The AIDA Display Processor System Architecture

S. R. Evans, R. L. Grimsdale, P. F. Lister and A. D. Nimmo

ABSTRACT This paper describes the Advanced Image Display Architecture, AIDA. The primary aims were to design a graphics display subsystem capable of satisfying the needs of both high performance workstations and vehicle simulator visual systems. AIDA can accept planar triangle primitives which have been transformed, clipped and projected by preceding stages. The system implements many desirable features including modularity, anti-aliasing, translucency, pixel-rate hidden surface removal and Gouraud shading. AIDA has been designed to take advantage of ASIC technology in the implementation of its processing units.

1 Introduction

The realization of a flexible Display Processor Architecture, capable of meeting the requirements of radically different applications, requires the use of modular subsystems. Application areas that the VLSI and Computer Graphics Research Group have targeted include high-performance 3D workstations and vehicle simulator visual systems. This paper will introduce AIDA, developed as part of the ALVEY PRISM project, in collaboration with Link-Miles Ltd. and GEC Hirst Research Centre, and initially designed to be used in a graphics systems with the MAGIC I [1] or MAGIC II [4] processors performing the required geometry operations. The algorithms used, the system architecture, its components and configuration details, and an analysis of the expected performance will be discussed. Several similar systems exist [3], but AIDA is considered to be more flexible in terms of configurability. Figure 1 represents the AIDA graphics display subsystems.

AIDA accepts triangle vertex data, face data and gradient information. Subsequent processing steps produce an image with several desirable attributes. These include anti-aliasing, Gouraud shading and translucency. The conceptual display pipeline is used to express the decomposition of primitive objects into spans, then into rendered, depth sorted pixels, without indicating the amount of data produced by each step. This depends primarily, on the characteristics of the database and the effect of preprocessing stages. Section 4 attempts to predict the results obtained from different database types.

2 Algorithms

The only primitive currently accepted by AIDA is the planar triangle—the decomposition of a database into triangular elements is assumed to be a relatively trivial task. This primitive was chosen to simplify the span generation process—the planar triangle is always convex, can be described using a small amount of data and can be characterised

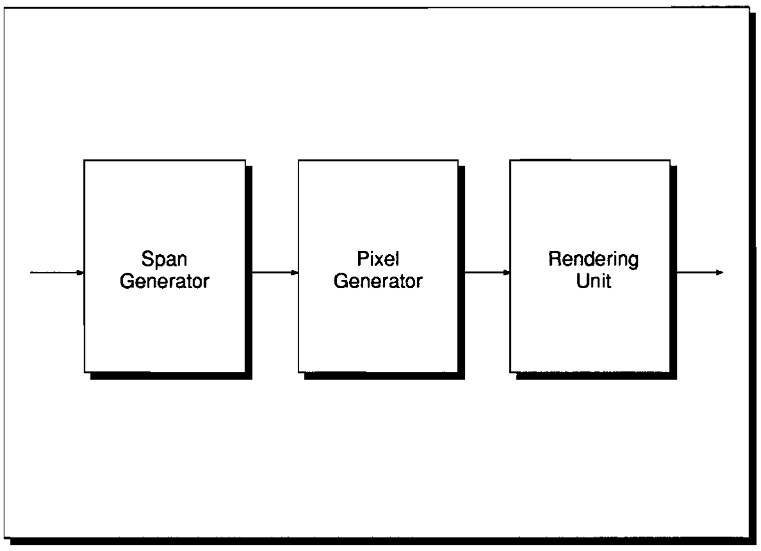

Fig. 1. The Conceptual Display Pipeline

simply. However, because the Pixel Generator and Rendering Unit only rely on span and pixel primitives, alternative Span Generators operating on different primitives—line, polygons, parametric surfaces—can be readily employed. Coherency is exploited by using interpolation methods in both the Span Generator and the Pixel Generator.

