Application of Transmission Line Matrix Theory on Modern Day Graphics Processing Unit Hardware

BSc Software Engineering Dissertation

Robert Jones

Project

Project Details

Student:	Robert Jones
Course:	BSc Software Engineering
Module:	3rd Year Project
Title:	Application of Transmission Line Matrix Theory on Modern Day Graphics Processing Unit Hardware
Date:	September 2006 – May 2007
Supervisor:	Steven Harris
Module Leader:	Fran Zbrozek

Acknowledgements

I would like to start my acknowledgments with my thanks to both Kristian Dor and Steven Harris; without their previous work this project would not have been possible in the first place.

I'd like to thank all the people on gamedev.net who have taken an interest in this project and who have kept reading my journal on there as this work has progressed. While I don't know who you all are I'd like to say a special thanks to those of you who took the time to make some comments and report my broken test application to me; Jack, Rick, Mike, Mars_999, Zed, Giallanon, Vilio, jjd, mrbastard, paulecoyote, Mushu and aidan_walsh.

I'd also like to thank those who managed to get the test application going, even if I didn't use their results in the end due to some last minute changes; Jack (again!), Garrick Chin, Rasmus Johansen and Rob R. Thanks guys, your time and effort was much appreciated.

I'd like to thank my various friends for keeping me at various levels of sanity and generally dealing with my bizzareness during the past 9 months (and beyond in some cases), in perticular; Craig, Claire, Claire, Seb, Jamie, Martin and Paula.

The Monday Night crew in general get a mention for some truly bizarre and cool nights out, all of which helps keep me relaxed (if not insane). *salutes*

More bands than I care to remember for producing some excellent music which I've been coding and writing to over the last few months; I really do need that background noise.

The Suffolk College Software Engineering group 2007; Ed, Rob, Kyle, Luke and Ian.. It's been surreal guys, always remember the Love Cheese!

And finally, my parents for supporting me during another 2 years of me having no cash and for putting me though uni the first time, Thanks a million ☺

Abstract

The Transmission Line Matrix (TLM) Theory allows for the simulation and modelling of wave propagation, primarily electromagnetic in nature. While it has been in use since the 1970s to date this simulation has been performed on the CPU of a computer system and due to their heavy computational nature are slow to complete, indeed for large matrices a real time solution might not even be possible.

Building upon the work carried out by Steve Harris and Kristian Dor in previous years this project aims to investigate the practicality of off loading the work required to perform the TLM calculations and mesh regeneration from the CPU to a graphics processing unit (GPU) to take advantage of their parallel processing power.

In order to gain an understanding of how the calculation speed is effected by the amount of parallelism used 5 different methods are tested ranging from CPU only versions to a final fully GPU based solution.

Each solution is encapsulated within its own class for clarity of programming and testing and each solution is tested at a number of different mesh sizes ranging from 40 by 40 to 1024 by 1024.

Table of Contents

List of Figures

Abbreviations

API –Application Programming Interface

D3D – Direct3D

DX – DirectX

GLSL – OpenGL Shading Language

GPU – Graphics Processing Unit

SDK – Software Development Kit

TLM – Transmission Line Matrix

GPGPU – General Purpose GPU

OGL – OpenGL

Introduction

Until recently the computing power of a system has been limited by the CPU, indeed if more power was required to work on a problem the general solution was to add more CPUs to a system or join multiple systems together to form a cluster of computers; each of these machines or CPUs would then working on a subset of the problem until a solution was found.

However, while this method can prove effective is has a number of draw backs, not least of which is the cost of implementation and the hardware required to cluster the machine as well as custom written software to manage the load balancing across the cluster to ensure optimal performance. These factors alone place the clustering of machines beyond the feasibility of many people or indeed college and university departments.

The rise of the graphics processor unit (GPU) however has effectively removed part of this barrier to entry, as is explained in more detail later these chips are effectively clustered ALU units, each capable of operating with a degree of independence from each other. This has given birth to a new area of computing; General Purpose GPU or GPGPU, an area which is dedicated to using the potential power in current and future GPUs to speed up calculations which would otherwise require far more computing power than a CPU alone can provide.

It is this potential power which has being investigated within this project and its application when it came to calculating the results of energy being placed into a Transmission Line Matrix (TLM), including the subsequent visualisation of this matrix.

Objectives

While some work has already been done on using the TLM to generate dynamic surfaces, in particular Kristian Dor [Dor, 2006] and Steven Harris [Harris, 2005] at Suffolk College, this work has been purely CPU based and thus the size of the matrix which can be calculated is somewhat limited.

The base of the following work is Steven Harris' test application which simulates a wave moving across a surface. Kristian Dor then carried out an extension of this work which moves the wave generated from a 2D coloured image to a full 3D representation of the wave moving across a number of surfaces using a number of different methods to show this simulation.

This work moved on a stage further in utilising the GPU in a system to carry out the calculations required to generate this mesh data.

The main question this work looked to answer was that of; "**What are the potential gains of using a GPU to carry out the calculations required to fully visualize a TLM matrix in 3D, and how does it compare to a CPU only solution?**"

To carry out this comparison a number of methods were developed to perform the generation of the TLM matrix and subsequent mesh for visualisation, each of which is covered in more detail later in this document

Research Areas

The areas of research required for this project were somewhat interdependent on each other. One area which didn't require much research however was that of the TLM itself. Thanks to the work done by Kristian Dor the details of the TLM and how it is implemented in code was already cleared explained, all that was required was the understanding of the theory and the implementation details from his work.

Due to Kristian's work being based in C# and Managed D3D some research was required into these areas to understand his code to allow for reimplementation or translation into another language and 3D API as was required; this led to a requirement to select a language and 3D API to carry out the project in.

Beyond the API and language side of things the biggest area of research was how to use the GPU to perform the calculations themselves which required research into both how it can be done with both APIs and some understanding of the underlying hardware so that the

limits of what was possible could be assessed. This required investigation into the various languages used to program GPUs and the supporting APIs built into the 3D APIs which could be used.

Deliverable

The deliverable in this project is a simple program to perform a number of tests on the various methods utilised to generate the mesh.

The test application is fully automated so that you can set it running and it will perform the given tests at various mesh sizes and produce text files with the details on execution times for each method-size combination.

This application was a newly written application based on the previous code by Kristian Dor but not directly utilising any of it directly due to the language and API that was selected for development.

Background

This project can be broken down in to 3 major component parts:

1. What is a Graphics Processing Unit (GPU)?
2. What is the Transmission Line Matrix (TLM) Theory?
3. How can the TLM be implemented on a GPU?

This section serves to provide some background information on these subjects so that reader can follow later discussion in this document.

What is a GPU?

A GPU or Graphics Processing Unit is piece of hardware designed primarily with the fast display of 3D graphics. Modern GPUs are an evolution of older technology designed to display graphics.

The history of the video card starts back in the 1960s when visual displays were introduced to replace printers as a means to generate output. However it wasn't until 1981 that the first video card was released with the first IBM PC; it had 4KB of memory, just one colour, and could only work in text mode at a resolution of 25*80 characters. Over the next 9 years several video cards were produced, each increasing the resolution and capabilities of what could be displayed.

However at this point graphics technology was still very primitive, dealing only with the displaying of text and simple graphics; it wasn't until 1991 that the first accelerating cards became common on the PC. These cards had dedicated chips to speed up the 2D rendering of window and other 2D graphics.

The mid-90s was when the first 3D accelerating chips started to appear on the market; however the first generation of these chips were simply 2D chips with 3D capabilities bolted on to them. They initially failed to catch on however the ground work for the subsequent explosion of 3D hardware was laid. A company called 3dfx released an add-on board which allowed for faster generation of 3D graphics via the now obsolete Glide API; this board was

called the Voodoo Graphics Card and was designed to accelerate the drawing of 3D based graphics; however most of the setup work was still done on the CPU.

This trend of chips which could speed up the rendering process continued and advancements, such as being able to render two textures to a given pixel at once, developed over time. However, it wasn't until 1999 that the largest initial advancement was made with the release of the GeForce 256 card by nVidia; this card represented a major step forward as it could perform the transformation of 3D objects and their lighting calculations on the chip before rendering them, this effectively freed the CPU from a large amount of work associated with 3D rendering.

While this was a significant step forward for 3D graphics, the functionality of these chips was still very much fixed. While simple aspects such as the colour of an object or a texture could be changed the overall pipeline was still very much fixed. The next key step didn't happen until 2001 with the introduction of nVidia's GeForce 3 series of cards and DirectX 8.1 (DX8.1); this revision saw a key feature added both to the API and the hardware; programmability.

With this release the application programmer, via the 3D API, had some control over the vertex data setup stage and the pixel processing stage. In the initial cards this was very limited; both by capabilities of the hardware and the speed of it, however with this series of chips the programmable graphics card age had arrived.

At this point the chips had two key abilities:

1. To be able to adjust vertex data
2. To control pixel processing

Vertex data is a position in 3D space coupled with other attributes, such as colour and coordinates to perform a lookup into a texture (which is just a region of memory representing image data). The original fixed functionality performed some basic tasks, such as transforming the position into what is known as 'eye space' so it can be seen and

performing calculations to allow the vertex to have lighting applied to it. With the advent of the vertex programmability this functionality could be replaced with the programmers own code to develop different lighting conditions or change how the position was processed.

The pixel processing stage is responsible for deciding what is displayed on the screen; with the original fixed functionality chips this was something simple such as sample a texture for some colour information and multiply it with the value calculated for lighting. With the advent of programmability this model could be changed to perform slightly more complex calculations.

However, at this point the programmability was still very basic, a year later things improved once more though with the release of the DX9 specification which granted much improved programmable control over the hardware. This specification allowed for longer programs to be loaded for processing both vertex data and pixels by the hardware as well as previously missing abilities such as being able to loop or jump over instructions during processing. The specification also for the first time introduced the idea of 'floating point' textures to main stream hardware.

Until this point all GPUs produced could only use textures which were at most a total of 32bit in size, with 8 bit allocated for Red, Green, Blue and transparency or Alpha information. With the release of the DX9 specification this restriction was lifted to a minimum of 24bit per colour channel allowing for 96bit textures or even 128bit if the hardware supported it. The coupling of floating point textures and the ability to write floating point data to them from the GPU, along with the ability to load longer programs is what first fuelled the General Purpose GPU or GPGPU excitement.

During this development process a critical series of steps was made; the parallelising of the GPU core. The calculations for 3D graphics are inherently parallel in that the result of one calculation doesn't affect the outcome of another at the same stage of the pipeline. With this in mind GPUs became effectively small parallel computers with multiple Arithmetic-and-Logic (ALU) cores for different stages of the pipeline. For example, after the DX9

specification was release a company called ATI released a graphics chip named the R300, this chip had 4 ALUs dedicated to vertex calculations and 8 ALUs dedicated to pixel calculations; this means that effectively the GPU can process 4 vertices and 8 pixels at the same time, compare this to a CPU which generally can only work on once piece of data at a time [1] and you should begin to get an idea of the kind of speed increase which can be gained.

Subsequent hardware releases have only served to improve the GPGPU interest, mostly due to the ever increasing instruction counts on programs, the speed of the processors, the increases in parallelism (the latest ATI chip, the R580, has 12 vertex pipelines and 48 pixel pipelines) and the ability to write to multiple 128bit textures at a time.

It's this rise in GPGPU processing and the increasing power of GPUs relative to CPUs which has led to games off loading parallel calculations to the GPU for processing and it is this power which can be utilised to perform the TLM calculations to generate a mesh for rendering.

What is the TLM Theory?

The TLM Theory was first put forward by Peter Johns and Raymond Beurle in 1971 and was used mainly as a method of modelling electromagnetic forces.

In its simplest terms, the model is a matrix of nodes joined to their immediate neighbours by means of lossless transmission lines [Christopoulos, 1995]. Using these interconnected nodes forces can be sent between nodes, with each node passing the force on to its neighbours at half the strength it received it at.

[1] Newer chips do have multiple cores or Hyper-threading technology which allows for multiple threads of code to be carried out at once; however these require significant programmer effort to utilise where as the parallelism of GPU is effectively free.

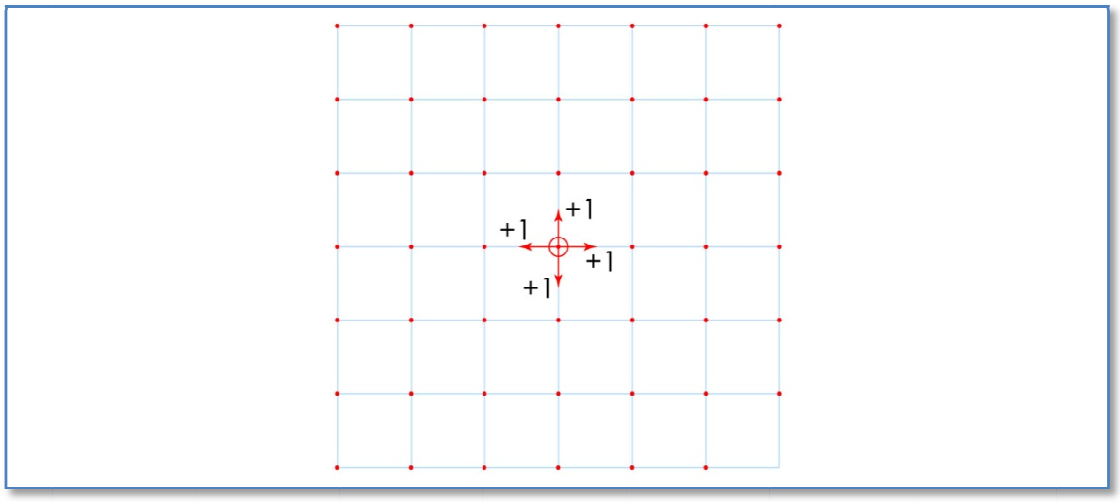


Figure 1 Transmission Line Matrix Showing Initial Excitation [Christopoulos, 1995]

As the above figure shows the initial input into the TLM is distributed as a force of 1 to its surrounding nodes. These nodes then retransmit that force at one half the strength to their surrounding nodes, however as indicated by figure 2 below the force returned to the source node of the previous step of the simulation is negated as it is a reflection of the incoming force.

