10 Testing Geometric Primitive Shaders

G. J. Dunnett
M. White
P. F. Lister
R. L. Grimsdale

ABSTRACT We present a design and test strategy for Geometric Primitive Shaders—integrated circuits which perform rasterisation of primitives such as vectors and triangles. The design strategy proceeds through various levels of detail, and we describe the need for testing as the design advances. A suitable set of test are given for a typical shader. Our experiences in applying the strategy to a real device are discussed, together with the tests which we devised, and practical compromises which we had to make.

10.1 Introduction

Advances in VLSI technology have made it possible to design complex chips to perform a wide range of graphics functions, and off-load image generation from host processors. As the functionality of these chips becomes ever greater, the importance of testing designs thoroughly before they are cast in silicon increases. At each stage of the design process, testing a model against a known standard helps ensure that these chips perform exactly as they were intended. This is particularly important when the evolution of a design is carried out by a geographically distributed design team who may have different understandings of how the design achieves its aim [2]. Previous papers have concentrated on device testability or design testing, for example [9]. Here we consider functional testing, and the validation of evolving designs. Nimmo [11] has described functional testing before.

In Section 10.2 we describe the functionality of a range of Geometric Primitive Shaders and distill from them, and our predictions of future devices, a list of generic functions which they perform. From this we present a typical Geometric Primitive Shader, and describe how we would construct a test strategy for it. Later, in Section 10.3, we outline how we applied the test strategy to a real shader—the SPIRIT-I ASIC Drawing Engine [7], designed as part of the SPIRIT Workstation graphics subsystem [16]. We describe in some detail the tests which we had to develop for this device.

10.2 Development of a Geometric Primitive Shader

During the last decade we have seen memory prices fall, and processor clock speeds increase remarkably. These two factors have allowed high resolution graphics on workstations to be supported. High resolution screens of over one million pixels, and 16 million displayable colours are now common [8]. To drive ever larger displays, however, takes vast amounts of processing power. Manufacturers of graphics oriented workstations are

increasingly relying on dedicated graphics subsystems with specialised hardware to off-load graphics processing tasks from the host processor in the same way that I/O tasks are now relegated to I/O processors.

Graphics users can now expect workstations to perform high speed shading of complex objects—chemists want to visualise molecular models, designers want to visualise car bodies, buildings, mechanical components, etc. Fast response rates to a request to render and display an object are required, and the combination of processing power, silicon density, and small size make custom VLSI devices a good solution. Typically, designs make use of highly pipelined architectures with wide data paths to calculate one pixel value per clock cycle [13]. This offers considerable advantages over conventional microprocessors which can at best calculate pixel values in tens of cycles, and do not offer optimised data types. Specialised VLSI hardware is now available which can support directly the generation of Gouraud shaded objects in near real time.

10.2.1 Overview of Current Geometric Primitive Shaders

Many authors have described custom chip designs suitable for inclusion in dedicated subsystems of high-performance workstations. Each chip appears to have different features reflecting the diverse applications for which they have been designed. Recent Geometric Primitive Shaders reported on include:

- Triangle Processor and Normal Vector Shader [3]
- Multipurpose Hardware Shader [13]
- Triangle Shading Engine [1]
- PROOF [15]
- PRISM [4]
- and many more.

In the table below we have summarised the features of some of these designs for the purpose of comparison.

		Proof	TP	TSE	MPHWS	AIDA
Shading	Gouraud	•		•	•	•
	Phong		•			
Windowing						
Parallelism	Object	•	•			
	Span					•
	Pixel			•	•	•
Vectors						
Antialiasing	Subpixel Mask	•				•
	Post filter		•			

where

TP is the Triangle Processor

TSE is the Triangle Shading Engine

MPHSW is the Multipurpose Hardware Shader and

AIDA is the Prism scan-conversion chipset.

10.2.2 Additional Geometric Primitive Shader Functionalities

As technology advances, we find that more functionality can be included on a given size of silicon, and that chip pin counts are increasing. These factors suggest that future Geometric Primitive Shaders will have additional features, and below we describe some desirable functions which we can expect to see.