2.1 The Span Generator

The Span Generator is the first processing element of AIDA, generating spans from the display list of triangle data. Triangle primitives are defined for a virtual screen of 4096 × 4096 elements, for viewing on a display screen of 1024 × 1024 pixels. A span therefore consists of four sub-scan lines and the start and end x coordinates, which, together with bounding information—sub-scan line coverage—provides enough information for anti-aliasing to be implemented in subsequent processing steps. One assumption is made—the triangles to be processed must have the top vertex (smallest y coordinate) passed first, and if two vertices share the smallest y coordinate the one with the smallest x coordinate is to be passed first. The vertices are then passed to the Span Generator in clockwise order. In order to generate spans in a largely autonomous manner, a processing step is employed to ascertain the relative positions of the triangle vertices. This process is known as *triangle typing*. There are four types of triangle vertex orientations that AIDA must distinguish between, although triangle type 0 defines two effectively equivalent orientations. The typing parameters, s_0, s_1 and s_2, are given in Equations 1, 2 and 3.

$$s_0 = y_{v_1} - y_{v_0} \tag{1}$$

$$s_1 = y_{v_2} - y_{v_0} \tag{2}$$

$$s_2 = y_{v_1} - y_{v_2} \tag{3}$$

Type qualifications are allocated using the simple magnitude comparisons,

```
if ($s_0 == 0$)||($s_2 == 0$)
  $type = 0$
else if ($s_0 < s_1$)
  $type = 1$
else if ($s_0 > s_1$)
  $type = 2$
```

Together, the typing parameters and the type qualification fully describe a planar triangle for the span generation process. The triangle orientations are shown in Figure 2 and gives the relevant typing parameters. A Bounding Mask is used to describe the span sub-scan line coverage, and is calculated using the current y coordinate and the two closest vertices, information which is type dependent. It is also used to determine the allocation of x coordinate interpolators to sub-scan lines to avoid reverse interpolation when a span does not start on the first sub-scan line.

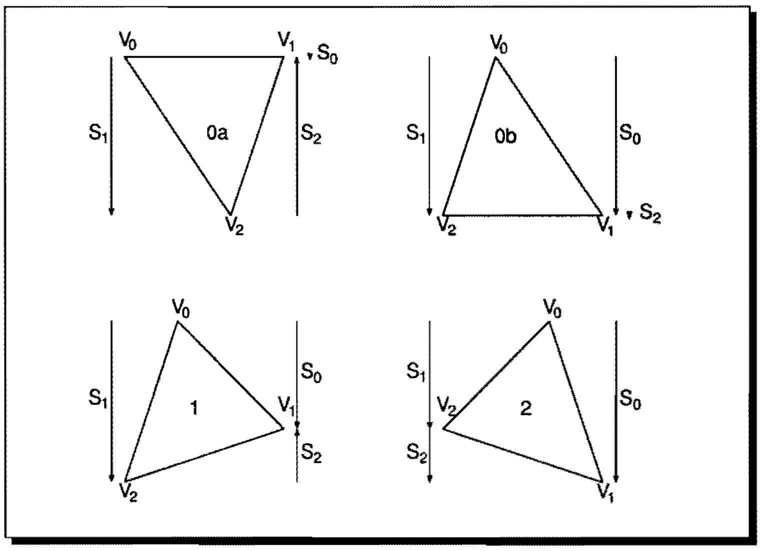

Fig. 2. Triangle Types

The development of a suitable algorithm for the Span Generator uses triangle typing and span coverage information, together with counters and comparators, to simplify and aid the production of similar routines for different triangle configurations. An overview of the algorithm is given.

```
per triangle {
    read vertex data, attribute data, write attribute data to face memory
    calculate data $s_0$, $s_1$, $s_2$, type triangle
    $y_c = y_{v_0}$
    if ($type == 0$) {
        calculate bound mask ($y_c, v_0, v_2$), allocate interpolators, write span
        while ($y_c$ != $y_{v_2}$) {
                $y_c$++, interpolate
                calculate bound mask ($y_c, v_0, v_2$), write span
        }
    }
    else if ($type == 1$) {
        calculate bound mask ($y_c, v_0, v_1$), allocate interpolators, write span
        while ($y_c$ != $y_{v_1}$) {
                $y_c$++, interpolate
                calculate bound mask ($y_c, v_0, v_1$), write span
        }
        calculate bound mask ($y_c, v_1, v_2$), allocate interpolators, write span
        while ($y_c$ != $y_{v_2}$) {
                $y_c$++, interpolate
                calculate bound mask ($y_c, v_1, v_2$), write span
        }
    }
    else { /* $type == 2$ */
        calculate bound mask ($y_c, v_0, v_2$), allocate interpolators, write span
        while ($y_c$ != $y_{v_2}$) {
                $y_c$++, interpolate
                calculate bound mask ($y_c, v_0, v_2$), write span
        }
        calculate bound mask ($y_c, v_2, v_1$), allocate interpolators, write span
        while ($y_c$ != $y_{v_1}$) {
                $y_c$++, interpolate
                calculate bound mask ($y_c, v_2, v_1$), write span
        }
    }
}
```

2.2 The Pixel Generator

The Pixel Generator follows the Span Generator and has three main functions:

- Interpolation in x for Gouraud shading.
- Generation of the edge mask [2], required to perform antialiasing.
- Depth sorting all the surfaces at each pixel for hidden surface removal.

