Memory Architecture for Efficient Utilization of SDRAM: A Case Study of the Computation/Memory Access Trade-Off

Thomas Gleerup tmg@it.dtu.dk
Hans Holten-Lund hahl@it.dtu.dk
Jan Madsen jan@it.dtu.dk
Steen Pedersen sp@it.dtu.dk

Technical University of Denmark
Department of Information Technology
DK-2800 Lyngby, Denmark

ABSTRACT

This paper discusses the trade-off between calculations and memory accesses in a 3D graphics tile renderer for visualization of data from medical scanners. The performance requirement of this application is a frame rate of 25 frames per second when rendering 3D models with 2 million triangles, i.e. 50 million triangles per second, *sustained* (not peak). At present, a software implementation is capable of 3-4 frames per second for a 1 million triangle model.

By using direct evaluation of certain interpolation parameters instead of forward differencing, writing back parameters to SDRAM is avoided. In software, forward differencing is usually better, but in this hardware implementation, the trade-off has made it possible to develop a very regular memory architecture with a buffering system, which can reach 95% bandwidth utilization using off-the-shelf SDRAM. This is achieved by changing the algorithm to use a memory access strategy with write-only and read-only phases, and a buffering system, which uses round-robin bank write-access combined with burst read-access.

Keywords

Memory architecture, 3D graphics, case study.

1. INTRODUCTION

Medical scanners such as CT (Computed Tomography), MR (Magnetic Resonance), and PET (Positron Emission Tomography) are in use in many hospitals for diagnostic and surgery planning purposes. The interpretation of the two-dimensional output of these scanners is difficult even for highly skilled physicians. To improve the quality and time consumption of the analysis of the two-dimensional "slices", it is desirable to be able to render a three-dimensional image of the data in real time.

A surface model consisting of triangles can be generated from the two-dimensional images that are output from the medical scanner. Such a model typically contains about two million triangles that have to be rendered in real time. This leads to a very high, sustained triangle rate of 50Mt/s (million triangles per second) at 25f/s (frames per second).

A tile-based rendering algorithm has been implemented in software [5]. This highly optimized software achieves a frame rate of about 3-4 f/s on a dual Pentium III 500MHz Windows NT PC when rendering the Buddha [9] model containing 1 million triangles, see fig 1. The Buddha model is commonly used for 3D graphics benchmarks. For comparison, a model of a hip joint is shown in fig. 1. This model has been generated from sub-sampled CT data to get a model with a low triangle count (32,000) to allow real-time rendering in software. Models like this are used in [3] for surgery planning.

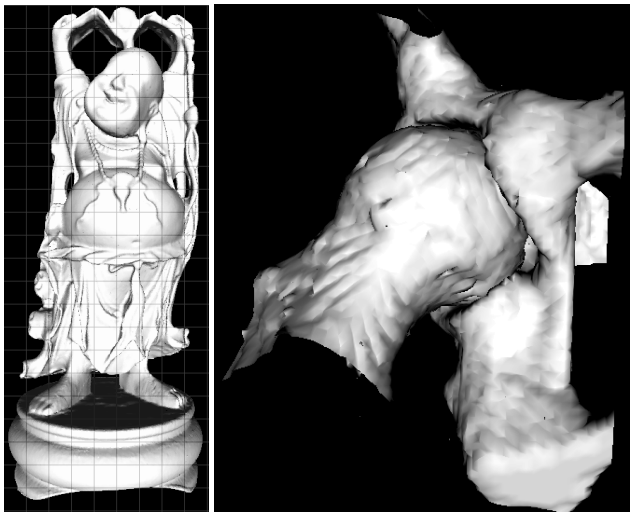

Figure 1: Buddha model [9] containing 1 million triangles and hip joint [3] containing 32,000 triangles, both rendered with the Hybris software [5]. The grid super-imposed on the Buddha indicates the tile size of the renderer (32 by 32 pixels).