10.2.2.1 Windowing

Windowing interfaces such as the X-Window system are becoming increasingly common, and we expect virtually all workstations will have windowing of one form or another within the next few years. It seems increasing likely, therefore, that Geometric Primitive Shaders will incorporate window management functions, typically clipping of pixels to rectangular regions.

10.2.2.2 Anti-aliasing

Computer generated images commonly suffer from "jaggies"—the well-known spatial aliasing effect. These can be distracting and should be eliminated wherever possible. Anti-aliasing techniques are well known and widely documented [5]. It is not unreasonable to expect that Geometric Primitive Shaders will perform anti-aliasing or provide support for it in the near future.

10.2.3 Proposal for a Geometric Primitive Shader

We believe that the following is a reasonable list of functionalities that should be provided by future specialised hardware capable of scan converting triangles and vectors.

- Interpolation of parameters such as R, G, B, Z, shading normals, texture coordinates, etc.
- Windowing
- Antialiasing

In Figure 10.1 we present a conceptual functional design of a Geometric Primitive Shader which we may wish to implement in hardware using VLSI technology. The functionalities of the units shown in the figure are as follows:

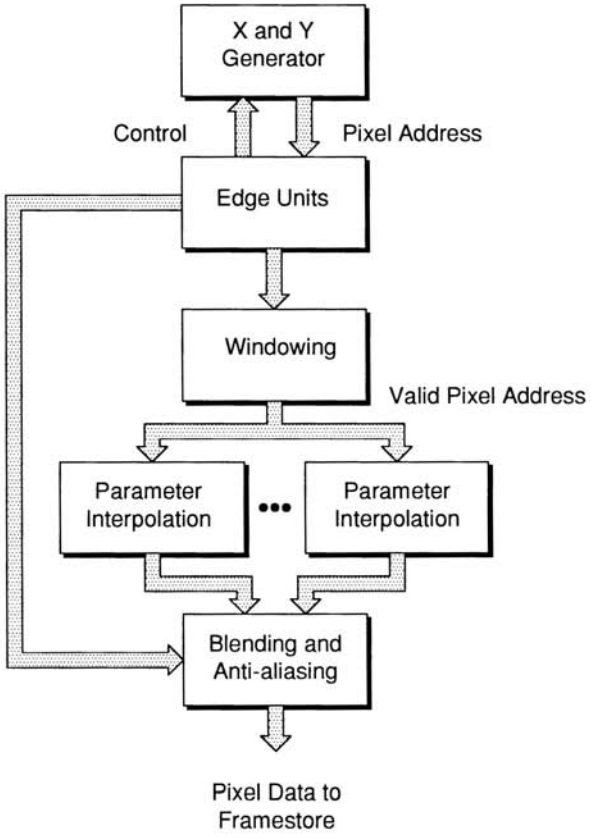

FIGURE 10.1. A Geometric Primitive Shader

X and Y Generator This block generates pixel addresses that lie within the primitive being rendered. Control information fed back from the edge units determine in which direction the next pixel lies in.

Edge Units These units determine when an edge of a primitive has been reached, and allow the XY generator to decide where the next pixel is found. Additionally, this unit calculates the fraction of the current pixel covered by the primitive to assist in the filtering of edge pixels.

Windowing Unit Pixels lying outside of the current window region are culled by this unit.

Parameter Interpolation Units From the pixel coordinates fed from the previous block, these units calculate pixel parameters such as R, G, B, and Z, etc. and where

necessary clamp outputs to the maximum/minimum values permitted.

Blending and Anti-Aliasing Unit Using all edge and colour data available, this unit calculates the final pixel value for sending to the frame store.