An outline of the Pixel Generator operations is given,

```
per pixel {
    if (span(s) cover(s) pixel) {
      for (all spans previously read and stored) {
          read span data
          calculate intensity, proximity and edge mask
          depth sort
          keep span if contribution to next pixel
      }
      for (all spans that start at current pixel) {
          read span data
          calculate edge mask
          depth sort
          keep span contribution to next pixel
      }
      while (pixel contributing surfaces exist, until maximum) {
            output pixel contributions
      }
    }
    else {
      output background surface
    }
}
```

The AIDA depth sort algorithm was developed to resolve two problems:

- Surfaces at different distances from the viewing position.
- Coplanar surfaces.

Surfaces with less depth (greater proximity) are placed in front of surfaces with greater depth (less proximity). Coplanar surfaces present an interesting dilemma. Take the example of superimposing one surface on another—typically used for road markings, where texturing techniques may be inappropriate. Each surface will have the same proximity value, making it impossible to predict the visible surface. A label, termed surface i. d. and allocated during the design of a database, comprising a surface identification tag and a priority word tag is used. Identical surface identification tags are given to primitives with the same parent object. The database designer can then choose a priority value for superimposed markings and details depending on the desired effect.

2.3 The Rendering Unit

Pixel rendering functionality is provided by the AIDA Rendering Unit. Situated between the Pixel Generator and the frame buffer, it takes depth sorted pixel contributions and outputs rendered RGB pixel data. Desirable features of the design include antialiasing and translucency. An algorithm for the Rendering Unit is given,

```
per pixel {
    set pixel colour zero
    per element {
        read element
    }
    per sub-pixel {
        find depth mask
    }
    do {
        find next unused depth mask
        mark all unused depth masks as used
        find number of depth mask
        calculate initial translucency factor
        for (every surface covering sub-pixel in depth order) {
            accumulate colour
            produce translucency factor
        }
    } while (not all subpixels used)
}
```

The Rendering Unit algorithm takes in a pixel packet, finding which sub-pixels are covered by the same pixel contributions, which will have the same final colour. The colour of each different coloured sub-pixel is calculated then multiplied by the percentage of sub-pixels with that colour. The values for all the different coloured sub-pixels are then added together to give the final pixel colour, Equation 4, where m is number of covering pixel contributions, n is number of differing depth masks, tf is the translucency factor, tl is the translucency level and w is the weighting factor.

$$c_{out} = \frac{\sum_{j=0}^{n} \sum_{i=0}^{m} c_i \times I_i \times (1 - tl_i) \times tf_i}{16} \quad (4)$$

$$tf_{i+1} = tf_i \times tl_i \quad (5)$$

$$tf_0 = w \quad (6)$$

To find which sub-pixels are covered by the same pixel contributions, the bit corresponding to a given sub-pixel in the edge mask is stored. Each sub-pixel has a word, or depth mask, in which the n^{th} bit represents the coverage by the n^{th} pixel contribution in the packet. Comparisons of the depth masks will show which sub-pixels produce the same colour. By resetting the bits in any edge mask covered by an opaque pixel contribution, the number of differing sub-pixels can be reduced, without affecting the final result. This significantly decreases the number of calculations needed to produce the final pixel colour.

3 System Description

The AIDA architecture exhibits an hierarchical structure, taking advantage of parallel processing and exploiting coherency. System performance can be optimised to meet a wide variety of applications by varying the number and configuration of processing elements.

3.1 The Span Generator

A complete Span Generator is implemented using two instances of the Span Generator ASIC—they are synchronised to track and interpolate the values at the left and right edges of a triangle, Figure 3.