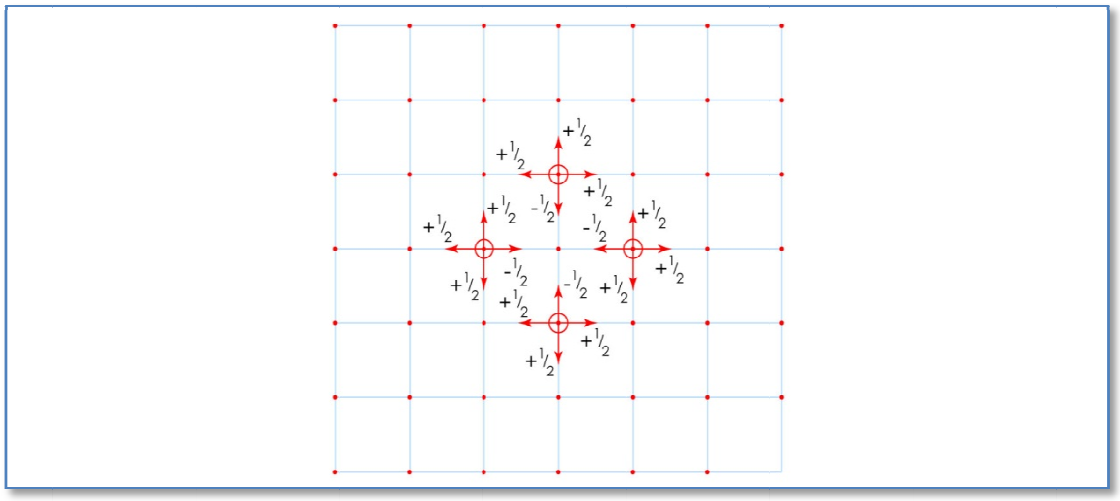


Figure 2 Transmission Line Matrix Showing First 'Scattering Event' [Christopoulos, 1995]

The four nodes highlighted in the above figure are now the leading edge of the simulated wave front, a point which becomes clearer after the second 'scattering event' occurs, at

which point the wave front has moved on another set of nodes and four ‘shunt’ nodes are introduced (highlighted in blue) which move a force of equal magnitude in the same direction the wave front is travelling.

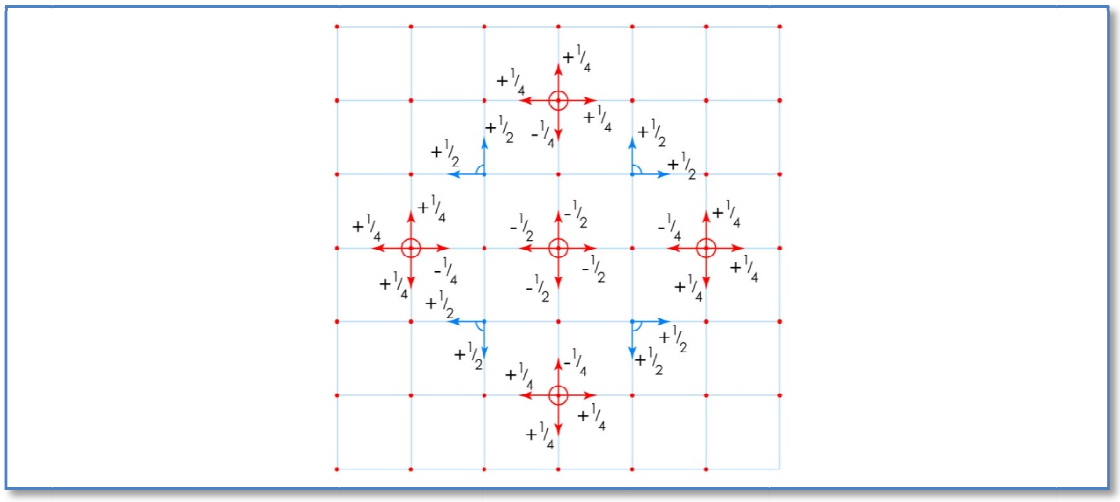


Figure 3 Transmission Line Matrix Showing Second ‘Scattering Event’ [Christopoulos, 1995]

These shut nodes come about due to the forces being transmitted around the matrix, the resulting scattering forces are either multiplied or cancelled out, leading to a wave front moving outwards from the initial point of insertion of force.

This 2D simulation can be run over multiple iterations in order to observe the wave fronts properties and it is this wave which will be used to generate a mesh on the GPU with the wave moving across it.

How can the TLM be implemented on a GPU?

As previously mentioned, due to their programmable nature, GPUs have move beyond simply providing accelerated 3D graphics and into the area of general purpose calculations, particularly ones which can be parallelised effectively. There is no better example of this

than the Folding@Home Project[2] which in 2006 released a client which made use of the ATI GPUs as processors for its data processing needs; this was seen as a major step forward for their processing abilities and the speed with which data could be analysed.[Folding, 2006]

The Folding@Home project makes use of a piece of technology called Close-To-The-Metal (CTM), which was released by ATI to allow programmability of their GPUs without using a secondary API such as OpenGL (OGL) or Direct3D (D3D) for the processing[3]. However, while this does afford great control of the GPU for processing the API doesn't have any functionality for 3D rendering, it instead is designed purely to allow for low level communication and control of the GPU. This is functionality more suited to offline simulation work than real time 3D graphics.

As such, for the primary purposes of this project a select had to be made between either OGL or D3D as the rendering technology.

However, in both cases the general method of working with the GPU is the same, the only difference is the API calls and support data used to achieve the required output.

When it comes to utilising both APIs for GPGPU operations there are 3 key areas to be concerned with:

1. Vertex data
2. Textures
3. Shaders

Vertex Data

Before anything can be drawn by a GPU it has to have specified its location and any other associated attributes, such as colour or a normal. Collectively a unique group of this information is simply known as 'vertex data'.

[2] http://folding.stanford.edu/
[3] http://ati.amd.com/companyinfo/researcher/documents.html

When it comes to GPGPU programming the vertex data sent is generally quite limited, consisting of a location and a texture coordinate pair. The location information is used to define the extent of the data the GPU should work on. For example, if your source data exists in a 512 by 512 array then the locations specified would generally define a square stretching from (0,0) to (512,512).

Texture coordinates are used to retrieve the information which is being processed from one or more textures which are specified for input.

Textures

Simply put textures are 2D arrays of data which exist in graphics card memory. In GPGPU processing they are both a source of data to be processed and a sink for the finally computed output. One important restriction with current generation hardware is that one texture cannot be both a source of data and a sink of data at the same time; doing so results in undefined behaviour.

Textures can be effectively any size up to a maximum defined level per graphics card; for example the ATI R580 based cards have a maximum texture size of 4096*4096. Originally there was a restriction on textures which required their dimensions to be a power-of-two (so a 128*256 texture was valid, but a 128*257 wouldn't be as the second dimension wasn't a power-of-two), however recent hardware lifts this restriction while enforcing others.

Texture data can also be in a number of various formats:

- 32bit floating point
- 16bit floating point
- Integer formats

The formats themselves can come in 1, 2, 3 or 4 component versions depending on how much data needs to be stored.

When it comes to GPGPU processing one important concept is that needs to be take into account is that of 'pixel centres'. Due to GPUs being used primarily for 3D graphics

operations the texturing system is setup in such a way that accessing the texture data at (0,0) doesn't return the texture data (or Texel as they are known) from that location, instead it effectively returns a filtered version from (0.5,0.5). If this is not taken into account this is can cause a problem when accessing data for GPGPU calculations as a filtered value is generally undesired. However it is possible to switch off this filtering for textures, so that (0,0) returns the unfiltered value at cell (0.0) as it would in a C or C++ problem, this functionality is provided by the graphics API.

Textures are the only way of getting per-cell information into a shader to be processed.

Shaders

Shaders are the programs of the GPU world, their name comes from their original use with the RenderMan program, used by companies such as Pixar to produce animated films (for example Cars), where they started life as small section of code used to change the properties of a surface. With the advent of programmable GPUs the name came with it.

Shaders come in three forms:

1. Vertex
2. Geometry
3. Pixel

Each of these is active at a different stage in the rendering pipeline and can perform different actions. However, for this project we are only using the Vertex and Pixel Shaders due to lack of hardware which can implement the Geometry Shaders.

By translating the various stages of the TLM calculation from traditional CPU code (such as C++ or C#) into shader code we can have the GPU perform the calculations for us and utilise the result.

When it comes to this translation we have to take a number of design considerations into account, mainly with regards to the limitations of what the various Shader units on the GPU can do.

Most algorithms break down into two components:

1. Scatter: Being able to write to any location in memory
2. Gather : Being able to read from any location in memory

The TLM is no different in that regard as it requires data from cells not being worked on at that moment; however the two operations cannot be performed equally by both the Vertex and Pixel Shaders.

Vertex shaders can effectively perform a 'scatter' operation as they can vary the location the vertex is in memory, effective adjusting the output location of the information. However, in the majority of GPU cases they can't read from an arbitrary location, instead they are restricted to the data they are fed in thus preventing it from performing a 'gather' operation. [GPU Gems 2, pg 497]

Pixel shaders on the other hand are free to read from any location on a source texture they wish, but are restricted to writing to a single pre-determined location; this makes them excellent at 'gather' operations but unable to perform scatter.[GPU Gems 2, pg 498]

When it comes to GPGPU applications the Pixel Shaders are typically used for data processing; this is because there are generally many more of them available thus increasing the parallel processing which can take place. Secondarily to this is that the pixel data goes more or less directly to the graphics cards ram, where as the output of the vertex shader has to be passed through the pixel shader as well as the later stages of the pipeline anyway making it far less straightforward [GPU Gems 2, pg 498]

Thus, by mapping the functionality required by the TLM calculation to shader and loading them into the GPU to be executed we can carry out the simulation in parallel across multiple pixel shader units. The details of this will follow in the implementation section later.

Implementation Choice: OpenGL vs. Direct3D

While the general design of the algorithm is indeed API agnostic some thought needs to be given to the API used to implement the final application. On a Microsoft Windows based PC

the choice is very firmly between OpenGL and Direct3D. While both have broadly the same performance characteristics and features making the choice non-critical to the overall speed of the application a choice must be made between the two none the less.

Direct3D is part of Microsoft's DirectX Software Development Kit (SDK); the SDK itself consist of a number of other parts, which while useful for game development have no bearing on the selection for the purposes of this application. While D3D's most recently released specification is D3D10 the version that was focused on was D3D9 due to D3D10 only being available for Windows Vista and a limited amount of hardware support. D3D9 however is more than up to the task as it has a number of key features which are required for this project:

- Support for shaders in the form of High Level Shading Language (HLSL)
- Support for rendering to floating point textures
- Support for using a texture as vertex data via an ATI extension

D3D also has a support library, D3DX, which has a number of classes and functions which can be used to aid the development of D3D based applications and significantly shorten the development time.

OpenGL on the other hand is an Open Standard which is managed by the Khronos Group and overseen by an Architecture Review Board (ARB) consisting of a number of major players in the graphics market. OpenGL itself grew out of IrisGL which was used by SGI for its machines. OpenGL came into existence in 1992 when the ARB was formed and OpenGL 1.0 was released. Since that time the standard has progressed to OpenGL 2.1, although many implementations currently only support OpenGL2.0.

Much like D3D, OpenGL has support for the 3 major components required for this project:

- Support for Shaders in the form of the OpenGL Shading Language (GLSL)
- Support for rendering to a floating point texture
- Support for copying data from a texture buffer to a buffer to be used as vertex data

The third point on the list is different as OpenGL doesn't have direct support in its current form for using data in one buffer type for another purpose; however this is being corrected in a newer version of the specification. For now a copy of the data has to be made, but thanks to a common extension this copy can be done asynchronously between areas of the graphics card memory with no CPU intervention thus not having a significant effect on the runtime costs.

Despite having no official set of support libraries there are plenty of utility libraries which exist to make working with OpenGL as easy as with D3D.

As noted, both APIs come with their own support for shaders; however there is a 3rd shading option which can be used with both APIs in the form of nVidia's C for Graphics or Cg library. This library allows you to write a shader once and reuse it in either 3D API with no changes. While this might have been important if a cross-API renderer was being developed as the selection was being made to select one of the two APIs the introduction of this library was seen as an un-required complication.

In the end this project used the OpenGL API; however the initial selection was to go with D3D for personal development reasons. This choice was changed very early in the application development phase due to reasons explained later in this document.

Surface and Vertex Normals

Something which hasn't been mentioned to date is the idea of 'normals'. A normal is a vector which is perpendicular to its associated surface. Normals are required in 3D programming to correctly carry out lighting of objects in 3D space.

As you can see from Fig 4 there are two types of normals; vertex and surface normals.

