Executive Summary

The fast processing speed and large memory bandwidth of the modern graphics processing unit (GPU) will make it a powerful tool in photo-realistic graphics programming and general purpose computing alike. Today's GPUs are highly programmable, accepting low-level assembly instructions for vertex and fragment operations. To improve compatibility over various platforms and simplify the programming environment, a number of high-level shading languages that target GPUs have been developed. As high-level shading programs become more complex, however, one must consider a serious problem: a program may exceed hardware limitations, such that it cannot be compiled in a single pass. These hardware constraints include the number of instructions, textures, constants, and storage registers. The GeForceFX 5200, for example, has a limit of 1024 fragment instructions and 65,536 vertex instructions, whereas a complex shading program often requires hundreds of thousands of instructions. If GPUs are to be used to their full potential, they must not be constrained by hardware resources.

A solution to the problem of resource constraints is the virtualization of hardware; that is, a mechanism must exist to allow high-level programs to use an indefinitely large number of resources. A 2002 paper from Stanford University presents the Recursive Dominator Split (RDS) algorithm for this virtualization. It partitions shading programs into multiple, hardware-appropriate passes while operating in polynomial time. Extending and replacing this algorithm are the Merging Recursive Dominator Split (MRDS) and Mio, which produce multiple outputs and perform priority-based instruction scheduling, respectively. While not widely used, RDS has been used successfully in shading languages like Ashli and Stranford's Real-Time Shading Language (RTSL). MRDS and

Mio have not yet been implemented.

Our project will involve writing and unit-testing the RDS algorithm, then the subsequent integration with the University of Waterloo's Sh complier, which currently fails for complex programs. The Sh compiler converts high-level code from Sh, a real-time shading language extending C++, into low-level code for programmable GPUs. Building on the Sh compiler will allow us to work on a solution to the virtualization problem without having to design an entire shading language. We will also consider extensions of the RDS algorithm: either expanding our code to handle multiple program types or implementing newer virtualization algorithms like MRDS or Mio. Additionally, if our project succeeds, our code will be released as part of the open-source Sh distribution.

Since the project consists only of software design, there are no requirements for funds and special facilities. The hardware required is a modern graphics card: an ATI card more recent than the Radeon 9500 or a NVidia card more recent than the GeForce FX 5200. One of the group members already has the necessary hardware, and all of the software being used for development is freely available from Waterloo.

Contents

1 Project Proposal

1.1 Introduction

1.1.1 Programmable graphics processing units

The graphics processing unit (GPU) has the potential to be a powerful tool in photo realistic graphics programming and general purpose computing alike. Non-programmable GPUs could perform only predefined algorithms and relied heavily on the control processing unit (CPU); however, modern GPUs are highly programmable and have been designed to perform certain computations, like vector operations, at very fast speeds. NVIDIA Corporation's recent GeForceFX 5200 operates at 350MHz, processing 68 million vertices per second and up to 8 pixels per clock cycle. The memory bandwidth of newer units is also becoming increasingly large: the FX 5200's is 6.4GB per second[**?**].

GPUs themselves accept low-level, assembly instructions. To improve compatibility over various platforms and simplify the programming environment, a number of high-level shading languages that target GPUs have been developed. Shading languages describe the effects – including texture, lighting, and colour – to be applied to each vertex and fragment (pixel) in a scene. These shaders allow large programs to be written without the use of hardware-specific instructions, such as moving values into registers. When compiled, a shading program is mapped to traditional low-level code, which is then processed by the GPU.

While attention to low-level details may be decreased, high-level shaders are still not free from the restrictions of hardware. GPUs have a limit on the number of assembly instructions, textures,

constants, and storage registers that can be used in one pass. The limits of some GPUs currently on the market are shown in Table 1. The GeForceFX 5200, for example, can handle up to 1024 pixel instructions and 65,536 vertex instructions[**?**]. In contrast, a complex RenderMan shading program, which relies primarily on the CPU, may require hundreds of thousands of instructions[**?**]. If GPU shading languages are to handle programs of such complexity, they must not be constrained by these hardware limitations.