Although the performance of the software is higher than what is achievable using game-oriented 3D hardware accelerators, it is evident that considerable speed-up is needed to reach 50Mt/s.

High-priced systems from e.g. Silicon Graphics are able to deliver the required performance, but the goal of this work has been to develop a PC-based cost-effective solution. The low cost will enable each physician to have his own 3D workstation, which can also be used for administrative purposes in a familiar environment.

The rest of the paper is organized as follows. Section 2 describes the 3D graphics system and discusses the trade-off between calculations and memory access. Section 3 motivates the use of SDRAM and briefly states the basic properties. Furthermore, the employed memory architecture is presented together with simulation results. Section 4 presents the synthesis and simulation results of the current implementation of parts of the 3D graphics system. Finally, section 5 gives some concluding remarks.

2. 3D GRAPHICS SYSTEM

The job of the tile rendering algorithm [1] is processing and rendering millions of triangles to create the final image on the screen, as seen in fig. 1. As the triangles may be arbitrarily transformed, we do not know where on the screen a triangle may be placed. Since the screen has been divided into tiles (see Buddha in fig. 1), we have to determine to which tiles a triangle belongs. When all triangles belonging to a tile have been identified, we can start rendering the tile by filling in the pixels covered by each triangle with specific color values. The pixels are held in the tile buffer while processing the tile, and later written to the display buffer. Another renderer based on this idea is presented in [6], however, they do not solve the bandwidth problems of the triangle heap, which will be discussed later.

Rendering a triangle requires the following steps. First, the geometry is transformed to screen coordinates, and a lighting equation is evaluated at each vertex. Then, the triangle interpolation parameters are calculated, and the triangle is inserted into the triangle heap (data structure for storing triangles). After all triangles have been processed this way, the tile renderers can begin to process the triangles in the triangle heap. The rendering pipeline is shown in fig. 2.

Rendering a triangle is done by determining which screen pixels are touched by the projected triangle, and which color to set them to. To do this we interpolate parameters for each triangle (edges, colors, depth). A triangle has three edges, which are connected by their vertices. Interpolation starts from the topmost triangle vertex. The parameters are first interpolated along the edges of the triangle, to determine starting interpolation values for interpolation along each scanline. Fig. 3 illustrates interpolations for rendering a triangle. While interpolating the parameters along scanlines, each parameter is checked against the current value in the tile buffer, and the final pixel value is written back to the tile buffer. Because of the high bandwidth requirement of this read-modify-write cycle, we use high-bandwidth on-chip memory for the tile buffer. This also allows for arrangement in parallel of the triangle rendering, as several tile buffers can be in use at once.

Since the order of the triangles is not known, the triangles must be sorted according to which tile they belong prior to processing each tile. The sorting, which facilitates the use of a small local memory for the tile processing, is done by inserting the triangles into the triangle heap. If a global frame buffer had been used (one large tile), the triangle heap would not be necessary. However, this would make parallel tile processing impossible, and requires very

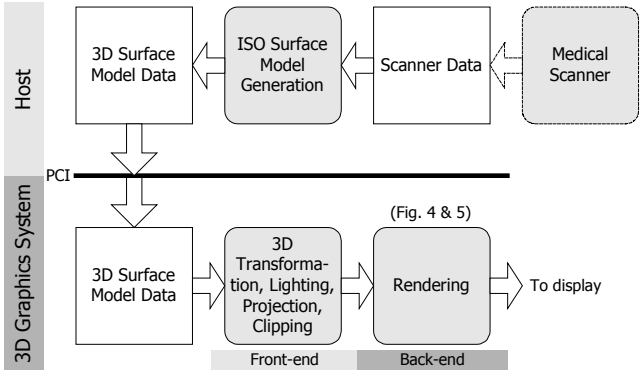

Figure 2: 3D graphics system. The back-end is shown in more detail in fig. 4.