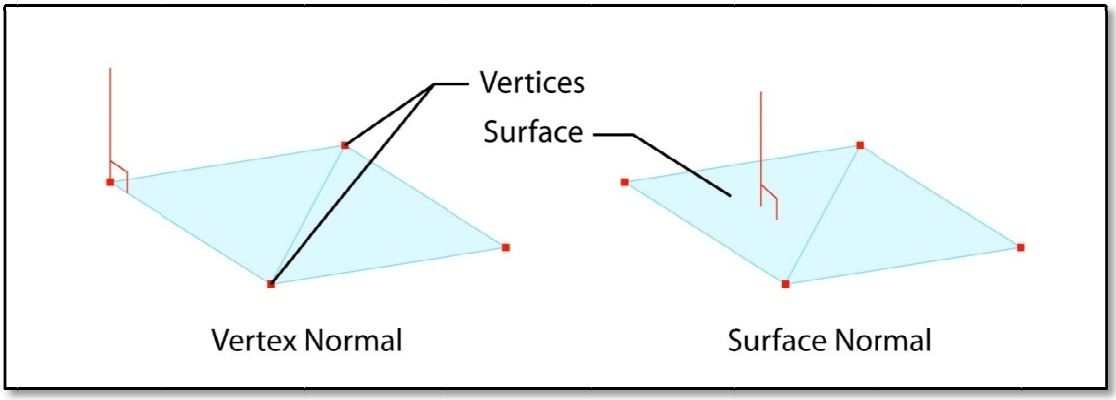


Figure 4 Vertex and Surface Normals [Dor, 2006]

When lighting is being calculated on a per-pixel basis the vertex normals are interpolated across the surface to create the correct normal at the point being processed. When vertex based lighting is being used the colour per pixel is worked out at each vertex and the value is interpolated between them to create the final colour.

To work out the normal of a vertex you require two vectors as shown in Fig 5, from there you perform a cross product as shown in Fig 6.

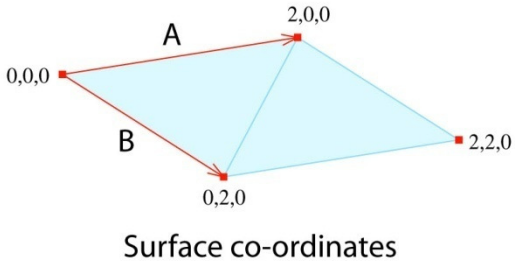


Figure 5 Surface Co-ordinates needed for normal calculation [Dor, 2006]

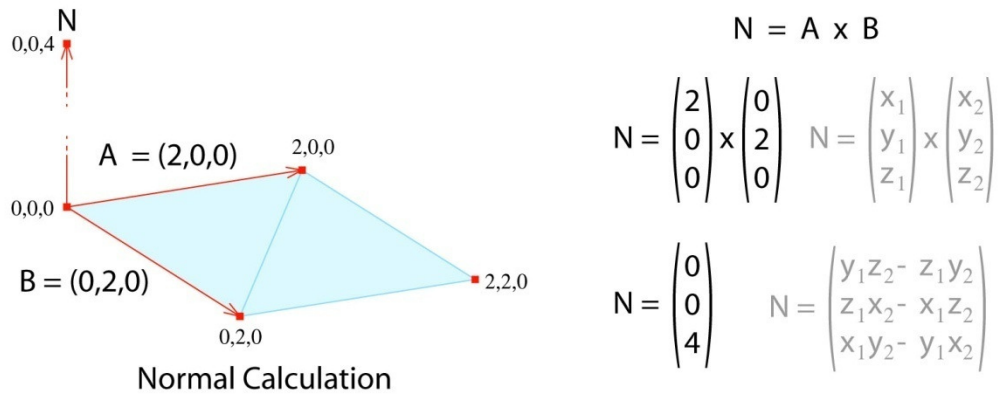


Figure 6 Cross product calculation [Dor, 2006]

The final step is to 'normalise' this vector so that it has a length of 1, this is required to ensure that lighting equations function properly when evaluated.

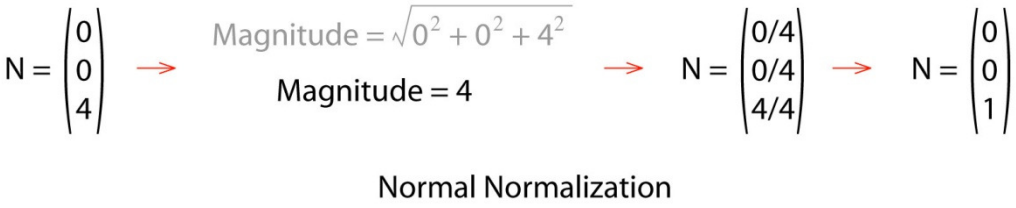


Figure 7 Normal normalisation [Dor, 2006]

In this project, although they are not used for the visualisation process, the normals are calculated using an '8 tap' system, where by the surrounding 8 vertices are sampled and used to create 8 surface normals, these normals are then summed and divided by 8 before being normalised to create the correct per vertex normal.

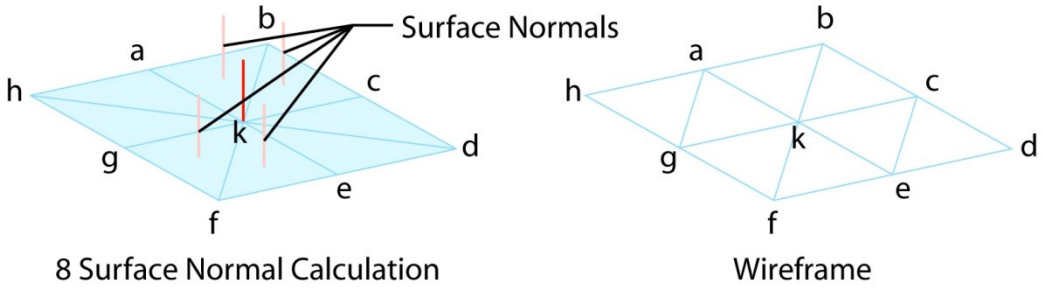


Figure 8 8 tap surface normal calculation [Dor, 2006]

Project Management

Planning and Time Management

In order for the project to be completed successfully with meaning full data to analysis a number of stages had to be completed along the way.

Investigate TLM
Select 3D API, Supporting Language and any supporting libraries required
Develop Application
Collect Results and Analysis
Draw Conclusions

While the initial plan was to stick to the time table as laid out in the project proposal some adjustments were made with regards to the ordering of the elements due to considerations made at the time of implementation.

Development Methodology

Due to project being none critical nature the choice was made to go with a Rapid Application Development (RAD) model while programming. Each stage of the project built

upon or adapts the stages before it in order to move the project forward; this process was back with a revision control system in the form of a piece of software called Subversion.

This choice was made so that if any critical braking change were introduced into the program's source code the changes could be rolled back to a known working copy before moving forward again. It also had the added bonus of having a backup of the code off the development system in case of a critical system failure resulting in code loss. At each major code change stage the code was compiled for correctness and the output tested before committing the code into the repository.

Application Requirements

The final requirements for the test application are in fact quiet succinct in nature, requiring it only to perform the following tasks:

- Open a window to render into
- Run the TLM simulation
- Be able to collect information with regards to the performance of the simulation
- Be able to clean up and shut down cleanly

The choice was made to split the TLM simulation into 3 stages;

1. Drive
2. Update
3. Render

Each of these could then be timed independently of the others during data collection.

Application Code Considerations

The system was required to test a number of algorithms which while functionally equivalent different in varying amounts implementation wise; with this in mind the placing of these different implementations into their own class, all inheriting from one common base class was a logical design choice to make.

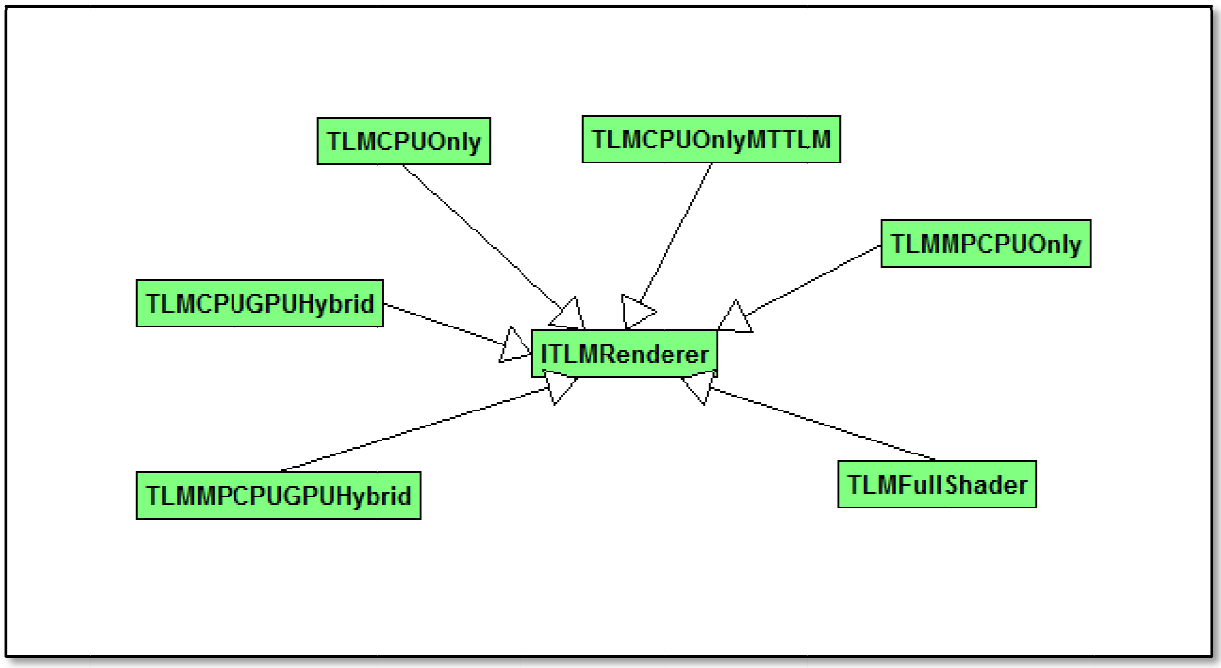


Figure 9 Renderer UML Diagram

The use of this common base class allowed for the test loop to work on any of the different implementations without requiring any knowledge of them beyond the common calling interface as detailed in their base class. This reduction of coupling between the implementation and the tester means that the tester could be used with more methods without any consideration for how they are required to be setup. Indeed, the methods used in this application had differing requirements when it came to construction.

The application also made use of a 'mesh' class which was used to hold the data for the target mesh; this series of classes also had a common base which allowed each TLM implementation to make use of any type of mesh providing it implemented the correct interface. While this coupled the TLM classes to the Mesh interface the lack of requirements to know about their internals means that the coupling is at the lowest level possible when it comes to the classes used as source data.

Application Development

The development of the application was broken down into a number of phases, each one shown in the time plan in Appendix A.

3D API Selection

The first major choice was that of the 3D API to use during development. Initially the choice was made to go with D3D due to the potential to learn more coupled with the D3DX support library which has classes for dealing with vectors, camera setup, Graphical User Interfaces (GUIs) as well as provide a framework for developing the application around based on stock code. The usage of this library would have also made the job of porting Kristian's code from his application to the newly developed one much easier.

However, the functionality of the D3D implementation hinged on the use of an extension to D3D by ATI to allow for texture data to be reused as vertex data; this reliance on a relatively new piece of technology is what caused the change of order in the development, so that instead of the code requiring this extension being developed last it would be developed first, that way if there were any problems a change of direction could be effected and the project wouldn't fail due to the inability to carry out the main goal; the execution of the TLM function on the GPU to develop a render-able mesh.

As it turned out the worse did indeed happen and, for reasons which still remain unexplained, the functionality could not be used, despite the code appearing the same as numerous examples which did indeed work. Thus, it was decided that instead of continuing to waste time and resources on a dead end a retooling for OpenGL, with which a higher level of experience was already available, would be the correct step forward.

To that end the swap to OpenGL was made and the API dependant parts of the code re-written to the new interface. Fortunately a number of libraries already existed to cover the majority of the functionality which would now be missed due to the removal of D3DX from the toolbox of utilities which could be used on the application.

OpenGL and 3D Support Libraries

With the removal of D3DX from the equation a number of support libraries were required in the project;

OpenGL Window Framework

This in-house library, available as an open source project online[4], was used for the construction of the OpenGL window into which rendering could be performed.

The Maths library

The maths library used in this project was taken from Game Programming Gems [Gems, 2000]. Created by [[insert programmer names]] this library forms the backbone of the maths operations on vectors and matrices used in this project.

The camera library

Another piece of in-house software was a 'camera library', designed to control the projection matrix and view matrix of an OpenGL scene. This code is currently in development and not available online.

OpenGL Shading Language Support Files

A simple two class in house library was used for the loading and control of OpenGL Shading language shaders during the project.

Language Selection

Language selection was another key point for the project and as previously mentioned in this document the choice was made to go with C++. When the initial plans were drawn up it was considered too far reaching to develop an application while learning about an API and a separate programming language, instead the choice was made to leverage the 6 years of C++ experience which already existed.

[4] http://oglwfw.sf.net

This turned out to be an advantageous choice as while OpenGL can be accessed via C# it is much easier to do so from C++. Also, the library used to create an OpenGL rendering surface is unavailable with a .Net/C# interface.