10.2.4 A Design and Test Strategy for the Shader

The design and implementation of the shader described in Section 10.2.3 is a complex task, but is made simpler if we adopt a design strategy which proceeds in a logical, incremental manner and which allows testing to be performed at each step. The ability to detect design errors at an early stage helps deadlines to be achieved, and makes errors less likely in the later stages. Also, if we acknowledge the comparative ease in which changes can be made to a software prototype, compared to the redesign of a hardware device, it makes sense to invest the time to create the software model and test it fully [11]. In Figure 10.2 we illustrate such a design and test strategy for the Geometric Primitive Shader. The main points are listed here:

1. The behavioural description of each unit is specified mathematically, e.g. linear edge functions, linear interpolation functions, etc.

2. This description is implemented in a high level language as an algorithmic behavioural model.

3. Test stimuli are generated automatically to drive the algorithmic behavioural model and used to refine this model until it is accepted. The pixel output from this model is now defined as the known correct pixel output—this produces acceptable 3D images.

4. A logic design is formalised and the behavioural description is evolved into a functional description or model that more closely resembles this design.

5. The same test stimuli are used to drive the functional model and the pixel output from the functional model is compared with the known correct pixel output. The functional model is refined until its pixel output matches the known correct pixel output.

6. The logic design and functional model is used as a specification for the chip design. This design is performed using a silicon compiler, VHDL, etc. A structural model is developed which represents the design using the technology selected.

7. The test stimuli are used to drive the structural model developed by the silicon compiler, etc. The pixel output from the structural model is compared with the known correct pixel output and the structural model refined until the tests are successful.

8. Validation tools provided with the silicon compiler, etc. are used to check that the design conforms to design rules. This *design testing* verifies data paths and gate level logic.

9. At this stage when all three model pixel outputs are the same, i.e. when functional and design testing are complete, we can commit to silicon with a high degree of confidence.

10. The test stimuli can then be used to drive the actual chip and the results compared with the known correct pixel output to validate the silicon implementation.

To guard against the introduction of errors between stages it is important to thoroughly test each model against provably correct results, and this means results obtained from the mathematical description of the functional units. Provided the mapping from the mathematical description to the algorithmic behavioural model is correct—and this is subject to the algorithmic behavioural model producing acceptable images—the pixel output from the algorithmic behavioural model can be used as the known correct values. This is desirable because the data provided by the algorithmic behavioural model will be in machine readable form.

10.2.5 Test Parameters for the Geometric Primitive Shader

Each of the functional units of the Geometric Primitive Shader must be tested at each of the test points in the design strategy. Ideally we would like to test each unit in isolation, however, this will not be possible for the final design due to the pipelined nature of the chip. Instead we recognise the fact that each test may exercise other units in the design and attempt to minimise any unwanted interaction. This means holding parameters constant and varying them one at a time. Below we list the tests which we need to perform in order to validate each separate design. We call each testable function a test parameter.

- Test that the XY generator does not attempt to generate pixels outside of the physical screen range.
- Test that the XY generator can scan convert small (pixel-sized) and large (screen-sized) primitives.
- Test that the XY generator can scan convert vectors independent of octant.
- Test that the Edge units make correct decisions about when a triangle primitive has been left. This in fact requires the scan-conversion of many primitives at many orientations.
- Test that the Edge units generate correct coverage information. This requires the scan-conversion of primitives with edges at many different slopes.
- Test that the Windowing unit clips pixels lying in each of the 8 rectangular regions formed by the window.
- Test each Parameter Interpolation unit for correct interpolation.
- Test each Parameter Interpolation unit clamps its output to the maximum or minimum permitted value when necessary.