Hardware Limits For Fragment Shaders

	GeForce4	Radeon 9500	GeforceFX
Instructions	128	16	16
Static Instructions	4	32	1024
Registers	8	64	1024

Hardware Limits For Vertex Shaders

	GeForce4	Radeon 9500	GeforceFX
Instructions	128	1024	65536
Static Instructions	128	256	256
Registers	12	12	16
Constants	96	256	256

1.1.2 The Sh shading language

One GPU shading language currently in development is Sh. Part a project lead by Michael McCool at the University of Waterloo, this language is an extension of C++, allowing vertex and fragment programs to be written in familiar, high-level code. Programs are compiled into intermediate Sh assembly instructions, which are further compiled into GPU-specific instructions[**?**]. The Sh

compiler is single pass, and consequently, it fails when a program surpasses the available hardware resources. This is a severe drawback of Sh, especially when compared to recent multi-pass shading projects like Ashli and Standford's Real-Time Shading Language (RTSL).

1.2 A multi-pass solution

1.2.1 Hardware virtualization

Graphics hardware continues to improve, but single-pass rendering will never allow for programs of arbitrary complexity. To overcome the problem of resource constraints entirely, hardware must be virtualized; that is, a mechanism must exist to allow high-level programs to use an indefinitely large number of resources.

A 2002 paper from Stanford University presents an algorithm *the Recursive Dominator Split (RDS)* for this virtualization by partitioning shading programs into multiple, hardware-appropriate passes. RDS aims to minimize the number of partitions, while operating in polynomial time[**?**]. While not yet widely used, RDS has been successfully integrated with both the Ashli and RTSL compliers.

More recent papers present additional algorithmic solutions and extensions to the virtualization problem. Standford's Merging Recursive Dominator Split (MRDS) builds upon RDS to generate multiple outputs[**?**]. Mio, suggested in a paper at the University of California, finds optimal partitions by scheduling instructions according to minimum remaining running time[**?**].

1.2.2 Project objective

With this solution in mind, our primary goals include:

- implementing the RDS virtualization algorithm
- integrating our code with the Sh back-end for stream programs
- verifying that our code finds nearly optimal partitions
- implementing an extension to our code, improving flexibility or performance

1.3 Technical Methodology

1.3.1 RDS

For the implementation of our virtualization extension we will use the RDS algorithm. The RDS algorithm, although relatively new, is acknowledged by the industry and academia having been implemented by Stanford in their Real Time Shading Language compiler and by ATI in their Ashli compiler.

This algorithm operates by organizing the program into a tree structure to define dependencies between instructions. This tree is partitioned into sub-trees, with each sub-tree representing a rendering pass. The output of a rendering pass will be saved and passed to future passes to reconnect dependent nodes.

RDS has a complexity of $O(n^3)$ which generates multi-pass shaders which are very close to optimal

(5% on average). There is a second algorithm presented, RDS_h, which uses a heuristic to partition the program. This decreases the complexity to $O(n^2)$ at the expense of the shader's performance. We will implement both, since it is very simple to replace the code from the original algorithm. This will give users the option of using RDS_h for debugging to speed up the compile time.[**?**]

1.3.2 Sh

Once the virtualization extension is written, we will need to attach it to an existing compiler. The Sh compiler seems to be a good choice: it is an open source project being developed by the university of Waterloo. This also means that if our project is successful it may become part of the Sh compiler.

In its current state Sh will not virtualize any resources, the resulting shader will just break when it tries to access non-existent resources. We will need to add checking for the hardware limits to Sh. Some other changes will also be necessary. Sh currently has one back-end, it uses OpenGL to issue instructions to the graphics card, but this may change in the future. Since Sh is a higher level language than typical shader languages (such as OpenGL's), it would be inefficient to try to partition instructions at that level. What we will do is to use an intermediate representation of low level Sh instructions, and partition the program before it is translated and sent for execution on the appropriate back-end. Fortunately, it has been the intent of the Sh developers all along to have an intermediate representation so there is already a considerable amount of code written.

Our first objective will be to implement RDS and integrate it into Sh stream programs. Stream programs are executed as normal programs, issuing commands to the graphics card as necessary.

This makes it a simple matter to buffer the commands and partition them. Sh also supports two other types of programs: vertex and fragment. These programs are loaded onto the video card, and are executed by the card when other graphics commands are issued (e.g. a metal shader would be loaded onto the graphics card, then a sphere is drawn and the graphics card executes the metal shader when necessary). In this case we will need to determine how one shader can be executed before the others. We are considering this as a possible extension to our project.

1.3.3 MRDS and Mio

Another possible extension for our project would be to implement a different partitioning algorithm. Recently, two papers were published which propose algorithms for partitioning on hardware which supports multiple outputs (this means that more than one result could be saved and used in the next rendering pass). The original RDS algorithm was conceived when graphics hardware was limited to a single output per pass.