Language Support Libraries

With C++ selected as the development language of a choice the requirement to select some support libraries needed to be met.

Boost

The first of these is a library called Boost[5]. It is designed to add support for functionality considered 'missing' from C++ and has members of the C++ language design board, as well as many others, who contribute to the library.

In particular this library has functionality for data output formatting and threading support. Although the threading won't be utilised in this project as it requires far too much code over head to implement.

OpenMP

"OpenMP[6] is a specification for a set of compiler directives, library routines, and environment variables that can be used to specify shared memory parallelism in FORTRAN and C/C++ programs".[OpenMP,2004] Unlike Boost's threading library the interface to OpenMP is a series of #pragma statements in the source code directly before the sections it is required to parallelise. The task of setting up the threads, the number of threads and handing out work is handled by the OpenMP runtime without any programmer intervention, this allows it to adapt dynamically to the system in use.

[5] http://www.boost.org
[6] http://www.openmp.org/drupal/

While it might not be a efficient as hand tuned multi-threaded code, the easy with which OpenMP can be used to add the functionality generally out weights the cost and it is this ease of implementation which made it the correct threading choice for this project.

Implementation

One of the key points of the project was to expand upon the work previously done by Kristian Dor and Steven Harris in previous years, to that end their existing code was looked at to see how it could be reused, if at all.

Given that the original code by Steven Harris was written in C it was decided that the main TLM calculation could be taken directly from his code and transplanted, with the correct support code, into the CPU based implementations. While this was only a small piece of code it would not have changed between the C and C++ versions as such a direct copy and paste seemed the correct way forward.

However, this was only part of the implementation of the functionality as for many aspects we were building off of Kristian's code base and not Stevens. Due to the difference in language selection a copy and paste could not be used, however the same algorithms, adapted for C++'s way of doing things, could be transplanted almost directly.

A key example of this is the normal generation function which takes 8 taps from around the central point which is having its normal generated. In Kristian's code the temporary variables are constructed each time the main loop body is run , while this is optimal for the C# implementation due to the characteristics of the .Net Garbage Collector[7] it doesn't suit C++'s memory allocation system all that well. Instead, as they are reinitialised each pass though the loop the variables were declared outside of the loop and used internally, thus

[7] A garbage collector is a piece of code which runs at set intervals and based on the state of a section of memory automatically returns it to the heap of free memory. In C++ you have to return this memory yourself, in C# this memory return it handled automatically and as a result of this scheme small and often memory allocations are cheaper in CPU time than in C++.

prevented wasted cycles with memory allocation and zero initialisation (this was changed slightly for the multithreaded version of the same code due to a class of memory usage, instead the temporaries were moved into the first loop). The other change was how the code dealt with the normals at the edge, instead of performing a mathematical operation a zero constructed vector was just assigned to the variable later used. The final change was the point where the vertex normal was finally constructed, instead of performing each components addition and division one at a time the overloads in the vector class which was provided with the maths library in use was used to perform all the operations in one line of code.

Aside from those changes the code was used mostly as a guide to ensure the newly created code worked as expected and performed correctly.

Application Setup

As mentioned above the OpenGL window Framework was used to create the initial openGL capable window for rendering into. After that the a standard setup for OpenGL Was used based on code from the OpenGL Programming Guide (aka The Red Book) [OpenGL Guide, 2003], this sets the OpenGL 3D pipeline into a known state such as the background clear colour, depth testing setup, blending setup and any face culling which was required.

After that the main application code begins in earnest, code which changes significantly over the lifetime of the project.

3D Pipeline Setup

Transformations

When drawing objects in a 3D work the vertex information must undergo a series of changes in order to be correctly displayed on a 2D screen, a simplified version of this pipeline is shown in figure 10

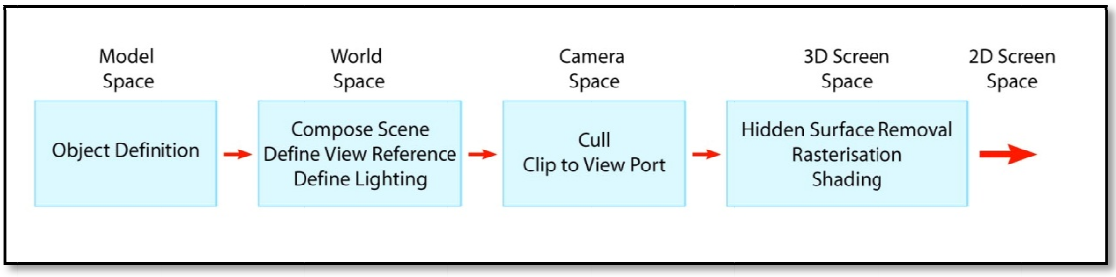


Figure 10 Simplified Graphics Pipeline – Adapted from [Watt, 2000]

All objects are drawn using what is known as a 'local coordinate system', that is all positions are relative to a local (0,0,0) point in 3D space, in order for the object to be position correctly in the 3D world these local coordinates must be transformed into world coordinates . In OpenGL this transformation is controlled by the 'model-view matrix', which acts as both a position of a 'virtual camera' and as the operations the vertex data must undergo. The term 'virtual camera' is used because, in effect, the camera never moves from the world's (0, 0, 0) point, instead the world moves around it to give the impression that the camera is moving.

Once this is done the resulting transformed vertex information must be projected from a 3D object to a 2D one; two forms of projection are used in this application

- Orthographical
- Perspective

Orthographical is like a builders blue print, in that regardless of how far away the object might be everything appears the same size on the screen; this projection is used during the TLM calculations.

The Perspective projection creates an image as you would be used to seeing in the real world, with objects which are further away appearing smaller than those which are close to the screen. However, so that the graphics card doesn't have to deal with infinitely close or infinitely far away objects two values are defined which form a near and far point, either side of which objects aren't seen; these are referred to a clipping planes.

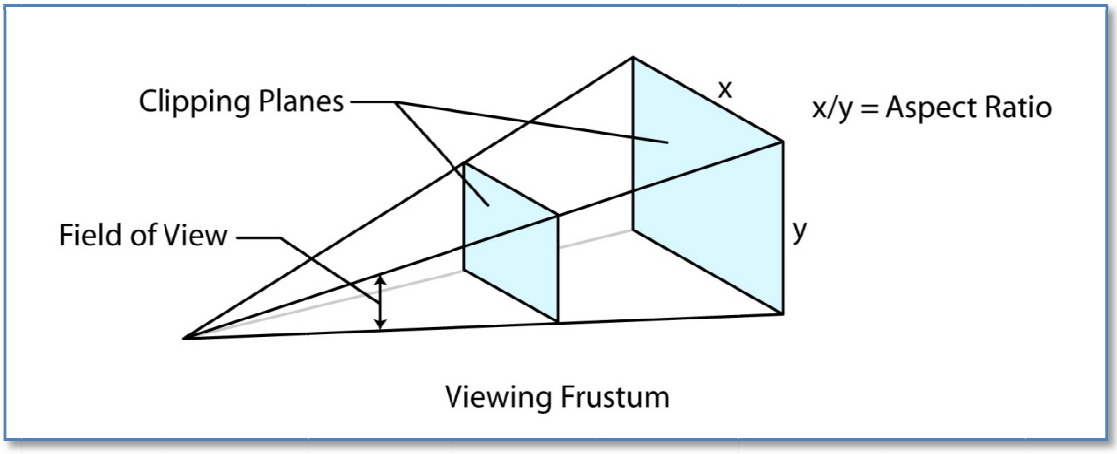


Figure 11 Clipping planes for view frustum [Dor, 2006]

The camera setup is handled via the support library code; in the main application a camera is used which gives the impression of orbiting around the object we are viewing, which is the mesh being produced from the TLM calculations.

Depth Testing

In real life, if object A is further away than object B and object A is behind Object B from a viewer point of view then it will be obscured. In 3D graphics however this isn't the case by default; graphic cards have no real perception of depth, thus to use the same example, if object A was draw after object B then we would still see object A over lapping Object B, which is of course incorrect in the real world.

To deal with this problem graphics cards have a section of memory called a 'z buffer', when an object is draw its depth into the screen is stored in this z buffer. If we then draw another object which overlaps the first the depth into the screen is compared to the values already in the z buffer; if that value indicates it is closer to the viewer than the previous object drawn we over write that first object's pixel, if however it is further away then we don't write any pixels out.

This allows us to draw objects correctly in 3D space without having to worry about the order they are drawn in, without a z-buffer and depth testing problems as those in figure x.xx can be seen.

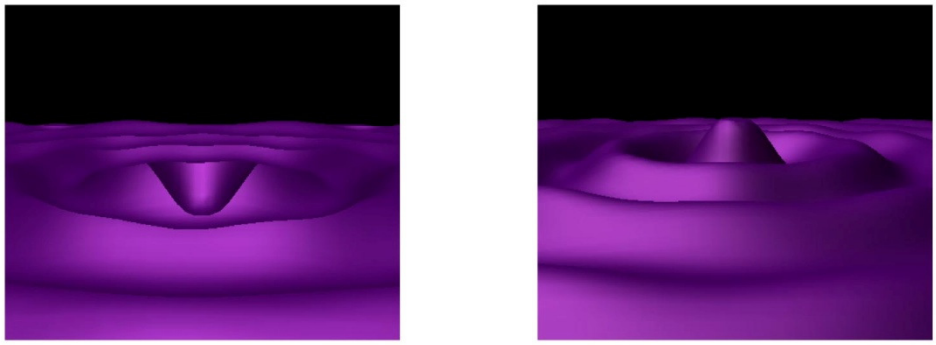


Figure 12 Scene rendered with and without a Depth buffer [Dor, 2006]

Initially depth testing is turned off in the application as it has no value and just consumes resources, however for the final rendering it is turned on again to generate the correct image.

Blending Setup

When pixels are drawn to the screen they can be done in one of two manners;

- Replace that which already exists
- Blend with existing data

The blending operations allows developers to create effects such as stained glass windows or indeed glass; where you have an object which has some colour but is also semi-transparent in nature. However, this blending comes at a cost and as it is of no advantage to this project it is set to off and pixels draw simply replace those which are drawn before them.

Face Culling Setup

Calculating the colour of pixels can be a time consuming operation, in order to generate images quickly we often want to reduce the number of pixel processed. Face Culling allows us to do just that by only drawing the faces of object we can see. Much like depth testing face culling is disabled for the majority of the application; it is only during the final rendering stage that it is enabled in order for us to cut down on the amount of work that is required to do.

TLM Simulation Modules

In order to compare the different methods of generating a TLM mesh a number of different classes were developed, all using the same basic interface, one for each method.

- CPU Only
- CPU Only with threaded TLM Calculation
- Fully Threaded CPU Only
- CPU/GPU Hybrid
- Threaded CPU/GPU Hybrid
- GPU Only

In order to ensure a fair comparison the work load of each modules was planned to be the same, this includes normal generation where only the 8 vertex method used in Kristian's code was implemented as it has the highest computational over head and thus would stress each method the most.

GPU Only Implementation

The first implementation which was developed was for the GPU only; the idea behind this method was to free the CPU from as much work as possible and instead utilise the GPU for all the work required to generate the mesh.

The first order of work was to perform the translation from the original CPU based TLM code into OpenGL Shading Language code; this required some adjustments in order to have it working optimally on the GPU.

The first problem which needed to be solved was that of data storage; the only data sources the pixel processing units can read are values passed from the vertex shader or read from textures. In the both the original TLM code by Steven and Kristian's version the forces which drive the TLM simulation are split into a number of arrays; two groups of four values to hold the values being scattered out and gathered in at each node.

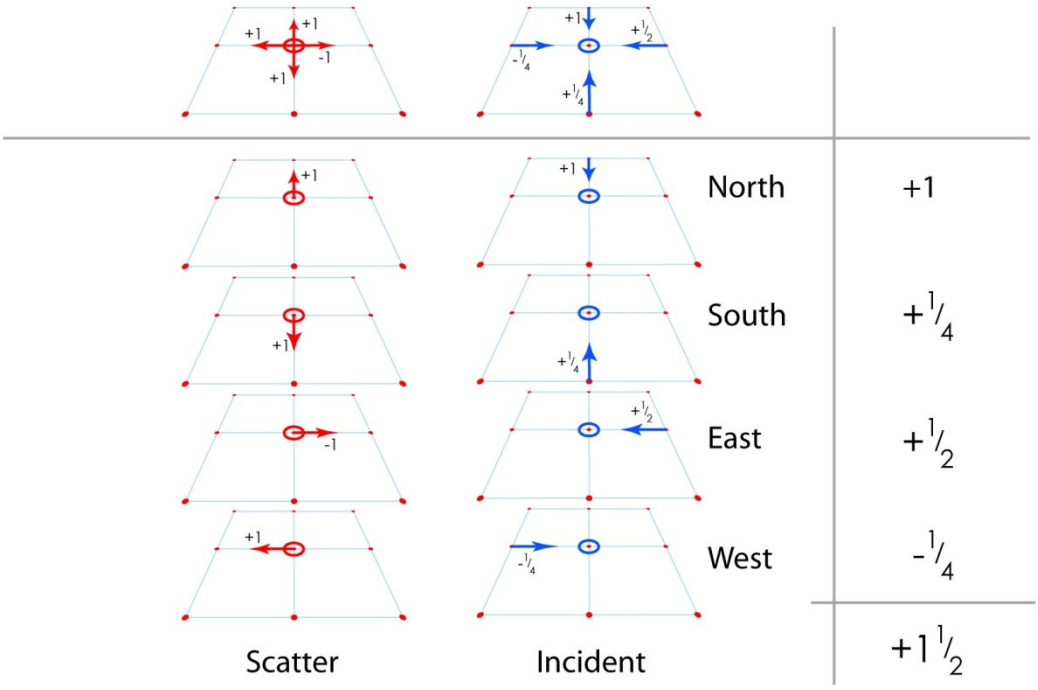


Figure 13 TLM Node layout in memory [Dor, 2006]

This grouping of data for the arrays mapped perfectly to using textures as a storage medium. Textures can have a number of component to them, a common number however is four; generally representing red, green, blue and alpha colour components. With this in mind the storage problem was solved by using two textures which could hold floating point values; these two textures between them would represent the 8 TLM arrays for the incident forces and the scattering forces.

