Exact Bucket Sorting for Segmented Screen Rendering

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

Segmented Screen Rendering (or Bucket Rendering) technique can considerably improve performance and/or lower external memory bandwidth requirements by segmenting the screen into small rectangles, and rendering these rectangles independently. Since the size of the buffers for a segment is significantly reduced compared to the full-screen buffers, it is possible to use on-chip buffers, while at the same time segments can be processed in parallel. The drawbacks of this technique are the necessity of the bucket sorting algorithm which computes the overlapped segments for every triangle (the basic element of 3D rendering), and the redundant work caused by triangles overlapping more than one segment.

This paper presents hardware architecture for exact bucket sorting, which – in contrast with bounding box bucket sorting – computes the segments overlapped by a given triangle exactly, reducing the redundant work in the following parts of the rendering pipeline.

Keywords
3D graphics architecture, FPGA, system on chip

1. INTRODUCTION

Traditional hardware architectures render the scene triangle by triangle, which requires random accesses to the working buffers; hence large, screen-sized buffers are employed. By segmenting the screen into small rectangles and render these rectangles independently, the size of the working buffers can be reduced so they can be fitted into on-chip memory. Segmented Screen Rendering also allows high level of parallelism by rendering multiple segments by multiple rasterization units. The efficiency of the shading part of the pipeline can also be increased considerably with deferred shading; deferred shading first compute the visible triangle for the screen pixels, and then only these visible pixels are shaded, making this process much more effective than traditional renderers where pixels not visible on the final image may be also shaded.

As any other techniques, this method also suffers from some drawbacks. First, the computation of the overlapped segments requires additional hardware unit and an additional external buffer is required to store a list for every segment containing the triangles overlapping those segment. Second, relatively large triangles overlap more than one segment, so later in the pipeline the attributes of these triangles are read more than once from external memory which clearly increases bandwidth usage.

Segmented Screen Rendering technique is used in many software and hardware systems, such as Pixar's RenderMan software, SGI Reality, Microsoft Talisman ([3]), PowerVR ([5]), Intel Extreme Graphics, PixelPlanes and PixelFlow ([4]) and ATI R300 architectures. However, the bucket sorting algorithm in these architectures is quite different. The simplest solution, employed by SGI does not do any bucket sorting, but processes all triangles in all segments. This method is quite ineffective, since a lot of unnecessary work is done when dealing with triangles not having pixels in the processed segment. The Talisman architecture, earlier PowerVR designs and Intel Extreme Graphics do bucket sorting in software, using the CPU. As long as the transformation part of the rendering pipeline is done on the CPU, this is a viable solution, but nowadays transformation is done in the graphics unit, so software bucket sorting requires too much bi-directional bus transfers to be effective (and moreover, CPUs in typical embedded systems may not have enough processing power to do this extra task fast enough). PixelPlanes and PixelFlow uses bounding box bucket sorting; this means that the triangle is processed in all segments inside its bounding box. For small triangles this is effective, but as triangles gets larger (or more exactly, "narrower"), or the segment gets smaller efficiency decreases. For example, Figure 1 shows a triangle where the bounding box method marks 360 segments as overlapped, while exact bucket sorting marks only 72 segments.

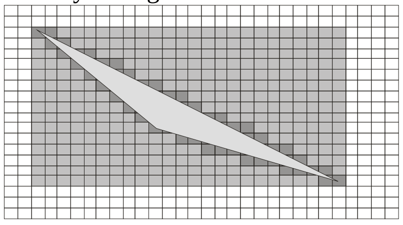
Figure 1. Bucket Sorting

Selecting the correct segment size is a key point in both bounding box- and exact bucket sorting architectures ([1]). However, the optimal size is both resolution and application dependent. Moreover, in our rendering architecture, larger segment size decreases the efficiency of the Hidden Surface Removal (HSR) unit ([7]), as with larger segment size, the ratio of covered/non-covered pixels within the segment decreases. While earlier papers discussed several algorithms ([2]), hardware implementations are typically limited to the bounding box method.

