Reprint from Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines, IEEE Computer Society Press, 2000.

Customising Graphics Applications: Techniques and Programming Interface

Henry Styles and Wayne Luk
Department of Computing, Imperial College
180 Queen's Gate, London SW7 2BZ, England

Abstract

This paper identifies opportunities for customising architectures for graphics applications, such as infrared simulation and geometric visualisation. We have studied methods for exploiting custom data formats and datapath widths, and for optimising graphics operations such as texture mapping and hidden-surface removal. Techniques for balancing the graphics pipeline and for run-time reconfiguration have been implemented. The customised architectures are captured in Handel-C, a C-like language supporting parallelism and flexible data size, and compiled for Xilinx 4000 and Virtex FPGAs. We have also developed an application programming interface based on the OpenGL standard for automatic speedup of graphics applications, including the Quake 2 action game.

1 Introduction

Despite tremendous progress in the last thirty years, real-time generation of realistic images still offers many challenges. The demand for realism requires improvement in algorithms, frame rate and resolution, while the variety of applications involves an increasing number of image formats. The situation is exacerbated by the need for reducing development time and costs, and for short turnaround due to changes in specification.

There are ASICs (application-specific integrated circuits) and high-end workstations for graphics acceleration, and microprocessors with graphics instructions have been introduced. However, they may not be cost effective or fast enough, for instance, when dealing with customised processing or non-standard image formats.

This paper presents a new approach to meet the challenges of real-time image generation. The key ideas are to:

- identify opportunities and techniques for customising architectures for graphics applications;
- prototype the customised architectures using hardware compilation and FPGAs;
- develop an API (application programming interface) based on the OpenGL standard for automatic speedup of graphics applications.

Two applications involving infrared simulation and geometric visualisation will be used to illustrate our approach. Table 1 shows the average results we have achieved, comparing software on a 400 MHz Pentium-II PC, software on this PC accelerated by an FPGA, and ASIC or a high-end workstation. Performance is given by frame rate, the number of frames that a system can generate each second.

The results for the geometric visualisation application show that the FPGA platform is approaching the performance of a dedicated graphics ASIC for general-purpose graphics applications. Infrared simulation, on the other hand, requires a custom pixel format which is not supported by standard graphics ASICs. The Onyx2 Reality Engine from SGI, a high-end graphics workstation, can deal with custom pixel formats. The frame rate estimated from its peak performance is given in Table 1 [12]. However this machine costs approximately 140,000 dollars in January 2000: it contains two 180 MHz MIPS processors, two Geometry Engine processors and two rasteriser ASICs, with a memory bandwidth of 6.4 GB/sec.

Table 1 Performance of case studies. Frame rate is given by the number of frames per second (fps).

Application	Implementation medium	Clock rate (MHz)	Frame rate (fps)
Infrared simulation	Software on PC	400	96
	Xilinx XC40150	20	167
	Xilinx XCV1000	40	330
	SGI Onyx2 Reality	180	2750 (estim.)
Geometric visualisation	Software on PC	400	24
	Xilinx XC40150	25	26
	Xilinx XCV1000	40	41
	NVidia TNT2 Ultra	170	55

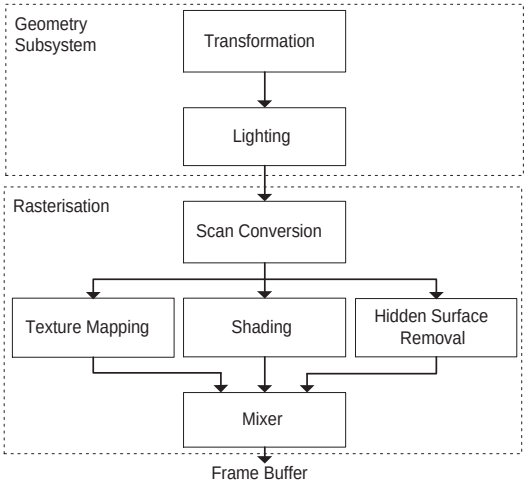

Figure 1 The graphics pipeline.

This represents approximately ten times the cost and memory bandwidth of our FPGA-based platform.

The performance of the infrared application demonstrates that the FPGA renderer is an effective low-cost platform for custom graphics applications. Although the Onyx2 workstation is approximately eight times faster, a lower-cost alternative would be preferable for applications not requiring the additional performance. The FPGA-based renderer meets these requirements. The development time of a customised FPGA renderer is comparable to optimised software, enabling a reconfigurable rendering platform to be used effectively for custom graphics applications.