With the storage problem solved the next problem to tackle was that of efficiently map the TLM algorithm from a CPU favourable approach to one which would be favourable to a GPU. One of the main considerations here was that of parallelism; current GPUs work best on data with more than one component to it; this is due to the normal operation of these chips as 3D graphics processing devices, as such the hardware has specialisations built in which allow it to work more efficiently with multiple component data types. There was also the consideration of the amount of data being read into the GPU from memory and the amount of time that would take; while a graphics card does have a large amount of memory bandwidth reading floating point values is still reasonable slow and intensive so the least amount of data that can read the better during processing the better the speed should be.

With this in mind the TLM algorithm was inspected for ways it could be made more GPU friendly. The original function can be broken down into two parts;

- For the current node read all the incoming values and sum them as required and write them to the value we are going to scatter outwards
- For the current node read all the nodes around it for the value they are scattering out

It would have been possible to perform this task on the GPU as it standards, however for each node in the matrix it would have required the GPU to read in 4 32bit floating point values and write back out four floating point values for the completion of the first stage and then read in 16 floating point values and write out 4 more for the second stage; 12 of those floating point values being read would have been wasted as they are not used during processing. As each floating point value is 4 bytes, this meant that for each node the GPU would have had to read in 80bytes of data, of which 48bytes is wasted, and write out 32bytes.

However, if the algorithm is reconsidered slightly this amount of data can be reduced quite dramatically; it is possible to combine the two steps into one, as the values calculated in the

first step are never used directly, they were just passed back to the original array as a new value. With that in mind the algorithm could be modified so that it became:

- For the current node read all the nodes around it and for each node calculate the scattering value and write it out

This results in a total read of 64 bytes and a write of 16 bytes with no waste, a significant improvement on the previous method and a reduction on memory bandwidth pressure by 20% on reading and 50% on writing.

However, this doesn't completely cover the TLM process, as it also writes out a value representing the magnitude of the data at that point in the matrix, something the GPU version currently didn’t do. Initially, the code took advantage of the ability of recent GPUs to write to more than one memory buffer at once. This would have allowed the shader to do this magnitude write at the same time as the TLM was calculated; however a driver bug/issue caused this to have a an extremely detrimental effect on the speed of the simulation, as such the choice was made to move this to its own pass over the data.

The pass which followed takes the generated TLM and uses it to displace a mesh. Like Kristian's code the mesh is displaced along its original normal by the calculated magnitude at that same point in the TLM; for this the original vertex and normal data was supplied to the pixel shader via two textures and then combined with the TLM magnitude output data to produce the final mesh output.

This output was then copied from the output buffer to a vertex buffer on the card so that it could be used in the final visualisation stage.

Once this was done the TLM is 'driven' for the next iteration of the simulation; this simply involved reading a texture for a driving value and writing out a new value which represented the state after the driving value had been applied. Unlike the CPU version however it is impossible to adjust just one point of the calculated TLM values directly, instead a texture was used with one value in to act as a driving energy source this texture was sampled at

every point for a value; while this does effectively mean that extra incident points can be added for free it can waste a large degree of processing power and memory bandwidth as many unchanged values are simply read from memory and then written back. However, some of this waste can be reduced by using a single component texture as the source data which reduces the amount of driving data written or read to a single floating point value, 16 or 32bit, per cell in the mesh.

The final stage was to calculate the vertex normals for rendering; this was done in much the same way as it was done on the CPU by Kristian's code, by sampling the location of the 8 points around the point which was currently being processed and then building a face normal for each of the 8 triangular faces this group of points made up. These 8 face normals were then summed and divided by 8 in order to correctly produce a vertex normal, this data was then written to the output buffer before being copied from that buffer to a vertex buffer for the final rendering.

The final rendering method simply rendered the vertex data that was calculated in the previous stages to produce the final visualisation of the TLM affecting the supplied 3D Mesh.

CPU Only Implementations

The second implementation completed was that of the CPU only simulation, this was basically the equitant of Steven Harris' and Kristian Dor's TLM simulation, where by all the calculations were performed on the CPU; the main difference between this and Steven Harris' version was that, instead of sending the mesh data vertex by vertex, this code computed the whole mesh and then uploaded this data into a buffer on the graphics card in order to be rendered quickly. Kristian's version performed much the same process however that code first buffered the data to system memory before performing the upload in once batch. By contrast the version tested here both buffered the data to system memory, to be used later during the normal calculation step, and at the same time a copy was send directly to the graphic card's memory.

The new mesh was calculated in the same manner used by the GPU version, however it was done using the CPU only; the original normal was multiplied by the magnitude of the TLM forces at the current point and added to the original vertex position to yield the new position. It was this vertex data that was both buffered in system ram and copied to the graphics card.

Once the mesh had been calculated the next step was to pass it on to the normal generation routine; like the GPU only version of this code it took the location of the vertex for which the normal was being generated as well as the 8 vertices which surrounded it, the normals for each face that these vertices were a part of are calculated and the resulting face normals were summed and divided by the number of faces in order to generate the normal for this vertex. As with the mesh generation above the resulting normal was written to a buffer on the graphics card but it was not stored in system memory.

The final render used the vertex position and normal data which was computed in the preceding stages to generate and light the final visualisation of the 3D mesh.

CPU Only Full Multi-threaded Implementation

One of the possible simple improvements to the TLM based mesh generation was the usage of extra threads during the generation stage; with modern CPUs it is common to find that they already have two functional 'cores' to process data with, thus by utilising multiple threads the work load can be split between the two cores, potentially speeding the calculations up.

To this end OpenMP was used to quickly add multi-threaded capabilities to the algorithm; in particular it was used to accelerate the TLM simulation itself and the normal generation by splitting the work load between the available cores. While the test system only had a CPU with two cores OpenMP itself can dynamically assign threads to work on the data, as such testing on CPUs with more cores might be appropriate in the future (see future recommendations for more details).

While the TLM simulation was naturally separated and thus required no real work when it came to converting it to a multiple threaded solution the mesh generation and normal generation were a different matter.

The first problem with the mesh generation was the method used to writing to the graphic card's ram directly; to do this in OpenGL a function was used to request a pointer to some memory to write to (the pointer returned could be in video ram or it could be in system ram, the method used is up to the graphics driver), however this pointer was only valid in the thread which created it, when new threads are created to carry out the work the pointer becomes invalid and any writing to it will fail and cause the program to crash. As such the data was simply buffered to system ram and then uploaded once the process was completed.

Normal generation had much the same problems and thus the same solution was used, however due to code layout one other problem had to be solved. To generate the normals the following code snippet was used:

```
// Next we need to redevelop the normal data for the new mesh
// Using an 8 tap method
// Using k as the vertex to create normals for
//
//           h-----a-----b
//           | \   |   / |
//           |  \  |  /  |
//           |   \ | /   |
//           g-----k-----c
//           |   / | \   |
//           |  /  |  \  |
//           | /   |   \ |
//           f ----e-----d
{
    vector3 ka;
    vector3 kb;
    vector3 kc;
    vector3 kd;
    vector3 ke;
    vector3 kf;
    vector3 kg;
    vector3 kh;
    glBindBufferARB(GL_ARRAY_BUFFER_ARB, normalBuffer_);
    glBufferDataARB(GL_ARRAY_BUFFER_ARB, numVerts_*(sizeof(vector3)), NULL, GL_STATIC_DRAW_ARB);
    vector3 * finalNormals = static_cast<vector3*>(glMapBuffer(GL_ARRAY_BUFFER_ARB, GL_WRITE_ONLY));

    for(int x = 0; x < vertsperedge_; x++)
    {
        for(int y = 0; y < vertsperedge_; y++)
        {
            // Normal calculations here
        }
    }
    glUnmapBuffer(GL_ARRAY_BUFFER_ARB);
}
```

Figure 14 Normal generation code snippet

Notice how the temporary vector3 objects are created outside of the first for-loop; this is to save use the cost of reconstructing them each loop as previously explained. However, as this is shared memory when we introduce threading into the mix you get memory collisions and incorrectly calculated results which lead to a corrupted output.

As OpenMP was only used to create threads to process the external loop the vector3 objects were created inside that loop instead; while this meant we had to do some extra work on each iteration it was unavoidable, however this setup cost was expected to be trivial when compared to the gains made on the loops execution speed using the thread solution.

```
        // Next we need to redevelop the normal data for the new mesh
        // Using an 8 tap method
        // Using k as the vertex to create normals for
        //
        //           h-----a-----b
        //           | \   |   / |
        //           |  \  |  /  |
        //           |   \ | /   |
        //           g-----k-----c
        //           |   / | \   |
        //           |  /  |  \  |
        //           | /   |   \ |
        //           f ----e-----d
        {

            glBindBufferARB(GL_ARRAY_BUFFER_ARB, normalBuffer_);
            glBufferDataARB(GL_ARRAY_BUFFER_ARB, numVerts_*(sizeof(vector3)), NULL, GL_STATIC_DRAW_ARB);
#pragma omp parallel
            {
#pragma omp for
                for(int x = 0; x < vertsperedge_; x++)
                {
                    vector3 ka;
                    vector3 kb;
                    vector3 kc;
                    vector3 kd;
                    vector3 ke;
                    vector3 kf;
                    vector3 kg;
                    vector3 kh;
                    for(int y = 0; y < vertsperedge_; y++)
                    {
                        // Normal Generation here
                    }
                }
            }
            // copy normal data to gfx card
            glBufferDataARB(GL_ARRAY_BUFFER_ARB, numVerts_*(sizeof(vector3)), finalNormals_, GL_STATIC_DRAW_ARB);
        }
```

Figure 15 Multi-threaded normal generation snippet

Aside from these minor changes the functionality of the multi-threaded solution was much the same as the original version, with no changes at all being made to the rendering code as this would not have benefited at all from any sort of multiple thread approach.

CPU Only with Multi-threaded TLM Calculation

In order to see what kind of effect adding threaded has on the TLM mesh generation algorithm a multi-threaded version of the code which sits half way between the single thread and the full multi-threaded implementation was developed.

This version of the code only split the work done on the TLM calculations between multiple threads; the vertex normal generation and the final rendering code was left the same as the original CPU Only implementation.

CPU & GPU Hybrid Implementations

The final set of TLM modules to be completed were the CPU/GPU Hybrid modules; these modules split the task of generating the mesh between the CPU and the GPU, with the final rendering taking place on the GPU.

The TLM calculations were carried out the same as with the CPU only approach, however instead of writing the force's magnitude to system ram the result was written directly in to the graphics card's ram, this saved using local memory to store a result which was only going to be used once to upload to video ram.

Next, the GPU was setup to generate the mesh data; this was done the same way as with the GPU only method; by giving it three textures which contained the original mesh vertices, the original normals and the previously calculated magnitude information. A shader was then executed which performed the same mesh generation calculations as used in all the previous modules: the original normal is multiplied by the magnitude of the TLM forces at the current point and then added to the original vertex position to yield the new position which is then written out to a buffer before being copied to a vertex buffer to later be used as the source for the final rendering.

Next, the normals were generated in the same way as indicated before for the GPU only version, by sampling the location of the 8 points around the point we are currently processing, and building a face normal for the 8 faces around the point. These 8 face normals were then summed and divided by 8 in order to correctly produce a vertex normal, this data was then written out to the output buffer before being copied from that buffer to a vertex buffer for the final rendering.

Finally these two newly generated buffers were used in the final rendering stage to render the 3D mesh which represents a visualisation of the TLM simulation as applied to the source mesh.

CPU & GPU Hybrid Multi-threaded Implementation

While the CPU was not doing as much work in the hybrid method as with the pure CPU only one a multi-threaded version was still developed to compare the speed gain, if any, with that single threaded version.

As the normal generation was being done on the GPU (which effectively made it multi-threaded already) the only area into which the extra threading could be inserted was for the TLM simulation. Due to the aforementioned problems with the pointer which was obtained from OpenGL in order to write directly to the graphics card's ram only being valid in the original thread this version of the TLM simulation also had to use a memory buffer in much the same way as the original CPU only method did in order to store the calculated data before it was uploaded to the graphic card's ram.

The rendering method used remained the same as the one used in the non-multi-threaded CPU & GPU Hybrid code.

Testing and Result Collecting

In order to test each of the modules in turn a test application was developed which could run each of the modules in turn and collect timing data from each of them.

The application was split into two functions, a main function which housed the loops and constant data for the tests and a test function itself which ran each module a given number of times, collected timing results and wrote them to a file for later processing and analysis. The timing results were measured in clock ticks per second using the high precision timer available on most PCs and for each of the iterations of the algorithm, the time taken to update, render and drive the TLM simulation was recorded and then saved out to a file, along with the minimum, maximum and average times for each method-size combination. The number of frames per second was also calculated in order to give an easy to understand comparison between the various methods.