2. ALGORITHM OVERVIEW

Overlapping determination is based on a variable generated from the explicit equations of the triangle sides (Eq. 1.):

$$A(x,y) = (x - x_i) * Dy - (y - y_i) * Dx \qquad (1)$$

This variable is zero on the side, negative in one of the half planes and positive on the other half plane defined by the side. The exact sign on the two half planes depends on how the delta values in Eq. 1 are computed.

In our algorithm vertices are sorted by their *Y* coordinates and named accordingly, as Figure 2 shows.

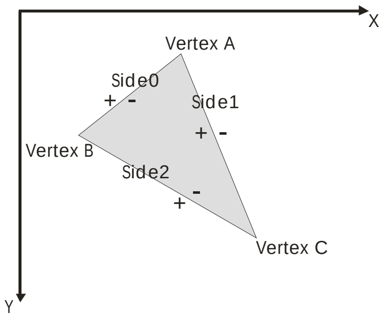

Figure 2. Covering determination

Triangle side numbers are also based on the vertices. *Side0* is defined by *Vertex A* and *B*, *Side1* is defined by *Vertex A* and *C* and *Side2* is defined by *Vertex B* and *C*. The figure also shows the sign of the mentioned variable (*A*) when delta values are computed by subtracting values at the vertex with greater *Y* coordinate from values at vertex with lesser *Y* coordinate; for *Side0* this means:

$$\begin{aligned} Dx &= x_A - x_B \\ Dy &= y_A - y_B \end{aligned} \qquad (2)$$

A point is inside the triangle if

$$\begin{aligned} &(sign(A_0(x,y))\ XOR\ sign(A_1(x,y)))\ AND \\ &sign((A_1(x,y))\ XOR\ sign(A_2(x,y)) \end{aligned} \qquad (3)$$

is true, where $sign(A_k(x,y))$ is the sign bit of the *A* coefficient of side *k* at *X,Y* screen coordinates (1 when the variable is negative, 0 otherwise). It can be easily seen that the coefficient must be incremented by Δy when stepping one pixel to the positive *X* direction, and decremented by Δx when stepping in the *Y* direction.

2.1 Segmentation algorithm

Processing a new triangle starts in the segment containing the top triangle vertex: *Vertex A*. After finishing this segment row, the algorithm steps down to the next row, and processes overlapped segments within that row. When finished, it steps down again. Processing a triangle is completed, when the row which contains *Vertex C* is ended. Processing a new row does not start in the leftmost or rightmost segment overlapped by the triangle, thus there may be overlapped segments in both horizontal directions.

Generally, five possible steps may be done in a segment (coordinate pairs – *X*, *Y* – in brackets show an example segment on Figure 3):

- Step right (positive *X* direction), eg. (4, 3)
- Step left (negative *X* direction), eg. (6, 5)
- Jump to the segment which is one segment left from the starting position (7, 3)
- Step one segment row, using the *A* values of the current segment, eg. (8, 4)
- Step one segment row, using the *A* values of a previous segment, eg. (4, 5)

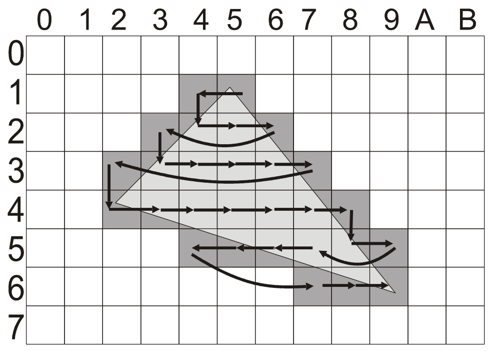

Figure 3. Stepping through a triangle

Step left and step right are self-explanatory, the algorithm steps in the current segment row. This horizontal stepping process starts with right stepping. After the right step procedure is finished, the algorithm jumps to the position which is one segment left from the starting segment, and starts left stepping. When both horizontal directions are completed, the algorithm steps down into the next segment row.