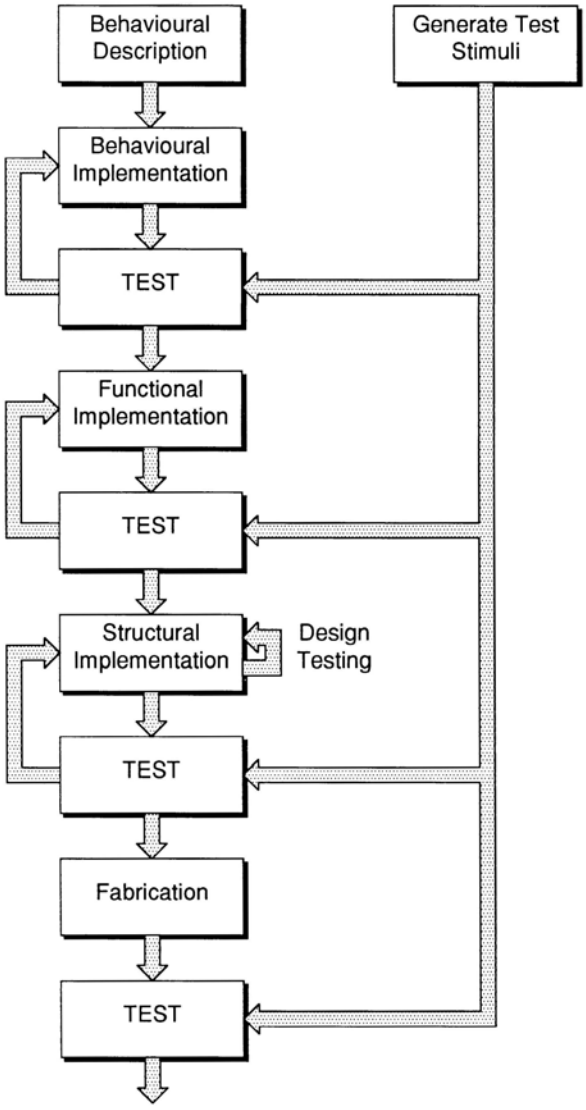

FIGURE 10.2. Design and Test Strategy

10.3 Application to the SPIRIT-I ASIC Drawing Engine

We now describe how we applied the design and testing strategy outlined above in a real situation, as we designed and tested a drawing engine with other members of the SPIRIT WORKSTATION CONSORTIUM.

10.3.1 Overview of the ASIC Drawing Engine

The ASIC Drawing Engine (ADE) is a custom VLSI device used as a pixel address generator, and can provide bilinearly-interpolated colour, depth and anti-aliasing information for each pixel. It performs Gouraud shading , and can assist Phong shading. The ADE scan-converts triangle and vector primitives using the Pineda [12] and Bresenham [5] algorithms respectively.

In addition to Gouraud shading, wireframe and hiddenline triangles can be generated. All primitives can be generated with anti-aliasing information, with a new pixel output available at every clock cycle. To achieve this, the internal architecture has been extensively pipelined.

10.3.2 Hardware Architecture

The hardware structure of the ADE can be broken down into the functional units [10], illustrated in Figure 10.3. These are:

1. A PLA state-machine which acts as a controller for the entire device.
2. Data loading unit and input registers.
3. Pixel address computation and depth interpolation units.
4. Colour computation units.
5. Distance to the ideal line computation units.
6. Colour and depth formatters.
7. Anti-aliasing information formatter using a subpixel bitmask memory.
8. Output formatter for the depth and colour values.

The ADE is designed around 9 bilinear incremental interpolators operated in parallel. These incrementers are loaded by a host processor through input registers which permit the incrementers to scan-convert one primitive whilst the setup information for the next is being written into the registers. At each clock cycle the incrementers can be made to increment or decrement in X, or increment in Y. Two of the incrementers are used to calculate the current X and Y values (pixel address).

The scan-conversion is controlled by a PLA state-machine which receives inputs from three edge units (three incrementers). For each pixel These calculate the distance of the pixel centre to each of the three edges of the triangle being displayed. The pixel centre

