Scalable Rasterizer Unit

Péter Szántó
Budapest University of Technology and Economics
Dept. Of Measurement and Information Systems
Magyar tudósok krt. 2., H-1117, Hungary
szanto@mit.bme.hu

Béla Fehér
Budapest University of Technology and Economics
Dept. Of Measurement and Information Systems
Magyar tudósok krt. 2., H-1117, Hungary
feher@mit.bme.hu

ABSTRACT

Modeling the light-surface interaction in real time 3D applications becomes more and more complex, as users require more lifelike images. Segmented screen rendering offers a viable solution to minimize the unnecessary work done in traditional rendering architectures. However, increasing the efficiency of the rendering pipeline also increases the required hardware resources for the 3D rendering unit. This paper presents a modular, scalable rasterizer architecture, which makes it appropriate in a wide range of applications.

Keywords
3D graphics architecture, FPGA, system on chip

1. INTRODUCTION

In real-time graphics rendering two approaches are prevalent. Immediate Mode Rendering (IMR) renders the scene triangle by triangle; rasterization and shading of a triangle immediately starts after it has been transformed into screen space. Contrary to this, Deferred Rendering (DR) waits for all triangles to be transformed before beginning the per-pixel operations. The latter method has two main advantages.

First, it guarantees maximum efficiency when computing the output color values (the shading part of the rendering process, which clearly becomes the most time consuming), as only the truly visible values are shaded – unlike IMRs, where pixels not visible on the final image may be also processed. This is possible by first doing the visibility test, and deferring the rasterization process, so it only starts when the whole frame is analyzed and the visible objects are determined for every screen pixel.

Second, it allows using on-chip memory for the Depth-, Stencil- and Frame Buffer, thus reducing external bandwidth requirements, lowering cost and power consumption. It must be noted, that theoretically IMRs can also use on-chip buffers, but these buffers have to be the same size as the final, rendered image – which currently cannot be manufactured. The ability to segment the screen into small rectangles and then render these rectangles as "independent, small screens" allows small, implementable buffers to be used.

The DR rendering process is just a little different from the IMR one:

```
for every triangle in the given frame{
    transform the triangle into screen space
    find overlapped segments
}
for every segment on the screen{
    for every pixel in the segment{
        do visibility test
    }
    for every pixel in the segment{
        compute output color values
    }
}
```

The first part of the article reviews different segmenting strategies and presents hardware architecture for segmenting. The second part presents a modular Depth/Stencil Unit.

2. SEGMENTATION

Basically, there are three possibilities when deciding about the segmentation strategy [1].

The simplest solution is to process all triangles in all segments, therefore completely skipping the segmentation part (SGI had architectures which work this way). This method has clear disadvantages, as the effective depth/stencil fill rate is especially decreased due to the unnecessary work done during the visibility test. On the other side, the hardware architecture is simplified, and there is no need to store a triangle list for the segments.

Bounding box method uses the bounding box of the triangles to define the overlapped segments. Even this simple method can increase efficiency considerably – especially with small triangles –, however there are cases when a lot of unnecessary segments are marked as overlapped. Figure 1 shows such a case. There are known architectures doing software segmenting this way, for example Intel Extreme Graphics [2] or Microsoft Talisman [3]. Hardware solutions are rarer, the PixelFlow [4] is surely employing bounding box method and benchmark results indicate that the only commercial deferred renderer (PowerVR Kyro [5]) also prefers this way.

The most efficient method is exact segmenting, when only segments having at least one pixel overlapped with the triangle are marked. The disadvantage is the required hardware resources to implement the functionality.

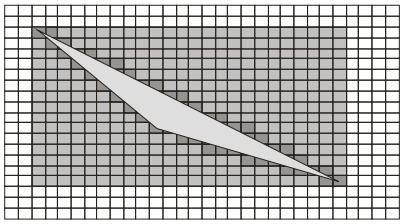

Figure 1. Overlapped segments using bounding box and exact segmenting

The following section shows the details of a hardware bounding box segmenting unit, and compares it with an exact segmenting solution, detailed in [6].

