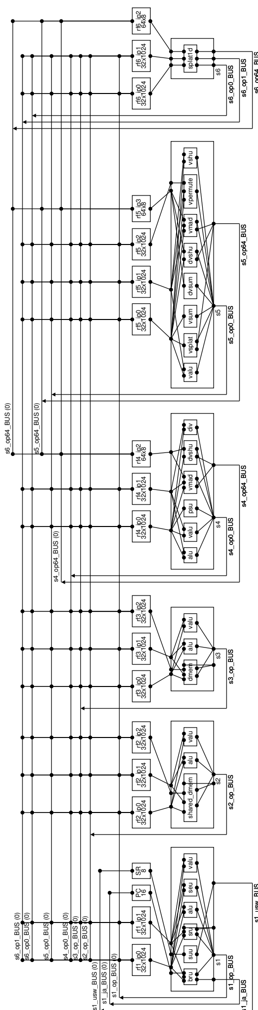
Figure F.3: "Temple of Retribution" test scene


© Koninklijke Philips Electronics N.V. 2004

© Koninklijke Philips Electronics N.V. 2004

# Appendix G

# Silicon Hive core



 © Koninklijke Philips Electronics N.V. 2004

Figure G.1: Example of a Silicon Hive core.

# Appendix H

# Source code

This Appendix contains various source code versions of the Exact primitive-to-tile-sorting algorithm that is mapped on a Silicon Hive core as the second part our of research.

## H.1 Common definitions

```
/** @file    defs.h
  * @author Wouter Burgers @ Silicon Hive
  * @description
  *   Defines several types and operations for
  *   fixed-point arithmetic.
  */
/*------------------------------------------------------------*/
typedef char                   int8;
typedef short                  int16;
typedef int                    int32;
typedef long long              int64;
typedef unsigned char          uint8;
typedef unsigned short         uint16;
typedef unsigned int           uint32;
typedef unsigned long long     uint64;
/*------------------------------------------------------------*/
typedef int                    fixed;
/*------------------------------------------------------------*/
#define FIX_PRC             5
#define MASK_FRACINT        0x7FFF
#define MASK_FRACTIONAL     0x001F
#define MASK_INTEGER        0x7FE0
#define MASK_SIGN           0x8000
/*------------------------------------------------------------*/
#define TO_FIX(X)           ( (X) << FIX_PRC )
```

```
#define TO_INT(X)          ( (X) >> FIX_PRC )
/*-----------------------------------------------------------------------*/
#define FIX_FLOOR(x)       ( ((x)>>FIX_PRC) << FIX_PRC  )
#define FIX_ABS(x)         ( ((x) > 0) ? (x) : (-1*(x)) )
/*-----------------------------------------------------------------------*/
#define FIX_DIV(x,y)       ( ((x)<<FIX_PRC) / (y) )
#define FIX_MUL(x,y)       ( ((x)*(y)) >> FIX_PRC )
/*-----------------------------------------------------------------------*/
#define FIX_BIGGER(x,y) \
   ( (((x)&MASK_INTEGER) > ((y)&MASK_INTEGER))\
     || (\
          (((x)&MASK_INTEGER)==((y)&MASK_INTEGER))\
          &&\
          (((x)&MASK_FRACTIONAL)<((y)&MASK_FRACTIONAL))\
        )\
   )
#define FIX_SMALLER(x,y) \
   ( (((x)&MASK_INTEGER) < ((y)&MASK_INTEGER))\
     || (\
          (((x)&MASK_INTEGER)==((y)&MASK_INTEGER))\
          &&\
          (((x)&MASK_FRACTIONAL)>((y)&MASK_FRACTIONAL))\
         )\
   )
/*-----------------------------------------------------------------------*/
```

## H.2 Initial code

```
/** @file    tti.hdf.c
  * @author Wouter Burgers @ Silicon Hive, October 2004
  * @description
  *   Our initial algorithm code.
  */
#include <custom.h>
#include <abbrevs.h>

/** @note
  *   ON(DMEM) AT(x):
  *   means the variabele will be placed in the local memory of
  *   the core at memory address x.
  */
int    ON(DMEM) AT(0)   in_tile_width;   /* 32, Tile width in pixels */
int    ON(DMEM) AT(4)   in_tile_height;  /* 32, Tile height in pixels */
int    ON(DMEM) AT(8)   in_tile_along_x; /* 20, # Tiles horizontal. */
int    ON(DMEM) AT(12)  in_tile_along_y; /* 15, # Tiles vertical. */
int    ON(DMEM) AT(16)  in_num_vertices; /* 4,  # Input vertices */
```