Our approach would be attractive in developing reconfigurable designs where an ASIC solution is not available or too expensive; it would also be useful in exploring desirable algorithms and architectures for ASICs. The techniques that we have developed will be described in the following sections.

The use of FPGAs for graphics acceleration has been advocated several years ago [13]; more recently FPGA-based rasterisers have been proposed [3]. Other relevant research includes volume visualisation architectures for the Teramac custom computer [2], procedural texture mapping for the TM–2 rapid prototyping system [15], and the graphics codesign framework based on C++ and VHDL [5]. However, we are not aware of work similar to ours on customising architectures based on hardware compilation and graphics API, and on using a commercially available platform (RC1000–PP from Embedded Solutions) for implementation.

2 Custom rendering architectures

This section motivates the development of custom rendering architectures to cater for the requirements of different graphics applications.

Graphics applications can be characterised by the rate at which images are generated. Real-time rendering applications require smooth animation which must be responsive to user input. A common method is to represent solid objects using sets of connected triangles which are projected onto the frame buffer. The system for this operation is commonly referred to as the graphics pipeline (Figure 1), where each pipeline stage can be implemented in software or hardware, and executed in sequence or in parallel.

The geometry subsystem transforms the polygon model from three-dimensional space to two-dimensional space. It then calculates the light intensity at each polygon vertex using a physical lighting model. During rasterisation, polygons are filled to maintain depth coherency by removing the hidden

Table 2 Graphics applications which customise the graphics pipeline in Figure 1. Each table entry denotes a non-standard rendering effect or property which is unsupported by ASIC rendering architectures.

Application	Transformation and lighting	Scan conversion	Hidden-surface removal	Shading	Texture mapping	Frame rate
Infrared simulation	point source	low resolution	simple scenes	per-pixel lighting	sparse	high
Stylised depiction	effect specific	effect specific		per-pixel lighting		
Deformable model	NURBS evaluator		complex scenes			
Scan line rendering	Phong lighting	non-polygon	complex scenes	per-pixel lighting		low
Radiosity methods			scene occlusion	spline based		

surfaces and mixing the textures and shading. The resulting image is then accumulated in the frame buffer.

Real-time rendering algorithms are characterised by parallel computation and parallel memory access patterns. General-purpose architectures are not always well-suited to meeting these two requirements. Modern microprocessors facilitate instruction pipelining to improve performance in tight inner loops. This optimisation cannot easily be applied to calculations in the graphics pipeline because of its length and the complexity of arithmetic operators. Furthermore super-scalar execution cannot improve the performance of coarse-grain parallel operations which account for much of the computation within the graphics pipeline. Traditional microprocessor memory architectures also serve as a bottleneck because parallel memory access patterns common in graphics effects, such as texture mapping, must be serialised.

Custom rendering architectures usually include one or more ASIC graphics processors and parallel memory. The rendering architecture can be integrated with a microprocessor to provide the required degree of device specialisation, ranging from tight coupling in a high performance workstation, to loosely-coupled PC-based consumer products.

The standard range of features supported by ASIC-based graphics rendering architectures is not sufficient for all graphics applications. High performance graphics applications in the fields of animation, visualisation, simulation, and cinematic effects often employ non-standard graphics algorithms, which are unsupported by ASIC-based graphics rendering architectures. Furthermore, the fabrication of a new high performance ASIC-based platform for each custom application is prohibitively expensive. These applications are usually serviced by software renderers, which are flexible but have lower performance than ASICs. Reconfigurable devices offer an attractive alternative, since they provide the flexibility of software and performance comparable to custom hardware.

Table 2 shows several applications which require non-standard customisation of the graphics pipeline. These customisations are not supported by low-cost off-the-shelf graphics accelerator chips. An application typical of this set is infrared simulation. Infrared simulators are used in applications involving heat sensing. The simulator must generate real-time images of objects as viewed by an infrared camera. Commercial infrared simulation applications, such as Paradigm Simulator's "SensorVision", operate at custom resolutions, refresh rates and pixel formats; typically 12 bits greyscale per pixel, 128 by 128 resolution at 60 Hz. Moreover, infrared simulation requires non-standard post-processing filter effects to model the interaction of the environment with an infrared camera. The infrared simulation application has been used to test the performance of our graphics rendering architecture for applications which require non-standard customisations of the rendering pipeline.

Applications such as Mathematica and Matlab use three-dimensional graphics for geometric visualisation. Geometric visualisation involves the use of stan-

dard graphics rendering techniques, such as texture mapping, uniform-direction lighting and Gouraud shading, with high volume polygon models at high texture fill rates. The fill rate limits performance when objects move close to the viewer, while polygon setup becomes the bottleneck when objects move away. This application allows us to compare the performance of our rendering architecture with existing products as it requires only standard graphics effects.