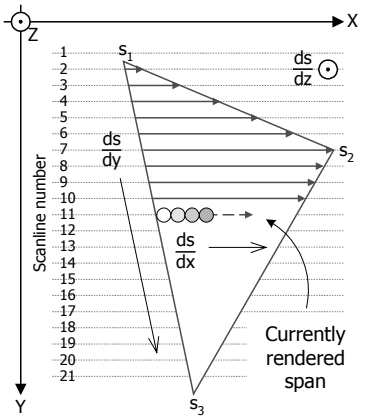

Figure 3: Interpolations when drawing a triangle on the screen.

high-bandwidth memory for the frame buffer, yet this is how most current PC 3D graphics hardware works, e.g. Nvidia Gforce 256 with 256 bits wide DDR SDRAM. A global frame buffer is best suited for rendering large triangles.

In this paper, we focus on the implementation of the back-end of the tile-based renderer. The parts not covered by this paper are the geometry transformation, culling, clipping and triangle setup, which occur prior to insertion into the triangle heap.

A connection to a host computer via a PCI bus is used to transmit triangle geometry data to the 3D renderer's memory. Prior to transmission to the renderer, the host computer generates triangle geometry data from the volumetric data of the medical scanner, using an iso-surface extraction algorithm [7].

2.1 Direct Evaluation versus Forward Differencing

When interpolating the parameters required for rendering a triangle, we can choose either direct evaluation or forward differencing to evaluate the interpolated parameters.

To evaluate an interpolation of a parameter between two values, the difference quotient, $\Delta p/\Delta x$, has to be calculated. Now, we can interpolate the values between the two endpoints by directly evaluating the equation: $p(x) = p_0 + x * \Delta p/\Delta x$. To cover N interpolated values, *direct evaluation* requires N additions + N multiplications.

Alternatively, we can interpolate the values between the two endpoints by incrementally adding the difference quotient, starting from the first value. Equations: $p(0) = p_0$, $p(x) = p(x-1) + \Delta p/\Delta x$. This is called *forward differencing*, and only N–1 additions are required. Thus, we have saved one addition and N multiplications. However, in more complex algorithms, such as rendering of triangles in 3D, the savings are even greater, although additional setup calculations are required.

The above example suggests that forward differencing is superior to direct evaluation, and this is usually true when an algorithm is implemented in software. However, when implementing an algorithm in hardware, only looking at the number of arithmetic operations doesn't tell the whole story, because memory issues are not taken into account. Memory issues are still important for software, but the memory architecture of the computer cannot be changed to optimize the software.

When using forward differencing, the current parameters have to be stored and later restored when the next data values are to be computed. In the simple example above, this is not really an issue, but in a highly complex hardware system using SDRAM, this read-modify-write behavior can have a very severe impact on performance.

In the tiled triangle renderer, one approach is to use forward differencing, and write the current values back to the triangle heap when reaching the tile border. This will allow the tile below to continue the interpolation from the values of the previous tile. While this works fine in software, the memory writes create problems for the hardware implementation. This is fixed by only using forward differencing within each tile, initialized by direct evaluation interpolation of parameters from the triangle heap. Fig. 4 shows how triangles are written into the triangle heap, and read from SDRAM by the triangle renderer. The extra calculations necessary for direct evaluation in hardware outweigh the cost of storing intermediate parameters. Note that in this case the parameters could not be stored on-chip since a large number of triangles may cross a tile boundary. The tiled approach allows the renderer to extensively utilize forward differencing without the memory overhead of parameter write-back, with a low overhead for direct evaluation.

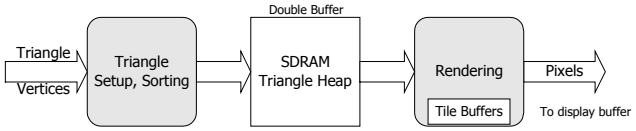

Figure 4: Overview of 3D graphics back-end.

