Codesign Of Graphics Hardware Accelerators

Jon P. Ewins, Phil L.Watten, Martin White†, Michael D. J. McNeill, Paul F. Lister

Centre for VLSI and Computer Graphics,
University of Sussex,
England

Abstract

The design of a hardware architecture for a computer graphics pipeline requires a thorough understanding of the algorithms involved at each stage, and the implications these algorithms have on the organisation of the pipeline architecture. The choice of algorithm, the flow of pixel data through the pipeline, and bit width precision issues are crucial decisions in the design of new hardware accelerators. Making these decisions correctly requires intensive investigation and experimentation. The use of hardware description languages such as VHDL, allow for sound top down design methodologies, but their effectiveness in such experimental work is limited. This paper discusses the use of software tools as an aid to hardware development and presents applications that demonstrate the possibilities of this approach and the benefits that can be attained from an integrated codesign design environment.

CR Categories and Subject Descriptors: D.3.2 [Programming Languages] Language Classifications - Object Oriented Programming; I.3.1 [Computer Graphics]: Hardware Architecture - Graphics Processors; I.3.3 [Computer Graphics]: Picture/Image Generation - Display Algorithms.

Additional Key Words and Phrases: 3D graphics, object oriented programming, codesign, VHDL, C++, component and design reuse

1 INTRODUCTION

The Centre for VLSI and Computer Graphics at Sussex University is involved in the research of advanced algorithms in computer graphics and in their implementation in state of the art ASIC and FPGA based architectures. The hardware description language VHDL (Very High Speed Integrated Circuit Hardware Description Language), is used as the main platforms for this work. The use of such a language permits the adoption of a 'Top Down' design methodology which allows the designer to focus on a more abstract level of functionality[11].

†Centre for VLSI and Computer Graphics,
University of Sussex, Falmer, Brighton
East Sussex, BN1 9QT, England
e-mail: M.White@sussex.ac.uk

1997 SIGGRAPH/Eurographics Workshop

This means that a design can be planned more quickly and accurately in it's entirety. The use of VHDL provides the hardware designer with other advantages, such as technology independence, etc. which are beyond the scope of this paper[1, 11].

In this paper we describe techniques in which software languages such as C++ can be used in conjunction with VHDL to increase the level of abstraction possible during the design process, and improve overall productivity. Much work has been carried out in the field of hardware/software codesign [2,4,7], but little has been specific to computer graphics[5]. We expand on the ideas developed in Galadriel [6]. This paper discusses the use of software as an essential tool in the design process of algorithms from a conceptual level through to their architectural implementation, as well as a tool for general hardware modelling and development. Specific emphasis is placed on the design and development of computer graphics related algorithms and the design of 3D graphics pipelines.

Throughout this paper the term software will be used to refer specifically to languages such as C++ and in particular Windows based applications. The term hardware will refer to physical hardware implementations and hardware description languages (HDLs) such as VHDL.

The following sections will start with a discussion on the comparison of the advantages of software applications versus hardware description languages, and describe how the two can be integrated together to produce an efficient and productive design environment. The following sections will then present the requirements for such an environment in the form of the system implemented at Sussex.

2 SOFTWARE VERSUS HARDWARE DESCRIPTION LANGUAGES

Although VHDL allows the designer to operate initially at a more abstract (behavioural) level before final Register Transfer Level (RTL) designs are implemented, simulation of models in behavioural VHDL do incur a relative time penalty when compared to a corresponding simulation in a software language such as C++, and it's efficiency in investigative and experimental work is limited.

Table 1 shows an example of the simulation performance of both languages. A comparison of both behavioural VHDL and C++ performing identical non-trivial operations was carried out. The operation performed was the complete setup computation requirement of triangles for scan conversion with linear edge functions [8], Gouraud shading, perspectively correct texture mapping[3] and antialiasing[9]. This involves 64 additions, 64 multiplications and 7 reciprocal operations. The tests were performed for both one million and ten million triangles using

behavioural VHDL written and run from the Model Tech V-System platform and Microsoft Visual C++ version 4.2. The C++ was compiled and run in both Release and Debug modes and all three were based on a Pentium Pro 200MHz machine.

