# High Performance Visibility Testing with Screen Segmentation

Péter Szántó, Béla Fehér
*Budapest University of Technology and Economics*
*Department of Measurement and Information Systems*
*szanto@mit.bme.hu, feher@mit.bme.hu*

## Abstract

*There are two factors determining the performance a 3D accelerator can achieve: the available computational power and the available memory bandwidth. In embedded systems, these resources are even more limited then in desktop environments, thus the efficiency of the hardware architecture and the exploitation of the logic resources become even more important. Most resources are wasted at the visibility testing process: traditional implementations require a lot of bandwidth, and process pixels which are not visible on the final image. By segmenting the screen, the presented architecture can use high performance, on-chip buffers to lower memory requirements and to provide high performance. The order of the processing guarantees that only those colors are computed, which are truly visible. The modular architecture allows satisfying different requirements; a trade off can be made between the number of processing units and performance.*

## 1. Introduction

Three dimensional image synthesis becomes more and more important even in embedded systems (such as handheld devices), as customers demand more functions, more entertaining applications and better quality. Although embedded CPUs get faster, the complexity of quality 3D rendering still requires dedicated hardware acceleration to provide convenient rendering speed. The increasing complexity and decreasing price of FPGAs allow very complex designs to be implemented in a single chip, as a system on programmable chip (SOPC).

From the 3D rendering perspective, embedded systems have fewer resources than desktop systems, while at the same time the lower resolution lowers performance requirements. As shown in the next section, most resources are wasted during image synthesis because of pixels which are processed, but are not visible on the final image.

## 2. 3D rendering basics

Real time 3D graphics rendering algorithms are based on triangles. Complex object surfaces are approximated by a so called triangle mesh – in general cases the more triangle is used to define an object, the more lifelike the resulting model is. The virtual objects are defined in their local coordinate system. To visualize the complete virtual world on a 2D display, the vertices of the objects are first transformed into a 3D space representing the whole world; with the camera placed into the origin and facing into the positive Z direction. The next step projects this space into the 2D screen space. Due to the non-linear behavior of this perspective projection, the 3D world space Z (depth) coordinate do not change linearly with the screen X, Y coordinates. Therefore, new Z values are generated during the process, which fulfill the linear requirement.

Another per-vertex operation is the vertex lighting. For every vertex, the effects of the light sources are computed according to the light source type and color, and the material properties of the object. The final result is a diffuse and specular color component at every vertex.

The per-vertex transformation is followed by the per–pixel rasterization process. For every screen pixel, the visible triangle is determined and its color is computed. Color computing can be done using the computed vertex colors by interpolating them across the triangle surface. This can be combined with other processes, such as texture mapping. Texture mapping assigns 1D, 2D or 3D arrays to triangles by defining an element of the array at the triangle vertices (texture coordinates) thus stretching it onto the triangle surface. In the simplest case, a 2D texture is nothing more than the photo of the real surface which is “painted” onto the triangle.

This paper focuses on one step of the rasterization process: the determination of the visible triangle for the screen pixels.

### 2.1 Wasting resources

In traditional implementations, rasterization happens triangle by triangle, while visibility testing is based on the well known Z Buffer algorithm. Usually there is a per-pixel buffer for color values (Frame Buffer) and a buffer for storing depth (Z) values (Z Buffer or Depth Buffer), both placed into external memory ([1]).

The Z Buffer algorithm finds the visible triangle for every screen pixel – this triangle is usually the one, which is closest to the camera in the given pixel, that is, which has the smallest Z value. At any given time, the Z Buffer stores the

minimum Z value of the already processed triangles. When processing a new triangle, for every pixel covered by it, a color value is computed together with a per-pixel depth (Z) value. The depth value is then compared with the value in the Z Buffer, and in case this depth test passes (the new Z value is smaller than the one read from the Z Buffer), the new color value is written into the Frame Buffer, while the new depth value is written into the Z Buffer. If the depth test fails, the computed color value is discarded.

Apparently, this process not only computes unnecessary color values, but requires a lot of memory bandwidth for Z Buffer reads and writes. Compared with desktop environment, in SOC systems, the available bandwidth is much less and it is shared between more processing units, thus architecture with the least amount of bandwidth demand is the most adequate.