3. OBTAINING HIGH BANDWIDTH UTILIZATION

This section will discuss the memory architecture that is used in the back-end of the 3D graphics pipeline (fig. 2 & 4) to utilize almost the full bandwidth of SDRAM. However, we will do this by taking a more abstract view on the architecture to make it more easily understandable.

3.1 Substantial Memory Requirements

Mainstream 3D accelerator boards claim to have high performance but in reality these boards are not capable of providing the required performance for large 3D models due to the PCI/AGP bus. To obtain a high, sustained triangle rate, the memory for the 3D model must reside on the accelerator board to avoid the PCI/AGP bottleneck. The memory requirement of this application is approximately 200MB, which implies the use of SDRAM due to cost constraints. The required bandwidth in some parts of the 3D-graphics pipeline is 1,600MB/s making memory bandwidth the performance bottleneck [4].

3.2 SDRAM Properties

SDRAM has a high potential bandwidth, but in practice, the bandwidth utilization is low due to the random-access nature and read-modify-write dependencies of an application. E.g., a 100MHz 64-bit SDRAM has a burst-access bandwidth of 800MB/s versus a single-word random-access bandwidth of only 100MB/s [8]. In addition, the latency of read- and write operations is asymmetric.

The reason that the random-access bandwidth is low for SDRAM (in contrast to SRAM) is that it takes some cycles to open a memory page for reading or writing. Consecutive accesses to the same page can be performed at full clock speed, e.g. 100MHz. Most SDRAMs are divided into four banks, which can each have a memory page open. This can be exploited to achieve very high bandwidth utilization by accessing the banks in a round-robin manner, thus hiding the page open/close operations.

3.3 Algorithm Properties/Requirements

The 3D-graphics rendering algorithm used in this application has a number of properties, which should be satisfied for the HW/SW architecture transformation to be effective:

1. The input data is a (long) stream of unsorted data records.
2. The algorithm operates on groups of data that need to be sorted prior to these operations.
3. The algorithm should be rewritten to allow for write-only and read-only phases, i.e. trading in more calculations by using direct evaluation instead of forward differencing.

In addition, some hardware-architectural properties should be satisfied:

4. SDRAM is used.
5. Memory bandwidth is the performance bottleneck.
6. Input data should be organized such that SDRAM burst mode can be utilized.

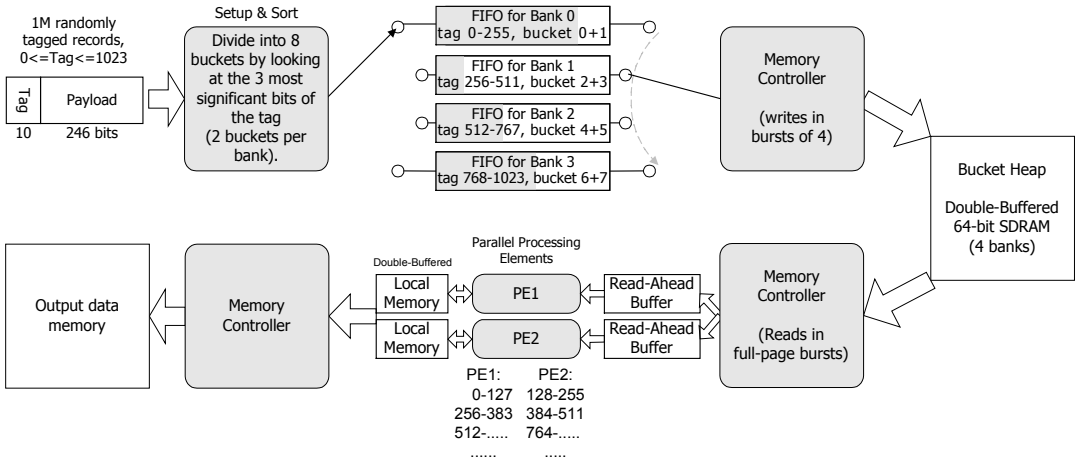

Figure 5: Read- and write buffer principle. See fig. 6 for memory layout of the bucket heap SDRAM.