Custom rendering architectures can be prototyped using an FPGA-based system. Such systems enable the use of reconfiguration to improve both functional diversity and performance. Table 2 demonstrates the breadth of graphics effects required by custom graphics applications. Most graphics applications, however, require only a small set of common effects which are supported by graphics ASICs. It is uneconomical to produce ASICs which support the vast breadth of graphics effects required by custom graphics applications. An FPGA-based architecture however, is capable of customising relevant hardware stages of the graphics pipeline critical to the performance of a given application.

For instance, hidden-surface removal and polygon culling algorithms can be varied according to polygon model complexity and level of scene detail; load balancing between pipeline stages can be achieved by adaptively managing resources and minimising computational bottlenecks. Moreover, parallel memory bank assignment can be varied to improve fill rate by controlling memory bandwidth for texture mapping. Reconfiguration can be applied when the application is loaded initially, or throughout execution in response to changes in application performance characteristics. These techniques will be described in later sections.

3 Design flow and partitioning

This section covers the prototyping of custom architectures using reconfigurable devices, and outlines the tools and platform that we use.

The design flow in our framework for developing custom rendering architectures is as follows. A software model of the application is created and profiled to determine performance characteristics. The developer then decides how the application is partitioned between hardware and software, produces the hardware components, and interfaces them to the revised software components.

We have developed a framework for rapid customisation of graphics architectures based on the Handel-C language from Embedded Solutions which supports hardware compilation. Compile-time parametrisation of the framework sets pixel format and datapath width, and its modularity allows developers to quickly plug-in custom designs for scan conversion, hidden-surface removal, shading and texture mapping.

The customised architecture is then compiled for the RC1000-PP board shown in Figure 2 which includes four parallel 2 MB memory banks, bus master PCI communication to the host, and a user-programmable Xilinx XC4000 or Virtex series FPGA.

There are several reasons for using Handel-C in this project. First, it enables rapid and incremental development, starting from a design almost directly translated from a software C description, to highly-optimised pipeline implementations, all in Handel-C. Second, the adoption of a syntax close to C facilitates variation of the hardware/software boundary. Third, Handel-C offers a degree of portability: designs can be re-targeted to different technologies. Fourth, the compilation strategy follows a particular timing discipline [9], allowing performance in number of cycles to be easily estimated. Fifth, the language contains constructs to specify bit-level operations, allowing precise resource management and control of circuit propagation delay. Finally, Handel-C is well-integrated with the RC1000-PP platform to provide a useful experimental vehicle. However, currently parallel designs have to be developed by hand, since parallelisation is not automated.

4 Customising data formats and operations

This section discusses the use of reconfiguration to improve the diversity of applications supported by hardware rendering platforms.

4.1 Custom output formats

Custom pixel formats are required by applications which service non-standard display devices. ASIC-

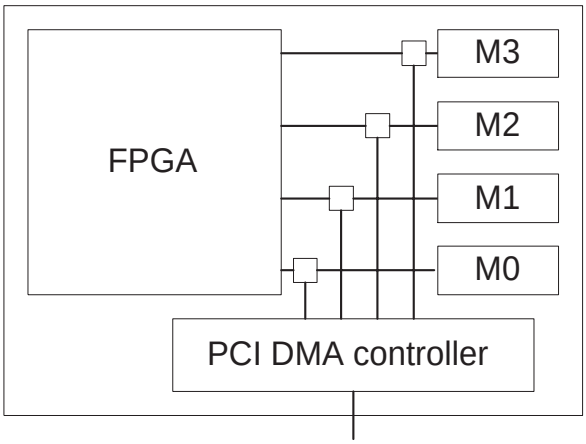

Figure 2 The RC1000-PP platform. The four memory banks are labelled M0. . .M3. The unlabelled boxes denote switches enabling the memory to connect to the FPGA or the DMA controller.

based renderers support up to 32 bits per pixel, made up of four 8-bit colour components. High accuracy pixel formats required by simulation applications are only supported by high-end graphics workstations. Reconfigurable platforms can be configured to support custom width datapaths and pixel formats.

Our graphics architecture framework adopts the C pre-processor for compile-time parametrisation of pixel format. The developer can select RGBα or greyscale colour components, and can set the accuracy of each component up to 32 bits per pixel.