Memory was also a factor between the various methods used, however as the application was designed to allocate all the ram it required up front this profile didn’t change during the

running of the algorithm (at least, not in any way the application could control, it is possible that the graphics card driver was reallocating buffers behind the scenes), as such the memory usage was computed 'off line' after the tests were completed.

The test application was designed so that it could run unattended though all the tests, this was required as a total test run was expected to take a significant amount of time to complete due to the sizes of the TLM meshes used.

In order to get a number of results to compare the following mesh sizes were selected for testing:

40 * 40
50 * 50
100*100
128*128
256*256
512*512
1024*1024

The first 3 mesh sizes were selected as they were the sizes used in Kristian's implementation of the TLM algorithm in his project and thus seemed a logical place to start, after that the sizes follow a simple binary series. While there was no limit on the size of TLM mesh the algorithm could deal with the size was constrained to a maximum of 1024*1024 due to limits imposed by the driver and hardware that were being worked with.

Finally, as the GPU can work in either 16 bit per channel or 32 bit per channel floating point format with regards to its texture reading and buffer writing abilities the choice was made

to include a series of tests which would test the GPU based versions at both of these precisions to see if any differences could be found in the speed

During the test each module was used as each mesh size and the rendering was done into a full screen display running at 800*600 in 32 bit colour.

Test machine details:

- AMD Athlon 64 X2 4400+ 2.2Ghz Socket 939 2MB L2 cache
- 4Gb RAM DDR333
- Radeon X1900XT PCI-Express Graphics card with 512Meg of onboard ram (at stock speeds)
- Drivers used were ATI/AMD Catalyst 7.4
- Windows XP x64 Professional Edition SP2
- 2 x 24” LCD Monitors
- V-sync disabled in control panel

As each test writes out its own data a secondary program was required to parse this data and convert it into a more useful output. This program had two stages;

- For all files read in the minimum, maximum and average times for each stage of the process and write them to 3 separate text files, the same 3 for each file processed, in a format which would allow them to read into Excel
- For all files processed skip the first 8 lines (of which 3 held the data used above) and output the rest of the per-iteration data into an Excel compatible comma separated value (CSV) file.

All results from this testing system were that of quantitative data and was analysed as such; comparisons between the various implementations was done purely in a numerical and factual sense and in a non-subjective manner.

Application Test Results

Having completed the tested and run the data though the secondary program in order to order it the results were analysed.

The test results in full can be found on the companion CD in the “Test Results” folder.

Mesh Size vs. Frames per Second

The first set of results looked was that of how the mesh size variations caused the number of frames per second to adjust; the results of which can be seen in the table below and the graph in appendix B.

Mesh vs. FPS	40	50	100	128	256	512	1024
CPU Only	1010	1010	144.286	84.1667	21.4894	3.99209	0.943044
CPU Only with MT TLM	1010	1010	168.33	101	22.4444	4.69767	1.09663
Multi-thread CPU Only	1010	1010	252.5	168.333	37.4074	6.3522	1.62119
CPU GPU Hybrid 16bpc	1010	1010	252.5	202	45.9091	7.31884	1.74138
CPU GPU Hybrid 32bpc	1010	1010	252.5	202	45.9091	7.21429	1.71477
Multi-thread CPU GPU Hybrid 16bpc	1010	1010	336.667	202	48.0952	9.71154	2.27477
Multi-Thread CPU GPU Hybrid 32bpc	1010	1010	336.667	202	48.0952	9.5283	2.21007
Full Shader 16bpc	1010	1010	336.667	252.5	63.125	16.0317	3.30065
Full Shader 32bpc	1010	1010	336.667	252.5	59.4118	15.7813	3.26861

During testing for some reason the multi-threaded CPU Only method at a mesh size of 40*40 produced an infinite result for the frames per second. The cause of this was unknown however given the performance of the surrounding implementations a decision was made to match results with them.

The first thing to draw from these results is that the testing length might not have been long enough for the first two mesh sizes; this is indicated by the returned value being the same as the number of iterations completed.

Once the mesh size reaches 100*100 the difference in implementations starts to be seen, with the single core CPU version of the code heading to the bottom of the group of results quickly. This is also the first indication that the time taken to perform the normal generation is a serious drain on resources; of the two CPU only multithreaded versions of the code the version which uses threading for both TLM calculation and for normal generation was 1.5 times faster than the other version; this extra speed would seem to indicate that the normal generation does indeed benefit from the use of another core to half the amount of time required to generate the data.

The difference in work load between the normal generation and the TLM is further highlighted by comparing the single threaded CPU version with the version which only implemented multi-threading for the TLM calculations; in this case the multi-threaded code only came out 1.167 times faster than the single threaded code, further indicating the amount of time required to process the normals.

As was somewhat expected the GPU only method performed best during the tests, being able to maintain double figures for the frames per second during all but the final test mesh. When compared to the single CPU version the GPU only solution performed between 2.3 and 3.95 times faster in 32 bit processing mode and between 2.3 and 4 times faster in 16bit processing mode.

It should be noted at this point that there was very little difference between 32bit and 16bit processing mode on the GPU; while the 16bit version was defiantly faster the margin was very small, which would seem to indicate that the memory to GPU bus bandwidth wasn't a massive factor in the speed decrease of the GPU only version.

Between the two hybrid version, single and multi-threaded, there was version little difference between them speed wise. At 100*100 the main factor seems to be the multi-

threading of the TLM code, as this is the only difference between the two version, which is giving it a large advantage in generation time. However after this initial dominance the results become more even for the next two mesh sizes, which would imply that the limiting factor speed wise has shifted to another part of the process and away from the TLM generation. Finally during the final two mesh sizes the speed switches back in favour of the multi-threaded method, again implying a switch in bottle neck.

Mesh Size	Approximate Percent speed difference
128*128	0.0%
256*256	4.0%
512*512	32.6%
1024*1024	29%

In all tests, at the size 128*128 and above all the methods saw the a drop in performance to approximately one quarter of that at the size before it, this would be consistent with the size increases in the mesh data.

Further Analysis

With the FPS giving an overview of how the various methods performed further analysis needs to be applied in order to see why there performed in such a manner, in particular the GPU only and Hybrid methods, with the latter showing some interesting switches in bottle neck as the tests progressed.

GPU Only Method Analysis

The GPU only method always stood the best chance of being the fastest; however it didn't perform as might have been expected. The GPU used during the tests has 48 pixel processing cores, meaning that it can effectively carry out 48 Arithmetic and logic (ALU) instructions on different data at any given time.

However, when compared to the CPU only results a 48 times speed increase was not seen, indeed at its best it could only manage a 4 times increase in 16bit processing mode.

When considering the reasons for this the first thing to take into consideration is the amount of work the GPU is doing; for the whole TLM update process it has to make 4 passes over the data compared to the CPUs 2, and in at least one of those passes the GPU processes a lot of data it strictly speaking doesn't have to. However, even if you take the amount of passes into account, you still would expect an increase in speed.

Another factor which plays against the GPU is in the architecture of the GPU itself; because it is primarily a game driven system a number of choices were made to optimise for games, one of these was the ratio of pixel processing units to texture fetching units. In the R580 this ration is 3:1, meaning that to get the best out of the GPU for every 3 ALU instructions one texture lookup is executed [Radeon,2006]. However, the TLM algorithm requires many texture fetches but is low on ALU operations, this means the code is probably bottle necking waiting for texture units to become available so it can get at the data needed; to prove this however further advanced testing would be required which is outside the scope of this document.

While these factors are important it was also important to look at the data generated from the GPU only tests to see if another reason for the slowdown could be spotted.

For this the table below is used which contains information on how long each part of the process took to carry out;

	40	**50**	**100**	**128**	**256**	**512**	**1024**
Render Max @ 32bit	582	4474	414	211	825	850	570
Render Min @ 32bit	48	51	80	57	99	133	234
Render Average @ 32bit	56	71	92	112	121	148	240
Render Max @ 16bit	310	262	630	692	724	274	672
Render Min @ 16bit	47	48	57	61	97	133	236
Render Average @ 16bit	52	54	89	109	122	146	243
Update Max @ 32bit	51886	54419	55946	58792	68043	237688	1104255
Update Min @ 32bit	3091	3636	10168	14103	57170	217136	1058247
Update Average @ 32bit	4203	4767	11046	15006	59053	223489	1081696
Update Max @ 16bit	4871	5344	13628	17904	68390	233845	1460421
Update Min @ 16bit	3029	3611	9842	13674	56238	214167	1050454
Update Average @ 16bit	3213	3750	10327	14513	58129	220733	1073709

Drive Max @ 32bit	408	748	736	994	1113	5185	18197
Drive Min @ 32bit	22	28	108	132	504	2732	13267
Drive Average @ 32bit	28	41	128	241	667	3170	13869
Drive Max @ 16bit	72	92	260	916	1077	4459	13952
Drive Min @ 16bit	30	38	102	151	612	2478	10360
Drive Average @ 16bit	32	42	134	221	626	2575	10689

This information is also shown in the graphical form in appendix D.

The graph gives the best indication of what is going on with the GPU only process; the rendering times are minimal which makes sense given the setup of the code, so this appears to have no effect on the overall time. The “drive” time as well scales as expected as the amount of data increases (by a factor of 4 each time once 128*128 was reached), however these times are also very low with very little contribution over all to the time.

By far the largest factor however was the time taken to update the TLM simulation and generate the mesh. This also shows the same increase in time as the data to process increases which very firmly puts the bottle neck in this area of the process.

Hybrid Method Analysis

Of all the methods tried out, the hybrid method showed the most interesting set of results, with respect to both the two different hybrid methods used and the other implementations in the test system.

Referring once again to the graph in appendix B and the table on page 41 it can be seen that at 100*100 the hybrid and multi-threaded CPU methods are both equally matched on speed, while the multi-threaded hybrid method is faster by a factor of 1.33. This would seem to imply that at this point the limiting factor is the TLM calculations and not the normal generation as the two hybrid methods would be closer together if this was the case.

However, at 128*128 the state switches again, with the two hybrid methods showing equal times and out pacing the multi-threaded CPU only method by a factor of 1.2. This would seem to imply that the amount of time required generate the normals has become the

limiting factor. This is indicated by the difference in speed between the multi-threaded CPU only implementation and the two hybrid implementations.

Once the mesh size makes it to 256*256 and beyond the two hybrid implementations are very closely matched; however the multi-threaded version slightly out paces the single threaded one. This would seem to indicate that the multi-threading had given that implementation an advantage in this instance as beyond this both implementations are the same.

Memory Usage

The final separating factor between the various versions is that of memory usage, as can be seen in the graph in Appendix C.

	40	50	100	128	256	512	1024
CPU Only	0.144	0.226	0.910	1.494	5.988	23.97	95.95
CPU Only with MT TLM	0.144	0.226	0.910	1.494	5.988	23.97	95.95
Multi-thread CPU Only	0.162	0.255	1.025	1.681	6.738	26.97	107.9
CPU GPU Hybrid 16bpc	0.199	0.312	1.254	2.056	8.238	32.97	131.9
CPU GPU Hybrid 32bpc	0.260	0.407	1.635	2.681	10.73	42.97	171.9
Multi-thread CPU GPU Hybrid 16bpc	0.199	0.312	1.254	2.056	8.238	32.97	131.9
Multi-Thread CPU GPU Hybrid 32bpc	0.260	0.407	1.635	2.681	10.73	42.97	171.9
Full Shader 16bpc	0.157	0.247	0.989	1.622	6.495	25.99	103.9
Full Shader 32bpc	0.231	0.361	1.447	2.372	9.495	37.99	151.9

Truncated Memory usage in megabytes

The lowest memory usage goes to the CPU Only and CPU Only with the TLM process multi-threaded. Due to how the algorithm is designed these two methods used the lowest amount of memory possible; as everything was being calculated on the CPU everything had to be buffered there, at which point it can be uploaded to the GPU in one go.

The next lowest memory usage goes to the 16bit Full shader version; while it uses the same amount of buffers as the 32bit version and a texture less than any of the hybrid versions as the data is stored at 16bit precision it is using considerably less ram; only the driving map in system memory uses full 32bit precision.

Using slightly more memory was the fully multi-threaded CPU version; this was down to it requiring an extra buffer to store the calculated final mesh and final normals; the former down to needing this data for the normal generation and the latter as it was required to buffer the normal data before upload to the graphics card.

There was a bit of a gap between the multi-threaded CPU version and the next highest memory usage which belonged to the two 16bit hybrid methods; both single and multi-threaded. This method has more memory usage than the previous two as it has to keep a number of full precision floating point buffers in memory; the TLM takes up the majority of this space.

The second highest memory usage goes to the 32bit Full Shader version, this beats out the 32bit hybrid method on memory as it doesn't have to keep extra buffers in system ram. Indeed, like its 16 bit counterpart the only buffer maintained in system ram is the small driving buffer used to update the driving texture. On the other hand the hybrid method must keep the TLM and the mesh data system ram as well as numerous textures and buffers in graphics ram.

Conclusions

The purpose of this research was to expand upon the work carried out by Steven Harris and Kristian Dor in previous years to translate the TLM algorithm to graphics hardware and away from the CPU for its processing requirements.