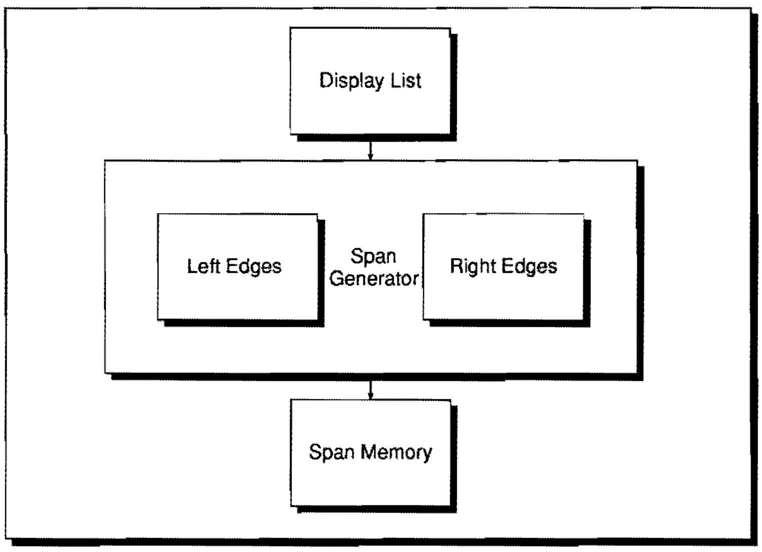

Fig. 3. The Span Generator System Architecture

Figure 4 shows the internal ASIC architecture. Interpolation of x values, proximity and intensity occur in parallel. The initial values for x are cascaded from the preceding interpolator, since each x interpolator deals with only a sub-scan line. The interpolated results are buffered in an output memory block, which is sent to the output block when a complete span is present—when the bottom bit of the bounding mask is set, or the bottom of the triangle has been reached. This information is obtained from the span coverage, using the Bounding Mask generator output present at the current scan-line.

3.2 The Span Memory

The Span Memory is the first stage of the pixel generation process, sorting all the spans, per frame or region, in XY order. The sort process creates, for each pixel, a null terminated linked list of spans which start at the referenced pixels. This method of span sorting simplifies Pixel Generator access to spans.

3.3 The Pixel Generator

Span coherency within the Pixel Generator, Figure 5, implies that only complete scan lines can be allocated per Pixel Generator.