Our main goals were to reduce unnecessary computations, reduce bandwidth requirements and achieve adequate performance for the implementation of more advanced visual effects (like shadowing). Another key point was to accelerate as much operation with dedicated hardware as possible to relieve the central processing unit.

## 2.2 Handling of transparent objects

One serious drawback of the original Z Buffer algorithm is the inability to handle transparent objects correctly. As the transparency effect depends on the order of processing, translucent triangles have to be sorted before processing them. For correct results, sorting should be done at the pixel level; however most hardware renderers do not support this operation, so only a rough sorting is done by the CPU before triangles are sent to the 3D accelerator.

To eliminate the need of this sorting by the CPU, the presented design handles transparent triangles in hardware with a multi-pass procedure ([2]).

## 2.3 Shadow effects

One of the most impressive and most important visual effects for lifelike image synthesis is the generation of shadows. A lot of research was invested into this area, and nowadays the two most widely used algorithms are shadow mapping ([3]) and stencil shadow volumes ([4]). These algorithms can be accelerated with an appropriate Z Buffer architecture. However, it is beyond the scope of this paper to discuss these algorithms in details, just a very brief description is presented to help imagine the required hardware functionality for fast processing.

**2.2.1. Shadow Map based shadows.** The Shadow Map algorithm renders the screen form the light sources point of view, and saves the resulting depth values into a texture (no color values should be computed during this process). The scene is then rendered from the camera point of view, and the saved depth values are used to determine whether a given pixel is in shadow or not. When multiple light sources are used, multiple depth maps must be computed, so to maximize depth mapping performance, the computed (and just as importantly, the saved) number of depth values per second should be maximized.

**2.2.2. Stencil Shadow Volumes.** The Stencil Shadow Volumes algorithm uses another per-pixel buffer, the Stencil Buffer. Just like the Depth Buffer, the Stencil Buffer has a comparison function, but it also has different functions depending on passing the stencil and/or the depth test. For example the stencil values for the pixels covered by triangle N. may be incremented when both the depth and stencil test passes, decremented when stencil test passes and depth test fails, and remain unchanged when both tests fail.

The algorithm first computes the silhouette of the shadow casting objects, then projects the vertices away from the light source, thus generating a volume which is in shadow (shadow extrusion process, see Figure 1). These are vertex based processes, so the CPU or the graphics hardware's geometry engine is responsible for them.

The rendering process is then divided into multiple passes (shown for one light):

1. The scene is rendered without the shadow casting light source, stencil buffer is set to 0
2. The front faces of the shadow volumes are rendered. When depth test passes (so the shadow volume is closer to the camera, then the object visible on that pixel) the stencil buffer is incremented**.**
3. The back faces of the shadow volumes are rendered. If the depth test passes, stencil buffer is decremented.
4. Render the scene with the light source, but only update pixel colors if the stencil value is zero (the pixel is not in shadow).

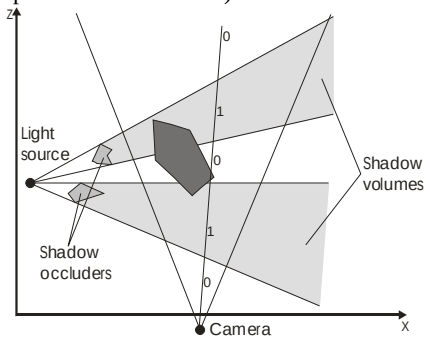


**Figure 1. Stencil Shadow Volumes**

This technique requires fast depth and stencil buffering to perform adequately.

# 3. Architecture overview

To spare external memory bandwidth, the presented architecture uses on-chip Depth- and Stencil Buffers. Naturally, the buffers for the entire screen cannot fit into on-chip memory (a 640*480 resolution screen requires more than one Mbytes for these buffers), so the screen is segmented into small rectangles (henceforward referred as tiles), which can be handled entirely on-chip. Rendering happens tile by tile – first the visible triangle is determined

for every pixel in the given tile (HSR – hidden surface removal – process), and then colors are computed only for the visible pixels.

