REMOTE EXECUTION FOR MOBILE 3D GRAPHICS

a thesis
submitted to the department of computer science
and the committee on graduate studies
of Worcester Polytechnic Institute
in partial fulfillment of the requirements
for the degree of
Master of Science

Kutty S Banerjee
May 2005

I certify that I have read this thesis and that, in my opinion, it is fully adequate in scope and quality as a thesis for the degree of Master of Science.

Emmanuel Agu
(Principal Adviser)

I certify that I have read this thesis and that, in my opinion, it is fully adequate in scope and quality as a thesis for the degree of Master of Science.

Fernando Colon Osario
(Reader - Computer Science)

I certify that I have read this thesis and that, in my opinion, it is fully adequate in scope and quality as a thesis for the degree of Master of Science.

Michael Gennert
(Head of Department,Computer Science)

Approved for the University Committee on Graduate Studies.

Abstract

Mobile clients such as PDAs, laptops, wrist watches, smart phones are rapidly emerging in the consumer market and an increasing number of graphics applications are being developed for them. However, current hardware technology limits the processing power on these mobile devices and wireless network bandwidth can be scarce and unreliable. A modern photorealistic graphics application is resource-hungry, consumes large amounts of cpu cycles, memory and network bandwidth if distributed. Besides running them on mobile devices may also diminish their battery power in the process. Bulk of graphics computations involve floating point operations and the lack of hardware support for such on PDAs imposes further restrictions. Remote execution, wherein part or the entire rendering process is offloaded to a powerful surrogate server is an attractive solution. We propose pipeline-splitting, a paradigm whereby 15 sub-stages of the graphics pipeline are isolated and instrumented with networking code such that it can run on either a graphics client or a surrogate server. To validate our concepts, we instrument Mesa3D, a popular implementation of the OpenGL graphics to support pipeline-splitting, creating Remote Mesa (RMesa). We further extend the Remote Execution model to provide an analytical model for predicting the rendering time and memory consumption involved in Remote Execution. Mobile devices have limited battery power. Therefore, it is important to understand if during Remote Execution, communication is more power consuming than computation. In order to study the same, we develop PowerSpy, a Real Time Power Profiler for I/O devices and applications. Finally, we add Remote Execution to an existing Distributed Graphics Framework targeted for mobile devices, namely, MADGRAF. In addition to Remote Execution, MADGRAF has another policy known as the Transcoder Based

Approach in which the original 3D graphics image is modified to suite the mobile devices' rendering capacity. Though this speeds up the rendering process, it affects photorealism. We propose an intelligent runtime decision making engine, Intelligraph, which evaluates the runtime performance of the mobile client and decides between Remote Execution and the Transcoder Based Approach.

Contents

List of Tables

List of Figures

Acknowledgments

First and foremost I would like to thank my amazing thesis advisor, guide, teacher and friend, Prof. Emmanuel Agu. His constant encouragement during some of the depressing points of my research and his never ending insistence for perfection at all times has helped me come up as a researcher, problem solver and programmer. I would also like to thank Prof. Fernando Colon Osorio for being the reader for this thesis.

I would like to thank my brother Shibram Banerjee, he has always been an inspiration for me. I owe most of what I have accomplished in life to him. My heartfelt thanks to my girlfriend Lini, for her patience and support. Many thanks to my friend Sasidhar Tadanki. Though I never really got a chance to rival his tally of publications, he nearly convinced me that reading and writing papers can be fun. Special thanks to my friends, Malav and Rahul and their ability to remain placid and cheerful at all times, it almost rubbed off on me too. My sincere gratitude towards the members of the OSR Online newsgroup for patiently listening to my novice queries on Windows Kernel Programming. I am also greatly indebted to Prof. Gary Police. His course on Software Engineering has been one of the best courses I have ever attended in my life.

Last but not the least I would like to thank my parents and GOD above all!!

Chapter 1

Introduction

High end graphics applications such as 3D games, interactive multimedia applications are becoming more and more popular as are mobile devices such as PDAs, laptops. However rendering high end 3D graphics on mobile devices has traditionally been a challenge due to the low processing power, memory and energy. Besides mobile devices lack a proper hardware floating point unit or graphics accelerator.

MADGRAF (Mobile and Adaptive Distributed Graphics)[2] is a distributed 3D graphics rendering engine targeted specially for mobile devices. One of the primary goals of MADGRAF is to render complex 3D image files stored on the server in *Real Time* on the mobile client machine. However, rendering on mobile devices with its hardware limitations can easily take a large amount of time besides draining its battery power. Therefore, the key challenge for MADGRAF is to use the services of a powerful server machine to aid the client in rendering the image.

1.1 MADGRAF System Architecture

Figure 1.1 describes the system architecture of MADGRAF. The MADGRAF server is a repository of 3D graphics models(files). The Mobile device requests the MADGRAF server for the 3D graphics models. These files contain data which is typically passed to a rendering engine to produce the final graphics output. This rendering process can be done either completely on the mobile device or as part of Remote Execution

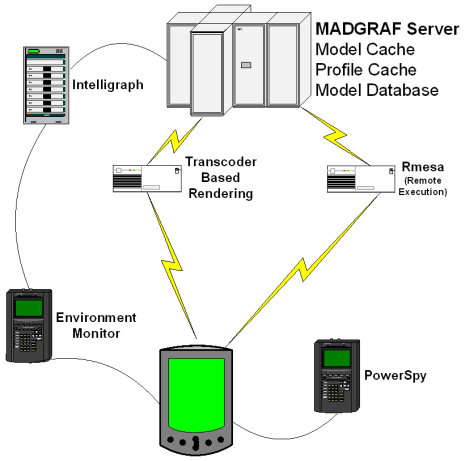


Figure 1.1: MADGRAF System Architecture

can be shared between the mobile device and the MADGRAF server. On the other hand, the MADGRAF server can modify the original model to a simplified model as part of the the Transcoder Based Rendering. This simplified model can then be rendered either entirely on the mobile device or shared between the client server using Remote Execution. The system performance on the mobile device is reported by the Environment Monitor. The rendering process on the mobile device comprises of either the actual rendering process or offloading rendering work to the server. This offloading process involves network communication. An important question being, does communication involve higher power consumption than computation. PowerSpy measures the power consumed by the rendering process and separates the computation power consumption from the communication power consumption. The results of PowerSpy and the Environment Monitor are fed to Intelligraph which decides between Remote Execution and the Transcoder Based Approach at runtime. Each of these approaches have their pros and cons as will be discussed later in this chapter.

1.2 Transcoder Based Rendering

Transcoder Based Rendering is an approach in which the original file stored in the MADGRAF server is transformed with the intention of reducing the complexity of

the graphics file. This simplified file is then transmitted to the mobile device for rendering. The time taken to render graphics files is linearly proportional to the size of the graphics file, therefore, this is a guaranteed method of speeding up the client side rendering time. The disadvantage of this approach as is apparent, is the reduction in the final image quality.

1.3 Remote Execution

The other approach is to use Remote Execution. In Remote Execution, the original file on the server is displayed on the mobile device without any modifications. However, the rendering process is split between the client and the server. Thus, instead of performing the entire rendering on the client machine, part (or all) of it is done on the server machine. The disadvantage of this approach is the increased communication between the client and the server which can be a challenge in a wireless medium. We introduce a Remote Execution Framework in MADGRAF, namely, RMesa (Remote Mesa)[1] which we will cover in details in chapter 2.

1.4 Environment Monitor

The two approaches, namely, Transcoder Based Rendering and Remote Execution, each have their own pros and cons. We propose to combine both approaches, intelligently at runtime, by gauging the runtime performance of the system periodically. Which gives rise to the notion of a component that measures the performance of the system at runtime and benchmarks the same. This is the role played by the Environment Monitor in the figure 1.1. In particular, the Environment Monitor dynamically monitors the following metrics on the mobile client,

1. *CPU Load*
2. *Free Memory available*
3. *Percentage of Battery Power remaining*

4. *Interactivity of the graphics application*

In 5.3, we will cover in details, why the above metrics are important and how they influence the execution of graphics applications on mobile clients.