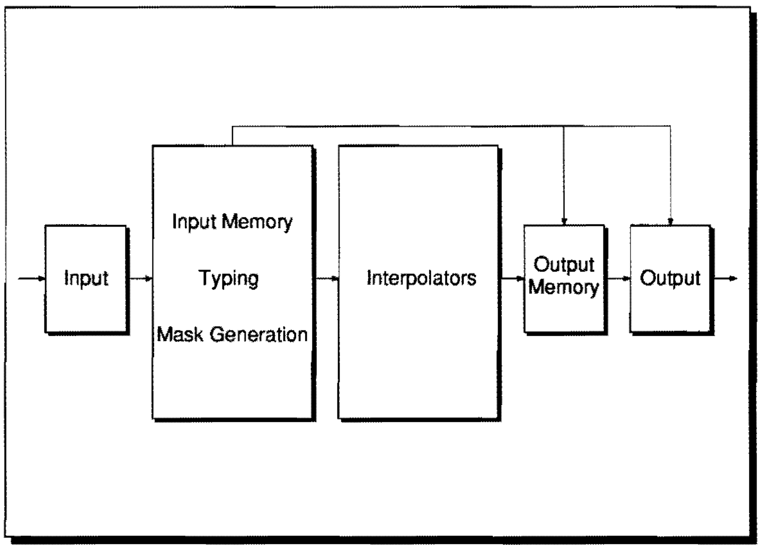

Fig. 4. The Span Generator ASIC Architecture

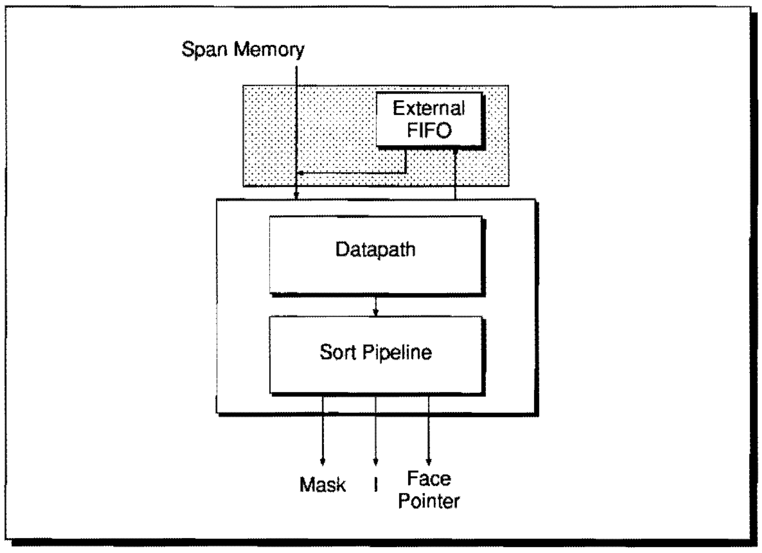

Fig. 5. The Pixel Generator ASIC Architecture

The Pixel Generator creates pixel packets, derived from all the spans covering a given pixel, and consists of a depth sorted list of pixel contributions. Two types of span, input through a 128-bit bus, are recognised:

- A *new span*, which starts on the current pixel.
- An *active span*, one which has previously started.

The essential difference between the two types of span is their source. New spans come directly from span memory and are input after all current active spans have been processed. Active spans are stored in an external FIFO instance—on-chip storage of every active span may not be possible because of the RAM size required.

The Datapath, Figure 6 generates the data packet of unsorted pixel contributions used by the Sort Pipeline. For a given pixel, the Sort Pipeline sorts the data in ascending depth order. The pipeline comprises n autonomous modules, where $(n-1)$ is the maximum number of resolved translucent surfaces. Data at the input of each module is either routed to the output or exchanged with the data present in the module.

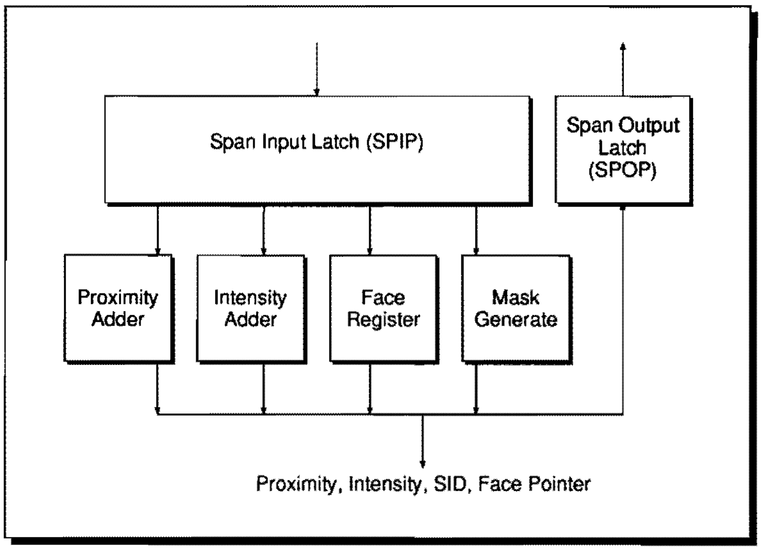

Fig. 6. The Pixel Generator Datapath

3.4 The Rendering Unit

The Rendering Unit ASIC Architecture, Figure 7, presents a hardware implementation of the Rendering Unit algorithm.

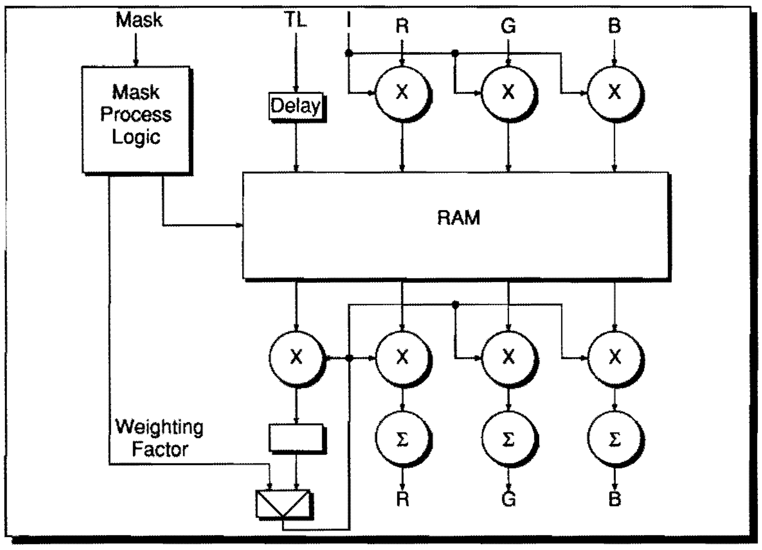

Fig. 7. The Rendering Unit ASIC Architecture

4 Performance

4.1 Introduction

It is widely recognised that accurate performance results for one system often have little real value for comparisons with other systems. More suitable performance indicators are obtained by using a variety of different data, to reflect different applications. The results presented here use several AIDA system configurations, to give an indication of the predicted peak performance.