MRDS (Merging Recursive Dominator Split) builds on top of the RDS algorithm. By adding one extra layer of computation, all the sub-trees from the RDS partitioning pass are considered for re-merging. The complexity of the algorithm increases to $O(n^4)$[**?**], however the resulting code performs very well. Also, it would be fairly simple to add an extra layer to the existing code rather than to write a new algorithm from scratch.

Mio has a very different focus than RDS: the creators of Mio suggest that minimizing the number of partitions is not as important on modern hardware. They suggest that minimizing the number of instructions will result in faster code, they also claim that as hardware becomes more capable

it will favour Mio over RDS. Mio has a complexity of $O(n \cdot log(n))$, so it compiles quite quickly, however it was not able to outperform the RDS algorithm in all cases.[**?**] Mio does present another opportunity: since the algorithm is different from RDS there are still options of optimization which have been suggested but not implemented by the authors. We will be looking into this as an option.

1.4 Work Plan

Our project is partitioned into three large tasks. The first task will be implementing the RDS algorithm, the second will be to integrate our work into the Sh compiler and the third will be to extend our implementation by adding more functionality or by increasing its compatibility. At the beginning of the project, there will also be some minor tasks: setting up a repository for our code and installing the Sh code on our machines.

Although the implementation of RDS and the integration into Sh could be done sequentially there are compelling reasons to work on them simultaneously. By working in Sh from the beginning we get the benefit of having an existing test suite to work from: Sh has a set of regression tests to determine if recent changes have broken any functionality. Also, there will not be any need to create sample programs for a standalone RDS library since we will have all the Sh demo code available. Merging these tasks does have a drawback: we will be learning Sh at the same time that we will be implementing RDS and this is certain to cause some confusion, however we hope it will be minimal.

The implementation of RDS will begin by creating a set of utilities for the dominator tree data structures. The data structures are not very unique and the methods are quite well documented in the papers we have read. After the data structure code is ready we can begin to convert Sh

programs into trees, this is an area where we may be able to build on something currently in the Sh code base. The next piece will be the actual partitioning module, this will probably be the most complex part of the RDS implementation. We will need to perform all the splitting and merging and writing the code for the heuristics used. Despite having testing utilities available from Sh, we are planning on writing unit tests for various pieces of RDS.

Integrating with Sh will probably be more challenging, at least initially. We will need to go through Waterloo's code, which in its current state of documentation leaves much to be desired. We will need to work on the back-end side of Sh and it is written in a nice modular fashion so we can abstract out the front-end. Rather than trying to code RDS directly in the Sh back-end, our plan is to add stubs to the existing code and build our code on top of that. Although this may involve some refactoring of the code later, it will be much simpler to maintain a branch of the code this way. The stubs will be added to the PBufferStreams (stream program) target in Sh, and we may also need to finish writing the ShTransformer which converts Sh code into the intermediate low level code.