Depending on the last processed segment in the row, two cases are possible: if the segment below the current one overlaps with the triangle the algorithm steps into that segment. If this is not the case, the algorithm uses the data of a previous segment – the last processed segment which has the underneath segment overlapped with the triangle – for the row stepping.

2.1.1 Stepping right or left

This section describes the horizontal stepping actions. Only right stepping is discussed here, left stepping is essentially the same, but uses different segment side for the intersection tests. From now on, *SVertex TL/TR/BL/BR* refers to the top-left/top-right/bottom-left/bottom-right vertex of a segment.

In trivial cases, the necessity of a right step can be determined by inspecting the two right vertices (*SVertex TR* and *SVertex BR*) of the segment. If any of them is covered by the triangle, a step is certainly required (Case0). Unfortunately, this is not enough in all situations. To handle other cases, intersection values are generated. A triangle side intersects with a segment side, if the sign of the appropriate *A* value is different at the two segment vertices defining the segment side. Figure 4a. shows an example situation, when intersection values are necessary to correctly identify overlapping segments.

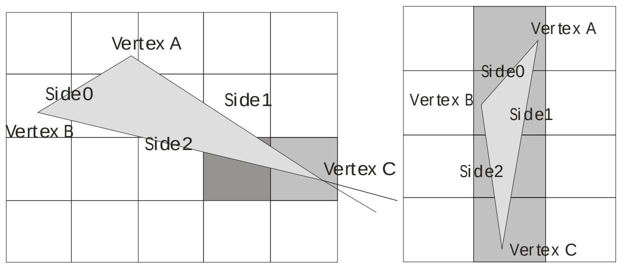

Figure 4. a. Side1 and Side2 Intersection; b. Step down

In the segment marked with dark gray a right step should be performed, because the right side of the segment has intersections with *Side1* and *Side2*. However, note that these intersections are also present at the light gray segment, where right stepping should be stopped, as *Vertex C* is reached.

There are similar cases for *Side0*/*Side1*/*Vertex B* and *Side0*/*Side2*/*Vertex B* triplets.

2.1.2 Jump in segment row

When processing a new segment row does not start in the leftmost segment a jump will be required after the step right process. The necessity of jump is determined by inspecting the need of right step and left step at the starting segment of the row; the *A* values at the left segment vertices (*SVertex TL* and *SVertex BL*) are saved (*AkSTRTL* and *AkSTRBL* in Table 1).

Jump takes the algorithm to the segment left from the starting segment (see Figure 3). The *A* values for *SVertex TR* and *SVertex BR* are the previously saved values, for *SVertex TL* and *SVertex BL* these have to be decremented by the delta values. If processing the current row is finished and the row containing *Vertex C* has not been reached, the algorithm steps down one row.

2.1.3 Row step

To ensure that row step takes the algorithm into a segment which is certainly overlapped by the triangle, the step does not occur in the last processed segment in the given row. Instead, a good place for the row step is calculated continuously; a segment is an acceptable step down position if the segment beneath it contains the triangle – which means (see Figure 3 and Figure 4b for easier understanding):

- Any of the segment's lower vertices are inside the triangle
- *Side0* and *Side1* have intersection points on the bottom side of the segment, and the segment row containing *Vertex B* has not been processed yet
- *Side1* and *Side2* have intersection points on the bottom side of the segment, and the segment row containing *Vertex B* has already been processed, and *Vertex C* is not in the currently processed segment row

If a currently processed segment is marked as an acceptable place for row step, its *X* and *Y* coordinates along with the *A* values of its bottom vertices (*SVertex BR* and *SVertex BL*) are stored (*AkSTLBL* and *AkSTLBR* in Table 1).