4.2 System Configuration and Triangle Throughput

We have given three example configurations, Figures 8, 9 and 10, together with peak performance rates and memory requirements. These configurations represent typical connection schemes. System Configuration A is used as a reference, Figure 8, with peak throughput and memory requirements given in Tables 1 and 2. System Configurations B and C, Figures 9 and 10, show more complex, higher performance configurations which would typically be used in workstation and vehicle simulator display systems, together with associated throughput and memory requirements.

Throughput		
Δ/sec		No. of
SG	PG	ASICs
1M	**100K**	4

Table 1.

Memory Requirements			
		Sub Regions	
Frame rate	Δ/frame	1	64
1Hz	100K	32M (21M)	19M (328K)
5Hz	20K	6.4M (4.2K)	3.7M (64K)
30Hz	3.3K	1M (700K)	600K (5K)

Table 2.

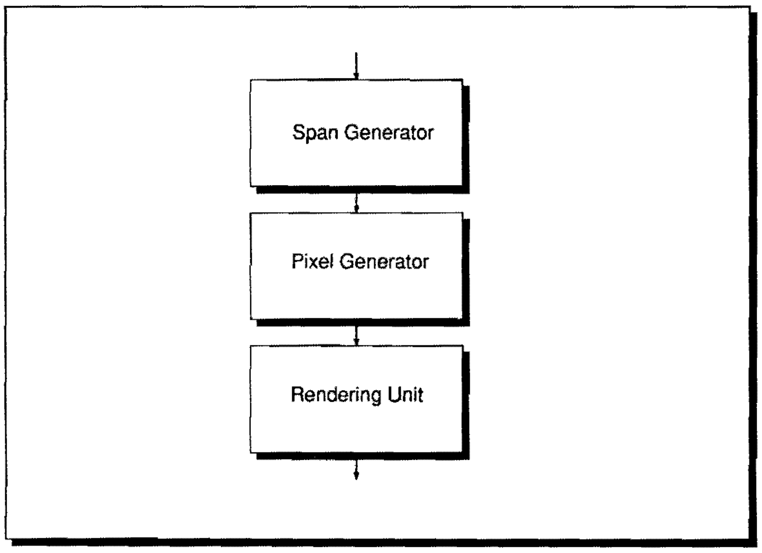

Fig. 8. System Configuration A

Throughput		
Δ/sec		No. of
SG	PG	ASICs
1M	**400K**	10

Table 3.

Memory Requirements			
		Sub Regions	
Frame rate	Δ/frame	1	64
1Hz	400K	130M (84M)	74M (1.3M)
5Hz	80K	26MB (17M)	15M (260K)
30Hz	13.2K	4.3M (2.8M)	2.4M (43K)

Table 4.

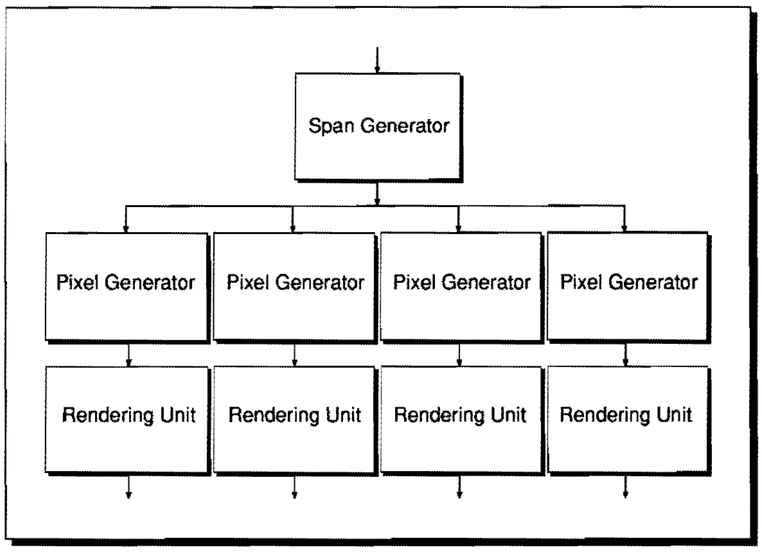

Fig. 9. System Configuration B

Throughput		
Δ/sec		No. of
SG	PG	ASICs
4M	6.8M	72

Table 5.