3.4 Hardware Architecture

Fig. 5 shows a conceptual diagram of the back-end of the 3D-graphics rendering system (fig. 4). To illustrate the workings of the system, a simple, abstract example is used, in which the incoming data consists of records with a 10-bit address tag. The input data records are sorted into eight buckets according to their tag; tags 0-127 go into bucket 0, tags 128-255 into bucket 1, etc. The bucket data are written to the SDRAM bucket heap in bursts of four. When all the data records have been sorted into buckets, the buckets are read from the bucket heap and processed individually, in this case by two processing elements running in parallel using their own local working memory. When the operations on a bucket is done, the results are transferred to an external SDRAM. The buffering system will be explained in the following sections.

3.5 Memory Read Access Strategy

The memory for each bucket in the bucket heap is allocated in large chunks (called a bucket buffer), typically an entire SDRAM page at a time as shown in fig. 6. When a bucket buffer is full, an empty buffer is allocated and linked to the full buffer. This LIFO organization can be used in the tile renderer, because it doesn't matter in which order the triangles are rendered within a tile.

By using the burst mode of SDRAM, almost 100% bandwidth utilization can be achieved when reading from the memory using the layout in fig. 6. Evidently, there will be a small overhead when changing to a new bucket buffer, because a new memory page (probably) has to be opened, but this is negligible compared to the

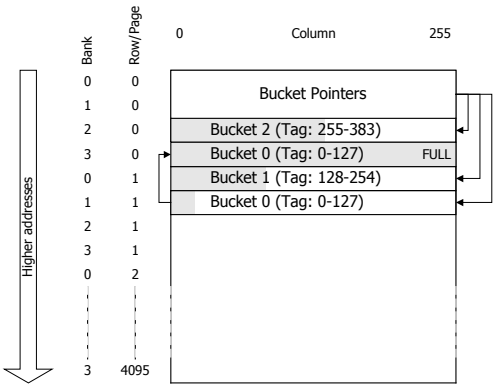

Figure 6: Memory layout of the bucket heap SDRAM in fig. 5.

bucket buffer size, which is limited by the page size of e.g. 256 elements.

To reduce unused space in the buffers, the bucket buffer size can usually be set lower than this to match the bandwidth utilization of the write side of the memory. In the case of the tile renderer implementation, the unused space accounts for about 6% on average, which is a low price to pay to obtain high bandwidth utilization.

3.6 Memory Write Access Strategy

In general, the arrival order of the data in the stream is unpredictable. Writing the data directly to the SDRAM in the layout of fig. 6 would cause severe page thrashing and therefore very low bandwidth utilization (app. 50%) if no cache or buffering were used. Therefore, we have to conceive a method of accessing the memory that can somehow avoid wasting memory cycles because of page open/close operations.

Such a method is illustrated in fig. 5. The idea is to alternate between writing to each bank, thus hiding the page open/close operations. This requires a queue for each bank and a stage to determine to which bank the incoming data records belong. This information, as well as the number of already used locations in each bucket buffer, is held in an on-chip memory until the frame is complete. After that, the bucket pointers are written to memory in one or several bursts depending on the number of buckets.

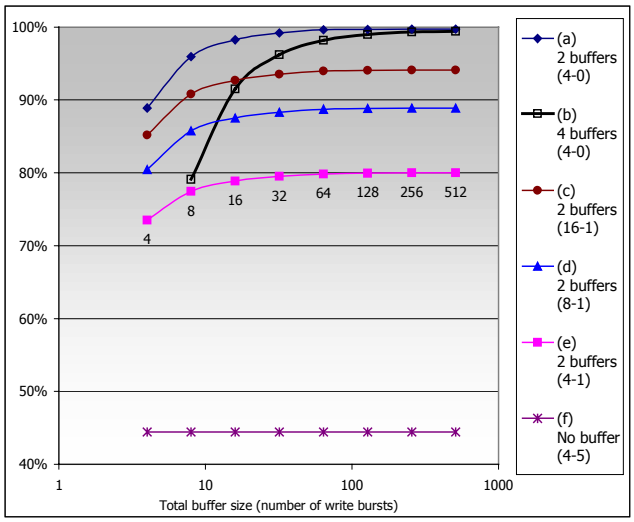

Figure 7: Memory bandwidth utilization as a function of buffer size. The input data is assumed totally randomized. The numbers in parentheses refer to (Write Cycles – Wait Cycles).