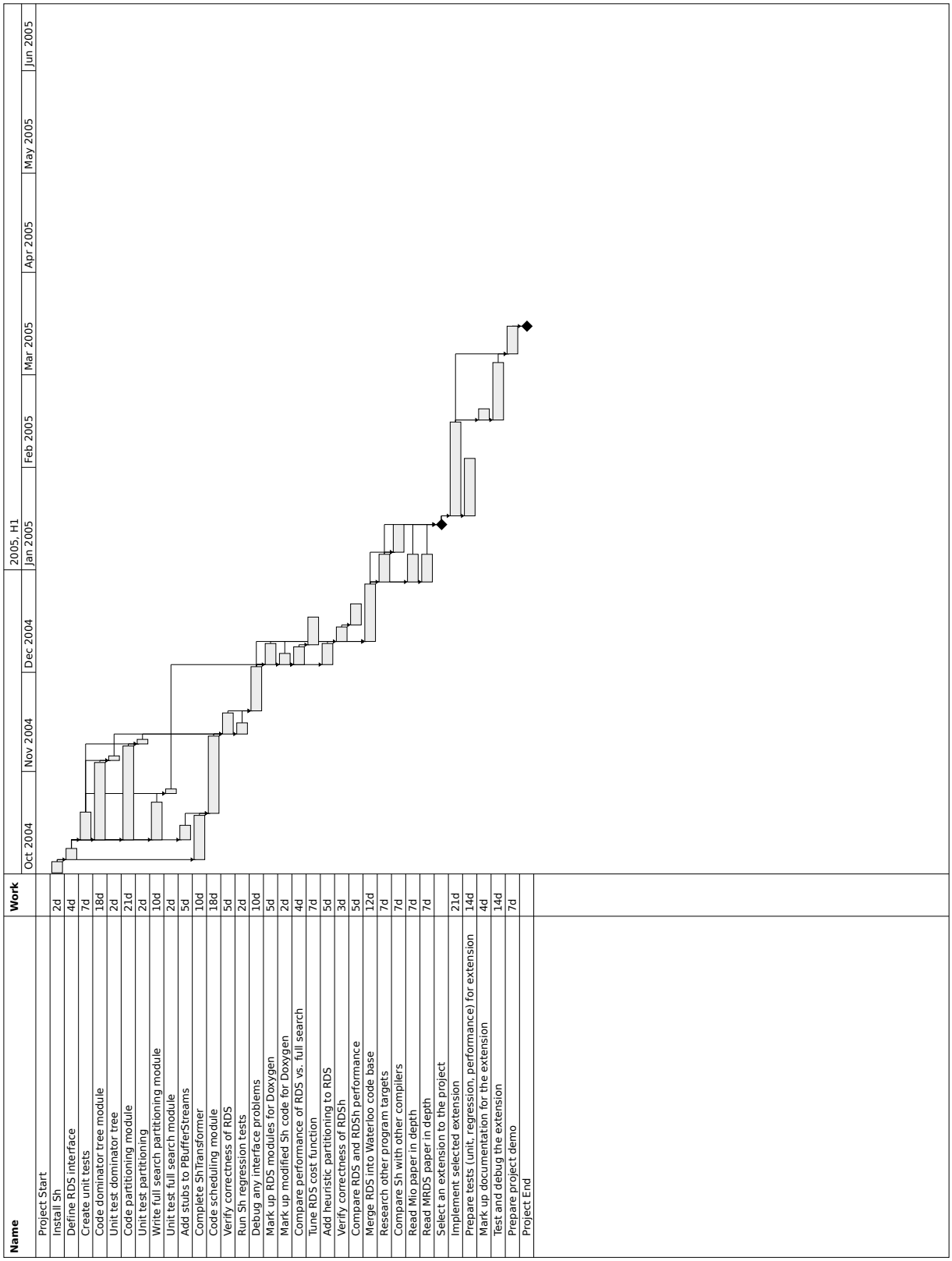
The extension of our project is still somewhat unclear, and we have several options to choose from at the moment. What we intend to do once we have finished integrating RDS into Sh is to look at some of the other shading language compilers, particularly Stanford's and ATI's. After that we will consider the feasibility of implementing one of the multiple output partitioning algorithms (Mio or MRDS). We intend to select one of these options and extend our project.

By making certain choices we have, as a side effect, decided which tools we will be using. For version control we will use Subversion, as this will make it easier to merge our code later. Sh is written in

C++ so we will be using C++, fortunately it seems ideally suited to writing compilers so we may have chosen it ourselves. Also, Sh uses Doxygen to create documentation and we will be doing the same.

Testing is also a case where circumstances have imposed a certain route. Only modern graphics chips are programmable, so all performance testing will have to be done by Aly (who has an NVidia FX series card). Although, there is an emulation back-end for Sh we are not sure of its status so all verification testing will also be done by Aly. This will mean that Cynthia will be responsible for writing the test cases and Aly will be responsible for executing them.

In terms of skill breakdown both of us are comfortable with C++ and have a strong programming background. Aly has an extensive programming contest background so he is quite familiar with the graph theory concepts behind the partitioning algorithms. He has not used C++ extensively, but has 9 years of experience with C. He also has some familiarity with OpenGL. Cynthia has taught game programming in C++, so she is quite familiar with both the graphics concepts and the language we will be using. Both of us are also taking many related courses such as compilers and interpreters, optimizing compilers and computer graphics. In terms of division of labour neither of us appears to be suited to a particular task, so any division is purely artificial.


Name
Work
2005, H1
Oct 2004
Nov 2004
Dec 2004
Jan 2005
Feb 2005
Mar 2005
Apr 2005
May 2005
Jun 2005
Project Start
Install Sh 2d
Define RDS interface 4d
Create unit tests 7d
Code dominator tree module 18d
Unit test dominator tree 2d
Code partitioning module 21d
Unit test partitioning 2d
Write full search partitioning module 10d
Unit test full search module 2d
Add stubs to PBufferStreams 5d
Complete ShTransformer 10d
Code scheduling module 18d
Verify correctness of RDS 5d
Run Sh regression tests 2d
Debug any interface problems 10d
Mark up RDS modules for Doxygen 5d
Mark up modified Sh code for Doxygen 2d
Compare performance of RDS vs. full search 4d
Tune RDS cost function 7d
Add heuristic partitioning to RDS 5d
Verify correctness of RDSh 3d
Compare RDS and RDSh performance 5d
Merge RDS into Waterloo code base 12d
Research other program targets 7d
Compare Sh with other compilers 7d
Read Mio paper in depth 7d
Read MRDS paper in depth 7d
Select an extension to the project
Implement selected extension 21d
Prepare tests (unit, regression, performance) for extension 14d
Mark up documentation for the extension 4d
Test and debug the extension 14d
Prepare project demo 7d
Project End