If the current position is a good step down point, the new *A* values should be computed from the values of the current segment; *A* values of *SVertex TL* are replaced with values at *SVertex BL*, while *A* values at *SVertex TR* are replaced with values at *SVertex BR*. New values for the bottom vertices are computed by decrementing the previous values with the appropriate delta values. If the current segment is not a good step down position, the values of the last good step down position should be used for the above computation.

3. HW OVERVIEW

The presented unit is part of a 3D rendering architecture, primarily targeted for embedded SO(P)C systems. Figure 5 shows the simplified block diagram of our system, the segmenting unit marked with gray.

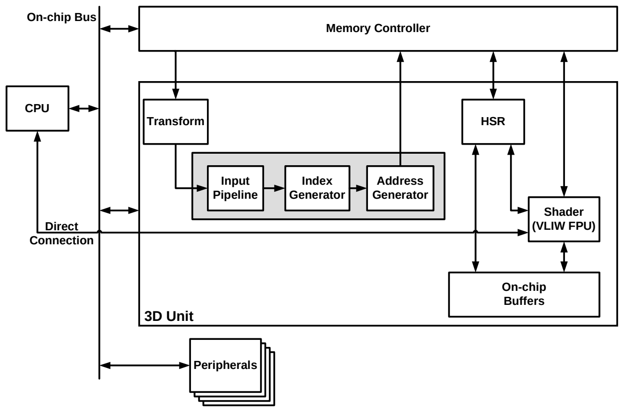

Figure 5. SO(P)C system with 3D rendering capability

As our development platform is a Xilinx XC2V6000-4 FPGA, the selected microcontroller is a Xilinx Microblaze.

The 3D Unit is connected to the processor through the standard OPB on-chip bus, while the Shader Unit will have a direct, fast connection (FSL), to act as a more general co-processor (the Shader is under development).

The Segmenting Unit uses the *X* and *Y* coordinates of the transformed and clipped vertices to determine the overlapped segments for every triangle. It consists of three main parts. The *Input Pipeline* calculates the required initial and incremental (Δx, Δy) values for the *Segment Generator*, which does the actual overlapped segment computation. The *Address Generator* handles communication with the external memory controller and generates an appropriate data format to be written to the memory.

Table 1 summarizes the required *A* value operations in different step cases. *AkTL*, *AkTR*, *AkBL* and *AkBR* represent the arrays of the three *A* variables at the different segment vertices; for example *AkTL* array contains the three *A* variables of the three triangle sides at the top-left segment vertex (*SVertex TL*). With the same logic, the *dkX* and *dkY* arrays contain the three Δx and Δy values (see Eq. 1) for the three triangle sides.

Table 1. Micro operations

	AkTL	AkTR	AkBR	AkBL
Load	AkTL_In	AkTR_In	AkTR_In – dkX_In	AkTL_In – dkX_In
Right	AkTR	AkTR + dkY	AkBR + dkY	AkBR
Left	AkTL – dkY	AkTL	AkBL	AkBL – dkY
Jump	AkSTRTL – dkY	AkSTRTL	AkSTRBL	AkSTRBL – dkY
Row + GP	AkBL	AkBR	AkBR – dkX	AkBL – dkX
Row	AkSTLBL	AkSTLBR	AkSTLBR – dkX	AkSTLBL – dkX

Load is the starting phase, when the *Segment Generator* loads the input values generated by the *Input Pipeline* – the three *A* values for *SVertex TL* and *SVertex TR*. The values for the other two segment vertices are computed in the *Segment Generator*.

Right, left and jump represent the appropriate horizontal stepping cases, *AkSTRTL* and *AkSTRBL* are the arrays of *A* values at *SVertex TL* and *SVertex BL* at the starting segment (left segment vertices).