Memory Requirements			
		Sub Regions	
Frame rate	Δ/frame	1	64
1Hz	4M	1.3GB	741.1MB
5Hz	800K	260MB	148.22MB
30Hz	133K	43MB	24.7MB

Table 6.

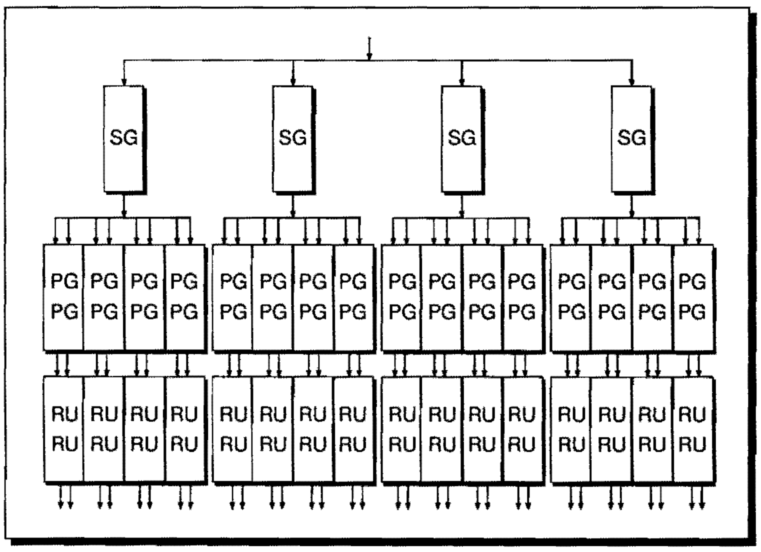

Fig. 10. System Configuration C

5 Conclusion

We have presented a modular graphics display subsystem, capable of high throughput and different system configurations, for the production of high-quality images. These designs have been developed using techniques presented in [5], and are being implemented in silicon using the GENESIL[1] silicon compiler.

Acknowledgements

The work presented here has been funded under the UK ALVEY programme, and performed in collaboration with Link-Miles Ltd. and GEC Hirst Research Centre. We would like to thank Alan Cunliffe and Daljit Bimrah of Link-Miles Ltd. for their contributions, and all the members of the VLSI and Computer Graphics Research Group, Graham Dunnet, Marie-Pierre Hébert, Michael D. J. McNeill, Bina Shah and Martin White.

References

[1] M. Agate, H.R. Finch, A.A. Garel, R.L. Grimsdale, P.F. Lister and A.C. Simmons.: A multiple application graphics integrated circuit—MAGIC. In A.A.G. Requicha, editor, *EUROGRAPHICS'86*, pages 67–77, 1986.

[2] L. Carpenter.: The A-buffer, an antialiased hidden surface method. In H. Christiansen, editor, *Computer Graphics—SIGGRAPH Conference Proceedings*, pages 103–108, 1984.

[3] M. Deering, S. Winner, B. Schediwy, C. Duffy and N. Hunt.: The triangle processor and normal vector shader: A VLSI system for high performance graphics. In *Computer Graphics—SIGGRAPH Conference Proceedings*, pages 21–30, 1988.

[4] H.R. Finch, M. Agate, A.A. Garel, P.F. Lister and R.L. Grimsdale.: A multiple application graphics integrated circuit—MAGIC II. In A.A.M. Kuijk and W. Strasser, editors, *Advances in Computer Graphics Hardware II*, pages 81–92. Springer-Verlag Berlin Heidelberg New York, 1987.

[5] A.D. Nimmo, P.F. Lister and R.L. Grimsdale.: A VLSI strategy for graphics. In A.A.M. Kuijk, editor, *Advances in Computer Graphics Hardware III*. Springer-Verlag Berlin Heidelberg New York, forthcoming.

[1]GENESIL is a trademark of Mentor Graphics Silicon Design Division