3.7 Simulation of Write Buffering

To examine the trade-off between FIFO size and bandwidth utilization, the write access strategy described above has been simulated by a C program. The simulation is done by applying 25,000 random pseudo-triangles to the buffering system, which is modeled by four circular arrays with corresponding push- and pop-functions. The number of different buckets (tiles in the tile renderer) is 768. The outcome is the number of cycles used, which can be converted to a bandwidth utilization figure.

From queue theory, it is known that the probability of a full queue is smaller if a fixed capacity is distributed on fewer queues. For that reason, it may be more efficient if the four FIFOs were joined to only two FIFOs, so that one FIFO holds data for bank 0 and 2, and the other for bank 1 and 3. If this is done, a four-bank SDRAM will be able to operate in an 8-1 or 16-1 manner, depending on the memory type. The 8-1 notation means that 8 cycles are used on data transfer and 1 cycles are spent waiting for page open/close.

The results of the simulations are shown in fig. 7, where the bandwidth utilization is plotted as a function of total buffer size measured in bursts; i.e. the total size of the physical buffer would be four times larger for a burst length of four.

Curve (a) is hypothetical, as it is not possible to operate in 4-0 mode using only two buffers. It is included to confirm the queue theory statement above. Curve (b) charts the bandwidth utilization of the proposed configuration shown in fig. 5. As expected, this curve lies below the hypothetical (a) curve, but above all other configurations except for buffer sizes of 16 and below. As a reference, curve (f) shows the bandwidth utilization when not using any buffering.

3.8 Input Data Correlation

The preceding simulation has assumed that the data arrive completely randomized. Due to the nature of the 3D model, this might not be true. As a 3D scene usually consists of different objects, there will generally occur some clustering of the triangles. The degree of this phenomenon is very difficult to predict and will vary greatly depending on the scene. Even so, the four-buffer system's sensitivity to this issue has been simulated. The clustering effect is modeled by a probability *p* that the next triangle is for the same bucket buffer as the previous. The results are shown in fig. 8.

It is difficult to decide what is the best trade-off, but if we conservatively assume that p = 50% and that we want at least 95% bandwidth utilization, a total buffer size of 64 elements is required (i.e. 16 for each FIFO).

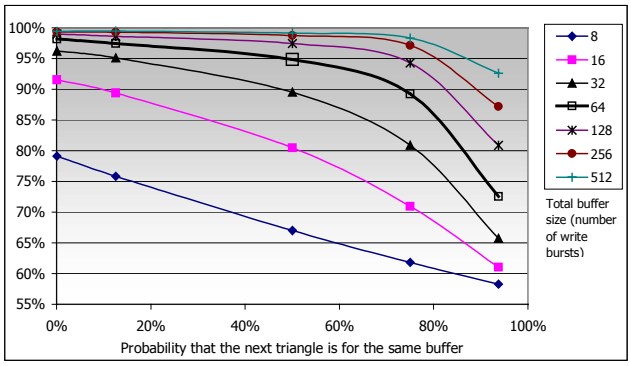

Figure 8: Memory bandwidth utilization for various total buffer sizes as a function of the probability that the next triangle is for the same bucket buffer.

As we have seen, the proposed scheme is quite simple, yet quite effective and efficient. However, it might be improved by using a shared buffer for all the queues, which would minimize the probability of a FIFO running full or empty. This would require extra memory for bookkeeping in addition to control overhead. Considering the good results of the simple solution, we have not pursued this further.