Row+GP is the case when the actually processed segment is a good step down point, while Row is the case when it is not. In the previous case, the *A* values of the current segment should be used for the row stepping, while in the latter case the *A* values at *SVertex BL* and *SVertex BR* of the last good step down point are used – namely *AkSTLBL* and *AkSTLBR*.

3.1 Input Pipeline

The *Input Pipeline* reads transformed vertex coordinates, computes the required input values for the Segmenting Generator and handles synchronization and load balancing with previous units via a FIFO. Architecturally, it consists of two pipelines. The first, short pipeline reads the FIFO and generates valid triangles with vertices sorted according to their *Y* coordinates. The second, longer pipeline computes the required values for the *Segment Generator*: incremental values (Δx, Δy) for the three triangle sides and the *A* values of the three sides at *SVertex TL* and *SVertex TR*. The pipeline stages do the following work:

- Sort triangle vertices by their *Y* coordinates
- For each triangle vertices, compute the segments which contain the given vertex
- Compute screen coordinates of *SVertex TL*
- Compute delta values in Eq. 1 from *SVertex TL*
- Compute products in Eq. 1
- Compute *A* variable for *SVertex TL*
- Compute *A* variable for *SVertex TR*

To limit resource usage, the *Input Pipeline* computes all required values for a new triangle in three clock cycles. This limits performance of the *Segmenting Unit* when triangles are very small compared to the segment size and only overlap pixels in one or two segments, however with programmable segment size and real objects this should not happen too frequently.

3.2 Segment Generator

Figure 6 shows the high level block diagram of the *Segment Generator*. Gray boxes represent registers, white ones are combinatorial logics.

SX and *SY* contain the processed segment's *X* and *Y* coordinates. When a new triangle is loaded these registers are updated with the *X* and *Y* coordinates of the segment containing *Vertex A* (*VA_Segment_X* and *VA_Segment_Y* on Figure 6). The coordinates of the segments containing *Vertex B* and *Vertex C* are also saved into *VB_Seg* and *VC_Seg* registers. The *ADDSUB* blocks right to the *SX*, *SY* registers consist of a large number of adders and subtractors to generate all possible coordinates for the different steps, while the *MUX* units select the appropriate values to write back into *SX* and *SY* using the output of the *SEL LOGIC* block (which generates the step to be done). Generating all possible results and multiplexing them requires more logic resources than multiplexing the inputs of the adders, but our target clock frequency required this method. The *COMPMUX* blocks compare the segments of *Vertex B* and *Vertex C* with all possible segments that are possible in the next cycle, and selects the appropriate using the output of the *SEL LOGIC*. This value is written into a register and used by *SEL LOGIC*.

The upper part of the figure is responsible for modifying the *A* values according to Table 1. The *Ak_REG* block is the array of the 12 *A* values: for all three triangle sides at all four segment vertices. The *ADDSUB* unit increments/decrements the contents of the *A* registers with the delta values (as with *SX* and *SY*, all possible results are generated for the fastest possible clock speed), while the *MUX* right to it selects the correct input according to the step being done (output of *SEL_LOGIC*). *SEL_LOGIC* generates the step to be done, using the "inside the triangle" signals (generated by *IN_TR*) of the segment vertices, the intersection values (generated by *INTERS*), and the contents of the *VB_InRow*, *VB_InCol*, *VC_InRow*, *VC_InCol* registers. These registers contain a flag which is set if the current segment row and column contains *Vertex B* or *Vertex C*.