	Behavioural VHDL	C++ Debug	C++ Release
1,000,000 triangles	135 seconds	25 seconds	14 seconds
10,000,000 triangles	20 minutes	250 seconds	140 seconds

Table 1 Performance Comparison of VHDL and C++

The times given here show that the difference in performance of C++ to behavioural VHDL is in the ratio of 5-6:1 for debug mode C++ and 9-10:1 for release mode. It might be argued that during the development of new algorithms a large proportion of time is spent in the debug mode. However, during long term simulation runs a 9-10 times release mode performance improvement is achieved.

As well as such quantitative performance issues, there are more qualitative issues that should be considered. The use of software languages and their advanced development platforms provide other benefits. The advanced debugging environments of these languages can greatly improve the productivity of designing and developing algorithms at a conceptual level. Also, we have produced VRML parsers and C++ 'plug-ins' for commercial modelling software packages, such as Kinetix 3D Studio Max, that provide an abundant model database supply and the simplified production of full multi-frame complex animations. Databases of any form can be created with ease to model any area of investigation that is required.

Of course, VHDL has other important advantages, but at the basic algorithmic level it is not interactive. The key to design efficiency is deciding where and when to invoke VHDL and when to use C++. An algorithm's functionality during the initial design phase can be modelled as abstractly as necessary, and it is more important at this stage to produce results and pictures with ease. Our software environment can provide abundant data bases and simplify generation of images, and even complex animations. The same software can then transform such data into formats for use by a VHDL testbench.

An alternative suggestion for the conceptual level development of algorithms is the use of interpreted languages such as Microsoft Visual Basic. These languages have the advantage that alterations can be made to the code as the program is running. Therefore, as an error is seen in the graphical output of say a new scan conversion algorithm, the code can be altered before the next pixel is drawn.

It is also possible for the software to be augmented to produce statistical evaluations of the algorithms under development. Multithreaded run-time performance monitoring is one such possibility. Through accurate modelling of hardware number representation and bit width precision, the software allows rapid experimentation with immediate visual results. The graphical interfaces of Windows based applications can allow the user to make run-time alterations to the data path bit widths as well as to alter modes of operation and algorithm selection. The Algorithm Prototyping Environment presented here is an example of an application developed at Sussex for just such operations.

3 AN INTEGRATED DEVELOPMENT ENVIRONMENT

As discussed above, a large selection of software tools can be developed to work alongside VHDL to aid in algorithm research and hardware design. Figure 1 shows the relationship between the software and hardware design processes of our integrated environment.

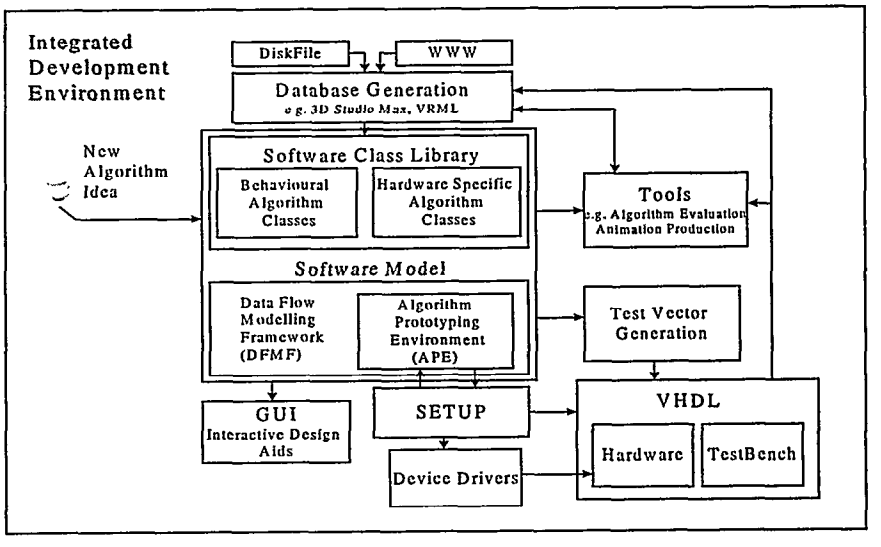

Figure 1: An Integrated Development Environment

The Data Flow Modelling Framework and the Algorithm Prototyping Environment software, form the main structure of the Software Model. In brief, the Data Flow Modelling Framework is used to simulate the structure of an architecture. It does this by allowing the code to be modularised into algorithmic blocks with the flow of data between these blocks and their organisation being definable by the user.

The Algorithm Prototyping Environment is a particular implementation of a graphics pipeline based on this framework. It provides a comprehensive graphical user interface that allows the designer to open a large selection of databases, make interactive alterations to algorithm selection and to alter bit width precision figures and supplies a range of on line design aid tools.