The infrared simulation application requires a non-standard pixel format for which colour component and pixel accuracy parameters are set to greyscale, 14 bits per pixel. Geometric visualisation requires a standard 16 bit pixel format, for which the RGBα colour model is selected with 5 bit red, 5 bit green, 5 bit blue, and 0 bit α colour components.

4.2 Custom lighting and shading models

A shading model approximates a mathematical lighting model across the surface of polygons. Current ASIC graphics architectures evaluate lighting model equations in software at each polygon vertex, then bi-linearly interpolate these intensities across each polygon face using hardware Gouraud shading.

Physically-based lighting models such as Phong and Torrance-Sparrow illumination, which require that lighting equations be evaluated at each pixel, are poorly approximated by Gouraud shading in exising ASIC renderers. Alternative approximations involve light-mapping, in which two textures are applied to each polygon: a surface texture and a dynamic light-map texture. However, this method compromises the performance gains of hardware rendering as light maps must be constructed and updated in software.

A reconfigurable rendering architecture can include a custom lighting and shading unit to support non-standard and accurate lighting models which are not supported by ASICs. For example, a custom shader can be constructed for spline-based interpolation methods based on quadratic Bezier patches, cubic Bezier patches, or Clough-Tocher interpolation to improve image quality when visualising radiosity illumination. Alternatively, one of many stylised depiction lighting models can be implemented in hardware to improve the readability of technical illustrations.

We have customised the graphics architecture framework with custom plug-in shading units for our two case study applications. Infrared simulation requires point-source lighting effects and depth cueing to approximate the radiance of heat between surfaces. The rasterising hardware has been customised to evaluate a point-source lighting model to approximate this effect. Each pixel intensity is calculated as proportional to the distance of the object from a heat source. This lighting model cannot be approximated using existing ASIC renderers.

The geometric visualisation application requires directional lighting: uniform lighting from a constant direction. It is accurately represented by vertex lighting and Gouraud shading, with straight-forward hardware implementations.

4.3 Texture mapping

Texture mapping is a standard function supported by all new graphics rendering hardware. Graphics applications require a diverse set of texture mapping effects, a subset of which is supported by ASIC-based renderers such as the NVidia TNT2 and graph-

ics workstations such as the Onyx2 Reality Deskside. An FPGA rendering architecture can be rapidly customised to accelerate unsupported texture mapping effects such as the ones below.

- Custom projection and mapping functions.
- Non-standard and variable quality texture sampling schemes such as summed area tables [1] for anisotropic filtering.
- A variable length texture blending cascade for multitexturing.
- Custom texturing methods, such as paraboloid environment mapping [11], and true bump-mapping [4].
- Procedural texture mapping [15].

As the quality of texture mapping effect increases, the complexity and size of hardware increases. Although there are many texture mapping algorithms and effects, a single graphics application typically employs only a few during execution. ASIC renderers de-activate circuits for extraneous algorithms, while an FPGA-based rendering architecture can instantiate a custom texture-mapping unit for each application. The latter is therefore more likely to be able to support a set of texture effects with greater diversity, since there is no need to have hardware for the complete set of effects concurrently.

We have customised hardware texture mapping for our two case study applications. Infrared simulation does not require high quality texture mapping, as the effect is sparsely applied. We have customised the texture mapping subsystem to support fixed delta affine texture mapping, which is a simple approximation used in ASIC-based rendering architectures such as the Sony Playstation.

Geometric visualisation benefits from accurate perspective-correct texture mapping which requires a highly-accurate division operation for each pixel. It has been implemented with a pipelined divider.

5 Optimising resources and performance

This section describes techniques which improve performance by reconfiguring the hardware renderer to exploit application specific optimisations.

5.1 Optimising datapath widths

Throughout rasterisation, fixed-point number representations are used for polygon setup, scan conversion and interpolations. Datapath width parameters control image quality and circuit size, and can adapt a design to FPGA devices with different capacity. For example, the Xilinx 40150 is sufficiently large to accommodate an RGBα rasteriser with 16-bit fractional accuracy for all interpolation calculations. However datapath width must be parameterised to 9-bit fractional accuracy to accommodate the reduced circuit area of a Xilinx 4085 device.

We provide sufficient control to decouple the quality of lighting and texture interpolation. This allows numerical accuracy and circuit area to be targeted at graphics effects which are important for each application. Once the datapath width is optimised, computational operators can then be effectively pipelined to maximise throughput.

The datapath parameter is set to 9 bits fixed point accuracy for infrared simulation, and 16 bits for geometric visualisation. Infrared simulation uses 128 by 128 resolution, compared to 512 by 384 for geometric visualisation, resulting in shorter interpolation runs. Furthermore, texture mapping is only applied sparsely, and artifacts introduced by computational inaccuracy have a minimal impact on image quality for this application.