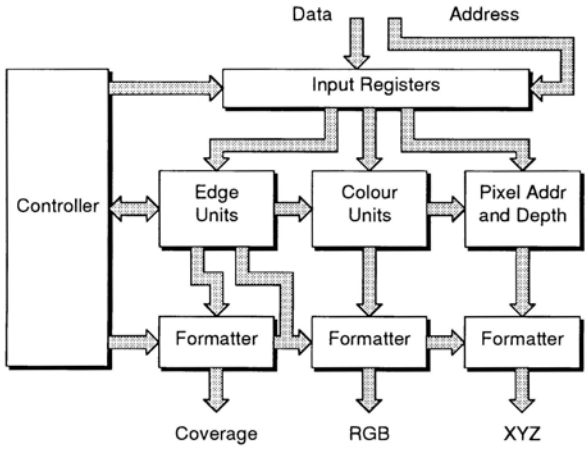

FIGURE 10.3. Architecture of the ADE

is inside the triangle if all three edge values are positive. Such a situation is known as thin geometry. These distance values, used in conjunction with the gradients of the edge functions in X and Y, and information from the windowing logic are used by the state-machine to determine which pixel should be visited next. The state-machine can *hunt* for the triangle from any start position above the triangle on the screen.

The remaining units are the three colour units (we use the RGB colour model), which calculate the colours used in Gouraud shading, and a depth unit which calculates the Z component for a hardware assisted Z-buffer test (which is not performed by this ASIC).

Anti-aliasing makes the operation of the edge units more complicated. Pixels which do not lie wholly with-in the triangle can be generated with a coverage value calculated using a lookup table [14]. This value is proportional to the fraction of the pixel actually covered by the triangle. To consider these extra pixels, the edge unit tests are now made against a small negative number rather than zero. This has the effect of slightly increasing the area of the triangle—a situation which we call greasy geometry. Pixels which only touch the triangle boundary can be generated in this way. The decision to include a pixel as being on the triangle boundary is particularly important when anti-aliased hidden-line triangles are being generated. To help this decision, edges are classified in 5 ways, *clipped, pseudo, virtual, real* or *silhouette*. Each edge unit can operate independently from the others, and process boundary pixels according to the status of the edge assigned to it.

10.3.3 The Design and Test Strategy for the ASIC Drawing Engine

At an early stage in the design of the ADE,a test vehicle known as the *Framework* was developed [6]. The Framework covered the entire graphics pipeline from modeling to the output of rendered pixels. The Framework was written in a high-level language using approximately 20,000 lines of code. The behavioural Framework allowed various algorithms

to be implemented and evaluated for hardware suitability. After algorithms were selected the logic design was started, and the rendering and scan-conversion routines of the Framework replaced by a functional model. The testing of this model against the behavioural model was not performed at this stage. The final functional code—the *asic* code—was then supplied to a foundry as a detailed specification for the drawing engine. The foundry used the Genesil silicon compiler for their implementation. We considered it most important that the Genesil design produced by the founder was thoroughly tested against the functional software before the chip was fabricated from the Genesil output. This was to guard against any misinterpretation of the specification.

The requirement to generate test data in a form suitable for supplying to the Genesil model meant that the input and output formats used by the Framework had to be extended. A register-level interface was implemented allowing register–data value pairs to be given, instead of the usual display list information such as vertex coordinates, colour, shading method etc. Similarly the output from the asic code was modified, allowing the values present on the output pins and buses to be captured, instead of pixel being plotted. Programs to translate to-and-from the Genesil format were written, allowing the output from both models to be compared. Figure 10.4 shows the interfaces between parts of the Framework, the asic code and the Genesil model.

10.3.4 Test Parameters for the Asic Drawing Engine

The test parameters used to test the SPIRIT-I drawing engine basically followed those given in Section 10.2.5, with additions to test features peculiar to our device. The additions were:

- Test that the asic can locate the triangle from any start position on the screen.
- Test that anti-aliasing information is correctly produced when the edge flags for each edge are combined in all possible ways.

Further information regarding the test parameters is given in the following section.

10.3.5 Description of Tests

Each test parameter gave rise to many separate tests to be performed. We describe the tests which we developed in more detail here.

10.3.5.1 Screen Position

Triangles are described to the ADE as 3 intersecting infinite lines, each splitting the plane into two half-planes. This makes it easy to specify triangles which do not lie in the screen, and may in fact partially overlap it. To test this we generated many triangles overlapping the screen. Triangles which intersect each of the 4 screen edges, pairs of edges, three edges at once, and all four edges were devised, for a total of 28 tests.