1.5 Intelligraph: Intelligent Decision Making

In this section, we will elaborate on the importance of an Intelligence Agency in the MADGRAF architecture. As mentioned in 1.4, we analyze the performance of the system (client) at runtime to decide between the Transcoder Based Approach and Remote Execution. This requires a component that reads in the system performance data and decides between the two approaches.

Further, the Transcoder Based Approach, as mentioned earlier, modifies the existing graphics files in the MADGRAF server. An example of this approach is a process called Simplification [12] in which, a graphics file containing a fixed number of vertices is reduced to a one containing lesser number of vertices. However, the extent of simplification is a parameter that needs to be supplied to the Transcoder. We require some form of intelligence to decide at runtime, what is the best extent of simplification for the current graphics file.

In Remote Execution, different stages of the graphics engine are offloaded to the server for execution. These different stages are not symmetrical. Besides, offloading stages can incur network overhead (read as increased latency). Therefore, we require some form of external intelligence to know which are the stages that may be offloaded for optimal performance.

All these intelligent decisions are taken by the component named Intelligraph in the figure 1.1. To summarize, Intelligraph needs to provide the runtime with answers to the following questions:

1. *When must the Transcoder Based Approach be used over Remote Execution?*

2. *If Remote Execution is used, then what are the different stages of the graphics engine that must be offloaded to the server?*

3. *If the Transcoder Based Approach is used, then what is the extent to which the server graphics files can be modified?*

We will cover the fine prints of Intelligraph in chapter 5.

1.6 PowerSpy and Energy Crisis in Mobile Devices

One metric of particular interest to us is the power consumed by applications. During Remote Execution, stages of the graphics engine are offloaded for execution on the server machine. A key research problem of interest to us is: *How does Remote Execution effect the power consumed by the graphics engine?*. In other words, if the stages were allowed to execute on the mobile device, would the processing power be higher than the power consumed in transmitting the information to the server? Separation of I/O device power consumption from processor level power consumption is a key problem in this area.

We will investigate further into this problem in chapter 4 and explain the component named PowerSpy[3] in figure 1.1.

1.7 Thesis Contribution

The purpose of this thesis is to develop the following components to augment the Distributed Graphics Rendering Framework, MADGRAF:

1. *RMesa*: Remote Execution Framework

2. *PowerSpy*: A low level fine grained Power Profiler.

3. *Intelligraph*: An Intelligence agency which controls the policies of the MADGRAF runtime.

4. *Environment Monitor*: Monitors the performance of a live running MADGRAF system and reports the same to Intelligraph.

The rest of the thesis is organized as follows. Chapter 2 discusses our Remote Execution Model in details while chapter 3 discusses an analytical model developed to study the rendering speed and memory consumption of Remote Execution. Chapter 4 talks about the Power Measurement tool that we have developed, namely, PowerSpy. Chapter 5 discusses the Intelligent Decision Making process in MADGRAF. Finally in chapter 6, we present our conclusion, some of the experiences in developing the MADGRAF system and brief comments about the overall performance of the MADGRAF system.

Chapter 2

RMesa - Remote Mesa

2.1 RMesa

MADGRAF is a Distributed Graphics Framework targeted for mobile devices that are low on processing power, memory, battery power and often lack a hardware floating point unit. Rendering 3D graphics can be an exhaustive process even on high end server machines. One of the primary goals of MADGRAF is to simplify the rendering process on the mobile client. Remote Execution and the Transcoder Based Approach are two alternative solutions provided by MADGRAF.

Geometric [4] and image-based simplification [11] and 3D compression [6]have all been proposed as part of the Transcoder Based Approach to improve the performance of highly interactive graphics components such as the foreground characters in flight simulators and computer games. However, these techniques involve some form of degradation which compromise the photorealism of rendered images.

Remote execution of portions or the entire 3D graphics pipeline is a better fit for photorealistic rendering where the quality of the image increases with the number of faces or polygons in the input mesh model. However, key challenges exist. 3D graphics libraries are generally deployed as closely-coupled parts which reside on a single client machine, which makes distribution challenging. Also, user interaction with a remotely executing stage may incur a penalty of a roundtrip delay.

We propose pipeline-splitting, a novel paradigm in which the stages of the 3D

graphics pipeline are isolated and networked such that each stage can be mapped to either the client or server. Thus, a weak mobile client can use a powerful surrogate server to execute parts or whole of low interactivity, compute-intensive graphics applications with the goal of increasing the overall execution speed and extending the battery life of the mobile host.

To validate pipeline-splitting, we create Remote Mesa (RMesa) by incrementing Mesa3D, a popular implementation of the OpenGL graphics API to support our novel pipeline splitting mechanism. We then run an OpenGL-based VRML browser as our test application and establish client-server mappings of the graphics pipeline stages which perform well, as well as conditions under which remote execution is an optimal solution for different input mesh model sizes. In addition to traditional performance metrics such as total rendering time, we extensively monitor the impact of various stage mappings on mobile client resources including battery and network bandwidth usage. We also carry out tests on actual PDA machines to determine the impact of remote execution in an environment that is not just constrained in terms of processing power, memory and energy but also lack of floating point support in hardware. Our results show a huge performance gain for PDAs executing floating point intensive operations of the graphics pipeline remotely.

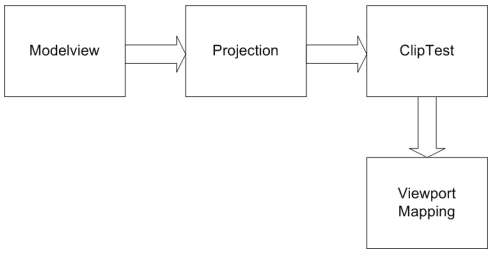
2.2 The 3D Graphics Pipeline

Rendering in computer graphics is typically organized such that input primitives such as triangles, quads, vertices are processed in a series of sequential steps in the form of a pipeline[1].

The main stages of the graphics pipeline are the geometry, per-fragment and rasterization stages, as shown in figure 2.1(a). The "Geometry" stage performs geometric operations such as transformation, projection and clipping on individual vertices and primitives. The output of the geometry stage is fed directly to the "Per-Fragment" stage which tests to check if the fragments (set of vertices are eventually transformed

[1]This is analogous to the process of refining crude petroleum wherein it goes through various purification processes

(a) Individual stages of the graphics pipeline. Software renderers execute them sesequentially. The Geometry stage operations comprise mostly floating point operations



(b) Geometry stage has further sub stages that are executed sequentially.

Figure 2.1: Graphics Pipeline Overview

to fragments) can be displayed on the screen. The fragments are converted to pixels in the "Rasterization" stage where operations on individual pixels are carried out.Each of the stages in figure 2.1(a) are made up of sub stages. In figure 2.1(b), the sub stages of the Geometry stage are shown.

2.3 Pipeline Splitting

Typically, the entire 3D graphics pipeline is implemented as a closely-coupled library (such as OpenGL[31] or DirectX[32]) which resides on a single machine. In this form, distribution requires that either the entire library resides on a client machine or on a surrogate server. We believe that this granularity of components is too coarse and propose a novel concept, called *pipeline-splitting* wherein we isolate and sub-divide the graphics pipeline into its stages such that the individual stages in figure 2.1(a) can execute on different machines. For instance, the Geometry stage could be rendered on the local client machine whereas the per-fragment and rasterization stages can be sent to a more powerful[2] server machine for execution. The functionality in each stage of the pipeline is replicated on both the client and server and the process of remote execution of a stage involves the following sequence of actions involving the client and the server: (1) Client packs the data required for executing the stage on

[2]in terms of CPU speed, memory, graphics card

the server into network format. This includes taking care of byte order, marshalling pointers[3].(2) Client sends this data across the network to the server.(3) Server unpacks the data and gets it into the native format used by its machine.(4) Server executes the stage.(5) Server packs the data back into the chosen network format.(6) Server sends this data across the network to the client.(7) Client unpacks the received data into the format used on its local machine.