5.2 Algorithm Selection

For each stage of the graphics pipeline, several algorithms exist which implement the required functionality but offer different performance characteristics. To maximise the performance of a specific application, the algorithm instantiated in graphics hardware must best match the performance characteristics of the application.

The importance of application-specific algorithm selection is illustrated by the contrasting performance characteristics of our case study applications. Two algorithms for hidden-surface removal, one based on span buffering and the other on depth buffering, will be introduced.

Infrared simulation requires high frame rates and low frame latency when rendering simple polygon

models. For this application, a custom hidden-surface removal algorithm is selected which optimises performance for these conditions.

Span buffer hidden-surface removal divides rasterisation into two operations: span insertion and fill. During span insertion, new spans are inserted into a database which maintains lists of visible spans. By determining full hidden-surface removal before each polygon is filled, expensive pixel overdraw is minimised, and the fill rate load is reduced. Clearing the span buffer between frames is an order of magnitude faster than the equivalent operation for the alternative hidden-surface removal algorithm of depth buffering. For applications such as infrared simulation which process low volume polygon models at high frame rate, the overall performance is improved by using span buffer hidden-surface removal.

There exist two possible hardware/software partitioning schemes when span buffer hidden-surface removal is employed. For large FPGAs, both span insertion and fill are placed in hardware, and polygon vertices are passed between software and hardware through the DMA controller (Figure 2).

Figure 3 shows a span buffer implementation for small FPGAs, in which span insertion is carried out in software. The span database is then passed to an FPGA-based fill unit. Progressive migration of the span buffer rasteriser to hardware using successive partitioning schemes affords the developer a fall-back position if the complete hardware rasteriser is found to be too large for the target FPGA device. Benchmark results for Xilinx XC40150 implementation in Table 1 show how application-specific selection of the span buffer algorithm directly benefits performance.

Consider now geometric visualisation applications, which process high volume polygon models. As scene complexity increases and occlusion reduces, span width tends towards one pixel, and span insertion becomes expensive. The performance of depth buffering degrades less rapidly as scene complexity increases, and the technique is better suited to rendering complex scenes with low occlusion: a depth buffer records the depth of the nearest primitive rendered to each pixel. Pixels within new primitives are only written to the frame buffer if they are nearer to the viewer than entries in the depth buffer, which is

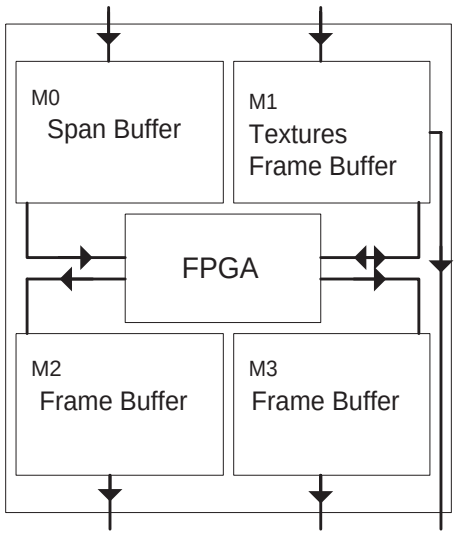

Figure 3 Span buffer hidden-surface removal on the RC1000–PP. Memory banks M0..M3 are shown in Figure 2

updated accordingly.

To be effective, the entire depth buffer rasteriser must be placed in hardware. Figure 4 shows the dataflow for the RC1000-PP depth buffer rasteriser. This circuit is comparable in size to a span-buffer rasteriser with both span insertion and fill in hardware.

These case studies show how application-specific selection of hidden-surface removal algorithms can benefit performance. In addition, the case studies have shown increased importance of this technique when targeting larger FPGA architectures. For instance, the Xilinx XC4085 can only support the span-buffer implementation, whereas the Xilinx XC40150 can support both span and depth buffering. The larger device demonstrates an increased capability for application-specific algorithm selection by enlarging the set of candidate algorithms.