2.1 Bounding Box Segmenting Unit

Just as the Exact Segmenting Unit (ESU), the bounding box version (BBSU) consist of three main parts: the first part (Input Pipeline) computes the necessary input values for the main processing unit (Segment Generator), while the third part (Address Generator) handles communication with the external memory. To – possibly – increase efficiency, the unit supports programmable segment size, which can be set as an application specific parameter, or can be even adjusted adaptively, based on the statistics of a previous frame(s). Although selecting an overall appropriate segment size was already discussed in [7], the effects of variable segment size require further research to be correctly analyzed as there is no known academic or commercial architecture supporting this feature.

2.1.1 Input Pipeline

The first unit receives screen space vertex data (*x*, *y* coordinates) from the transformation part and, as a first step, generates primitives – triangles – from them. For primitive generation, triangle strips and triangle lists are supported without additional requirements, while for triangle fans the shared vertex should be sent to the Segmenting Unit multiple times (thus, generating a triangle strip from the fan). The architecture of the Input Pipeline is shown on Figure 2.

For load balancing with the transformation part, vertex data is immediately written into a small, 64 word deep FIFO. The FIFO is followed by two 3-tap sorters, which can take a new input every clock cycle. After reading the appropriate number of vertices from the FIFO (eg. one for triangle strip, three for triangle list), the sorters output the minimal and maximal *x* and *y* coordinates of the current triangle (which define the bounding box with high precision). These values are then multiplied with the reciprocal of the horizontal and vertical resolution of the segment, generating the corner segments of the bounding box. To limit resource usage, only two multipliers are used, therefore in worst case it takes two clock cycles to generate the four new values.

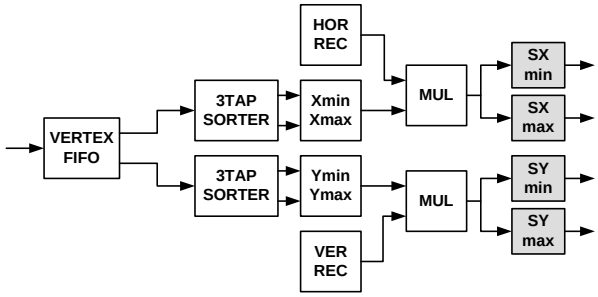

Figure 2. Input Pipeline

2.1.2 Segment Generator

The Segment Generator itself is very simple: it consists of two adders: one for incrementing the segment *x* coordinate, and one for incrementing the segment *y* coordinate. In the current implementation, the Segment Generator generates new (SX, SY) segment coordinate pairs every clock cycle, but with multiple adders it can be easily parallelized further (however, there is not too much sense in using more adders than the average maximum{bounding box height, width}).

2.1.3 Address Generator

The Address Generator builds a chained list for every segment. The list itself consists of 32-word blocks, from which 31 words are pointers to triangles, while the 32nd word is a pointer to the next 32 word block. The hardware implementation is the same for the BBSU and for the ESU, which was presented in details in [6].

2.1.4 Bounding Box vs. Exact Segmenting

Table 1. shows the main advantage of the BBSU, namely resource requirement.

		FF	LUT	MUL	BRAM
Input Pipeline	**ESU**	1100	1200	4	-
	BBSU	310	540	2	-
Segment Generator	**ESU**	900	2800	-	-
	BBSU	40	50	-	-
Address Gen.		270	380	1	4
ALL	**ESU**	2270	4380	5	4
	BBSU	620	970	3	4
BBSU/ESU, %		27.3	22.1	60	100

All in all, the BBSU requires about quarter as many FPGA resources as the ESU. Efficiency is more complex to answer, as it largely depends on the frame to be rendered, not to mention that it is not enough to analyze the Segmenting Unit alone, but together with the Hidden Surface Removal Unit.

If average triangle size is comparable to the segment size (eg. one triangle overlaps only 3-4 segments) the BBSU can be just as effective as the ESU. As triangle size increases, ESU becomes at least twice as effective. If the scene contains a lot of triangles with high aspect ratio (just as the one on Figure 1), the efficiency advantage of the ESU version increases further. To fully answer this question, real-world applications should be analyzed, as widely accepted fill rate tests (such as 3DMark [8]) use full screen quads – in this case the ESU is twice as effective as the BBSU.

3. Hidden Surface Removal Unit

The Hidden Surface Removal Unit (HSRU) consists of two main parts: a Vertex Processing Unit (VPU), which receives vertex data and computes all the necessary values for the next part, which does overlapping determination, depth buffering and stencil buffering. The latter block is made up from several, similar processing elements (HSR Cells), as Figure 3 shows. The function of the adders between the two blocks will be discussed later.