10.3.5.2 Geometry—Size and Orientation

The ADE edge units play a large role in the successful scan-conversion of triangles and vectors. Edges can be approached from both inside and outside of the triangle, and this

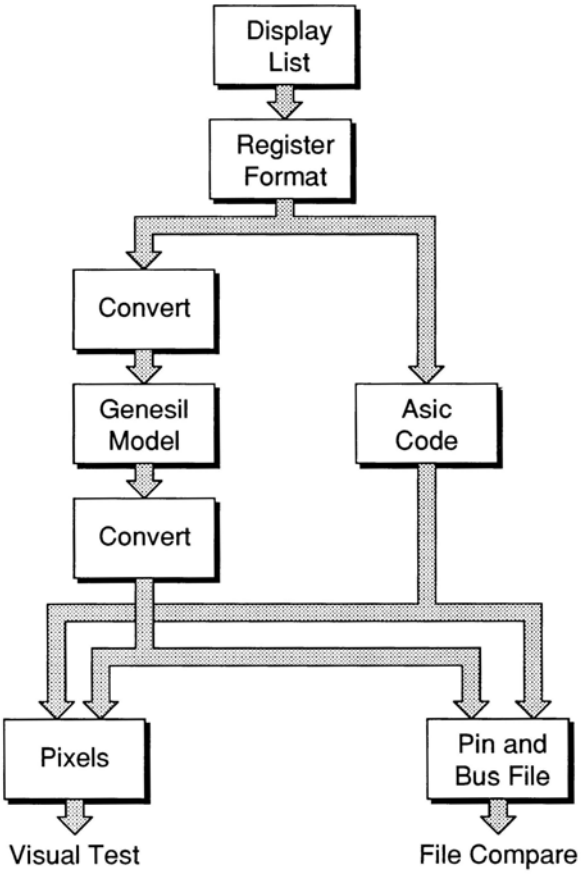

FIGURE 10.4. Interfaces Between Models

led us to identify 12 distinct geometric test cases. In order to test the scan conversion routine fully we rotated each of these 12 cases into each of the 8 octants and applied a small random perturbation to the vertices (intersection points). This forced the triangle to be non-aligned with the (integer) pixel grid. The tests were generated for both greasy and thin triangles. At this stage the anti-aliasing information produced was ignored. Of consideration was only whether the correct pixels were being located.

Vector primitives are scan-converted using two edge units. One unit describes the line on which the vector is located, and the second is used to count the number of pixels plotted. Scan-conversion finishes when this count reaches a previously calculated figure. The Bresenham algorithm is performed only in the first octant, and so we tested that the vector algorithm could perform as expected in each octant. Perturbing the end points, as for the triangle cases, exercised the Bresenham algorithm more fully.

The ADE cannot draw triangle primitives which are smaller than a pixel in size, unless the triangle is classed as greasy, enabling pixels around the primitive to be considered. Scaling the 12 geometric cases to be sub-pixel sized allowed us to test that the ADE could correctly scan-convert small primitives.

Four half-screen sized triangles were created, with vertices at each screen corner. These tested the scan-conversion of large triangle primitives. Scan converting 2 vectors placed diagonally across the screen tested the drawing of large vectors.

10.3.5.3 Anti-aliasing

As mentioned above, rotating a triangle causes its vertices to fall at non-integer screen locations. This causes pixels to be only partially covered by the triangle and thus require anti-aliasing. We took advantage of this to test the anti-aliasing information produced by the ADE. A "normal" and long-thin triangle were each used in these tests with a small rotation angle and vertices perturbed randomly by ± 1 pixel to generate many partially covered pixels. We kept the triangle size small—$\sim$ 9 pixels—to force most pixels to be partially covered. The treatment of anti-aliased, hiddenline triangle primitives is extended to allow border pixels to be considered as either inside the triangle, or part of the edge. Pixels which lie on the border can be anti-aliased or not, depending on the precise mode. Tests were created three times over, once for Gouraud, and twice for the two modes of hiddenline triangles.