5.3 Balancing the graphics pipeline

The FPGA-based graphics renderer can easily be modified to improve application-specific load balancing of the graphics pipeline. In particular, circuit area and memory bandwidth resources must be assigned correctly throughout to balance the graphics pipeline

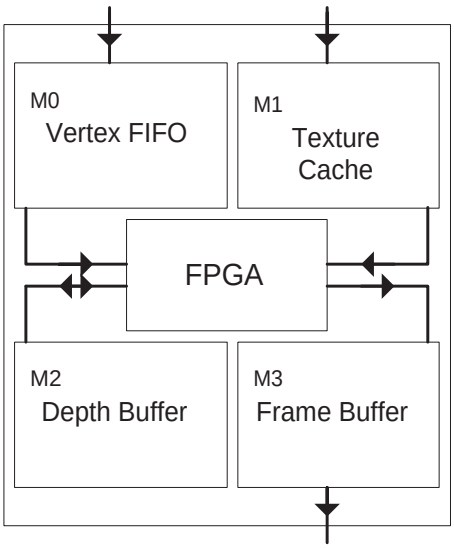

Figure 4 Depth buffer hidden-surface removal on the RC1000–PP.

and avoid performance bottlenecks.

Each stage of the hardware rasteriser, polygon setup, texture mapping and shading, can be customised and assigned hardware resources commensurate with its contribution to performance. For some applications, the computational and memory throughput of one subsystem may have to be reduced, in order that other pipeline stages can be improved to optimise overall performance.

For our case studies, hardware resources attributed to each stage of rasterisation are optimised to best match hardware and software performance requirements. Infrared simulation is characterised by low volume polygon models and by sparse application of texture mapping within each frame. The performance requirement for non-textured Gouraud shaded pixels is greater than that for texture-mapped pixels, which account for only a small proportion of rendering pipeline fill load. Hence memory capacity, bandwidth and circuit area associated with the texture mapping unit and the polygon setup unit are reduced, such that hardware resources available to accelerate Gouraud shading can be increased.

Specifically, two modifications have been made to the texture mapping and polygon setup units to best balance the graphics pipeline. Arithmetic operations such as multiply and divide are evaluated over several cycles, freeing circuit area to implement pipelined arithmetic for lighting calculations. Moreover, a single memory bank, M1, is used to store texture maps and to implement part of the frame buffer, as shown in Figure 3. The use of three memory banks, M1, M2 and M3, for frame buffering maximises memory bandwidth available to the shading unit. The rendering pipeline thus has a peak fill rate of six Gouraud shaded pixels every cycle, and one texture-mapped pixel every two cycles.

Geometric visualisation, on the other hand, processes high volume polygon models and requires high fill rate for texture mapping. More hardware resources are therefore used in optimising polygon setup and texture mapping throughput for the depth buffer. Arithmetic operations are more extensively pipelined, while pipelining in the shading unit is minimised. In particular, parallel hardware is used in the polygon setup unit to improve polygon throughput. As a result, setup time is reduced from 187 cycles per polygon in the span buffer to 54 cycles per polygon in the depth buffer. A single memory bank, M1, is dedicated to storing texture maps (Figure 4). The rendering pipeline has a peak fill rate of one Gouraud shaded, texture-mapped pixel every cycle.

These applications demonstrate how the graphics pipeline can be balanced to match the performance characteristics of hardware and software, resulting in higher performance.

6 Run-time reconfiguration

Section 5 has illustrated the benefits of reconfiguration to improve the performance of different applications using the same hardware platform. The hardware renderer is customised to maximise performance for each application, and the FPGA is configured before execution. The use of reconfiguration to improve performance is motivated by the differences in performance of different graphics applications, depending on the graphics effects applied during rendering, and the behaviour of the application in presenting graphics primitives to the rendering pipeline. Load-time reconfiguration can effectively optimise against vari-

ation in graphics effects, but cannot maintain optimality against the application's dynamic behaviour in presenting graphics primitives, which is a function of the application's run-time state.

For example, the geometric visualisation application transforms geometric objects and animates them in a path towards then away from the viewer. During animation, the performance of the application software varies according to the distance between the graphical objects and the viewer. When the objects are near to the viewer, the load within the graphics pipeline is concentrated on the shading and texturing fill units of the rasteriser. Fill rate limits performance, and polygon setup hardware stalls while waiting for the fill unit to complete. Conversely, when the objects move away from the viewer, the load on the fill units diminishes, and polygon setup load increases until the fill units are stalled, waiting for the polygon setup hardware to complete. Hardware optimality can therefore be maintained by run-time reconfiguration of the graphics renderer, to ensure that the graphics pipeline is always balanced.

Two rasterisers have been implemented: one is optimised for high polygon throughput, and the other for fill rate. Hardware for profiling performance has also been developed to measure the time taken by each stage of the rasterisation pipeline. As each frame is recovered from the frame buffer, profiling information is sent from the RC1000-PP to software which assesses the performance of the current rasteriser against the predicted performance of the alternative rasteriser. When the difference in performance reaches a threshold level, the FPGA is then reconfigured with the new optimised rasteriser and rendering continues.