 © Koninklijke Philips Electronics N.V. 2004

```
float   ON(DMEM) AT(20) in_win_x[ 4 ];   /*    X coordinates */
float   ON(DMEM) AT(36) in_win_y[ 4 ];   /*    Y coordinates */

int     ON(DMEM) AT(600) lm[ 20 ];       /* 20 scanlines Leftmost */
int     ON(DMEM) AT(700) rm[ 20 ];       /* 20 scanlines Rightmost */

void tti()
{
  int     i;
  int     j;
  int     k;

  int     inc;

  {
    float AJf;
    float AIf;
    float BJf;
    float BIf;

    int   AJ;
    int   AI;
    int   BJ;
    int   BI;

    float LJf;
    float LIf;
    float RJf;
    float RIf;

    int   LJ;
    int   LI;
    int   RJ;
    int   RI;

    float alpha;
    float beta;

    float lastJf;
    float stepJf;

    int   lastJ;

    /** @note
      *  A 32x32 tile size is assumed.
      */
    AJf = in_win_x[ in_num_vertices - 1 ] / 32;
```

```
AIf = in_win_y[ in_num_vertices - 1 ] / 32;
AJ  = floor( AJf );
AI  = floor( AIf );

for (i = 0; i < in_num_vertices; i++)
{
  /** @note
    *   A 32x32 tile size is assumed.
    */
  BJf = in_win_x[ i ] / 32;
  BIf = in_win_y[ i ] / 32;
  BJ  = floor( BJf );
  BI  = floor( BIf );

  if ( AJf < BJf )
  {
    LJf = AJf;
    LIf = AIf;
    LI  = AI;
    LJ  = AJ;

    RJf = BJf;
    RIf = BIf;
    RJ  = BJ;
    RI  = BI;
  }
  else
  {
    LJf = BJf;
    LIf = BIf;
    LJ  = BJ;
    LI  = BI;

    RJf = AJf;
    RIf = AIf;
    RJ  = AJ;
    RI  = AI;
  }

  if ( LJ == RJ )
  {
    if ( LI < RI )
    {
      inc = 1;
    }
    else
    {
```

 © Koninklijke Philips Electronics N.V. 2004

```
      inc = -1;
    }

    for (j = LI; j != RI + inc; j += inc)
    {
      if ( lm[ j ] > LJ ) lm[ j ] = LJ;
      if ( rm[ j ] < RJ ) rm[ j ] = RJ;
    }
  }
  else if ( LI == RI )
  {
    if ( lm[ LI ] > LJ ) lm[ LI ] = LJ;
    if ( rm[ LI ] < RJ ) rm[ LI ] = RJ;
  }
  else
  {
    if ( lm[ LI ] > LJ ) lm[ LI ] = LJ;

    alpha = (RJf - LJf) / (RIf - LIf);
    beta  = LJf - alpha * LIf;

    if ( alpha > 0 )
    {
      inc = 1;
      lastJf = (LI + 1) * alpha + beta;
    }
    else
    {
      inc = -1;
      lastJf = LI * alpha + beta;
    }

    lastJ  = floor( lastJf );
    stepJf = fabs( alpha );

    if ( rm[ LI ] < lastJ) rm[ LI ] = lastJ;

    j = LI + inc;

    while( j != RI )
    {
      if ( lm[ j ] > lastJ ) lm[ j ] = lastJ;

      lastJf += stepJf;
      lastJ   = floor( lastJf );

      if ( rm[ j ] < lastJ ) rm[ j ] = lastJ;
```

© Koninklijke Philips Electronics N.V. 2004

```
          j += inc;
        }

        if ( lm[ RI ] > lastJ ) lm[ RI ] = lastJ;
        if ( rm[ RI ] < RJ    ) rm[ RI ] = RJ;
      }

      AJf = BJf;
      AIf = BIf;
      AJ  = BJ;
      AI  = BI;
    }
  }
}
```

## H.3 Code after the 'fixing' stage

```
/** @author Wouter Burgers @ Silicon Hive, October 2004
  * @description
  *   Our algorithm code after the 'fixing' stage.
  */
#include <custom.h>
#include <abbrevs.h>
#include "defs.h"

/** @note
  *   ON(DMEM) AT(x):
  *   means the variabele will be placed in the local memory of
  *   the core at memory address x.
  */
uint32 ON(DMEM) AT(0)  in_tile_width;
uint32 ON(DMEM) AT(4)  in_tile_height;
uint32 ON(DMEM) AT(8)  in_tile_along_x;
uint32 ON(DMEM) AT(12) in_tile_along_y;
uint32 ON(DMEM) AT(16) in_num_vertices;
fixed  ON(DMEM) AT(20) in_win_x[ 4 ];
fixed  ON(DMEM) AT(36) in_win_y[ 4 ];

uint32 ON(DMEM) AT(600) lm[ 20 ];
uint32 ON(DMEM) AT(700) rm[ 20 ];

void tti()
{
  uint32 i;
  uint32 j;
```