Using this code, at the conceptual level, the Software Model simulates the behaviour of new algorithms both individually and as a part of an overall structure, e.g. a rendering pipeline. This model allows the rapid prototyping of new ideas and structural organisations. The algorithms can then be progressed further to allow the testing of performance issues such as fixed and floating point precision and how they effect the visible results. It is important that an accurate means of bit width precision modelling be used.

Finalised algorithms are added to a C++ class library for reuse as components in future software simulations of new architectures. Such software models, if well written and documented, act as excellent examples of the complexities and subtleties of each algorithm.

Another important part of the development environment is the use of good databases, e.g. small and large and varied databases which test the algorithms sufficiently. As mentioned above, to meet this requirement we have developed a 'plug-in' to 3D Studio Max as well as VRML parsers. These allow new designs (algorithms and architectures) to be tested with images and animations selected from a large range of databases compatible with 3D Studio Max.

As the VHDL architecture is produced, based on the initial algorithmic decisions reached from the software simulation, the software model is modified simultaneously to match any unforeseen changes in the architectural requirements and used to provide rapid feedback as to the implications of such changes.

During hardware development the software can provide test data for particular individual parts of the pipeline design. Eventually the development process will reach a point where the software platform is solely a means of supplying services such as database production, output data interpretation and visualisation and tools such as image compare operations.

It should be noted that the software model contains a version of the setup code identical to that to be used with the device drivers. In this way we can allow the same scene data to produce setup data for both the software simulation and the physical hardware itself. The image compare tools then allow comparison of expected and achieved results.

At the end of the design process the software development will result in a software based reference model to accompany the completed hardware design. Such a software based reference model was recently used by Microsoft, for the Chicken Crossing demonstration for the Talisman architecture[10].

3.1 Development Environment Features

1. *Algorithm Investigation*
   - Accelerates the implementation of new algorithms and provides an environment in which to test them both individually and as part of an entire pipeline
   - Allows drawing of individual pixels, primitives or entire scenes selectively
   - Provides methods of performance monitoring

2. *Architectural Modelling*
   - Change the organisation and order of execution of the algorithmic blocks to investigate and verify potential new architectures
   - Ability to select between different algorithms

3. *Low Level Algorithm Modelling*
   - Incorporation of accurate bit width precision operations
   - Change rapidly the accuracy and resolution of algorithms and analyse the effect, e.g. texture co-ordinate interpolation from floating point to 8.16 fixed point

4. *Production of Test Stimuli or Vectors*
   - Provision of real world scene data from, for example, commercial packages such as Kinetix 3D Studio Max and VRML files
   - Rendering of these databases using the software simulation for comparison purposes
   - Production of test stimuli for driving VHDL hardware modules or setup data for driving the physical hardware

5. *Provision of On line tools for result analysis*
   - Provision of image analysis tools, e.g. zoom in functions, image compare etc.

6. *Rapid production of real results*
   - Images and full animations
   - Statistical data for algorithmic evaluation

4 DATA FLOW MODELLING FRAMEWORK

The Data Flow Modelling Framework (DFMF) forms the backbone of our hardware modelling environment. It allows the developer to model the operation of the hardware structure through modularisation of the code structure into distinctive process elements. These process elements can simulate the breakdown of the basic hardware structure and the flow of data between them to whatever degree is chosen.

This breakdown may range from an early layout of the main functional algorithmic units of a graphics pipeline, such as scan conversion to texture read and so on, to a the modelling of the physical partitioning in a multiple FPGA based architecture, right down to full register transfer level partitioning. Although, RTL descriptions are usually done in VHDL.

The diagram in Figure 2 outlines the basic structure of this frame work and indicates the C++ class structures involved. Here the

functional process element examples are based on a projective rendering pipeline.