The reconfiguration time for the XCV1000 device on the RC1000-PP has been measured to be 23.6 ms, which allows for a 40 Hz frame rate with only one frame dropped during reconfiguration. If reconfiguration occurs once each second for every 40 frames, it causes no noticeable break in animation.

Run-time reconfiguration can also be applied to graphics applications such as radiosity form-factor calculations, which demand high throughput but do not require consistent low frame latency.

7 Application programming interface

A graphics API is an abstract software layer which improves compatibility between graphics applications and graphics rendering architectures. Graphics applications benefit as they do not need to be modified to support a diversity of graphics rendering architectures. Hardware vendors also benefit, as they have a standard for their rendering architectures allowing compatibility with existing graphics applications.

We have constructed an interface between our reconfigurable rendering architecture and a subset of OpenGL [8], a widely-used and supported API for two- and three-dimensional graphics. This API allows us to run OpenGL graphics applications on our reconfigurable rendering architecture, and benchmark our hardware designs against existing rendering platforms. Both infrared simulation and geometric visualisation can be supported by this approach.

Our API is based on Mesa3D [10], a graphics library for OpenGL applications. We have constructed a Mesa3D driver for our reconfigurable rendering architecture which routes calls to the API through the Mesa3D state machine to the hardware rasteriser described in Handel-C. Our Mesa3D device driver controls reconfiguration of the RC1000-PP, synchronises dataflow between hardware and software elements of the graphics pipeline, and manages texture placement on RC1000-PP memory.

The most challenging aspect of this work is the development of an effective way of caching textures on the RC1000-PP as they are required by OpenGL applications. The FPGA rasteriser can output one 32-bit texture-mapped pixel every cycle at a clock rate of 40 MHz, meaning 160 MB of texture map is read by the rasteriser each second. Clearly it is infeasible to use the PCI bus, with a peak bandwidth of 133 MB per second, to transfer texture-mapped pixels from main memory to the RC1000-PP on demand.

To maximise rasteriser throughput, the texture map associated with each polygon must be made resident on the designated RC1000-PP memory bank before each polygon is drawn. It is desirable to upload all the textures required for a sequence of rendering to the RC1000-PP before rendering begins to minimise PCI bus traffic during rendering. However, the typical

volume of texture data required to render a scene exceeds the 2 MB size of a memory bank dedicated for this purpose. We have therefore constructed a software texture cache, which manages run-time placement and removal of texture maps on the RC1000-PP texture memory bank, to minimise transfer of textures between host and FPGA memory, and minimise PCI traffic during rendering.

Figure 5 shows the action game Quake 2 running on the RC1000-PP through our Mesa3D driver. Table 3 compares the performance of the RC1000-PP against software and ASIC implementations. Other OpenGL applications, such as LightWave3D and VRML browser, have also run on the RC1000-PP using our API.

Table 3 Quake 2 benchmark performance. The first two demonstrations come from the Quake 2 package, while the other two can be downloaded from http://www.planetquake.com/sda/. Software runs on a 400 MHz Pentium-II PC. The ASIC is a 170 MHz NVidia TNT2 Ultra.

Demonstration	Software (fps)	XC40150 (fps)	XCV1000 (fps)	ASIC (fps)
demo1.dm2	0.2	6.5	14.4	71.6
demo2.dm2	0.2	6.4	14.2	68.8
jail5059.dm2	0.2	6.8	15.0	72.6
jail3a020.dm2	0.3	7.0	15.6	71.5

Figure 5 Quake 2 running on RC1000-PP.

8 Scalability

This section identifies opportunities for short-term improvements in performance, and prospects for long-term performance scalability.

As an example, Quake 2 is characterised by large textures which cannot be cached efficiently on the RC1000–PP. For this application, 8 or 16 MB texture banks would be more appropriate. Performance would benefit from moving the RC1000–PP from the PCI bus to the Advanced Graphics Port (AGP), a dedicated point-to-point connection which allows for up to 1 GB/sec bandwidth between graphics hardware and memory. The use of 8 or 16 MB texture banks and AGP connectivity would provide a marked short-term improvement in performance by increasing the texture cache hit rate, and reducing the texture cache miss penalty. Given that a texture cache hit occurs, an improved memory subsystem containing more parallel memory banks and fast memory would improve throughput. A single additional memory bank would allow the rasteriser based on depth buffer to output two pixels per cycle, effectively doubling the performance for applications limited by fill rate, such as infrared simulation.