At the same time it tried to answer the question "**What are the potential gains of using a GPU to carry out the calculations required to fully visualize a TLM matrix in 3D, and how does it compare to a CPU only solution?**"

This question can be broken into two parts to be answered;

Firstly there is the issue of how the GPU simulation compares to the CPU versions; when it comes to raw frames per second the GPU beats the CPU hands down practically every time with the Full GPU method leading the pack. However at small mesh sizes there isn't enough

information to accurately compare them and in one test at 100*100 the multi-threaded CPU version, which was only operational on 2 cores, was able to equal a version which was using the GPU for normal calculations.

When it comes to the potential gains, the GPU allows us to the do the same work much faster and provided the TLM mesh is kept relatively small current hardware could be used to generate effects based on this algorithm for games in real time with no problem. This allows us to off load this work to the GPU and have it processing the data while leaving the CPU free to perform other game related tasks in parallel. The usage of a16 bit data allows the speed to be kept high while reducing the amount of ram required to carry out the effect, and in a game situation and potential loss of precision is often traded for the extra speed.

When it comes to academic research the ability of a GPU to keep the fps high even in full 32bit processing mode could be seen as a large gain.

With the advancements of GPUs still moving at pace these advantages can only be improved upon as time progresses. Granted, CPU speeds are increasing and with multiple cores becoming more and more common the CPU simulation of this algorithm might well become more viable in the future requiring a switch back.

Further Research and Considerations

While this project does serve to answer the question posed more research could be done in this area to improve the simulations and indeed the data collected.

Streaming instruction support

Right now the code is pure C++, however this might not be making optimal use of the CPUs resources, in particular the usage of the built in floating point unit can be a potential bottle neck. To get around this the use of Streaming SIMD Extensions (SSE) could be used to process more than one group of data at a time, potentially improving the speed of the CPU versions. These instructions are now very common with all the major CPU manufactures supporting them.

To use these instructions some lower level programming would be required to be used as well as better understanding of the CPU's pipeline. Also, custom vector code would have to be written to generate faster mathematical functionality.

64bit support

While all the tests were executed in a full 64bit environment the application in use was only compiled to 32bit. By recompiling the tests for 64bit the compiler will have more registers and a greater instruction set to work with; as well as naturally using SSE2 instructions when dealing with floating point values.

There is potential for this to improve the TLM simulation's speed by being able to move more data at once as well as avoiding the floating point unit for mathematics work.

Custom Threading Support

While this solution makes use of OpenMP for its multithreading support this might not be optimal, instead a hand crafted threading system might well prove more optimal in balancing the CPU work.

GPU Bandwidth Saving

There are numerous methods available to improve on the usage of memory bandwidth in GPGPU applications. These generally revolve around data packing to transfer more data per clock to improve data flow.

Direct GPU Programming Support

One of the original optional goals of this project was to use the direct GPU programming API provided by ATI/AMD to directly program the GPU and by pass any overhead a 3D API might well introduce.

While this project wasn't able to do it, it might be an interesting project to see what kind of speeds can be gained from using ATI/AMD's CTM or nVidia's CUDA programming environments to directly program the GPUs from either company.

Implementation of a seamless mesh/matrix

This was one of the things which Kristian also put forward as an extension, in theory adapting the GPU code for the TLM shouldn't prove that hard, however applying it to a seamless mesh could prove challenging.

Improve granularity of the collected data

The current data collection process is limited to only collecting data about each discreet process in the TLM simulation; this made analysing some of the results difficult as it was impossible to drill down to the level of detail required.

Timing the various section of the TLM generation directly might well yield more useful details about potential bottlenecks and improvements.

Critical Evaluation

Initial Impressions and Ideas

As I've always had an interest in the future direction of technology and games programming I initially my thoughts for a final year project were revolving around a multi-threaded game engine investigation, however as soon as Kristian's project was brought to my attention I very quickly started to get many ideas drawn from my experience with graphical programming and my interest in looking into GPGPU programming at some point in the future; this project seemed the perfect chance to merge both of those interests into one.

Fortunately like Kristian I was able to draw on the fact that others before me had done work on the TLM implementation so that from my point of view all I had to do was understand their code and perform the adaptations to GPU programming instead. While I did spend a large amount of time reading over Kristian's work I decided in the end not to implement everything he or expand upon it too much. In hind sight I realise it might well have been worth implementing some of his extensions, such as the damping model, if only to see how they would function in a pure GPU world.

During the development of this project I've become quite intrigued as to how I could implement this algorithm in my own game projects; when I initially talked to Steven Harris

about this one of the ideas we both shared was the idea of some kind of ‘active armour’ effect which could be used when a player was struck with certain weapons; while I’m not sure that the current implementations or hardware could stretch to that on any large scale I can see potential in it which I hope to exploit in the near future in my own projects.

Meeting the Objectives

While I had initially hoped to generate a full simulation complete with a GUI interface which would allow dynamic adjustment and selection of the TLM methods this had to be shelved relatively early on into the project.

This was a disappointment to me, however it was one which I felt couldn’t be helped; with the failure of ATI’s D3D extension which was required to implement this project in D3D (see fig 16), I had to return to my OpenGL roots and with this give up the D3DX library and its GUI components. While GUI systems to exist which would work with OpenGL they are generally a relative pain to get working and I have ‘issues’ with the APIs as such I would have probably ended up spending more time (re-)designing and developing a GUI than working on the project proper which would have been a large waste of my time.

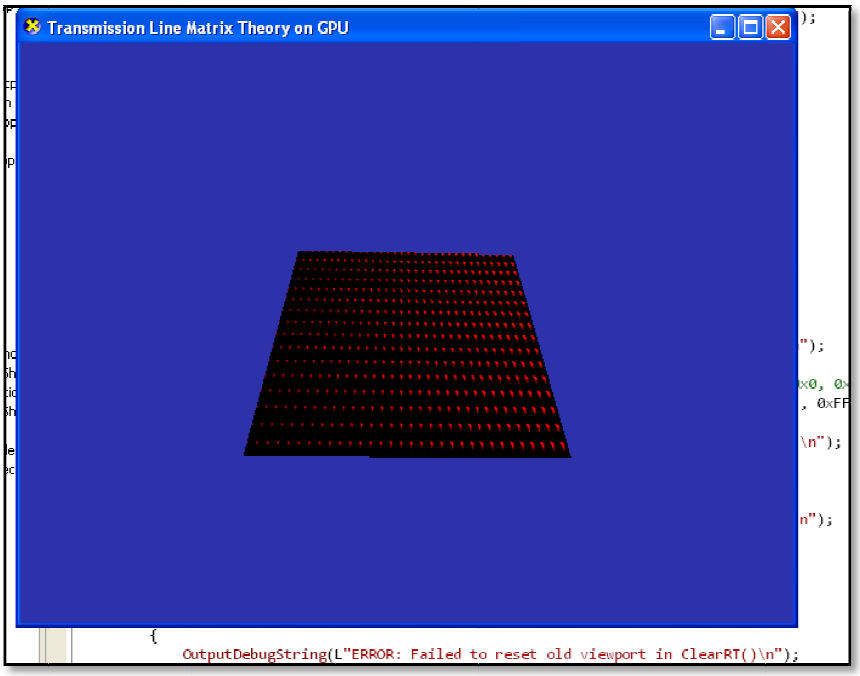


Figure 16 Failed D3D TLM Application

However, with the modified objectives in place, sans the GUI, I was happy with how the program turned out on the whole. The fact that I could leave it for long periods of time to collect all the data required in an automated manner made the job much less tedious than it could have been; however the downside of this was a complete run meant that my computer was out of action for around 2 hours leading to some sitting around and reading while the computer got on with the process. However this was better than having to select and activate each process in turn.

Documentation and Code Quality

I hope that the document produced to follow this project is understand able by most people, I admit that his is a problem I can have as I'm so used to dealing with these concepts that over time I tend to lose sight of what others don't know about the subject. This could be considered a problem with spending a large portion of your life talking with those whose interests lie along the same lines as your own.

At the same time GPGPU isn't a subject which is easy to explain without being able to assume some understanding of the GPU and 3D pipeline so I'm happy with the job I've done explaining it while not being too specific to the problem domain to allow others access to how things work.

The code itself is about as clear as I can make it; while comments are handy I tend to favour a more verbose variable naming scheme where possible to try to make it clear how things are working. That said, I did get into some problems near the start of the project when I confused the difference between a 'map' and 'buffer' variable leading me to see some very strange results until I got it fixed.

Planning the Project

My project plan was pretty accurate with my projected times, having been programming for a number of years now I can get a feel for how long things are going to take for me to complete and then pad that to try to allow for unforeseen situations arising.

One of my regrets is that I didn't make better use of Steven Harris to keep me on task; while I didn't drift too far off and managed to self motivate myself during the process there is a possibility that with his input I could have covered a few of the ideas from the 'future research' list above. However, as I'm used to having to self motivate both when I program as a hobby and when I did it as a job I very slipped back into the mind set of 'I have a task to do for this date so I'll just sit down and do it'.

In the case of the deliverable, as I've already mentioned I wasn't happy that I had to drop the GUI, however the rest of the project lent itself to the evolutional RAD approach taken; by build step by step on it I was able to maintain forward momentum during the project.

I found that, much like Kristian, I became more interested in the development of the program than in the final testing and analysis of the project. This is another area where if I had been talking to Steve I might have gained some extra focus and been able to perform some more in-depth analysis of the project. Due to how I develop applications there are in fact very few or no written notes in existence with regards to how this project was developed. Most of the project was developed on the fly or in my head as I walked to and from various locations; as such no formal design process exists for the project.

Final Thoughts

In developing this project I got a chance to indulge in my interest in 3D graphics as well research into some areas I wanted to look into at some point in the future. I hope that, much like my predecessors on this project this document and project will be used in the future, I certainly know that I will be looking to use what I have learnt here in my own applications in the future.

The point when I knew that this project was going to work properly was most definitely the moment when I finally had a full TLM simulation working on the GPU of my PC; while I had no doubt in my ability to do it the feeling you get from seeing something you've worked on for some time work properly can't be matched. Of course this was shortly followed by a realisation of just how slow it was going which lead me to discover a bug in ATI's OpenGL

driver as well as much yelling and gnashing of teeth however it has been established by others that you can't have everything in this life and the bug was easy to work around in the end.

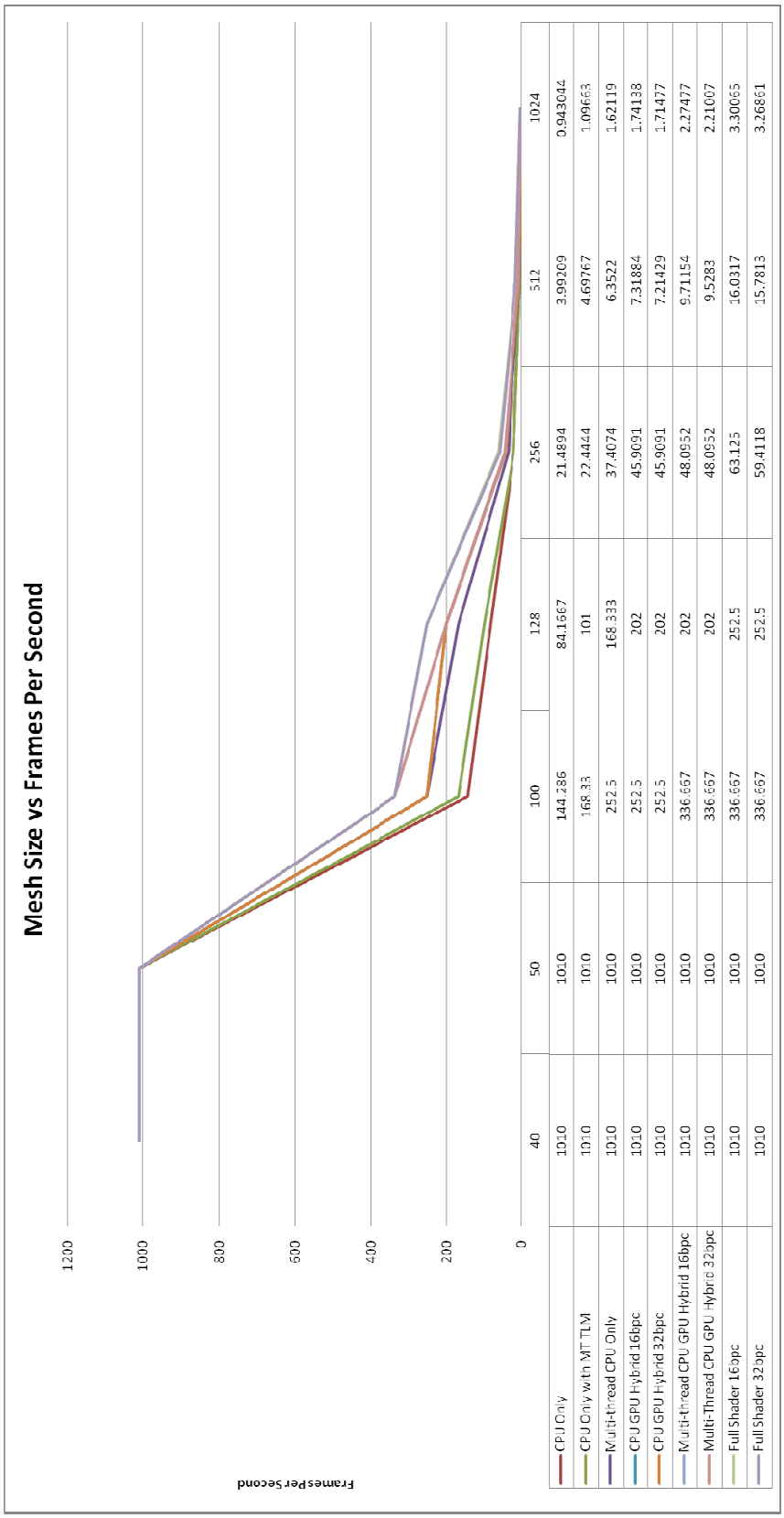
I'm looking forward to finding the time to implement what I have learnt here in my own projects in the future and I can only hope that this document goes some way to giving someone else something to work on after me.