A typical scene in real-time applications contains ten thousands of triangles. Traditional rendering hardware architectures work triangle by triangle, so every object is processed once. With the screen divided into tiles, processing every triangle in every of them would make the architecture very inefficient, so before the actual rasterization starts, another hardware unit (Indexing Unit) processes all triangles in the scene, and generates a list for every tile containing pointers to triangles, which covers at least one pixel in the given tile.

Figure 2 shows the high level schematic of the whole Rasterization Unit.

The Memory Controller handles communication between the external memory and the internal processing units.

The Vertex Buffer stores data associated to vertices (x, y coordinates, depth value, color, and so on). Its output is routed to all processing units.

The Pointer Buffer stores the output of the Indexing Unit in 32 word blocks. The first N number blocks (where N is the number of tiles) are associated with the appropriate tiles, and further blocks are reserved as required. To implement this scheme, the first 31 elements of the blocks are pointers to triangles, while the 31$^{st}$ is a pointer to a new 32 word block.

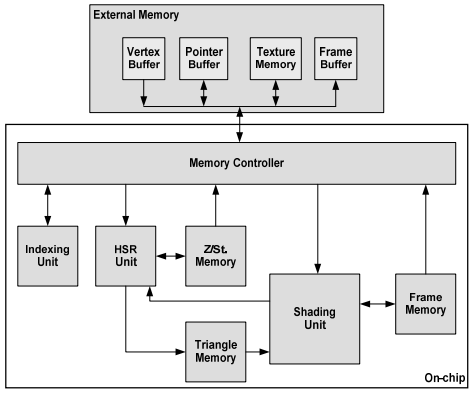


**Figure 2. Rasterization Unit**

The Pointer Buffer is read by the HSR Unit. For every given tile, only those triangles are processed by the HSR Unit, which are listed in the pointer Buffer.

The texture Memory stores 1D, 2D and 3D arrays of data. For fast shadow map generation Texture Memory can be written with data from the internal Z/Stencil Buffer; or for other special effects from the internal Frame Memory.

The Shading Unit is responsible for computing output color values by using vertex colors and texture data. It has an on-chip Frame Memory containing the output colors of one tile, which are saved into external memory after processing of the tile finished. Bidirectional communication with this on-chip memory is required to implement transparency (the computed opaque color has to be read, modified with the transparent color and written back).

The Indexing- and HSR Unit are responsible for the previously mentioned visibility determination. This paper presents the architecture of the HSR Unit.

# 4. The HSR Unit

The HSR Unit works tile by tile. Within a tile, triangles are processed one after the other, reading the Pointer Buffer and then the vertex data for the triangles listed in the buffer. The unit has three functions: determination of the covered pixels, depth testing and stencil testing.

The whole HSR Unit consists of sixteen HSR Cells and an Input Processing unit, as Figure 3 shows. Each cell has its own Z/Stencil- and Triangle Memory; the former stores per-pixel Z and Stencil values in a single 32 bit word (8 bits for stencil value and 24 bits for depth value), while the latter contains per-pixel triangle pointers (which are the input of the Shading Unit).

You may notice that the HSR Cell has input from the Shading Unit. This is necessary because some special rendering techniques require the modification of the Z value during the shading process.

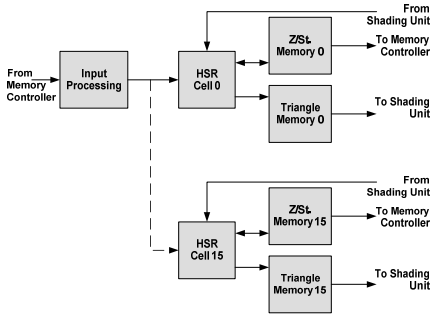


**Figure 3. HSR Unit**

Due to the separate memories, the sixteen cells can work in parallel, achieving fast depth- and stencil testing performance.

The tile size is configurable, so it can be adjusted for different applications to perform optimally. The minimum size is 16 lines by 32 pixels, while the maximum is defined by the memory size.