For illustrative reasons, let us consider that the time taken to execute the rasterization stage on the client machine is 40ms and 10ms on the server for a certain graphics model. So, for remote execution of the rasterization stage, the time taken to transmit, pack, unpack data and execute the stage on the server must take less than 40ms for remote execution to be beneficial (the total roundtrip time should be much lesser than 40ms). Thus, the gains from remote execution is directly related to the cost of executing individual stages on a given client and server plus the incurred network delay. Intuitively remote execution performance improves as the speed of the client, server and network increases. Low powered mobile clients could greatly benefit from utilizing a powerful surrogate server, across a fast wireless network (such as a wireless LAN), for rendering.

2.3.1 Granularity of Pipeline Splitting

As previously explained in section 2.2, graphics pipeline is composed of individual stages which in turn are composed of sub-stages that could be considered for local or remote execution. If each pipeline sub-stage can be executed either locally or remotely, then we say that the granularity of pipeline splitting is a *single sub-stage of the graphics pipeline*.

2.4 RMesa

As explained in Section 2.2, the graphics pipeline has various stages. In order to offload certain Portions of the graphics pipeline for the server to execute, it is necessary

[3]pointers need to be dereferenced before they can be sent across the network

Figure 2.2: VRML File displayed on a PDA

to *split the stages of the graphics pipeline*. This implies being able to have location independence in terms of execution of a single stage/sub-stage of the graphics pipeline. Thus, the stage/sub-stage could be rendered on the client machine, or could be rendered on a remote server machine. To validate pipeline-splitting, we instrument Mesa3D [34], an open source implementation of OpenGL with socket networking code between the graphics sub-stages to create Remote Mesa (RMesa). Thus, RMesa provides infrastructure that facilitates sub-stage level pipeline-splitting granularity. In this paper, we describe a basic configuration-file-based stage mapping mechanism that allows us to flexibly map the stages of the graphics pipeline to the client or server and thus perform performance analysis. However, it is our vision that in the future, we shall develop more sophisticated external decision engine that controls the mapping of pipeline stages, as well as full operating system support.

2.4.1 RMesa System Architecture

The RMesa system architecture is shown in figure 2.3. The base components include the RMesa Client and the RMesa server. The RMesa Client also has a 'Stage Map Control' unit to control the current stage mapping of the client.

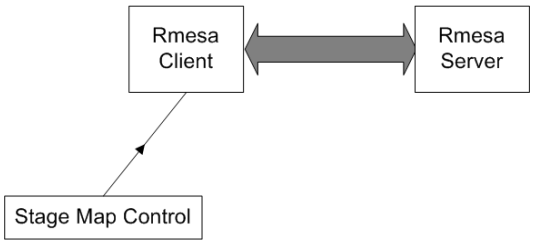


Figure 2.3: RMesa System Architecture. The RMesa Client runs the graphics pipeline in which each stage can also be computed at the RMesa Server. This decision is dynamically controlled bythe StageMap Control.

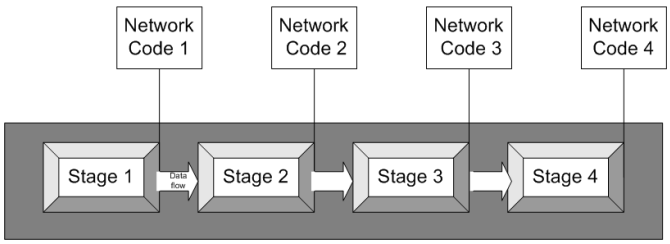


Figure 2.4: Networking in the Pipeline. The N/W code at the start of each stage sends data to the server whereas the N/W code at the end involves fetching results of the stage operations from the server.

2.4.2 RMesa Client

The RMesa client is a software only OpenGL implementation. The basic OpenGL implementation used is that of Mesa[34]. The principle behind pipeline splitting is shown in figure 2.4.

At the end of each stage in figure 2.4, there is a placeholder for networking code such as sockets or Remote Procedure Calls (RPC). At each such point, the flow of a quick decision is made (or pre-determined mapping is looked up) as to whether the data will be allowed to flow ahead or whether the data needs to be transferred to a remote machine for the execution of the next stage. For e.g., at *Network Code 1* in figure 2.4, a decision could be taken to send the data to a remote machine which has a similar pipeline structure. Thus instead of going over to *Stage 2* on the local machine, the *Stage 2* could be executed on the remote machine. Again at the end of *Stage 2*, at *Network Code 2*, a similar decision can be made. *This is the basic principle behind pipeline splitting.* The RMesa Client consists of (1) *An OpenGL implementation with*

hooks in the graphics pipeline stages. (2) *The capability at each sub-stage to execute the next stage on a remote machine with a mirror graphics pipeline.* (3) *A Provision to accept input that controls whether a particular stage is to be executed locally or remotely.*

2.4.3 Stage Map Control

The Stage Map Control Unit on the RMesa client determines on which machine (client or server) a given OpenGL sub-stage will run. The Stage Map Control is a separate process that connects to the RMesa Client using TCP/IP. Thus, this unit does not have to be present on the same machine as the RMesa client. The Stage Map Control unit in turn reads the stage map data from a configuration file which can be controlled either directly by the user or can be modified by any external component. Currently, we manually write to this configuration file. However, in the future , an external *Intelligent Unit* might decide on stage mappings based on more sophisticated algorithms and change this configuration file based on dynamic conditions of the wireless network and the machines in order to optimize speed, memory, power consumption and bandwidth usage.

2.4.4 RMesa Server

The RMesa server continuously waits for input from different client machines asking for particular stages of the graphics pipeline to be rendered, and processes such requests on a *First-In-First-Out (FIFO)* basis.

Each RMesa server can concurrently process many RMesa clients and state information about each connection is maintained by the RMesa client. Thus each RMesa client, informs the server of what stages it needs rendered and data (such as input or partially processed vertices) required for those stages. After executing the requested stages, the server closes the connection and does not maintain any further information about the closed connection. It should be noted that the RMesa server can concurrently handle multiple RMesa client requests which involve different stages of the graphics pipeline.

```
Server Address=sochi.wpi.edu
Server Port=3354
```

Figure 2.5: RMesa Client Config File

```
Geometry:
ModelView=client
Projection=client
ClipTest Perspective=client
Viewport=server
```

Figure 2.6: Stage Map Configuration File

2.5 RMesa Implementation

In this section, we take an in-depth look at the instrumentation of Mesa carried out in order to obtain pipeline splitting.

The graphics pipeline that is described in greater detail in [28] is instrumented to enable each of the sub stages to be rendered on either a client machine or a remote server machine. This instrumentation of the regular graphics pipeline is called the RMesa graphics pipeline which is shown in figure 2.7(a). Our implementation facilitates flexible mapping of 15 sub-stages to either the client or server. All stages and substages in figure 2.7(a) have location independence with respect to execution. ClipTest and Perspective Divide substages have been kept together due to the tight coupling of the data structures used within both in the Mesa implementation.

Configuration Files: The RMesa client connects to the remote RMesa server using a configuration file. A sample configuration file is shown in figure 2.5.

The Stage Map Unit uses a configuration file shown in part in figure 2.6 which *dynamically controls the current stage mapping of the RMesa client.* Thus if the values of the configuration file change at runtime, then the stage mapping used by the RMesa client also changes.

Referring to figure 2.6, the modelview, projection, cliptest and perspective sub

stages of the geometry stage of the RMesa graphics pipeline are rendered on the RMesa client machine. The viewport sub stage is rendered on the server.

Platform: RMesa is currently instrumented to work on a Win32 port of Mesa OpenGL. Work is in progress to port the RMesa client and the server to other operating systems. Under Windows, RMesa produces a dynamic link library, OpenGL32.dll which must be placed in the current directory of the application. This causes windows to load OpenGL32.dll of RMesa into the applications' process memory instead of the default Mesa or any other OpenGL vendor libraries. Since RMesa is a modified Mesa library, therefore it has the basic OpenGL capability in addition to ability to split the graphics pipeline and communicate with a remote server.