Appendices

A - Time Plan

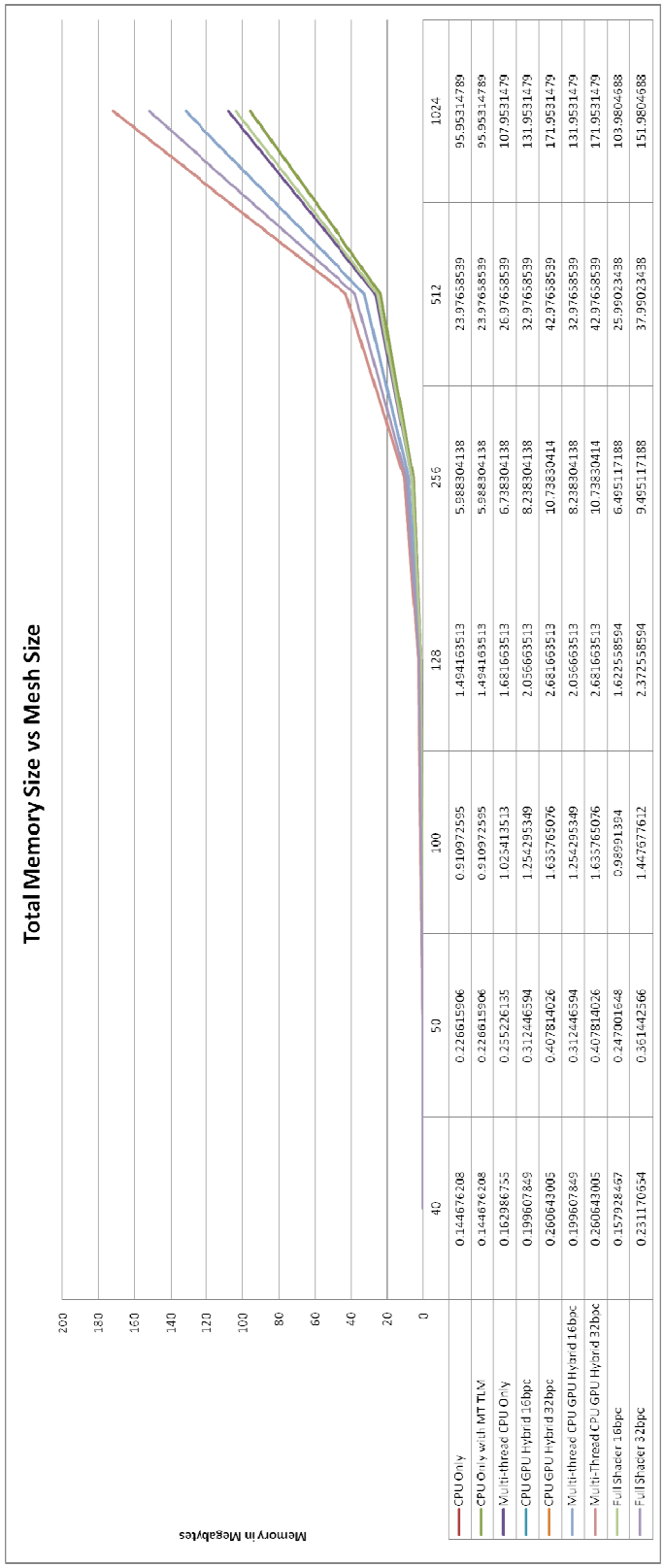
ID	Name	Duration	Start	Finish	Predecessors	Outline Level	Notes
1	Investigate TLM Theory	10.d	30/10/2006 08:00	10/11/2006 17:00		1	
2	Investigate 'close to the metal' API	75.d	13/11/2006 08:00	23/02/2007 17:00	1	1	
3	Select API	15.d	13/11/2006 08:00	01/12/2006 17:00	1	1	
4	Produce Basic Framework	10.d	04/12/2006 08:00	15/12/2006 17:00	3	1	
5	Adapt previous code to new framework	20.d	18/12/2006 08:00	12/01/2007 17:00	4	1	
6	Produce Hybrid CPU/GPU implementation	10.d	15/01/2007 08:00	26/01/2007 17:00	5	1	
7	Produce GPU Only implementation	10.d	29/01/2007 08:00	09/02/2007 17:00	6	1	
8	Produce D3D10 implementation	20.d	12/02/2007 08:00	09/03/2007 17:00	7	1	
9	Produce 'Close To the Metal' implementation	10.d	12/03/2007 08:00	23/03/2007 17:00	8,2	1	
10	Produce Final Application	15.d	26/03/2007 08:00	13/04/2007 17:00	9,8,7,6,5	1	

B – Mesh vs. Frames per Second



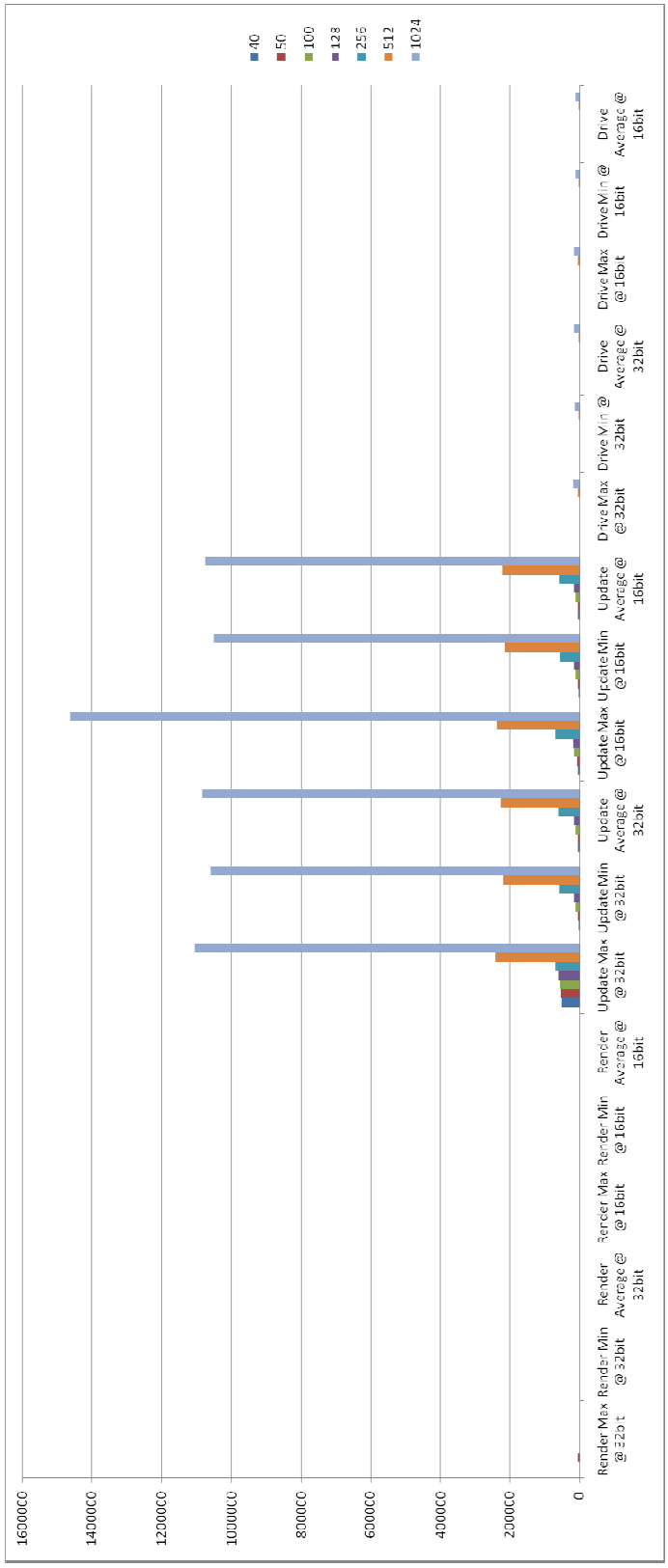
	40	50	100	128	256	512	1024
CPU Only	1010	1010	144.286	84.1667	21.4894	3.99209	0.943044
CPU Only with MT TLM	1010	1010	168.33	101	22.4444	4.69767	1.09663
Multi-thread CPU Only	1010	1010	252.5	168.333	37.4074	6.3522	1.62119
CPU GPU Hybrid 16bpc	1010	1010	252.5	202	45.9091	7.31884	1.74138
CPU GPU Hybrid 32bpc	1010	1010	252.5	202	45.9091	7.21429	1.71477
Multi-thread CPU GPU Hybrid 16bpc	1010	1010	336.667	202	48.0952	9.71154	2.27477
Multi-Thread CPU GPU Hybrid 32bpc	1010	1010	336.667	202	48.0952	9.5283	2.21007
Full Shader 16bpc	1010	1010	336.667	252.5	63.125	16.0317	3.30065
Full Shader 32bpc	1010	1010	336.667	252.5	59.4118	15.7813	3.26861

C – Mesh Size vs. Memory



	40	50	100	128	256	512	1024
CPU Only	0.144676208	0.226615906	0.910972595	1.494163513	5.588304138	23.97638539	95.95314789
CPU Only with MTTLV	0.144676208	0.226615906	0.910972595	1.494163513	5.588304138	23.97638539	95.95314789
Multi-thread CPU Only	0.162986755	0.255226135	1.025413513	1.681663513	6.738304138	26.97638539	107.9531479
CPU GPU Hybrid 16bpc	0.199607849	0.312246594	1.254295349	2.056663513	8.238304138	32.97638539	131.9531479
CPU GPU Hybrid 32bpc	0.260643005	0.407814026	1.635765076	2.681663513	10.73830414	42.97638539	171.9531479
Multi-thread CPU GPU Hybrid 16bpc	0.199607849	0.312246594	1.254295349	2.056663513	8.238304138	32.97638539	131.9531479
Multi-Thread CPU GPU Hybrid 32bpc	0.260643005	0.407814026	1.635765076	2.681663513	10.73830414	42.97638539	171.9531479
FullShader 16bpc	0.157928457	0.247001648	0.989911394	1.622558594	6.495117188	25.99023438	103.9804688
FullShader 32bpc	0.231176564	0.361442566	1.447577612	2.372558594	9.495117188	37.99023438	151.9804688

D – GPU Processing Time Graph



Bibliography

Akenine-Möller & Haines, 2002	**Real-time Rendering**, second edition, A K Peters, Ltd, Natick, MA. ISBN: 156881-128-9
ATI, 2006	**ATI CTM Guide: Technical Reference Manual**, available from http://ati.amd.com/companyinfo/researcher/documents/ati_ctm_guide_beta.pdf
Brook, 2004	**Brook for GPUs: Stream computing on Graphics hardware**, Buck et al, ACM Press, http://doi.acm.org/10.1145/1186562.1015800
Christopoulos, 1995	**The Transmission-Line Modelling Method**, IEEE Press/Oxford Press, 1995
Dempski & Viale, 2005	**Advanced Lighting and Materials with Shaders**, Wordware Publishing, Inc. Plano, Texas. ISBN: 1-55622-292-0
Dor, 2006	**Dynamic Surfaces using Transmission Line Matrix Theory**, Suffolk College, Ipswich, Suffolk
Folding, 2006	http://folding.stanford.edu/FAQ-ATI.html
Gems, 2000	**Game Programming Gems**, Mark Deloura el al, Charles River Media Inc, ISBN: 1-58450-049-2
GPUGems 2, 2004	**GPU Gems 2,** Matt Pharr et al, Addison-Wesley, ISBN: 0-321-33559-7
Harris S, 2005	**TLM Test Application (code)**, Suffolk College, Ipswich, Suffolk.
More OpenGL Game Programming, 2006	**More OpenGL Game Programming, OpenGL Shading Language Chapter**, Robert Jones, Thomsom Course Technology, ISBN: 1-59200-830-5
MS, 2006	**DirectX SDK**, available from http://msdn.microsoft.com/directx/
OpenGL Guide, 2003	**OpenGL Programming Guide**, 4th Ed, Woo et al, Addison Wesley, ISBN: 0-321-17348-1
OpenMP, 2004	http://www.openmp.org/drupal/node/view/11
Radeon, 2006	**Radeon X1x00 Programming Guide**, Guennadi Riguer, Part of ATI/AMD SDK, available from http://ati.amd.com/developer/radeonSDK.html
Shader Algebra, 2004	**Shader Algebra**, McCool et al, University of Waterloo, ACM Press, http://doi.acm.org/10.1145/1186562.1015801
The Direct3D 10 System	**The Direct3D 10 System**, David Blythe, Microsoft Corporation, Available from http://download.microsoft.com/download/f/2/d/f2d5ee2c-b7ba-4cd0-9686-b6508b5479a1/Direct3D10_web.pdf
The OpenGL Shading Language, 2004	**The OpenGL Shading Language**, Randi J. Rost, Addison Wesley, ISBN: 0-321-19789-2
Persson E, 2006	**R2VB Programming**, available as part of the March 2006 SDK from ATI at http://ati.amd.com/developer/radeonSDK.html