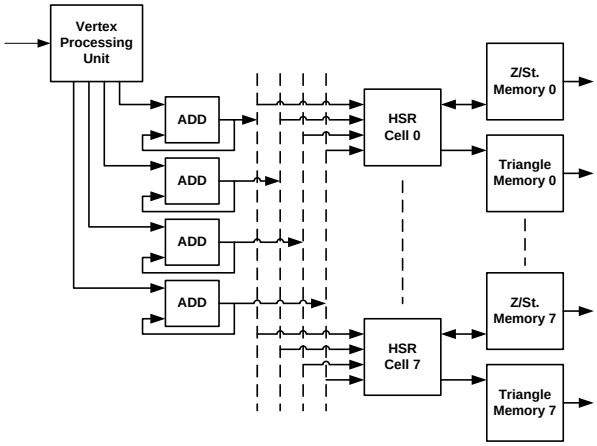

Figure 3. HSR Unit

Depending on the specified segment size, and the number of HSR Cells, each of them works on one or more segment lines. To correctly identify covered pixels and interpolate the depth values, the cells require initial values and delta values, which are generated by the VPU.

3.1 Vertex Processing Unit

Overlapping determination is based on variables generated from the explicit equation of the triangles sides:

$$S_j(x,y) = (x_i - x) * \Delta y - (y_i - y) * \Delta x \quad (1)$$

where x_i and y_i are points on the side, Δx and Δy are the differences between the two vertex coordinates forming the side. During interpolation, $S_j(x,y)$ is incremented or decremented with the delta values as the HSR Cells step through the pixels assigned to them. Covering is determined using the sign of the three S variables (S_0, S_1, S_2 represents the three variables for the three sides):

$$(sign(S_0(x,y)) \; XOR \; sign(S_1(x,y))) \; AND \; sign((S_1(x,y)) \; XOR \; sign(S_2(x,y)) \quad (2)$$

All in all, for the three sides the following calculations are required:

$$\begin{aligned} S_0(x_{strt},y_{strt}) &= (x_{strt} - x_0) * (y_0 - y_1) - \\ &- (y_{strt} - y_0) * (x_0 - x_1) \\ S_1(x_{strt},y_{strt}) &= (x_{strt} - x_0) * (y_0 - y_2) - \\ &- (y_{strt} - y_0) * (x_0 - x_2) \\ S_2(x_{strt},y_{strt}) &= (x_{strt} - x_1) * (y_1 - y_2) - \\ &- (y_{strt} - y_1) * (x_1 - x_2) \end{aligned} \quad (3)$$

In (Eq. 3) x_{strt} and y_{strt} are the coordinates of the starting pixel for the HSR Cells (without anti aliasing, the top-left pixel of the processed segment).

Initial depth values and incremental values are generated using the following equations:

$$\begin{aligned} z(x_{strt},y_{strt}) &= E_z * x_{strt} + F_z * y_{strt} + G_z = \\ &= (-\frac{A_z}{C_z}) * x_{strt} + (-\frac{B_z}{C_z}) * y_{strt} + (-\frac{D_z}{C_z}) = \\ &= (-\frac{A_z}{C_z}) * (x_{strt} - x_1) + (-\frac{A_z}{C_z}) * (y_{strt} - y_1) + z_1 \end{aligned} \quad (4)$$

The coefficients in (Eq. 4) are computed using the depth values defined at the three vertices and the plane equation of the triangle.

$$\begin{aligned} A_z &= (z_1 - z_2) * (y_1 - y_0) - (y_1 - y_2) * (z_1 - z_0) \\ B_z &= (x_1 - x_2) * (z_1 - z_0) - (z_1 - z_2) * (x_1 - x_0) \\ C_z &= (x_1 - x_2) * (y_1 - y_0) - (y_1 - y_2) * (x_1 - x_0) \\ D_z &= -(A_z * x_1 + B_z * y_{1.} + C_z * z_1) \end{aligned} \quad (5)$$

In the hardware realization clipped screen space x, y coordinates are 16 bit fixed point values, while vertex depth (z) values are 24 bit floating point numbers. For the above computations, the VPU uses these formats, but at the last step $z(x_{strt},y_{strt})$, E_z and F_z are converted to 24 bit fixed point format, preserving only the fractional part of the generated depth values (the transformation of triangles from 3D world space to screen space maps depth values to [0, 1] range).