In the long term, our graphics rendering architecture framework can be scaled up by retargeting new FPGA architectures. We have demonstrated design re-use when targeting the same design to Xilinx XC4085, Xilinx XC40150, and Xilinx XCV1000 devices on the RC1000–PP board. For each new FPGA platform, an initial version of the rasteriser is created by recompiling our Handel-C source code. Future FPGA architectures will offer scalable performance and image quality through increased clock rate and higher density of FPGA resources.

Improved density will allow earlier stages of the graphics pipeline to be integrated into hardware.

ASIC graphics architectures, such as the NVidia GeForce256 [7], already use dedicated hardware for geometry subsystem operations. A larger FPGA will increase the candidate set of algorithms for each hardware stage of the graphics pipeline, and allow for more aggressive algorithm selection and pipeline balancing optimisations. Multiple FPGAs can also be used, although finding an optimal chip partitioning may not be straightforward. Finally, an FPGA customised for graphics will need to have sufficient on-chip memory for the frame buffer and texture data to maximise fill rate.

9 Concluding remarks

We have described various techniques for improving cost effectiveness of graphics applications. The key elements of our approach include the adoption of custom data formats and datapath widths, the optimisation of common operations such as texture mapping and hidden-surface removal, and the deployment of pipeline balancing and run-time reconfiguration where appropriate. Customised architectures have been prototyped using hardware compilation and FPGAs, and an OpenGL-based API has been developed for automatic speedup of graphics applications.

The compiled hardware at present has not exploited many device-specific features, such as embedded memory or partial dynamic reconfiguration. Current and future work includes refining our designs to take advantage of such features, and exploring automatic techniques for producing optimised pipelines [14] and run-time reconfigurable designs [6].

Acknowledgements

Many thanks to Matt Bowen, Glynn Clements, George Constantinides, Arran Derbyshire, Tom Egan (DERA UK), Roger Gook, Patrick Kane, Brian Paul, Charles Sweeney, David Thomas, Richard Sandiford, Shay Seng, Tim Todman, and Markus Weinhardt for their comments and assistance. The support of Embedded Solutions Limited, UK Engineering and Physical Sciences Research Council (Grant number GR/24366, GR/54356 and GR/59658), and Xilinx, Inc. is gratefully acknowledged.

References

[1] F.C. Crow, "Summed-area tables for texture mapping", *SIGGRAPH'84*, July 1984, pp. 207–212.

[2] W.B. Culbertson et. al., "Exploring architectures for volume visualisation on the Teramac custom computer", *FCCM'96*, IEEE Computer Society Press, 1996, pp. 80–88.

[3] L. del Pino, "FPGA-based 3-D graphics rasterizers", in *Advances in Information Technologies*, J.-Y. Roger et. al. (eds.), IOS Press, 1998, pp. 514–521.

[4] I. Ernst et. al., "Hardware supported bump mapping: a step towards higher quality real-time rendering," *10th Eurographics Workshop on Graphics Hardware*, 1995, pp. 63-70.

[5] J.P. Ewins et. al., "Codesign of graphics hardware accelerators", *Proc. SIGGRAPH/Eurographics Hardware Workshop*, 1997.

[6] W. Luk, N. Shirazi and P.Y.K. Cheung, "Compilation tools for run-time reconfigurable designs", *FCCM'97*, IEEE Computer Society Press, 1997, pp. 56–65.

[7] NVidia, "Geoforce 256: The world's first GPU", http://www.nvidia.com/geoforce256.nsf.

[8] "The OpenGL library", http://www.opengl.org/.

[9] I. Page, "Constructing hardware-software systems from a single description", *Journal of VLSI Signal Processing*, Vol. 12, 1996, pp. 87–107.

[10] B. Paul, "The Mesa 3D graphics library", http://www.mesa3d.org/.

[11] H. Seidel et. al., "View-independent environment maps", *1998 Eurographics/SIGGRAPH workshop on graphics hardware*, 1998, pp. 39-45.

[12] SGI, "Silicon Graphics Onyx2 Product Guide", http://www.sgi.com/.

[13] S. Singh and P. Bellec, "Virtual hardware for graphics applications using FPGAs", *FCCM'94*, IEEE Computer Society Press, 1994, pp. 49–58.

[14] M. Weinhardt and W. Luk, "Pipeline vectorization for reconfigurable systems", *FCCM'99*, IEEE Computer Society Press, 1999, pp. 52–62.

[15] A.G. Ye and D.M. Lewis, "Procedural texture mapping on FPGAs", *FPGA'99*, ACM Press, 1999, pp. 112–120.