2.6 Performance Metrics for Evaluating Pipeline Splitting

In evaluating the merits of pipeline splitting, it is useful to monitor important metrics which give a good idea of the gains due to remote execution. Important metrics include total rendering time, power consumption on mobile client. We consider these metrics in answering the following questions *(1) Does remote execution improve performance? (2) Which client-server mappings of stages/sub-stages lead to the best overall performance?* We now present an intuitive discussion of these metrics, which we shall confirm in later sections via measurement.

2.6.1 Rendering Time

For a mobile device with low processing power, limited battery, memory and graphics resources, a powerful server can speed up the rendering time of a graphics scene. However, if the client is itself a powerful graphics machine, for e.g., a desktop machine with high graphics capabilities, then by executing the stages/sub-stages on the server, we will not gain much as the network overhead involved in remote execution will dominate the rendering time. The difference in the client and server processing power must be substantial for remote execution to yield a faster rendering time. So rendering

time for the scene helps decide whether remote execution should be used.

Graphics scenes are of various types. So, for a graphics scene involving 3 million vertices and with very little complex lighting, the bulk of its operations are in the geometry stage. Therefore remote execution of the geometry stage is beneficial. Thus the complexity of the graphics scene helps in deciding what stages/sub-stages are to be remote executed.

2.6.2 Battery Power Consumption

The rendering process also drains the battery power level of the mobile host. Consider that we have a scene involving 3 million vertices. Even if the mobile host is a laptop with a high performance graphics accelerator, executing the stages on the local client diminishes the battery power, even though the rendering process is fast. Therefore, in conjunction to the rendering time, the battery power consumption is another important force that could compel remote execution. So if by remote executing the above scene, the battery could last for 45 minutes instead of 25 minutes, then remote execution clearly helps save battery power.

2.7 RMesa Test Cases

In order to evaluate the performance of a graphics application using RMesa, we chose to investigate the following stage maps in detail:

1. *Case I:* The Modelview substage of the Geometry stage is rendered on the server as is shown in figure 2.7(a) . All other stages and substages are rendered on the local machine.

2. *Case II:* The Normal substage of the Lighting stage is rendered on the server whereas all other stages and substages are rendered on the local machine.

3. *Case III:* The Depth-Test substage of the Rasterization stage is rendered on the server as is shown in figure 2.8(a) whereas all other stages and substages are rendered on the local machine.

	Mem(MB)	**CPU(MHz)**
Client	64	1.7
Server	1024	2.4

Table 2.1: Client - Server resources

Specifically, using these test cases we will study the timing, power consumption details and network usage of graphics applications using RMesa library.

2.8 RMesa - Performance Analysis

We set up simple experiments to validate our pipeline-splitting concept and RMesa implementation. The key goal of these experiments was to determine specific mesh sizes and graphics workloads for which remote execution is an optimal strategy. After prototyping RMesa, we performed the following tests with meshes of different sizes and different permutations of locating graphics pipeline stages on the server. Each time that the client sends a sub stage to the server for rendering, the following sequence of steps take place: (1) Pack the vertex data in a network worthy format. (2) Transmit the data over the network. (3) Unpack the vertex data on the server side. (4) Perform the server stages on the data. (5) Pack the resultant vertex data. (6) Transmit the resultant data back to the client. If we assume that the time taken to pack and unpack on both the client and the server is "Y", the time taken to render the sub stage on the server is "X", the time taken to render the sub stage on the client alone is "Z", then: $X + Y < Z$ for the sub stage rendering on the server to be a faster process than a local machine rendering.

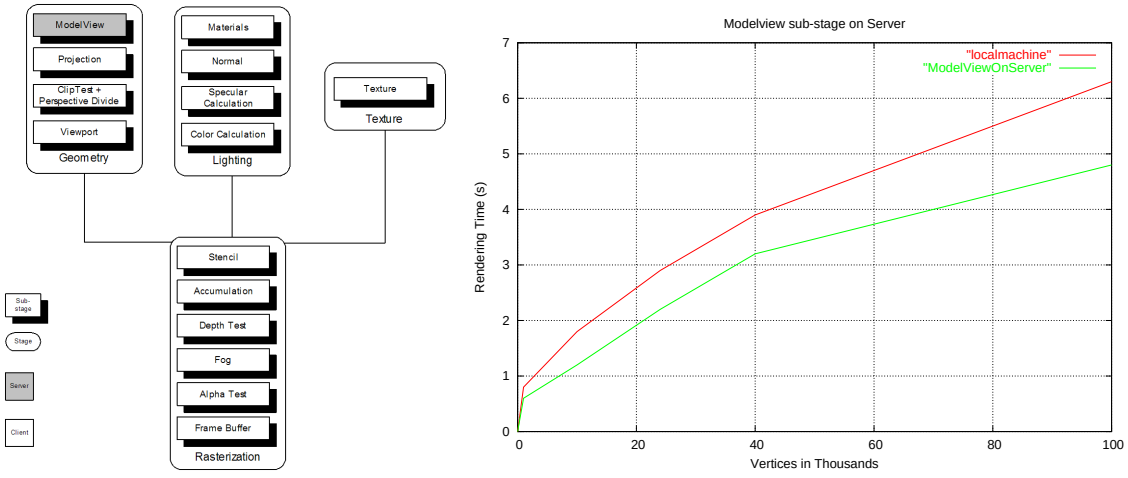
2.8.1 Experimental Setup - Laptop

The hardware configuration of our client and server are as shown in table 2.1 while table 2.2 shows information about our input VRML files.

In order to control the number of vertices generated and sizes of our input VRML files, we created them using the AC3D[33]. A cylinder rendered with 'X' number of segments produces a VRML file containing roughly '2X' number of vertices as shown

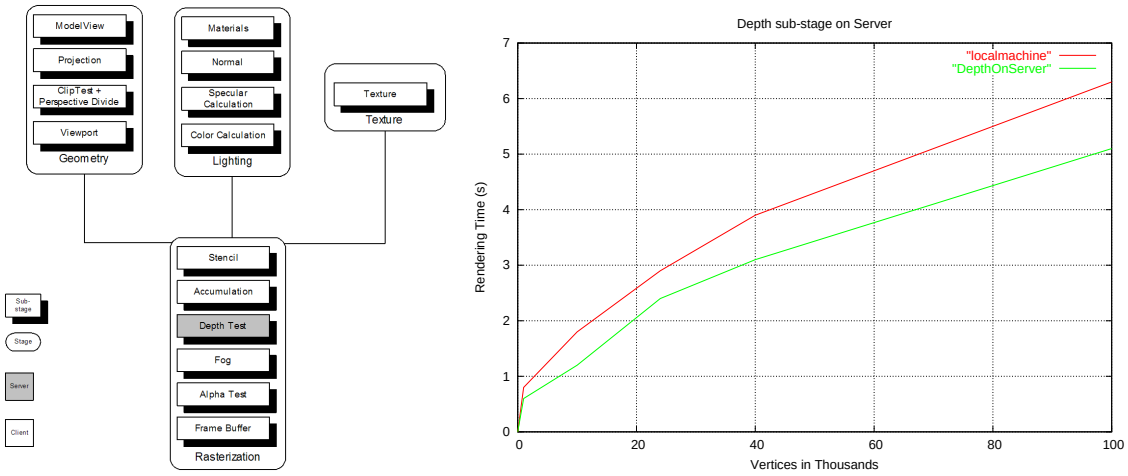
File	#Segments	#Vertices
1K.wrl	500	1,021
10K.wrl	5000	10,023
24K.wrl	12,244	24,564
40K.wrl	20,000	40,044
100K.wrl	50,000	100,212

Table 2.2: Test Case VRML files



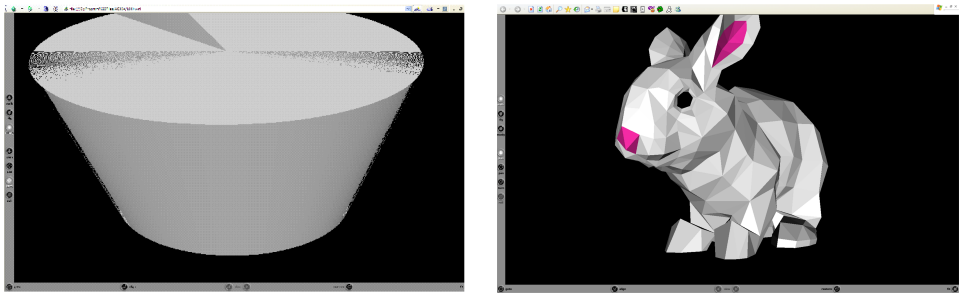
(a) Pipeline View (b) Rendering Performance

Figure 2.7: Model view rendered on server



(a) Pipeline View (b) Rendering Performance

Figure 2.8: Depth Test sub-stage Rendered on the Server



(a) 100k.wrl (b) Bunny.wrl

Figure 2.9: Sample VRML Files

in the table 2.2. Each of these files are then rendered using OpenVRML[35], a VRML File Browser. OpenVRML uses OpenGL for rendering. OpenVRML was made to render using RMesa instead of the original Mesa OpenGL library. A sample rendered image of the file '100k.wrl' is shown in figure 2.9(a). Another high resolution VRML file, 'bunny.wrl' is also shown in 2.9.