3.1.1 Hardware Architecture

Because of the architecture of the HSR Cells, the above computations can be done in 16 clock cycles without limiting performance. Therefore, the trivial dataflow implementation is not the best option, as it requires too many resources, while it is needlessly fast.

A programmable solution not only requires fewer resources, but it is also more flexible, which – together

with the HSR Cell architecture – allows flexible anti aliasing implementation (more on this later). Figure 4 shows the architecture of the arithmetic unit.

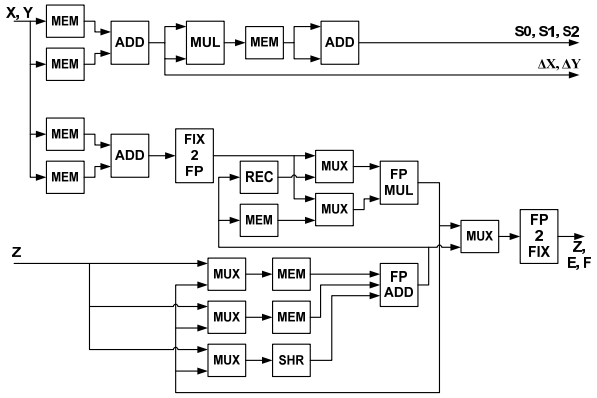

Figure 4. Vertex Processing Unit

The inputs of the VPU are the vertices' *x*, *y* and *z* coordinates and the screen coordinates of the top-left pixel in the processed segment.

The whole processing unit consists of two, almost separated portions. The upper part on Figure 4 is able to compute the required values for covering determination, while the lower part is responsible for the depth related computations.

The first units in the overlapping determination part are small, dual-ported memories to store the three *x* and *y* coordinate pairs of the currently processed triangle. The two memories fed a 16 bit adder/subtractor which has write enable signals at the input registers – its function is to compute the delta coordinate values in Eq. 3. The output is routed to a multiplier (which also has write enable signals at the input registers) to compute the partial products in Eq. 3 – to increase flexibility, these values are stored in a small memory. The final 32 bit adder can calculate the final edge variables (S_0, S_1, S_2), which – together with the delta values – are the input values for the covering determination part of the HSR Cells.

The depth related part supports mixed formats; in order to reduce latency, the subtraction of the screen coordinates are done using fixed point arithmetic, but all other calculations are performed using the 24 bit floating point format of the vertex depth values. Accordingly, the fix point adder which generates the coordinate differences is followed by a fixed point to floating point converter before connecting to the floating point multiplier. The coordinate differences are multiplied with the output of the floating point adder (depth differences), generating the partial products of A_z, B_z and C_z. The output of the multiplier is routed to the input memories of the adder to be able to compute the A_z, B_z, C_z results. The A_z, B_z results are then multiplied with the reciprocal of the C_z value (hence the multiplier can use the result of the adder and the reciprocal unit), generating the final E_z and F_z values. To generate the initial depth value, E_z and F_z are multiplied with the starting *x* and *y* coordinates, and then these results are added together with the z_1 vertex depth value. To avoid the operand cancellation when adding together these values (that is when two large, almost equal values with different sign hide the impact of a small third operand, but then cancels each other), the adder has three inputs, and it always generates correct results; however, the third input is only used when the initial depth value is computed, otherwise it is set to zero (the shift register at the input allows the appropriate delay to be applied to z_1 compensate for the latency of the computations).

The whole arithmetic unit can be programmed by controlling the read address of the memories; the select signals of the multiplexers; the add/subtract control signals and the write enable signals of the different units. Scheduling analysis shown that the depth computation has nearly 48 clock latency (obviously, this depends on the program), but it is able to start processing a new triangle every 16th clock cycle. Of course, when programmed differently, latency and performance may differ, the above reported results are valid for an actual program which can handle anti aliasing.

As the different arithmetic units have different latencies which are not hidden from the programmer in any way, programming the unit is not the easiest task. However, the creation of simple compiler (data flow compiler) is obviously possible.