1.5 Financial Analysis

Finances should not be a problem for our group as we already own all the necessary equipment required. Instead, we will use this section to outline which resources we will be using and where we can find them.

Sh compiler with source code	Available from `http://libsh.org`
C++ compiler	Using gcc which is free software
Test system	Aly's system: OS - Linux, Graphics - NVidia FX5200

1.6 Technical risks

A major technical risk is the failure of our code to successfully integrate with Sh. To avoid this, we will communicate regularly with the developers at Waterloo, such that we are aware of any significant changes to the Sh code. Additionally, to minimize error in our stand-alone code, our work plan involves writing our algorithms in small, thoroughly tested parts. If, however, our code still does not integrate, then the purpose of our project will be changed to the successful implementation of RDS itself.

A minor risk pertains to legal issues. Since the Mio and MRDS algorithms are very new, there is the possibility that they will be patented during the course of our project. If this occurs, we will either choose to implement another algorithm or not release our code with the Sh distribution.

1.7 Benefits

1.7.1 Computer industry applications

Multi-pass rendering comes with several benefits. Firstly, an unlimited number of textures and instructions will allow GPUs to generate photo-realistic graphics in real time. This will serve useful in such fields as film special effects, virtual reality, and scientific imaging.

Unlimited resources will also provide an advancement to arbitrarily large general purpose GPU programs. GPUs are being used for non-graphics applications like linear algebra and scientific simulation, and their high speeds make them ideal co-processors for the CPU. Once hardware virtualization is fully established, the computing possibilities for GPUs will be endless.

Little attention to hardware details and limitations in shading programs will increase their portability over various units dramatically. High-level programs of an arbitrarily large size will be broken down to accommodate the resources of virtually any programmable GPU.

1.7.2 Improvement of Sh

Successful integration of our code with Sh will greatly improve its imaging capabilities, as the system will be able to handle the most complex of shading programs. Our work will be released as open source along with the Sh distribution.

2 Schedule A - Self Assessment Form

1. UNDERSTANDING OF THE TECHNICAL PROBLEM AND ITS APPLICATION CONTEXT

   C: The team has completed a detailed analysis of the technology that demonstrates they fully understand the technical problem and the context of the application.

   GPU specifications were provided to show how hardware can limit the complexity of shading programs. Some more examples to contrast GPU and CPU shaders could be provided.

2. PROJECT OBJECTIVES/PROPOSED SOLUTION: ADVANCING THE STATE OF THE ART

   B: An advancement of one aspect of existing technologies or an application area.

   Though not widely used, RDS has been implemented with other projects. After writing the RDS code, we will attempt to integrate it with the Sh compiler. We will also work on extending RDS using algorithms that have not yet been implemented.

3. TECHNICAL METHODOLOGY

   C: The area of virtualization for shading languages is still very new; however we covered most of the existing algorithms.

4. WORK PLAN

   B: A detailed Gantt chart was included, but the work breakdown was not completed. The Gantt chart does contain dependency information.

5. FINANCIAL PLAN

   C: Our project requires no financial support. Explanations were given for what resources we do need and how we will acquire them.

6. TECHNICAL RISKS, INCLUDING RISK MITIGATION

   B: The proposal has identified and outlined the technical-risks likely to be faced by the project and has presented a rudimentary risk mitigation plan.

Some risks and mitigation strategies were mentioned, but more details about the likelihood and overall impact of the risks could be provided.

7. MARKET ANALYSIS, BENEFITS TO CANADA/SOCIETY

B: Provided limited evidence to suggest that there is a market need for their proposed technology and some economic benefits to Canada or to society. An understanding of the need for the proposed technology and an initial understanding of the scope and scale of the markets.

We gave some examples of where multi-pass rendering may serve useful in the graphics industry and in general purpose computing. Some more specific examples of where complex GPU programs would be needed would more effectively illustrate its benefits. Additionally, a more in-depth discussion on how our work will benefit Sh itself and any impacts of an improved version of Sh could be provided.