Test Results Analysis

Case I: The Modelview substage is rendered on the server machine with all other

stages and substages being rendered on the local machine. The graph in 2.7(b) is plotted for rendering time taken by VRML files of varying number of vertices by mapping Modelview substage onto the server against a complete local machine rendering. The graph shows that for VRML files with lower number of vertices, the rendering time is by and large the same. Close to around 1000 vertices, the rendering time taken on the server starts decreasing.

Case II: The rendering performance for remote mapping shows a steady improvement in performance with increase in number of vertices. However, close to 40,000 vertices, the difference in rendering performance between the remote and local mappings gets marked, with the remote mapping being the better one.

Case III: Referring to 2.8(a), the Depth Test substage being rendered on the server machine shows a similar response to that in Case II.

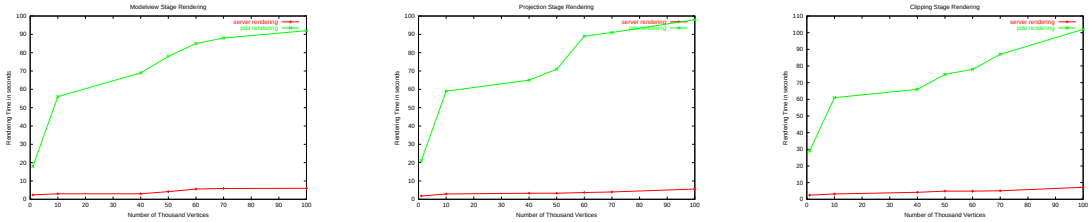
We observer that with a nominal difference in cpu processing power, remote execution starts showing a healthy trend. The trend clearly shows that as the number of vertices in the graphics image increases, remote execution starts to give a clear advantage over local execution.

2.8.2 Experimental Setup - PDA

Encouraged by the improvement in performance on two processors with differing speeds, we executed our tests on a *iPAQ h4300* with 64MB memory and 400MHz processor *without floating point support in hardware*. The server used is the same as in 2.8.1. The matrix multiplication operations involved in each of the geometry graphics pipeline stages

- Modelview
- Projection
- Clipping

were executed on the PDA as also the server. The goal behind this experiment was to determine:



(a) Modelview Stage (b) Projection Stage (c) Clipping Stage

Figure 2.10: Individual stages of Geometry being rendered on the pda and the server. The huge difference between the server and client rendering time, is clearly visible

- What is the impact of increasing difference in processing speeds of server and client. We have already seen a gain in 1 second with a nominally weaker client in 2.8.1. By how much does this 1 second margin increase, or does it remain stagnant?

- What is the impact of executing floating point in software on a PDA? Since PDAs normally do not have floating point support in hardware, by how far does the difference in rendering time increase as a result.

As seen in figure 2.10, the difference in rendering time, increases by a large margin. Our analysis being that since PDAs lack floating point support in hardware and the Geometry stages of the graphics pipeline are floating point intensive, rendering them on a server using remote execution improves performance largely.

2.9 Power Profiling of RMesa

RMesa provides the facility to map different stages of the graphics pipeline on either the client or the server machine for execution. In this section, we discuss the power consumed by an open source VRML browser, OpenVRML [35]. OpenVRML is run with the modified OpenGL library, RMesa. The power measurement is carried out using PowerSpy[3].

Stage Mapped to Server	Power Consumed-(mWH)
Geometry Stage	8
Lighting Stage	10
Rasterization Stage	5
Texture Stage	3
Application Stage	5
Client Only	3
Server Only	5

Table 2.3: Power Profile for a 1000-Vertex VRML File

2.9.1 Experimental Setup

The client machine used for this purpose is a Dell Inspiron laptop with 2.4 GHz CPU and 1GB memory. The server machine used is a desktop with a processor of 1.2 GHz and 128 MB of memory. The server machine has a lower processing power than the client machine but since we are measuring the power consumed by the laptop, the decline of speed caused on the client on account of a slower server does not affect the power consumed.

Different VRML files of varying number of vertices is being used for the power profiling purpose. Each VRML file is tested under the following test conditions: (1) *Geometry Stage on server* (2) *Lighting Stage on server* (3) *Rasterization Stage on server* (4) *Texture Stage on server* (5) *Application Stage on server* (6) *Client Only*

The total power consumed by the application for each of the above stage mappings is tabulated. Rendering with the *Application Stage on the server* implies that the VRML file is present on the server side and that, only processed vertices are transferred back to the client.[4]

2.9.2 Power Profiling Results

VRML File - 1K vertices: A VRML file with 1K vertices was used for power profiling RMesa. The power consumed by using different stage maps is as shown in table 2.3.

The power consumed by performing all rendering on a local machine is the least

[4]processed vertices being those that have been operated upon by stages of the graphics pipeline.

Stage Mapped to Server	**Power Consumed-(mWH)**
Geometry Stage	47
Lighting Stage	45
Rasterization Stage	49
Texture Stage	46
Application Stage	47
Client Only	45
Server Only	21

Table 2.4: Power Profile- 10K Vertices VRML File

Stage Mapped to Server	**Power Consumed-(mWH)**
Geometry Stage	351
Lighting Stage	348
Rasterization Stage	356
Texture Stage	354
Application Stage	319
Client Only	314
Server Only	245

Table 2.5: Power Profile- 100K Vertices VRML File

as tabulated in the table 2.3. The power consumed by rendering entirely on the server and returning completely processed vertices back to the client is also high suggesting that the power consumed in networking proves expensive when the number of vertices is on the low. *VRML File - 10K Vertices:* A VRML file with 10K vertices was also profiled for power consumption and the results are contained in 2.4.

The power consumed in processing the entire graphics file at the server side is much lower than the other stage maps. Thus with 10k vertices, there is a substantial power saving in rendering all stages on the server.

VRML File- 100K Vertices: A VRML file with 100K vertices profiled for power consumption revealed the results tabulated in table 2.5.

The power consumption for a *server only* rendering process is the least among all the other stage maps. This power saving increases as the number of vertices in a graphics file increases.

2.10 Related Work

We will classify the related work into the following, graphics architectures, remote execution and parallelization of OpenGL.

Graphics Architecture: Popular graphics architectures that follow the principle of breaking the graphics pipeline operations into network independent execution models include *WireGL*[10] and *Chromium* [9]. Both these architectures are targeted at high end clusters and do not specifically target mobile environments. Their configurations are also not dynamically reconfigurable at runtime. WireGL, though not adaptive neither geared for small sized mobile clients, does provide the framework for splitting the graphics pipeline. The granularity of pipeline splitting in WireGL is a single stage of the graphics pipeline. Chromium, is built on WireGL and improves upon it by providing support for stream operations.