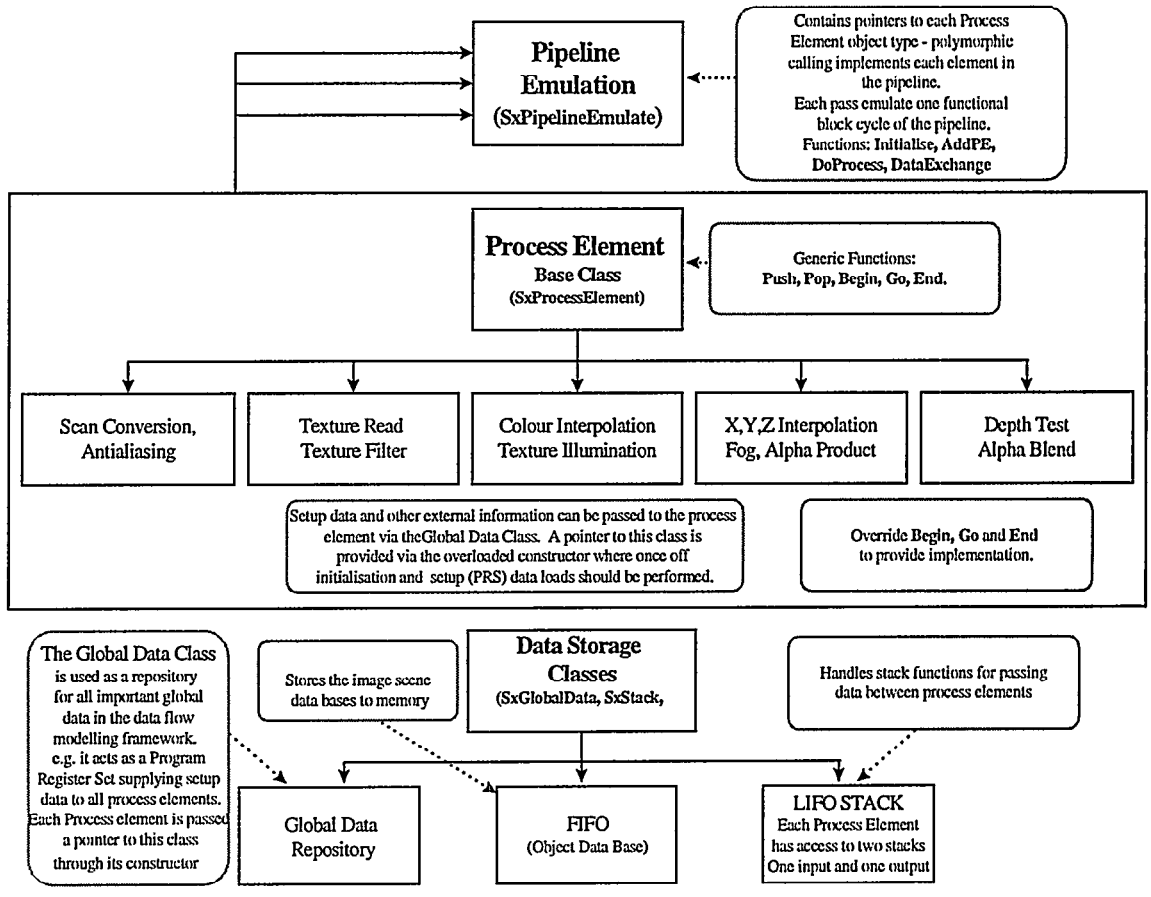

Figure 2: Data Flow Modelling Framework Class Structure

4.1 Pipeline Emulation

The Data Flow Modelling Framework is designed primarily for the modelling of a graphics pipeline. For this reason the main operation is encapsulated within the class SxPipelineEmulator and controlled from four main functions: Initialise(), AddPE(), DoProcess() and DataExchange(). Different Process element blocks are derived as required from the SxProcessElement class and formed into a pipeline structure with the function Initialise() which in turn uses the AddPE() function to include the process elements in the order desired. The code fragment in Figure 3 provides an example of this operation.

```
//-------------------------------------------------------
BOOL Pipeline::Initialise(SxGlobalData *p)      //Supply pointer to Global Data Class
{
      BOOL bContinue;
      pGlobalData = p;
      bContinue = pGlobalData->Setup();         //Call setup functions in SxGlobalData
      if(bContinue)
      {
            //Creation of pipeline in order of process elements organisation
            //Note that AddPE accepts pointers to instances of derived process
            //element classes, each of which is provided with a pointer to SxGlobalData

            AddPE(new PEScanConvert(pGlobalData));
            if(bTextureFlag)
            {
                  AddPE(new PETextureRead(pGlobalData));
            }
            AddPE(new PEColourInterpolate(pGlobalData));
            AddPE(new PEIlluminate(pGlobalData));
```

```
            AddPE(new PEXYZInterpolate(pGlobalData));
            AddPE(new PEBlendFunctions(pGlobalData));

            //Pipeline Organisation Control
            m_pDataIn = GetFirstProcessElement()->GetInStack();
            m_pDataOut = GetLastProcessElement()->GetOutStack();
            return TRUE;
      }
      else return FALSE;
}
//-------------------------------------------------------------
```

Figure 3 Code fragment showing the pipeline initialisation process

Using this basic structure any combination of process element blocks can be placed together to form different structures. The use of user specified conditional blocks in this code allows process element reorganisation during run time. Figure 4 shows a schematic representation of the pipeline organisation created with the above code.