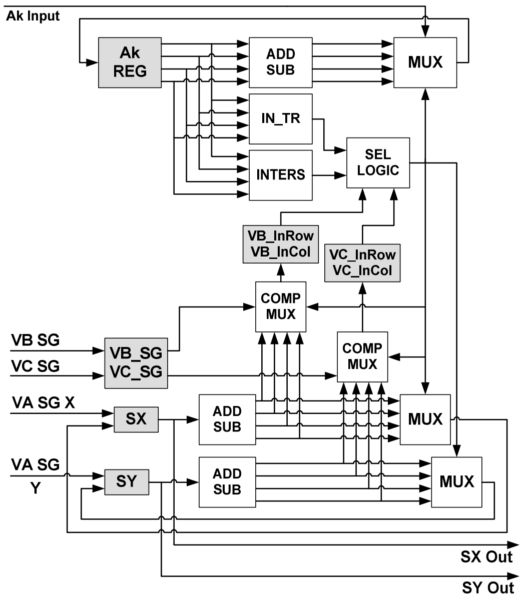

Figure 6 Segment Generator architecture

Figure 7 shows the more detailed schematic of the control logic (the generation of the step signal). The registers at the left side contains the array of *A* values for each segment vertex. The control logic uses only the MSB of these values (sign bit). The "*INSIDE SVTL ... SVBL*" blocks are XOR/AND networks, which generate "inside the triangle" signals for the four segment vertices according to Eq. 3.

The blocks beneath (*INTERS RIGHT/BOTT/LEFT*) generate the necessary "intersection" signals for the left, bottom and right sides of the segment according to the cases in 2.1.1. The *LEFT_LOGIC* and *RIGHT_LOGIC* generate a left and right step required signal using the intersection control signals, the "inside the triangle" signals and the *VB_InCol* and *VC_InCol* signals. The *STEP_SEL* block is a priority encoder which selects the appropriate step from the mentioned signals and the synchronization signals. *LOAD* is active for one cycle when new triangle is read, *BUSY* is set by *LOAD*, and reset when processing the current triangle is finished.

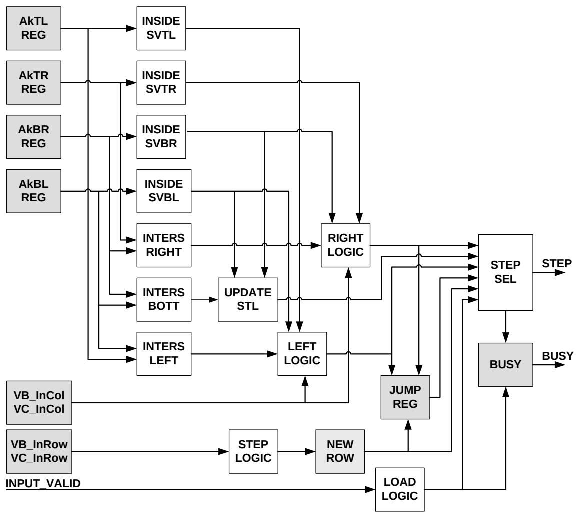

Figure 7. Control Logic

3.3 Address generator

The address generator accepts the output of the *Segment Generator* as an input (triangle pointer and segment coordinates), and generates an appropriate external memory address for the triangle number (pointer) to write to. Pointers for a given segment are stored in 32-word blocks in the external memory. The first 31 element of these blocks are triangle pointers, while the 32nd is a pointer to the next 32-word block. This organization allows the HSR Unit to read the pointers in bursts, and does not waste too much memory (depending on the average number of triangles falling into a segment). Figure 8 shows the schematic of the unit.

The *Address Memory* stores one word for every segment: the memory address where to write the next triangle pointer. The *SEGMENT_NUM* block generates the read address for this memory from the *SX* and *SY* coordinates of the segment. This address is stored in a register for later use. The five least significant bits of the output of the memory is compared with *COMP_VALUE* (which equals to 31) to determine if the write address is the last element in a 32-word block – so new block should be reserved. The result of the comparison is stored in a register, just as the output of the *Address Memory*.

When no new block is required, the output of the *Address Memory* becomes the address for external memory write (*EXT_WR_ADDR*), and one new value can be written at every clock cycle. The word assigned to the given segment in *Address Memory* is updated with the incremented value of the write address.