 © Koninklijke Philips Electronics N.V. 2004

```
uint32 k;

int32  inc;

{
  fixed AJf;
  fixed AIf;
  fixed BJf;
  fixed BIf;

  int32 AJ;
  int32 AI;
  int32 BJ;
  int32 BI;

  fixed LJf;
  fixed LIf;
  fixed RJf;
  fixed RIf;

  int32 LJ;
  int32 LI;
  int32 RJ;
  int32 RI;

  fixed alpha;
  fixed beta;

  fixed lastJf;
  fixed stepJf;

  int32 lastJ;

  /** @note
    *  A 32x32 tile size is assumed.
    */
  AJf = in_win_x[ in_num_vertices - 1 ] >> 5;
  AIf = in_win_y[ in_num_vertices - 1 ] >> 5;
  AJ  = TO_INT( AJf );
  AI  = TO_INT( AIf );

  for (i = 0; i < in_num_vertices; i++)
  {
    /** @note
      *  A 32x32 tile size is assumed.
      */
    BJf = in_win_x[ i ] >> 5;
```

```
BIf = in_win_y[ i ] >> 5;
BJ  = TO_INT( BJf );
BI  = TO_INT( BIf );

if ( FIX_SMALLER( AJf, BJf ) )
{
  LJf = AJf;
  LIf = AIf;
  LI  = AI;
  LJ  = AJ;

  RJf = BJf;
  RIf = BIf;
  RJ  = BJ;
  RI  = BI;
}
else
{
  LJf = BJf;
  LIf = BIf;
  LJ  = BJ;
  LI  = BI;

  RJf = AJf;
  RIf = AIf;
  RJ  = AJ;
  RI  = AI;
}

if ( LJ == RJ )
{
  if ( LI < RI )
  {
    inc = 1;
  }
  else
  {
    inc = -1;
  }

  for (j = LI; j != RI + inc; j += inc)
  {
    if ( lm[ j ] > LJ ) lm[ j ] = LJ;
    if ( rm[ j ] < RJ ) rm[ j ] = RJ;
  }
}
else if ( LI == RI )
```

 © Koninklijke Philips Electronics N.V. 2004

```
{
  if ( lm[ LI ] > LJ ) lm[ LI ] = LJ;
  if ( rm[ LI ] < RJ ) rm[ LI ] = RJ;
}
else
{
  if ( lm[ LI ] > LJ ) lm[ LI ] = LJ;

  alpha = FIX_DIV( (RJf - LJf), (RIf - LIf) );
  beta  = LJf - FIX_MUL( alpha, LIf );

  if ( alpha > 0 )
  {
    inc = 1;
    lastJf = FIX_MUL( TO_FIX( LI + 1 ), alpha ) + beta;
  }
  else
  {
    inc = -1;
    lastJf = FIX_MUL( TO_FIX( LI     ), alpha ) + beta;
  }

  lastJ  = TO_INT( lastJf );
  stepJf = FIX_ABS( alpha );

  if ( rm[ LI ] < lastJ) rm[ LI ] = lastJ;

  j = LI + inc;

  while( j != RI )
  {
    if ( lm[ j ] > lastJ ) lm[ j ] = lastJ;

    lastJf += stepJf;
    lastJ   = TO_INT( lastJf );

    if ( rm[ j ] < lastJ ) rm[ j ] = lastJ;

    j += inc;
  }

  if ( lm[ RI ] > lastJ ) lm[ RI ] = lastJ;
  if ( rm[ RI ] < RJ    ) rm[ RI ] = RJ;
}

AJf = BJf;
AIf = BIf;
```

```
      AJ   = BJ;
      AI   = BI;
    }
  }
}
```

## H.4 Code after the 'optimization' stage

```
/** @author Wouter Burgers @ Silicon Hive, October 2004
  * @description
  *   Our algorithm code after the 'optimization' stage.
  */
#include <custom.h>
#include <abbrevs.h>
#include "defs.h"

uint32 ON(DMEM) AT(0)  in_tile_width;
uint32 ON(DMEM) AT(4)  in_tile_height;
uint32 ON(DMEM) AT(8)  in_tile_along_x;
uint32 ON(DMEM) AT(12) in_tile_along_y;
uint32 ON(DMEM) AT(16) in_num_vertices;
fixed  ON(DMEM) AT(20) in_win_x[ 4 ];
fixed  ON(DMEM) AT(36) in_win_y[ 4 ];

uint32 ON(DMEM) AT(600) lm[ 20 ];
uint32 ON(DMEM) AT(700) rm[ 20 ];

#define WORKAROUND_BUG565

void tti()
{
  uint32 i;
  uint32 j;
  uint32 k;

  int32  inc;

  {
    fixed AJf;
    fixed AIf;
    fixed BJf;
    fixed BIf;

    int32 AJ;
    int32 AI;
    int32 BJ;
```