The antialiasing of vectors was tested in a similar manner, with a short vector—$\sim$ 3 pixels long. This was rotated and perturbed to create many test cases. These tests were repeated twice, once with the ADE configured in one-pass mode, and the second in two-pass mode. The background colour was important for the one-pass tests, and so the tests were generated on a black, and on a white background to exercise both cases—background lighter, and background darker than the primitive.

10.3.5.4 Windowing

Testing the interaction between primitives and windows caused the largest number of tests to be developed. Two cases were important, a window clipping a primitive corner to corner, and a primitive crossing the edges of a window. These cases were treated separately. We identified 38 cases of the first type and 182 of the second. We generated thin and greasy

versions of each to determine the behaviour when additional pixels were included.

Thirty-five cases of a vector clipping a window were also discovered and tests generated for. We were particularly keen to test that the ADE could handle vectors which started just inside the window and ended just outside, since this had caused problems before.

10.3.5.5 Interpolation

To test the interpolation units, triangles and vectors were generated with random colour and depth information.

10.3.5.6 Overflow and Underflow

Pixels which are mathematically outside of a triangle, but which are included for anti-aliasing purposes can be generated with colour and depth information outside of the valid range. The ADE should identify these cases and clamp the output to the minimum or maximum value depending on the direction it has errored in. Tests were devised with greasy triangles with each of the red, green, blue and depth parameters in turn varying from minimum to maximum across the primitive. With one minimum and two maximum values at the vertices we created overflow test cases; With one maximum and two minimum values we created underflow tests. A total of 8 cases were devised.

10.3.5.7 Start Position

The ADE can start scan-converting a triangle from any screen position, since it can locate the triangle before it starts plotting pixels. By placing a bounding box around an arbitrary triangle, we defined 13 regions from which the scan-conversion could start. We defined a total of 26 test cases with all 13 start positions being tested for thin and greasy triangles. The start position was generated randomly in each of the 13 regions, allowing many tests to be produced easily.

10.3.5.8 Edge Status

There were 38 combinations of edge flags possible. These tests can be considered as an extension to the anti-aliasing tests, and in fact we re-used the geometric information used for those cases to test edge flag combinations. This led to some unnecessary duplication of tests. Testing the behaviour of the ADE at partially covered vertices with different edge flags on each edge was important. This led us to use small triangle with only 9 pixels. We generated tests with perturbed vertices to increase the number of tests.

10.3.6 Automatic Test Stimuli Generator Suite

To generate the test stimuli, we wrote a suite of programs which could produce all the tests from a particular section at once. This allowed the tests to be generated at each of the separate sites which would need them. Some of the tests were difficult to write, since they relied on particular data. The windowing cases were a good example. The data for these tests were created by hand, using a drawing package to produce windows and triangles. The files stored by the drawing package were then parsed to extract the geometric information, and this was translated into a form usable by the software. This

combination worked particularly well, since for the test documentation we would have had to have drawn the test cases anyway. To test that the translation from the drawing package to the test generators was correct, more code was written, this time to display the tests on our workstations. This allowed us to compare the cases drawn using the drawing package to what we saw on our screens. When the test stimuli were generated we could be reasonable sure that they were free of any errors. In the process of generating the test stimuli, we fed cases through the Framework to check that the formatting of the display lists was correct, that offsets onto the screen were being calculated correctly, and many other things. Indirectly, this demonstrated errors between the functional model and behavioural model—the Functional Framework demonstrated anamolies. This caused us to regression test the functional Framework and modify the specification given to the foundry. Of course, we should have performed this test much earlier.

We could vary the number of tests we generated with most passes of the generator suite by varying the number of times each primitive was perturbed. The list of tests described in Section 10.3.5 in fact is only a minimum test set, and we increased the total number of tests to approximately 5000.

10.4 Conclusions

We have presented a design and test strategy which we believe can be applied to a wide range of hardware accelerators, and help produce working silicon. This strategy has been used to design a new ASIC device and proved itself useful, both in producing simulations at a variety of levels, and identifying errors in our final specification. The device is still being fabricated and so it is too early to know whether it will work entirely as expected, although we strongly believe that it will.