3.2 HSR Cell

The unit doing the actual hidden surface removal is made up from a number of similar cells. Every cell has its own buffer (storing depth, stencil and triangle pointer), and some segment lines assigned to it; using N cells, every nth line is processed by the nth cell (for example, with 8 cells and 32*16 resolution segment, the 1st and 9th line is processed by the 1st cell, the 2nd and 10th lines are processed by the 1st cell, and so on). After processing one of the associated lines, the HSR Cells require new input values, as they are unable to interpolate in the *y* direction. Irrespectively of the number of overlapped pixels, for every triangle every cell processes all of their associated pixels, so efficiency decreases if a triangle only overlaps a small number of pixels in a segment. However, this makes scheduling predictable, and this is why the VPU can have 16 clock cycles to generate new input data for the cells.

When the VPU has finished generating the new values for the starting pixel, the 1st cell loads these data. At the same time the input data is modified for the next cell (stepping one line in the *y* direction), which loads them one clock cycle later compared to the 1st cell. This method only requires one input at any time, so only one adder per

variable is needed to interpolate them in the *y* direction (these are the adders on Figure 3).

A cell itself consists of three distinct units. Covering determination identifies if a pixel is inside the processed triangle – and allows the write enable signal of the buffers to become active.

3.2.1 Covering Unit

After loading the initial values of the three S_j variables, a HSR Cell decrements this value with Δy (see Eq. 2-3) every clock cycle. The decision about overlapping is done using Eq. 2.

3.2.2 Depth Unit

The Depth Unit reads the depth buffer, compares the read value with the interpolated one (using the set comparison function) and in case the comparison returns true, allows the write back to the depth buffer. The pipelined architecture is shown on Figure 5, gray blocks represent registers, and white blocks are logic functions.

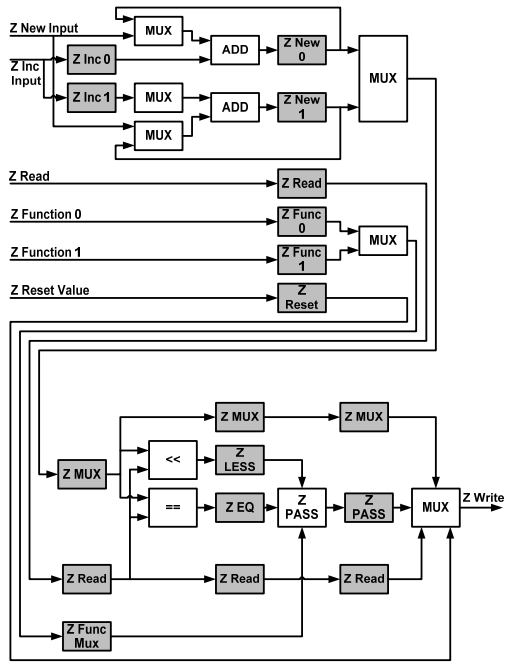

Figure 5. Depth Unit

This unit supports all possible comparison functions (Z Func – always, never, less, less-or-equal, equal, greater-or-equal, greater). The Depth Unit has two processing modes: opaque and transparent.

In opaque mode, two pixels are processed per clock cycle, the two Z Inc registers stores the same delta value and the depth functions are also similar.

Transparent, multi pass mode can be activated after all opaque triangles are processed. In this mode, the unit can process one pixel per clock cycle. In every pass, the transparent triangle which is farthest from the camera, but closer than the already processed triangles is determined for every pixel. This is done with two comparisons and using two depth buffer location: one location stores the depth value of the already processed (shaded) triangle, while the other is the working buffer. The former is compared with the interpolated depth value using less comparison function, while the latter is compared using larger. After finishing a pass, the two buffer locations changes function.

3.2.3 Stencil Unit

To keep up with the Depth Unit, the Stencil Unit also supports two stencil tests per system clock using 8 bit stencil values. As these two units share the same memory, the pipeline latency is also the same. Just as the Covering- and Depth Unit, it also generates write enable signal for the buffer, using the defined stencil reference value, read mask value, write mask value, comparison function and stencil operation. The comparison function has the same options which were listed for the Depth Unit, while the stencil operation can be set for “stencil test fails”, “stencil test passes and depth test fails” and “stencil test passes and depth test passes” cases. Available operations are: keep previous value, set to zero, replace with reference value, increment (with or without saturation), decrement (with or without saturation) and invert.

4. Arbitrary Anti Aliasing

The programmable Vertex Processing Unit together with the programmable segment size allows implementing anti aliasing (AA) with arbitrary number of samples and arbitrary sample positions.