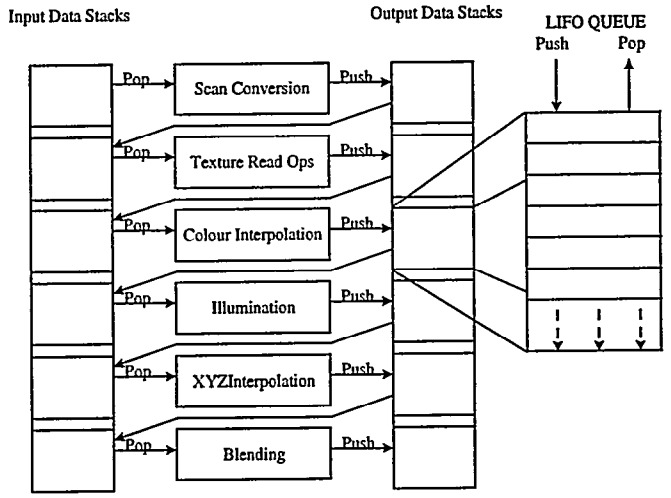

Figure 4: Pipeline Modelling with Data Stacks

As can be seen in Figure 4, each process element has a separate input and output stack, the format of which is discussed in section 4.3. The use of separate independent input and output stacks allows concurrent data flow to be modelled sequentially. The operation of the pipeline is performed with cyclic calls to the functions DoProcess() and DataExchange(). DoProcess() implements the functionality of each process element in turn with data read from the input stack, processed and then written to the output stack. When all the process elements have been processed, the function DataExchange() passes the stack pointers so that the output stack of one process element becomes the input stack of the next and so on, thus simulating the flow of data through a pipeline. This operation is shown in Figure 5.

```
//-------------------------------------------------------------------------
p = new Pipeline();                     //Create instance of pipeline emulation class
bDRAW = p->Initialise(m_pGlobalData);   //Initialise the pipeline
if(bDRAW) Ctl = 0;
else   Ctl = _STOP;

while(!(Ctl & _STOP))
{
      p->DataExchange();          //Performs Stack Data Flow Operation
      p->DoProcess();             //Performs functionality of each process element

      // Retrieval of pixel data from final process element
      Ctl    = p->GetOutStack()->Pop();
      blue   = p->GetOutStack()->Pop();
      green  = p->GetOutStack()->Pop();
      red    = p->GetOutStack()->Pop();
      y      = p->GetOutStack()->Pop();
      x      = p->GetOutStack()->Pop();
}
//-------------------------------------------------------------------------
```

Figure 5: Code fragment showing the call to Initialise() and the framework operation

The process element functional blocks are derived as required from a base class called SxProcessElement that consists of:

- A set of core functions and data members that each derived class inherits. These include pointers to input and output LIFO data stacks and the functions Pop() and Push() for accessing and writing the data to and from these stacks.
- A pointer to a global data repository class that is provided in the constructor that can be used to provide access to any external data source.
- Three main controlling virtual functions Begin(), Go() and End() that the user is invited to override to include his/her own functionality.

The basic format of this class as presented to the user is shown in the code fragment in Figure 6. Note that it is laid out in a familiar 'Wizard' manner.

```
//-------------------------------------------------------------------------------
class YOUR_CLASS_NAME : public SxProcessElement
{
public:
                ...
                ...
                ...
                void Begin();
                void Go();
                void End();
                //Add your member functions below
                //i.e. void YOUR_FUNCTION_NAME
protected:
                //Add your member variables below
                //i.e. int m_YOUR_NAME;
};

void YOUR_CLASS_NAME::Begin()
{
        //TO DO: Obtain your variables from the input register
        //i.e. m_YOUR_VARIABLE = Pop();
}

void YOUR_CLASS_NAME::Go()
{
        //TO DO: Add you implementation code
        //i.e. m_YOUR_VARIABLE ++;
}

void YOUR_CLASS_NAME::End()
{
        //TO DO: Add your variables to the output register
        //i.e. Push(m_YOUR_VARIABLE);
}
//-------------------------------------------------------------------------------
```