AnyGL, takes off from where WireGL leaves and provides a finer still granularity in breaking the graphics pipeline. Thus AnyGL provides support for splitting the pipeline into geometry, rasterization and per-fragment operations with each of these having location independence in execution. The idea inherent from the above works is that finer levels of granularity provides greater flexibility and a better scope to schedule execution on client or server. In RMesa, we obtain a finer still granularity than that of WireGL or AnyGL yet providing a framework for lightweight mobile clients to operate upon.

Remote Execution: Project Aura [8] developed at Carnegie Mellon provides a flexible framework for remote execution of components in an adaptive manner. Though, not graphics centric, the ideas of resource monitoring and adaptive offloading of work to the server are similar to the framework being built as part of RMesa.

Changing the OpenGL implementation: The Parallel Mesa[7] and OpenGL Design[29] works are instructive of ways of parallelizing OpenGL and tailoring a given OpenGL implementation to ones' specialized needs. Parallel Mesa is implemented on top of Mesa 3D library similar to the way we are building our model on top of Mesa 3D though Parallel Mesa concentrates on providing parallelism to the existing graphics

pipeline in a multi-server architecture.[5]

2.11 Summary

In this paper, we have proposed a novel pipeline-splitting concept that facilitates remote execution of applications which use OpenGL. RMesa provides the capability to flexibly render 15 graphics pipeline sub-stages on different machines. However, different stage mappings provide varying performances in terms of speed, interactivity, power consumption. Although we include specific results of remote execution for a specific client, server, optimal mappings will vary for different machines and the true power of RMesa lies in the flexibility it provides and the potential it has for fitting into applications using OpenGL for rendering.

[5]Having one operation performed by multiple servers at the same time.

Chapter 3

Modeling the RMesa Rendering Process

The Graphics Rendering process uses the pipeline architecture comprising of individual stages. Consider for example, that the pipeline consists of only 2 stages, namely, A and B. Each of these stages can be rendered on the client machine or on a powerful server machine. Thus the pipeline can be executed with 4 permutations of stage mapping, i.e., 2^2. The problem that we are trying to solve in this chapter is, *given a pipeline with 'n' stages, what is the stage map permutation that leads to optimal execution time and memory consumption on the client machine?* Given the nature of graphics applications, we also state that the rendering time of the pipeline is a function of the *network roundtrip overhead*, *processing power of the client and server machines* and *the number of input vertices*.

We have identified the following two approaches to solving the above problem. Considering the above example of a pipeline with two stages, the 4 permutations for stage mappings can be executed on a pair of client server machines using a given network. The execution time for varying number of input vertices can be recorded. This is the learning phase which is done offline (meaning not during the rendering process). Later, when a model is to be rendered, using a look up method, the stage map leading to the optimal execution time can be identified. A similar approach can be used for memory consumption too.

In the second approach, we can model the execution time of the pipeline as a function of the input number of vertices, the processing power of the client and server machines and the network roundtrip overhead. In this approach, there is no offline learning phase. During the rendering process, the mathematical function can be solved to yield the optimal stage map.

The lookup based method has the drawback that for a pipeline with 15 stages, the total number of permutations to be tested increases to a massive 2^{15}. Besides, during the learning phase, tests including these permutations will have to be carried out for each client, server pair. The modeling approach does not suffer from the same drawback. In this chapter, we present a mathematical model used to predict the performance of the Remote Execution rendering process.

3.1 Modeling RMesa

The most optimal mappings of RMesa sub-stages to either a mobile client or a surrogate server, depend on several factors including the degree of asymmetry in processing power between the client and server, the mobile client's memory size, battery power constraints, as well as the speed of the wireless network. In our experiments and performance evaluation studies, we have been encouraged by findings that even with the significant roundtrip network delay overheads incurred, mobile devices with no hardware floating point units or graphics cards showed rendering speed improvements of up to 10 times while using RMesa[1]. Generally, server-side execution of floating-point-intensive stages such as the geometry (per-vertex) stage yielded the most dramatic results. Figure 3.1 is a sample of our results which shows the improvements in the overall rendering time for a PDA when the modelview stage is rendered on a server, over a Wireless LAN (WLAN).

The actual decision on where each of RMesa's stages should be rendered is determined by a centralized server-side MADGRAF module called IntelliGraph. This problem is non-trivial given the large number of permutations (2^{15}) of possible client-server mappings of pipeline stages that could be chosen in RMesa. To facilitate this decision, we have built models for (1) *Execution time of the graphics pipeline on client*

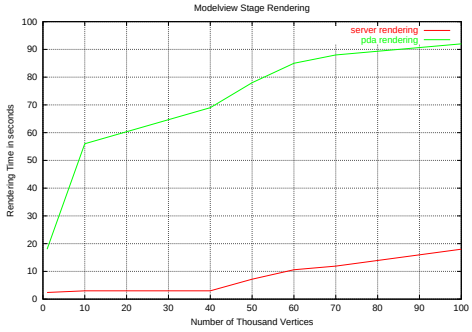


Figure 3.1: PDA rendering time with modelview on server

and server machines (2) *Memory requirements of a given mesh* (3) *Network overhead incurred in transmitting data between client and server* which the IntelliGraph can use to determine where each stage of the pipeline could be optimally executed. Our models for overall rendering speed and resource consumption are the subject of this paper.

Our work extends earlier pipeline modeling work [13] to encompass remote execution in RMesa, the different rendering capabilities of the the client and server, consider memory requirements of mobile clients and network overhead incurred during remote execution. The rest of this paper is as follows. Section 3.2 discusses our model for predicting the rendering time of RMesa, whereas section 3.3 discusses our model to predict the memory consumption by individual stages. Finally, section 3.4 analyzes the roundtrip network delay incurred during remote execution.

3.2 Modeling Execution Time

In this section, we present a model to predict the execution time of our modified RMesa graphics pipeline on a given mobile-client and server. We use a hybrid approach in which we directly instrument RMesa to measure the per-vertex rendering time of each stage of the graphics pipeline on both the mobile client and server and use these measured times as inputs to our analytic models for overall rendering time.

Section 3.2.1 discusses the equation used to predict the execution time whereas 3.2.2 discusses how the individual parameters in the equation are measured or derived at.

3.2.1 Measured Parameters

Our model for the total rendering time of a given client-server mapping of RMesa builds on the equation 3.1, proposed in [13] and models the rendering time, T of a single machine software implementation of OpenGL as

$$\begin{aligned} T =& T_{Fixed} + \\ & (V + V_C)[T_{MP} + (1 - p_{clipped})(L.p_{shaded}.T_L + T_{VL})] \\ & + (F + F_C)T_{RF} + S.T_{RS} + P.T_{RP} \end{aligned} \tag{3.1}$$

where T_{fixed} is the time required to execute parameter-independent code such as clearing color and initialization code. T_{MP} is the time needed for the modelview transformation, the projection, perspective division and clip tests for 1 vertex. V is the number of vertices in the original mesh model and V_C are additional vertices introduced as a result of clipping. Likewise, F is the original number of triangles and F_C are triangles introduced by clipping. T_L and T_{VL} are the times taken to shade one vertex and the time required to initialize shading plus the time required for viewport mapping of 1 vertex respectively. $p_{clipped}$ is the probability that a given vertex falls outside the view frustrum and hence clipped and p_{shaded} controls if light sources are close enough to the vertex to be shaded. T_{RF}, T_{RS} and T_{RP} are the rasterization times per triangle (F), line span (S) and pixel (P).

The parameters from equation 3.1 can be classified into the following 3 categories, namely, (1) *Input Parameters*: V, F, P, S (2) *Scene-dependent Parameters*: V_C, F_C, $p_{clipped}$ and p_{shaded} (3) *Per-Machine Parameters*: T_{Fixed}, T_{MP}, T_L, T_{VL}, T_{RF}, T_{RS}, T_{RP}, L. The Per-Machine parameters remain constant for a given machine and for any graphics scene, and are measured separately for the client and server in our remote execution system. The Input Parameters are scene dependent as they include the number of vertices fed into the graphics pipeline and the number of faces fed to the rasterization stage of the graphics pipeline respectively. The

scene-dependent parameters are the most difficult to measure and are mostly scene and view dependent. Ideally for remote execution of a given stage to be profitable, $T_{stage_{server}} >> T_{stage_{client}} + T_{network}$, where T_stage could be any of the per-machine parameters and $T_{network}$ is the roundtrip time for network transfers.