The pixel rows of the tiles are assigned to HSR Cells. In the case of 16 line tile, every row is processed by a different cell. As the vertical size of the tile increases, every M$^{th}$ row is processed by the M$^{th}$ HSR Cell. In a given row, a cell steps through all pixels, irrespectively of the pixel being covered by the triangle or not.

## 4.1. Covering Determination

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

$$A_s(x,y) = (x - x_0) * \Delta y - (y - y_0) * \Delta x, \quad (1)$$

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. Figure 4 shows the situation when vertices are sorted according to their y coordinates, and numbered accordingly. Triangle sides are also numbered based on the vertex they are in front of. The figure also shows the sign of the mentioned variable ($A_s$) when delta values are computed by subtracting values at the

higher numbered vertex from the lower numbered vertex's data; for *side2* this means:

$$\Delta x = x_0 - x_1, \quad \Delta y = y_0 - y_1 \tag{2}$$

A point is then inside the triangle if

$$(sign(A_0(x,y))) \wedge sign(A_1(x,y))) \,\&\, (sign(A_1(x,y)) \wedge sign(A_2(x,y))) \tag{3}$$

is true, where $sign(A_k(x,y))$ is the sign bit of the $A_s$ coefficient of side $k$ at $X,Y$ screen coordinates (1 when $A_s$ is negative, 0 otherwise), '^' is XOR while '&' is AND operation.

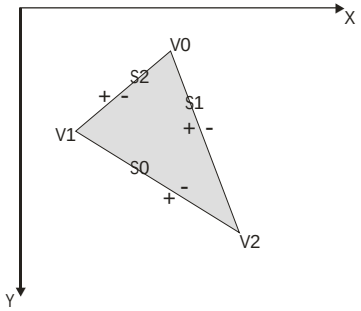


**Figure 4. Covering determination**

It can be easily seen that the $A_s$ variable must be incremented by $\Delta y$ when stepping one pixel to the positive $X$ direction, and by $\Delta x$ when stepping in the $Y$ direction.

## 4.2. Interpolating depth values

The perspective projection generates Z values in the vertices which change linearly with the screen coordinates across the triangle surface. This means that per-pixel values can be computed using the plane equation

$$z(x,y) = E_z * x + F_z * y + G_z = \frac{A_z}{C_z} * x + \frac{B_z}{C_z} * y + \frac{D_z}{C_z} \tag{4}$$

where the coefficients can be computed using values given at the vertices (Eq. 2):

$$\begin{aligned} A_z &= (z_1 - z_2) * (y_1 - y_0) - (y_1 - y_2) * (z_1 - z_0) \\ B_z &= (x_1 - x_2) * (z_1 - z_0) - (z_1 - z_2) * (x_1 - x_0) \\ C_z &= (x_1 - x_2) * (y_1 - y_0) - (y_1 - y_2) * (x_1 - x_0) \\ D_z &= A_z * x_0 + B_z * y_{0.} + C_z * z_0 \end{aligned} \tag{5}$$

## 4.3. Input processing

The input processing unit uses vertex data from the Memory Controller to produce the necessary input values for the HSR Cells.

First, it computes the three $A_s$ values for the three triangle sides, and the coefficients for interpolating the Z value. Then, it computes the $A_s$ values and a Z value for the top-left tile pixel – or in other words, the first pixel assigned to the $0^{th}$ HSR Cell. In the following clock cycles, the Input Processing Unit increments the computed $A_s$ values by the appropriate $\Delta x$ values, and the Z value by $F_z$ – with that, generating these values at the next pixel row's leftmost pixel.

The $N^{th}$ HSR Cell therefore reads its input N clock cycles after the $0^{th}$ cell read its values. This means that loading one triangle's input data into all cells takes 16 clock cycles. This way, only four adders (three for the $A_s$ variables, one for the Z value) are required to generate the input values for all pixel rows.

## 4.4. The HSR Cell

The HSR Cell has four subunits: the Address Generator, the Covering Unit, the Depth Unit and the Stencil Unit, as Figure 5 shows.

**4.4.1. The Depth Unit.** The global architecture of the cell is mainly defined by the functionality of the Depth Unit. As mentioned earlier, the presented architecture supports hardware sorting of transparent triangles. This is achieved with storing two depth words for every pixel: one at even Z/Stencil Memory address, and one at odd address.