We found that our approach generated far too many test vectors to apply to the Genesil model in a realistic time. Each clock cycle takes many minutes to simulate with the Genesil package, and this meant that many of our tests could not be used. Having all the tests to hand, however, allowed more accurate detection of errors when problems did appear.

Acknowledgements:

The authors wish to acknowledge France Glemot, Pascal Gros and the CAPTION SPIRIT team in general for their efforts in the design of the ASIC and their contribution to this paper. This project is part of the Esprit II program supported by the European Commission.

10.5 References

[1] Hans-Josef Ackerman and Christoph Hornung. The Triangle Shader Engine. In: R.L.Grimsdale, A. Kaufman (Eds.):*Advances in Computer Graphics Hardware V*,EurographicSeminars. Springer-Verlag, Berlin, 1992.

[2] The Spirit Workstation Consortium. The Spirit Workstation, A High Performance Technical Workstation, Part II, Detailed Project Description. Technical Report Annex-1053-ACE, ACE, July 1990.

[3] Michael Deering, Stephanie Winner, Bic Schediwy, Chris Duffy, and Neil Hunt. The Triangle Processor and Normal Vector Shader. *Computer Graphics*, 22(4), August 1988.

[4] S. R. Evans, P. F. Lister, R. L. Grimsdale, and A. D. Nimmo. The AIDA Display Processor System. In: R.L.Grimsdale, A. Kaufman (Eds.):*Advances in Computer Graphics Hardware V*,EurographicSeminars. Springer-Verlag, Berlin, 1991, p.15-28.

[5] James D. Foley, Andries Van Dam, Steven K. Feiner, and John F. Hughes. *Computer Graphics Principles and Practice.* Addison Wesley, second edition, 1990.

[6] Pascal Gros. Framework Documentations. Technical Report 0270, Caption Sarl, May 1989.

[7] Pascal Gros. ASIC Documentation. Technical Report 0277, Caption Sarl, June 1990.

[8] David Kirk and Douglas Voorhies. The Rendering Architecture of the DN10000VS. *Computer Graphics*, 24(4), August 1990.

[9] Ralph Marlett and Stephen R. Pollock. Guaranteeing ASIC Testability. *VLSI Systems Design*, 9(8), August 1988.

[10] Paul Munsch and Pascal Gros. Architecture of the ASIC for Spirit-I. Technical Report 0264, Caption Sarl, March 1990.

[11] Andrew Nimmo, Paul Lister, and Richard Grimsdale. A VLSI Design Strategy. In A. A. M. Kuijk, editor, *Advances in Computer Graphics Hardware III.* Springer-Verlag, 1988.

[12] Juan Pineda. A Parallel Algorithm for Polygon Rasterization. In *Computer Graphics*, pages 17–20. Association for Computing Machinery, July 1988.

[13] Josef Popsel and Eckard Tikwinski. A Multipurpose Hardware Shader. In: R.L.Grimsdale, A. Kaufman (Eds.):*Advances in Computer Graphics Hardware V*,EurographicSeminars. Springer-Verlag, Berlin, 1991, p.39-51.

[14] Andreas Schilling. Some Practical Aspects of Rendering. In: R.L.Grimsdale, A. Kaufman (Eds.):*Advances in Computer Graphics Hardware V*,EurographicSeminars. Springer-Verlag, Berlin, 1991.

[15] Bengt-Olaf Schneider and Ute Claussen. PROOF: An Architecture for Rendering in Object Space. In: A.A.M.Kuijk (Ed.): /it Advances in Computer Graphics Hardware III, EurographicSeminars. Springer-Verlag, Berlin, 1991, p.121-140.

[16] Martin White, Graham. J. Dunnett, Paul L. Lister, and Richard L. Grimsdale. The Spirit Workstation—Graphics Hardware. In: R.L.Grimsdale, A. Kaufman (Eds.):*Advances in Computer Graphics Hardware V*,EurographicSeminars. Springer-Verlag, Berlin, 1991.