3.2.2 Measurement Policy

We shall now briefly describe our measurement policy for the *Input, Scene-Dependent and Per-Machine* parameters.

Input Parameters: The Input Parameters are V, F, P and S. When the graphics pipeline starts execution, only V is known, with which we can solve the geometry part of equation 3.1 and decide whether it is best to execute the geometry stage on the client or server. F, P and S are dependent on V as well as viewing and mesh characteristics. Hence, we measure them by placing a hook between the geometry and rasterization stages and deferring the decision on where to execute the rasterization till F, P and S are known.

Scene-Dependent Parameters: The *actual number of vertices, triangles* that are generated as a result of the clipping operation is detected by placing a hook at the end of the clipping stage. These parameters are also measured during each run of the graphics pipeline. [13] observed an average case of 0.5 for $p_{clipped}$. However, to yield a conservative estimate, we set $p_{clipped}$ to 0 and p_{shaded} to 1.

Per-Machine Parameters: Each of these parameters roughly correspond to the amount of time taken to carry out a set of floating point operations (or stages) on a given machine, and can be determined by averaging several executions of that stage.

3.3 Memory Consumption

In this section we discuss our algorithm to predict the maximum memory requirements of an individual graphics pipeline stage. Mobile hosts have limited physical memory. Therefore, our work in memory estimation ensures that given a client-server stage

map, that the mobile client has enough memory to execute its assigned stages and hence avoid the failure of memory allocation system calls. We assume for this model, that the server always has the required amount of memory since servers may swap memory, or use virtual memory. In 3.3.1, we discuss the different types of memory used by an application while in 3.3.2, we discuss the memory estimation policy for each stage of the graphics pipeline. Further in 3.3.3 we compare the results from 3.3.2 with those obtained from actual measurements.

3.3.1 RMesa Memory Usage

The Mesa source code largely used the heap during, dynamic memory allocation to store vertex and rendering context information. Stack memory usage is highly optimized. We therefore focus our attention on measuring dynamic memory allocation per stage of the graphics pipeline.

3.3.2 Analytical Model

Our memory requirement analysis of the graphics pipeline is broken into: (1) *Per-Vertex operations*: including the geometry, lighting and normal substages. (2) *Per-Fragment operations*: including accumulation, alpha Test, depth test, stencil Test and color substages. By monitoring dynamic memory allocation calls, we analyze the memory requirements for each of these stages.

Per-Vertex

The per-vertex stage operates on individual input vertices and contains the geometry, lighting and normal substages.

Geometry substage: includes modelview transformations, projections and clipping operations that are carried out on vertices. The memory consumption of the geometry substage can be expressed as:

$$Memory_{Geom} = M_{GC} + I_{pv} * V + I_{pcv} * V_C \tag{3.2}$$

where M_{GC} is the size of data structures required to render context data which is unique to the geometry stage. This data is independent of the number of input vertices and remains constant. Extra memory is also required for alignment at it at specific boundary sizes. V is the number of input vertices, V_C is the number of vertices generated by the clipping stage and I_{pv} is extra information stored per vertex such as color, normal, fog, and texture coordinates. I_{pcv} is additional information stored for vertices generated by clipping.

Lighting and Normal substages: includes the per-vertex operations carried out for normal data and for light calculation, and its memory consumption can be expressed as:

$$Memory_{Light} = M_{LC} + V * GLSize \tag{3.3}$$

$$Memory_{Normal} = M_{NC} + V \tag{3.4}$$

where M_{LC} is the size of data structures that are required to hold basic lighting data, and data is independent of the number of input vertices or light sources. M_{NC} is size of constant data structures required for maintaining normal data.

Per-Fragment

The total memory required for the Per-Fragment stage is aggregate of the memory requirements for the accumulation, alpha, depth, stencil and color substages. These stages depend on the viewport (or screen) dimensions of width W and height H in pixels.

Constant	**Value (Bytes)**
M_{GC}	23132
I_{pv}	52
I_{pcv}, $maxDepth$, $maxColor$	4
M_{LC}	352
$GLSize$	16
M_{NC}	32

Table 3.1: Constant Enumeration in Memory Profiling

$$Memory_{accum} = W * H * sizeof(GLaccum) \tag{3.5}$$

$$Memory_{alpha} = W * H * sizeof(GLchan) \tag{3.6}$$

$$Memory_{stencil} = W * H * sizeof(GLstencil) \tag{3.7}$$

$$Memory_{depth} = W * H * maxDepth \tag{3.8}$$

$$Memory_{color} = W * H * maxColor \tag{3.9}$$

where *maxDepth*= either 2 bytes of 4 bytes depending upon the platform and maxColor=1,2 or 4 depending upon whether stereo and back buffers are used. A summary of the constants used in the above equations and their corresponding values on an x86 32 bit architecture machine are shown in table 3.1.

3.3.3 Model Validation by Actual Memory Measurement

In order to validate our model developed in 3.3.2, we compare the predicted results with the actual results obtained by measurement at runtime. In this section, we present a validation of the memory requirements as predicted for the *per-fragment* stage. First we discuss our approach for measurement of memory consumption at runtime and later discuss how these values compare with those predicted by the mentioned equations.

Memory measurement methodology: In order to measure the memory requirements for the per-fragment stage, we choose a graphics application linked with

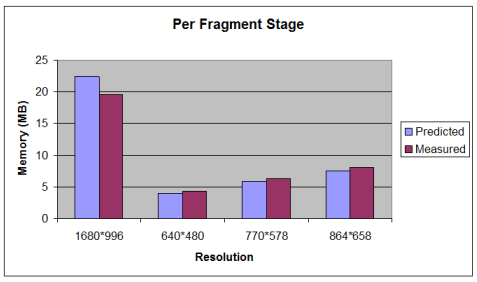


Figure 3.2: Predicted Vs. Measured Memory Allocations

OpenGL and using fixed number of vertices. We then display this application at varying window resolutions of *1680 * 996*, *640 * 480*, *770*578* and *864*658* . Each time we change the window resolution, the graphics pipeline needs to allocate and reallocate memory for the 'per-fragment' stage. The memory for the geometry stage remains constant since changes in screen resolution do not affect the geometry stage. For each resolution, we use Win32 Performance Counters [41] to measure the amount of total memory consumed by the application. We then attribute the difference in memory usage for any two resolutions to the difference in screen resolution and solve for the memory usage per pixel.

Comparison of Measured and Predicted Values: Figure 3.2 shows the difference in the predicted and measured values for the per-fragment stage. There was a slight difference between the measured and predicted values, which we attribute to the fact that the operating system typically allocates memory in blocks such that odd-sized requests typically get a full page of memory, F that is highly dependent on the operating system and processor architecture.

3.4 Network Overhead

In this section, we estimate the network overhead involved in sending data from the mobile host to the server. Figure 3.3 shows the measured network overhead incurred for meshes of different sizes over an 802.11b wireless network operating at 11Mbps. Apart from a few spurious noisy spikes, the graphs are mostly linear, with a constant overhead (y-axis intercept). Thus, we can model the network time using the straight

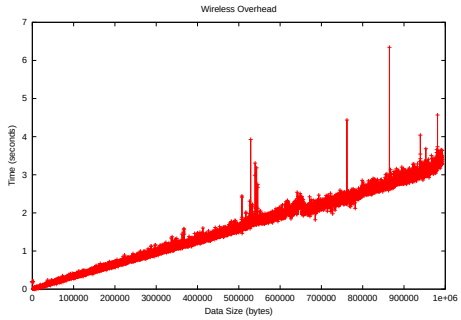


Figure 3.3: 802.11b Wireless LAN Round Trip Time

line equation of the form $y = mx+c$, where y is the total time taken for the roundtrip journey, x is the size of the packet in bytes and c is a constant overhead involved for every network transfer. This constant overhead is machine and operating system dependent and includes the time required for servicing network card interrupts, as well as copying and waiting time at the network card.