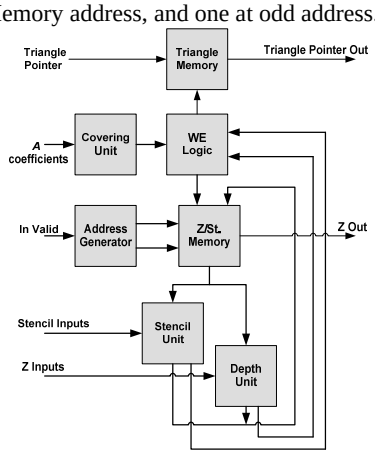


**Figure 5. HSR Cell**

Processing starts with opaque triangles, and happens just like with the ordinary Z Buffer algorithm, using even addressed Z Memory words. At the end of the process, the Z Memory contains the Z value of the visible opaque triangle at every pixel, while Triangle Memory contains a pointer to this triangle. The Shading Unit computes an opaque output color value for every pixel using these pointers, and saves them into the internal Frame Memory.

The next processing step is an even transparent pass. In this pass, transparent triangles are processed to find the transparent triangle for every pixel which is farthest away from the camera, but closer than the opaque triangle. This can be determined by two Z comparisons per pixel: the Z value of the processed triangle must be compared with the opaque Z value (even addressed Z Buffer word), and with the already processed transparent value (odd addressed Z Buffer word) – the depth test passes if both comparisons pass. In this case, the odd addressed Z Buffer word is updated with the new transparent value together with the pointer in Triangle Memory. After the end of this pass, the color of the transparent triangles in the Triangle Memory are computed and blended with the opaque color in the Frame Memory.

HSR process continues with an odd transparent pass. In this pass even and odd Z Buffer words change role: now odd words act as already processed Z values, while even addressed words are used to find the transparent triangle which is farthest away from the camera, but closer than the previously processed transparent triangle (or in other words: second farthest away from the camera but closer than the opaque triangle).

Processing takes as many passes as many transparent layers are on the scene. Figure 6 shows the schematic of the Depth Cell (for smaller size, inputs directly connected to registers are not labeled – they are the appropriate input values from Input Processing Unit).

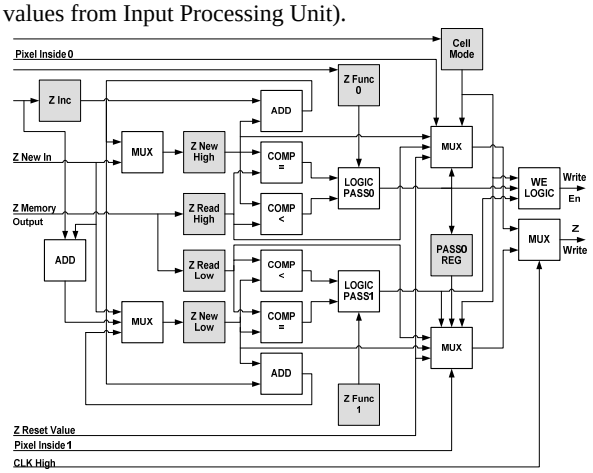


**Figure 6. Depth Unit**

To be able to process one transparent pixel per system clock, the cell contains two depth comparison logics, while the Z Memory is read with doubled system clock (referred as fast clock). In the schematic, grey blocks represent registers, while white blocks are logics. Registers are:

- Z Read High: the output of the Z/Stencil Memory, sampled at the falling edge of the system clock
- Z Read Low: the output of the Z/Stencil Memory, sampled at the rising edge of the system clock
- Z Inc: the value to increment Z as the cell steps to the x direction: $E_z$ (see Eq. 4), or double of it
- Z New High: a new (interpolated) Z value, updated at the falling edge of system clock
- Z New Low: a new (interpolated) Z value, updated at the rising edge of system clock
- Z Func0, Z Func1: a comparison function for the depth test (never, always, less, less-equal, equal, greater-equal, greater)
- Cell Mode: operating mode of the cell (opaque, transparent even pass, transparent odd pass)

**Opaque mode.** In this mode the HSR Cell can perform depth test for two pixels. The Z/Stencil Memory is read and written at twice the system clock, however the complex comparison and selection operations has too high latency to be performed at twice the system clock, therefore the memory's output are stored in two registers (Z Read Low and High), both updated at system clock rate. For the same reason, the interpolated Z values are also stored in two registers; in opaque mode the Z New High stores interpolated Z values for even numbered pixels in the processed pixel row, while Z New Low stores the value for odd pixels. This means that these registers should be incremented with $E_z$ multiplied by two (stored in Z Inc).