Figure 6: Code fragment showing the layout of the Process Element Class

As can be seen, an implementation of this class will involve the inclusion of user created member variables that are added by the system to the header file. A pointer to a global data repository class is passed to each instance of the PE class via it's constructor where all 'once off' data load and initialisation procedures are then performed. The Begin(), Go() and End() functions of each process element are then called in turn by the DoProcess() function for each cycle of the pipeline emulation. The normal operation can be summarised as:

Begin() 'Pop'ing of data from the input stack onto the member variables

Go() Functionality of the process block. Instantiations of algorithm classes and processing of pixel data

End() 'Push'ing the required output data onto the output stack

The exact operations performed by these member functions are completely dependent on the specific implementation. However, the suggested method of implementation is use instances of framework independent algorithmic classes within a bare framework structure. The idea is to maintain a level of abstraction between the pipeline functional blocks and the actual algorithm classes. This is discussed further in section 5. The data flow modelling framework should be treated as a platform on which to build specific implementations.

4.2 The Data Stacks

The data stacks form the basis of the data flow mechanism. There are two for each Process Element, an input and an output stack. In order to give the software the flexibility it requires, the user implementing a new design or reorganising an existing one should have the freedom to alter the number and type of the data values

passing into and out of the process elements as required, with the minimum effort.

To this end the data stacks need to be dynamic in size and demonstrate a degree of data type independence. Dynamic sizing is achieved through the use of the Last In First Out (LIFO) stack operation with data pushed on and popped off the stack as needed. This is basically a linked list class structure. Pseudo data type independence is achieved by utilising our own user defined data type class in the data storage stacks. By applying this class to all data in the stacks and to the process element algorithmic classes as required, we introduce a consistency in the stack operation and provide a powerful means of data flow control. The data flow type class can be written to deal with numerous data structures such as floating point and fixed point operations.

In our implementation a class was created that allowed not only C standard *double* 64 bit floating point precision, but also an accurate custom floating point and fixed point precision modelling system that allows user defined bit width decisions to be implemented. Here the use of C++ allows overloaded operators to implement these operations in a completely seamless manner. Also the careful use of these stacks allows the pipeline to exhibit any degree of parallelism required.

4.3 Test Vector Generation

At this point we can see how this software environment can then be used to integrate with the VHDL design work being carried out. At the simplest level, the pipeline structure can be split open at any point to allow the inclusion of a VHDL test bench as shown in Figure 7.

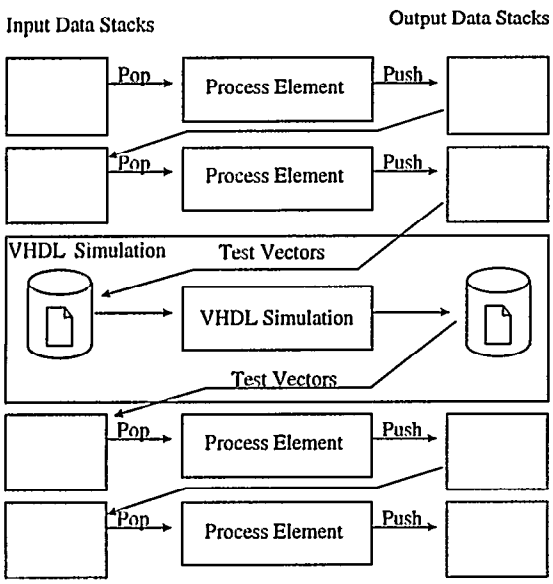

Figure 7: Integration of VHDL Test Bench with the Software Model

As well as just using the software to write out test data for the VHDL test bench at any desired point, a file sharing mechanism has been developed that allows the run time integration of the included portion of VHDL with the software model. Other approaches currently under development include the use of Windows sockets operations for network communication.

5 ALGORITHM PROTOTYPING ENVIRONMENT

The Algorithm Prototyping Environment (APE) is an application written in C++ based on the DFMF platform. It provides the user interface and algorithmic functionality of the Integrated Development Environment. The functional operation of each process element unit was performed using instances of algorithmic classes. Maintaining the algorithmic functionality and pipeline emulation functionality as distinct class structures both simplifies program maintenance and allows for the creation of a large component library of algorithm classes that are independent of any particular implementation. The future reusability of these classes is ensured through the use of abstract base classes for the general functionality of each algorithm block and by ensuring that all further functionality of derived classes is implemented with virtual functions.