 © Koninklijke Philips Electronics N.V. 2004

```
int32 BI;

fixed LJf;
fixed LIf;
fixed RJf;
fixed RIf;

int32 LJ;
int32 LI;
int32 RJ;
int32 RI;

fixed alpha;
fixed beta;

fixed lastJf;
fixed stepJf;

int32 lastJ;

/** @note
  *  These variables were introduced in the optimized version.
  */
int32 dummyB, dummyQ;
int32 a, c;
int32 LI_;

AJf = in_win_x[ in_num_vertices - 1 ] >> 5;
AIf = in_win_y[ in_num_vertices - 1 ] >> 5;
AJ  = TO_INT( AJf );
AI  = TO_INT( AIf );

for (i = 0; i < in_num_vertices; i++)
{

  BJf = in_win_x[ i ] >> 5;
  BIf = in_win_y[ i ] >> 5;
  BJ  = TO_INT( BJf );
  BI  = TO_INT( BIf );

  /** @note
    *  The 'if' block below can be replaced by a series
    *  of 'double mux'.
    *
    *  OP dmux (int32 a, b, c) -> (int32 p, q)
    *    double mux: first output selects a or b, depending on c.
    *    second output gives other value.
```

© Koninklijke Philips Electronics N.V. 2004 

```
   *
   * if ( FIX_SMALLER( AJf, BJf ) )
   * {
   *    LJf = AJf;
   *    LIf = AIf;
   *    LI  = AI;
   *    LJ  = AJ;
   *
   *    RJf = BJf;
   *    RIf = BIf;
   *    RJ  = BJ;
   *    RI  = BI;
   * }
   * else
   * {
   *    LJf = BJf;
   *    LIf = BIf;
   *    LJ  = BJ;
   *    LI  = BI;
   *
   *    RJf = AJf;
   *    RIf = AIf;
   *    RJ  = AJ;
   *    RI  = AI;
   * }
   */
dmux( Any, AJf, BJf, (AJf<BJf), LJf, RJf );
dmux( Any, AIf, BIf, (AJf<BJf), LIf, RIf );
dmux( Any, AJ,  BJ,  (AJf<BJf), LJ,  RJ  );
dmux( Any, AI,  BI,  (AJf<BJf), LI,  RI  );

if ( LJ == RJ )
{
  /** @note
    *
    *
    * if ( LI < RI )
    * {
    *    inc = 1;
    * }
    * else
    * {
    *    inc = -1;
    * }
    */
  inc = LI < RI ? 1 : -1;
```

 © Koninklijke Philips Electronics N.V. 2004

```
    for (j = LI; j != RI + inc; j += inc)
    {
      /** @note
        *   The compiler does not yet recognize the if statement
        *   below als a 'minimum' instruction. We therefore replace
        *   it explicitly using a 'double minimum signed'.
        *
        *   OP dmins (sint32 A, B, C) -> (int32 R, Q)
        *     double signed minimum:
        *     calculates the minimum of signed values A and C,
        *     and of B and C.
        *
        * if ( lm[ j ] > LJ ) lm[ j ] = LJ;
        */
      dmins( Any, lm[j], dummyB, LJ, lm[j], dummyQ );

      /** @note
        *   The compiler does not yet recognize the if statement
        *   below as a 'maximum' instruction. We therefore replace
        *   it explicitly using a 'double maximum signed'.
        *
        *   OP dmaxs (sint32 A, B, C) -> (sint32 R, Q)
        *     double signed maximum:
        *     calculates the maximum of signed values A and C,
        *     and of B and C.
        *
        * if ( rm[ j ] < RJ ) rm[ j ] = RJ;
        */
      dmaxs( Any, rm[j], dummyB, RJ, rm[j], dummyQ );
    }
  }
  else if ( LI == RI )
  {
    /** @note
      *   Similar to above.
      *
      * if ( lm[ LI ] > LJ ) lm[ LI ] = LJ;
      */
    dmins( Any, lm[LI], dummyB, LJ, lm[LI], dummyQ );

    /** @note
      *   Similar to above.
      *
      * if ( rm[ LI ] < RJ ) rm[ LI ] = RJ;
      */
    dmaxs( Any, rm[LI], dummyB, RJ, rm[LI], dummyQ );
  }
```

```
    else
    {
      /** @note
        *    Similar to above.
        *
        * if ( lm[ LI ] > LJ ) lm[ LI ] = LJ;
        */
      dmins( Any, lm[ LI ], dummyB, LJ, lm[ LI ], dummyQ );

#ifdef WORKAROUND_BUG565
      /** @note
        *
        *
        * if (RIf - LIf < 0)
        * {
        *    alpha = -1 * FIX_DIV( (RJf - LJf), -1*(RIf - LIf) );
        * }
        * else
        * {
        *    alpha = FIX_DIV( (RJf - LJf), (RIf - LIf) );
        * }
        */
      c = RIf - LIf < 0;
      a = c ? -1 : 1;
      alpha = a * FIX_DIV( (RJf - LJf), a * (RIf - LIf) );
#else
      alpha = FIX_DIV( (RJf - LJf), (RIf - LIf) );
#endif

      beta  = LJf - FIX_MUL( alpha, LIf );

      /**
        * @note
        *
        *
        * if ( alpha > 0 )
        * {
        *    inc = 1;
        *    lastJf = FIX_MUL( TO_FIX( LI + 1 ), alpha ) + beta;
        * }
        * else
        * {
        *    inc = -1;
        *    lastJf = FIX_MUL( TO_FIX( LI     ), alpha ) + beta;
        * }
        */
      c = alpha > 0;
```