**Transparent mode.** In this mode only one pixel is processed per system clock, as two comparisons are required per pixel. Z High now stores the already processed Z value, while Z Low stores the transparent Z value. The result of depth test 0 is stored in a register to be available during the time of depth test 1 (half system clock delayed). Write is only possible once, after both depth tests are evaluated – at this time only Z Read Low and Z New Low contains valid data. The Depth Cell does not differentiate between even and odd passes; this is handled by the Address Generator.

**4.4.2. Stencil Unit.** The Stencil Unit is also capable of two stencil tests per system clock. Architecturally it is very similar to the Depth Unit, with the key difference being the different function.

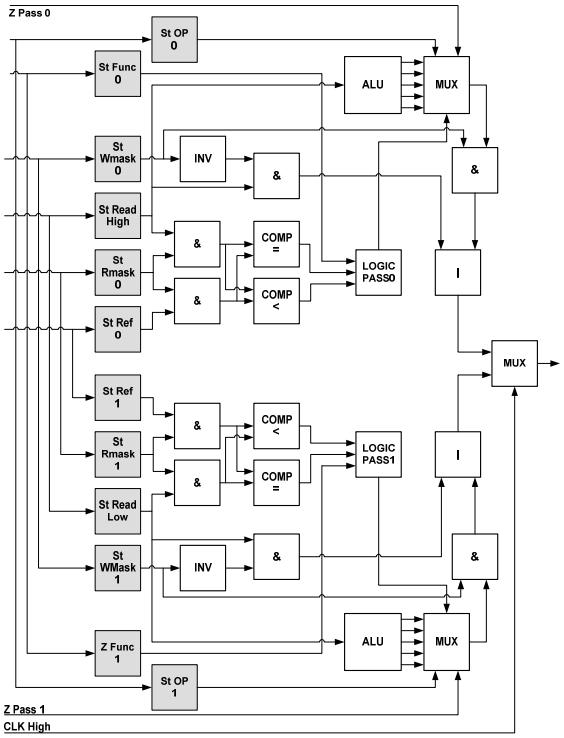


**Figure 7. Stencil Unit**

A stencil test is passed if

$$(Ref \ \& \ RMask) \ COMP \ (StBuff \ \& \ RMask) \qquad (6)$$

is true, where Ref is a reference value, RMask is a read mask, StBuff is the stencil buffer value and COMP is the comparison functions (same options as with the Depth Unit). The stencil buffer is updated with the following value:

$$(StBuff \ \& \sim WMask) \,|\, (WMask \ \& \ OP(StBuff)) \qquad (7)$$

where WMask is a write mask, and OP is one of the possible operations (among others: replace with reference, increment, decrement). Unlike the Z Buffer, the Stencil Buffer is not only updated when the stencil test passes; different operations may be set for "stencil test fails", "stencil test

passes and depth test fails" and "stencil test passes and depth test passes" cases.

Figure 7 shows the schematic of the Stencil Unit (again, trivial inputs are not labeled). Like the Depth Unit, the Stencil Unit is also pipelined, so most registers (masks, functions) are doubled – the only difference being them is the load time: registers with 0 index loads its values on the falling edge of the system clock, while the others loads on the next rising edge when processing of a new line starts.

The logic next to the registers generates the stencil pass signals. The ALU computes the results of the possible operations, and a multiplexer selects the appropriate result based on the stencil and Z pass signals and the set operation. The next logic blocks generate the value to be written into the Stencil Buffer according to Eq. 7. The final multiplexer selects one of the two results considering the system clock state.