4. RESULTS

The rendering stage of the back-end (lower part of fig. 5), except some minor modules, has been synthesized with Synopsys Design Compiler 1998.08 using STMicroelectronic's HCMOS7 0.25µm library [2].

The total area of the rendering stage, which does not employ parallel processing elements, is 13.7mm^2. The area of the sort/setup stage of the back-end is estimated at 13.2mm^2. This estimate is based on adding up the area of synthesized sub-modules for all major components.

The triangle heap in this system consists of 2x64Mbyte 128-bit 100MHz SDRAM modules in a double-buffer configuration for maximum throughput. Each buffer can hold around 1 million triangles corresponding to 2 million triangles in the 3D model prior to backface culling (removal of back-facing triangles, which are not visible).

The write buffer for the triangle heap uses 2.63Kbyte on-chip RAM for bookkeeping purposes (bucket pointers and number of available spaces), and 4Kbyte total in the four FIFO's. The read-ahead buffer uses 8Kbyte.

The ASIC needs app. 483 pads, making the design pad-dominated and leaving a large amount of silicon area that could be used for more tile rendering processors running in parallel, or it could be used to implement the front-end on the same chip.

At 100MHz, the performance has been simulated to be 58 f/s when rendering the Buddha [9] in fig. 1 at a resolution of 1024 by 768 pixels with a tile size of 32 by 32 pixels. The triangle rate is 64.1Mt/s in the front-end, 31.6Mt/s in the sort/setup stage of the back-end, and 15.4Mt/s in the rendering stage of the back-end. Note that these numbers are sustained (not peak) rates for a very large actual 3D model.

5. CONCLUSION

This case study has shown that there is an important trade-off between calculations and memory accesses, which is much more prominent in the hardware implementation than in the software implementation.

Specifically, the hardware implementation of the tile renderer uses more compute-intensive direct evaluation of certain parameters – instead of forward differencing – to avoid writing back parameters to the SDRAM. This trade-off has made it possible to develop a very regular memory architecture, which can reach 95% bandwidth utilization using off-the-shelf SDRAM.

The on-chip memory requirements for the buffering system are quite low. The presented curves (fig. 7 & 8), which are based on modeling and simulation of the buffering system, make it possible to find the optimal trade-off between buffer size and bandwidth utilization.

6. REFERENCES

[1] Foley, van Dam, Feiner and Hughes, "Computer Graphics, Principles and Practice, Second edition", Addison-Wesley, 1990.

[2] Thomas Gleerup, "ASIC for 3D Graphics Pipeline Back-End", Master's Thesis, Technical University of Denmark, Dept. of Information Technology, Lyngby, Denmark, 1999.

[3] Hans Holten-Lund, Mogens Hvidtfeldt, Jan Madsen and Steen Pedersen, "VRML Visualization in a Surgery Planning and Diagnostics Application", Web3D+VRML2000 Symposium, Monterey, February 2000.

[4] Hans Holten-Lund, Jan Madsen and Steen Pedersen, "A Case Study of a Hybrid Parallel 3D Surface Rendering Graphics Architecture", SASIMI 1997 Proceedings.

[5] Hybris software renderer, http://www.it.dtu.dk/~hahl/hybris.html.

[6] Michael Kelley, et.al., "A Scalable Hardware Render Accelerator using a Modified Scanline Algorithm", Siggraph Proceedings, 1992.

[7] W.E. Lorensen & H. E. Cline, "Marching Cubes: A High Resolution 3D Surface Construction Algorithm", Computer Graphics, vol. 21, no.3, pp. 163-169, July 1987.

[8] Micron Technology Inc., "Synchronous DRAM Data Sheet, 64Mb SDRAM", rev. 10/98, Micron Technology Inc., 1998.

[9] The Stanford 3D Scanning Repository, http://www-graphics.stanford.edu/data/3Dscanrep/.