 © Koninklijke Philips Electronics N.V. 2004

```
inc = c ? 1 : -1;
LI_ = LI + c;
lastJf = FIX_MUL( TO_FIX( LI_ ), alpha ) + beta;

lastJ  = TO_INT( lastJf );
stepJf = FIX_ABS( alpha );

/** @note
  *   Similar to above.
  *
  * if ( rm[ LI ] < lastJ) rm[ LI ] = lastJ;
  */
dmaxs( Any, rm[ LI ], dummyB, lastJ, rm[ LI ], dummyQ );

j = LI + inc;

while( j != RI )
{
  /** @note
    *   Similar to above.
    *
    * if ( lm[ j ] > lastJ ) lm[ j ] = lastJ;
    */
  dmins( Any, lm[ j ], dummyB, lastJ, lm[ j ], dummyQ );

  lastJf += stepJf;
  lastJ   = TO_INT( lastJf );

  /** @note
    *   Similar to above.
    *
    * if ( rm[ j ] < lastJ ) rm[ j ] = lastJ;
    */
  dmaxs( Any, rm[ j ], dummyB, lastJ, rm[ j ], dummyQ );

  j += inc;
}

/** @note
  *   Similar to above.
  *
  * if ( lm[ RI ] > lastJ ) lm[ RI ] = lastJ;
  */
dmins( Any, lm[ RI ], dummyB, lastJ, lm[ RI ], dummyQ );

/** @note
  *   Similar to above.
```

© Koninklijke Philips Electronics N.V. 2004

```
             *
             * if ( rm[ RI ] < RJ    ) rm[ RI ] = RJ;
             */
          dmaxs( Any, rm[ RI ], dummyB, RJ, rm[ RI ], dummyQ );
        }

        AJf = BJf;
        AIf = BIf;
        AJ  = BJ;
        AI  = BI;
      }
    }
}
```

 © Koninklijke Philips Electronics N.V. 2004

# Appendix I

# Semantics of Mosca instructions

```
OP dmux (int32 a, b, c) -> (int32 p, q)
{
  // double mux. first output selects a or b, depending on c.
  // second output gives other value.

  SEM p = if c then a else b fi;
  SEM q = if c then b else a fi;
};


OP dmins (sint32 A, B, C) -> (int32 R, Q)
{
  // double signed minimum,
  // calculates the minimum of signed values A and C, and of B and C.

  SEM R = if A<C then A else C fi;
  SEM Q = if B<C then B else C fi;
};


OP dmaxs (sint32 A, B, C) -> (sint32 R, Q)
{
  // double signed maximum,
  // calculates the maximum of signed values A and C, and of B and C.

  SEM R = if A>C then A else C fi;
  SEM Q = if B>C then B else C fi;
};


OP dasr (sint32 A, B, int32 C) -> (sint32 R, Q)
{
```

© Koninklijke Philips Electronics N.V. 2004

```
  // double arithmetic shift right. outputs A >> C and B >> C

  SEM R = A >> C;
  SEM Q = B >> C;
};
```

 © Koninklijke Philips Electronics N.V. 2004