5.1 Origins Of The Algorithm Prototyping Environment

The original specification for this pipeline implementation was to model an architecture under development at Sussex, the basic functional blocks of which are shown in Figure 4. The purpose of this work was to allow the investigation of several different algorithm developments in areas such as scan conversion, antialiasing and texture mapping. To do this the DFMF platform was expanded to include, as well as the basic implementation of this pipeline, the ability to select between the different algorithms and compare the results. To this end a comprehensive graphical user interface was developed from which numerous features have been and continue to be added. These include:

- Online choice of a large and comprehensive data base of models
- Selection between algorithms for scan conversion, texture mapping, level of detail calculation, antialiasing, etc.
- Ability to override by hand every mode of operation in the pipeline: antialiased edges, depth compare modes, blending modes, etc.
- Run-time adjustment of data path bit widths
- Production of output data files from different parts of the pipeline—communication with VHDL models
- Analysis of the performance of the different algorithms, both as a visual comparisons and as numerical/statistical output.
- With C++, multithreaded operations can be implemented to allow multiple simulations and run-time performance monitoring
- Link to Commercial packages—animation creation
- Image compare tools—image difference compare, zoomin, etc.

6 CONCLUSIONS AND FURTHER WORK

In this paper we have discussed the benefits of an integrated software-hardware codesign environment for the design of 3D graphics accelerator hardware. We presented details of a frame work for such an environment and a specific implementation, the Algorithm Prototyping Environment. Although this implementation is related to computer graphics, the principles discussed in this paper can be applied to other areas of hardware design.

The Algorithm Prototyping Environment software and associated tools have been of tremendous use to the Centre for VLSI and Computer Graphics and it's application and functionality continue to evolve. Several areas have been identified for future work:

- Development of full appwizard driven framework
- Drag and drop schematic capture methods
- Schematic creation by the software
- Provision of details on gate counts and performance figures for components based on particular technologies
- Combined VHDL/C++ library for verified components to allow group design experience to be retained and utilised in a far more efficient manner
- Use of above component library and schematic capture methods for generation of VHDL structural code
- Further interaction between VHDL and C++ data flow components.

7 REFERENCES

[1] Peter J. Ashenden, *The Designers Guide To VHDL*, Pub. Morgan Kaufmann, 1996.

[2] Rajesh Kumar, Gupta, *Co-Synthesis Of Hardware And Software For Embedded Systems*, Kluwer Academic Publishers, 1995.

[3] Paul S. Heckbert, Henry P. Moreton, "Interpolation For Polygon Texture Mapping And Shading", *State of the Art in Computer Graphics: Visualisation and Modelling,* Springer Verlag, New York, 1991, pages 101-111.

[4] *Joint MCC/OMI Hardware / Software Codesign Study Report*, 1996.

[5] J. Madsen and J.P.Brage, "Codesign Analysis Of A Computer Graphics Application", *Design Automation For Embedded Systems, pages 121-145*, Kluwer Academic Publishers, 1996.

[6] M. D. J. McNeill, M. White and P. F. Lister, "Graphics Codesign For Multimedia Systems", *Proceeding of the Eleventh International Symposium on Computer and Informations Sciences*, Turkey, November 1996.

[7] Giavanni De Micheli and Mariagiovanna Sami, *Hardware/Software Co-design*, Kluwer Academic Publishers, 1995.

[8] Juan Pineda, , "A Parallel Algorithm For Polygon Rasterization", In *Computer Graphics*, Volume 22, Number 4, pages 17-20, Addison Wesley (1988).

[9] Andreas Schilling, "A New Simple And Efficient Antialiasing With Subpixel Masks", In *Computer Graphics*, Vol. 25, No. 4, July 1991.

[10] J. Torborg, J. T. Kajiya, 'Talisman: Commodity Realtime 3D Graphics For The PC', *Computer Graphics Proceedings, Annual Conference Series,* 1996, pp353-363. The Chicken Crossing Presentation shown at Siggraph '96 Electronic Theatre.

[11] Martin White, Marcus D. Waller, Graham J. Dunnett, Paul F. Lister, and Richard L. Grimsdale, "Graphics ASIC Design Using VHDL", In *Computers and Graphics*, Volume 19, No. 2, pages 301-308, 1995.