**4.4.3. Address Generator.** The unit is responsible for generating the read- and write address for the internal memories, according to the operation mode. Figure 8 shows the block diagram of the unit.

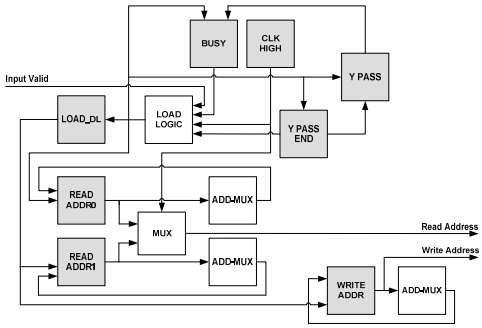


**Figure 8. Address Generator**

In opaque mode, both the read- and write address are incremented by two every fast clock, thus reading and writing even addresses in the memory.

As mentioned earlier, when processing transparent triangles writing can only happen after both depth tests are evaluated, so only Z Read Low and Z New Low are valid sources. In even transparent pass, this is not a problem, as odd addresses must be written, and the valid sources contain the appropriate data, if addresses are incremented by one every fast clock cycle. In odd transparent passes, still only the mentioned registers are available for writing – so the address generator must ensure that those registers contain "transparent" data. As now data at even addresses act as transparent, from every two-word block the odd must be read first, followed by the even. This means that read address series such as 1-0-3-2-5-4 are correct. This is why two read address registers are present, one working at rising edge, and the other at the falling edge of the system clock. These addresses are multiplexed according to the system clock state to generate the final read address. Table 1 lists the initial addresses, the load time (delay in number of fast clocks) and the increment value in the different modes (opaque/transparent even/transparent odd).

The Load Logic synchronizes the cell with the Input Processing Unit, the LOAD_DL delays this load signal for registers loading some fast clocks later.

**Table 1. Address values**

| | Init | Increment | Load time |
|---|---|---|---|
| **Read 0** | 0/0/1 | 4/2/2 | 0/0/0 |
| **Read 1** | 2/1/0 | 4/2/2 | 1/1/1 |
| **Write** | 0/0/-1 | 2/1/1 | 3/3/3 |

# 5. Conclusion

The presented units fulfill our preliminary requirements. External bandwidth requirement is decreased thanks to the on-chip buffers. The parallel memories allow very fast z- and stencil buffering, thus the architecture can greatly cope with shadow computing and complex scenes. While the handling of transparent object may be faster when they are pre-sorted, the presented multi pass algorithm lowers CPU requirements, and allows correct, per-pixel transparency effects to be achieved. As typical embedded systems still use SDRAM memory, the target clock speed of 66 MHz fits into this segment. Resolution in such devices hardly reaches 640x480, so with 100.000 triangles on screen, the HSR unit can achieve more than 40 frames/second average.

Further improvements can be made by utilizing the programmable tile size and change it adaptively; by analyzing the rendered images an optimal tile size can be derived based on the average number of pixels covered by a triangle in a tile, and on the average number of tiles affected by a triangle.

Another possibility for improvement is to join two triangles which are on the same plane and process them as a single quad in the HSR Unit, effectively doubling the ratio of covered and non-covered number of pixels in a tile.

Table 2 summarizes the required logic resources for the presented units (number of 4 input LUTs, flip-flops and BlockRAM memory blocks).

**Table 2. Logic Resources**

| | LUT | FF | BRAM |
|---|---|---|---|
| **Depth Unit** | 330 | 76 | 2 |
| **Stencil Unit** | 210 | 36 | |
| **Covering Unit** | 355 | 123 | |
| **Address Gen.** | 47 | 111 | |

# 6. References


[1] A. Watt, *3D Computer Graphics*, Addison-Wesley, 2000.

[2] P. Diefenbach, *Pipeline Rendering: Interaction and Realism Through Hardware-Based Multi-Pass Rendering*, Ph.D. Thesis, University of Pennsylvania, 1996.

[3] Kilgard, M. J., Everitt C. *Optimized Stencil Shadow Volumes.* Game Developer Conference, 2003.

[4] Y. Wang, S. Molnar, *Second-Depth Shadow Mapping.* Technical Report TR94-019, 1994.