Reserving a new 32-word block requires two external memory writes: the pointer to the new block (stored in

NEXT_BLOCK register) should be written to the last element of the current block, while the triangle pointer should be written to the first element of the new block. In order not to exceed one memory write per clock, the *Address Generator* disables the *Segment Generator* for one clock cycle in such cases (this is also needed when the memory is busy with other tasks). In the first clock cycle, the triangle pointer is written into the first element of the new block. In the second clock cycle, the pointer to the new block is written into element 32 of the current block, using the content of the *AMEM_OUT* register. The appropriate word of the *Address Memory* is updated with the incremented pointer of the new block, while the *NEXT_BLOCK* register is incremented by 32.

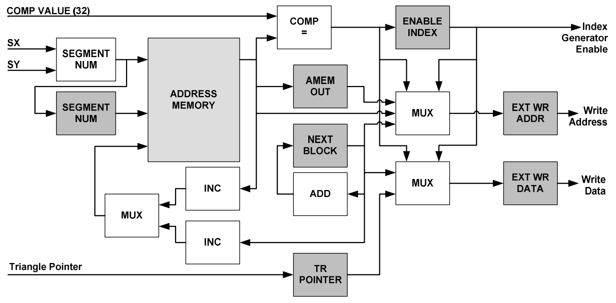

Figure 8. Address Generator

When processing a new frame starts, the memory containing the write addresses should be set to the initialization value, which is segment number multiplied with 32. This is done by rendering two triangles which act as a full-screen background.

4. CONCLUSION

The *Segmenting Unit* can generate a new valid segment coordinate at every clock cycle, and the structure of the Pointer Buffer allows fast, burst reads in the HSR Unit.

Preliminary clock speed requirements were defined by two factors. Our development platform has embedded multipliers, which can achieve about 100 MHz. The selected central microprocessor (MicroBlaze) also works at this frequency, so 100 MHz was selected as a target for our modules to achieve after place and route. The presented architecture is able to fulfill this, using Synplicity Amplify synthesizer.

4.1 Theoretical performance numbers

With the achieved clock speed some general statements can be made about worst case and best case performance of the architecture.

The worst case is when every triangle is relatively large, and segment size is set to minimum (32*16 pixels). For example, with an average 100 segments/triangle, and 640x480 resolution, one triangle takes 100 clock cycles to process, so performance is 1000000 triangles/second. The best case is when all triangles cover pixels in only three segments (below this, the *Input Pipeline* is the limiting factor). In this case the unit can process one triangle every clock cycle, so peak performance is 33 million triangles/ second.

With 30 frames/second rendering speed, the worst case situation allows 33000 triangles/frame, while the best case is 1.1 million triangles/frame. The first number is not that much; however it is still enough for simpler scenes. The second number is far higher – real-time applications do not use such a large number of triangles today, and it is not likely to be reached in the near future.

Real applications are somewhere between the two presented situations, typically somewhat closer to the best case.

4.2 Logic utilization

The following table summarizes the required logic resources for the presented functional blocks in our target architecture. FF shows the required number of flip-flops, LUT is the number of 4-input look up tables, MUL is the number of embedded multiplier blocks, while BRAM is the number of the required BlockRAM blocks.

Table 2. Logic utilization

	FF	LUT	MUL	BRAM
Input Pipe.	1100	1200	4	-
Segment Gen.	900	2800	-	-
Address Gen.	250	250	1	2

5. REFERENCES

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

[2] 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

[3] Jay Torborg, James T. Kajiya, *Talisman: Commodity Realtime 3D Graphics for the PC*, ACM Computer Graphics Proceedings, 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] Imagination Technologies Ltd, *PowerVR SDK*, www.pvrdev.com

[6] ATI Technologies, *Radeon X800 Architeture White Paper*, http://www.ati.com/products/radeonx800/RADEONX800ArchitectureWhitePaper.pdf

[7] Péter Szántó, Béla Fehér, *High Performance Visibility Testing with Screen Segmentation*, ESTIMedia 2004