Without AA, the x_{strt} and y_{strt} coordinates in Eq. 3 and Eq. 4 are set to the screen space coordinates of the top-left segment pixel, and the segment size is set to the real (pixel) resolution of the segment. The VPU processes a triangle once, computing the required delta values and the initial values for the top-left segment pixel. The HSR Cells use these values to determine covering and generate depth values at pixel centers.

Setting for example two-times AA requires only minor programming changes. Let define sampling positions as shown on Figure 6. The desired effect can be achieved by setting the vertical resolution of the segment to twice of the real size, and then modifying only the x_{strt} and y_{strt} values. First, the VPU computes the same delta values as without AA, but generates the start values using (*top left pixel x + pix_size/4*) as x_{strt} and (*top left pixel y – pix_size/4*) as y_{strt}. As long as the HSR Cells process the first sample positions, the VPU modifies x_{strt} to (*top left pixel x – pix_size/4*) and y_{strt} to (*top left pixel + pix_size/4*).

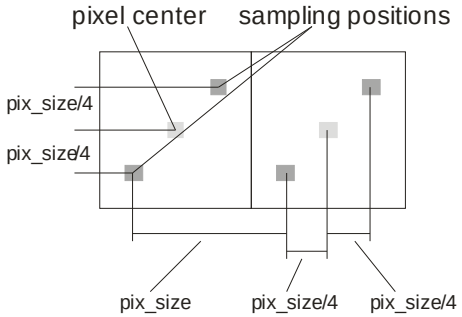

Figure 6. AA sampling pattern

Using this method, any number of AA samples can be generated within a pixel, as long as the precision of the coordinate computation allows it, and there is enough buffer memory. For example, a 32*16 resolution segment allows 64-times AA to be used. Obviously, depth/stencil buffering fill rate decreases linearly as the number of samples increases. It must be noted that the presented architecture only deals with oversampled covering determination and depth testing. To generate AA-ed images, the other units (especially the shader unit) have to support this arbitrary mode. Actually, the type of AA (multisampling – only object edges are filtered; supersampling – the whole image is filtered) applied is also dependent on those units – the output of the HSR allows both methods.

5. Conclusion

The article presented some hardware units which can be used to create an efficient rasterizer unit. In an FPGA implementation using XC2V6000-4 FPGA, all units can achieve 100 MHz system clock speed (with parts of the HSR Cells operating at double frequency), translating into 100 million segments/sec for the segmenting units, and up to 1.6 GPixels/sec depth/stencil fill rate for an 8 cell HSR Unit.

The two segmenting methods have to be analyzed using real-world applications to explore the overall performance increase exact segmenting offers. Similarly, to identify the potential benefits of the programmable segment size, the performance of real applications should be measured under different conditions. These measurements then may be used to create an algorithm to adaptively alter segment size between frames, which is simple enough to be implemented in hardware.

6. REFERENCES

[1] Michael Cox, Narendra Bhandari, *Architectural Implications of Hardware-Accelerated Bucket Rendering On the PC*, Siggraph/Eurographics Workshop On Graphics Hardware, 1997

[2] *Intel Zone Rendering Technology 3 Whitepaper*, http://support.intel.com/design/chipsets/applnots/302625.htm

[3] J. Torborg and J.T. Kajiya, *Talisman: Commodity Realtime 3D Graphics for the PC*, Proc. ACM Conf. on Computer Graphics Conference(SIGGRAPH '96), 1996

[4] Eyles, John, Steven Molnar, John Poulton, Trey Greer, Anselmo Lastra, and Nick England, *PixelFlow: The Realization*, Proceedings of the Siggraph/Eurographics Workshop on Graphics Hardware, 1997

[5] *PowerVR Tile Based Rendering Whitepaper*, http://www.pvrdev.com/pub/PC/doc/idx/whitepapers.htm

[6] Péter Szántó, Béla Fehér, *Exact Bucket Sorting for Segmented Screen Rendering*, GSPX 2005 Pervasive Signal Processing, 2005

[7] I. Antochi, B.H.H. Juurlink, S. Vassiliadis, P. Liuha, *Scene Management Models and Overlap Tests for Tile-Based Rendering*, EUROMICRO Symposium on Digital System Design, 2004

[8] *3DMark05 Whitepaper*, http://www.futuremark.com/companyinfo/?companypdfs