3.5 Summary

In this chapter, we have seen a mathematical modeling approach to answering the following question: *given a pipeline with 'n' stages, what is the stage map permutation that leads to optimal execution time and memory consumption on the client machine?* Given that the rendering times of the pipeline is a function of the *network roundtrip overhead*, *processing power of the client and server machines* and *the number of input vertices*. The rendering time of the pipeline on a machine can be modeled using Input Parameters such as the number of vertices, fragments, the Scene Dependent Parameters such as the number of clipped vertices, culled fragments and the Per Machine Parameters such as the geometry and rasterization rendering time for one vertex.

The memory consumption of the pipeline has also been summarized by tracking the dynamic memory allocation calls made by the pipeline and has further been verified by comparing the virtual memory sizes of different applications using the

graphics pipeline for different number of vertices.

Finally, the rendering time has been updated to reflect the time taken for data transfer between the client server. This includes the time taken for the client to ship the data to the server and for the server to ship the processed data back to the client.

Chapter 4

PowerSpy

4.1 PowerSpy

Battery power capacity has shown very little growth, especially when compared with the exponential growths of CPU power, memory and disk space. Hence, battery power is frequently the most constraining resource on a mobile device. As a foundation for optimizing application energy usage on mobile devices, it is increasingly important to profile system-wide energy usage in order to accurately determine *where the energy is going?*. Previous work on profiling energy usage has either required external hardware multimeters, provided coarse grain results or required modifications to the operating system or/and profiled application. Battery power drain is important in the context of Remote Execution since it is important to know if the communication involved in Remote Executing stages is consuming higher battery power than a local execution process. We present PowerSpy which separates the battery power consumption by computation from that by I/O devices thereby helping us to better analyze Remote Execution.

The goal of our research is to monitor accurately thread level and I/O device level power consumption. When an application is monitored, its energy consumption is noted at finite intervals of time. However, this measured energy is not just consumed by the threads of our monitored application but also includes the energy consumed by the hard disk, network card display, other I/O devices, and potentially the threads of

other applications in a multithreaded environment. Our profiling approach involves two passes. First, battery power usage is accurately sampled at fine intervals while also tracking all system activity. We then filter (subtract out) the energy consumed by other applications, I/O devices and other *noise*, leaving us with an accurate estimate of energy usage by our monitored application.

Using PowerSpy, using only their executables, we profile five diverse commercial-strength applications including a web browser, VRML graphics browser, compiler and video player. Our results show that the network interface consumed the most power for networked applications, followed by disk I/O operations and then CPU.

4.2 Previous Work

The continuous power sampling method for profiling application energy usage was adopted by PowerScope [18]. An external hardware multimeter with an in-built clock is used to sample the monitored computer. At each sampling time, the CPU status of the monitored computer is taken. This status consists of the current value of the *Program Counter* and the *Process ID- PID* along with interrupt handling details. At the same time, the multimeter records the instantaneous current, voltage values. Later, during a energy profiling post-process stage, the CPU values are associated with the multimeter readings for each sampled interval in order to reconstruct the power consumption details of the monitored computer.

Another approach to measuring I/O power consumption is to pre-determine the power consumed by each I/O device when it is in a given fixed power state[19]. Thus, if the power consumption of the hard disk is known in all its states, and the power consumption in transitions between states is pre-determined, given the state of the I/O device, the I/O power consumption can be derived as in [19].

Energy Estimation is also performed by formulating fine level building blocks of which an application is composed [20]. One such way is to measure the power consumed by different procedures within the Operating System. Once we know which of these components a given application uses, we can then estimate its power consumption.

4.3 PowerSpy

In this section, we present the two-pass profiling approach used our energy profiling tool, PowerSpy. The functionality is presented in two stages, namely, *(1) Event Tracking* and *(2) Analysis*. Further, the Analysis stage has 2 separate passes which we shall expound on.

4.3.1 Event Tracking

In this stage, the application to be profiled for energy consumption is run. Simultaneously we profile this application for CPU Time, I/O Activity and Energy Consumption.

CPU Time: From the time that the application is started to the time that it ends, the thread IDs of all threads created by the application are kept in an 'in-core' database maintained by PowerSpy. The CPU Profiler records in the *cpu.log* file, the thread id of the thread being run at the end of each context switch along with the time stamp.

I/O Activity: Simultaneously, all I/O requests made to the attached devices are also recorded along with their time stamps and the request specifics in an *io.log* file. An I/O request in this context is an asynchronous call made to an I/O device. We assume that the device starts servicing the request from the time that it receives this I/O call.

Energy Consumption: The energy consumed by the system is also profiled in a separate *energy.log* file. This file contains a chronological listing of *time vs energy consumed*.

Thus, at the end of the Event Tracking stage, we have tracked and stored in three files, all CPU events, I/O events and system energy usage over time while the application was running.

Power					
Requirement	+5VDC (±5%)	+5VDC (±5%)	+5VDC (±5%)	+5VDC (±5%)	+5VDC (±5%)
Dissipation (typical)					
Startup (max. peak)	4.7 W	4.7 W	4.7 W	4.7W	4.7W
Seek (average)	2.3 W	2.3 W	2.3 W	2.3W	2.3W
Read (average)	2.1 W	2.1 W	2.0 W	2.0W	2.0W
Write (average)	2.2 W	2.2 W	2.1 W	2.1W	2.1W
Performance idle (average)	1.85 W	1.85 W	1.85 W	1.85W	1.85W
Active idle (average)	0.95 W	0.95 W	0.85W	0.85W	0.85W
Low power idle (average)	0.65 W	0.65 W	0.65W	0.65W	0.65W
Standby (average)	0.25 W	0.25 W	0.25 W	0.25W	0.25W
Sleep	0.1 W	0.1 W	0.1 W	0.1W	0.1W
Power consumption efficiency (watts/GB)	0.008	0.011	0.016	0.022	0.033

Figure 4.1: Sample Hardware Spec Sheet

4.3.2 Analysis Stage

In this stage, we process the data acquired at the end of the *event tracking stage* in order to recreate a snapshot of the system during the time that the application being monitored was being executed, so that we can understand the power consumption by the different parts of the system. The Analysis stage takes as input the cpu.log, io.log, energy.log event tracking files as well as a fourth file called spec.log. The spec.log as shown in figure 4.3.2, file contains the estimated energy required by various devices to perform specific tasks, as deduced from the specification sheets provided by the device's manufacturer. A typical spec.log entry is the energy consumed by a disk to read 1K bytes of data. The Analysis Stage is carried out in two separate passes: the I/O filtering process and the CPU thread-level accounting.

Pass One - I/O Filtering Process

This pass takes as input the io.log, energy.log and spec.log and modifies the energy.log file. In this pass, we filter out (subtract) the estimated energy consumed by each I/O device that ran in a given time interval, from the total measured energy consumed in that interval. The remaining energy is attributed to CPU threads.

Let us consider for example, that the energy consumed in a given time interval is *'A'* units. And that in this interval threads *a,b,c* were run (note that we carefully choose time interval *'A'* as the smallest time interval for which we notice a change in remaining battery energy). Additionally, I/O devices *e,f,g* were operating in this same interval. However, from spec.log, we know that devices e,f and g when performing

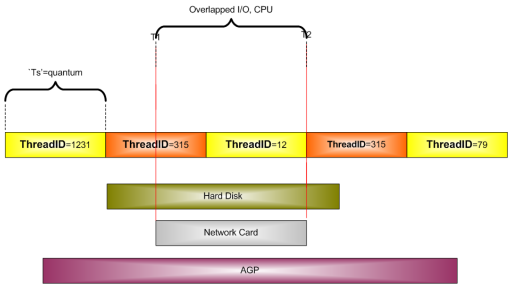


Figure 4.2: Overlapped CPU and I/O Operations

certain tasks consume finite amount of energy, say P_e, P_f, P_g respectively. We can deduce the power consumed by threads a, b and c for their cpu-centric operations. Mathematically, if P_a is the amount of energy consumed by thread *'a'* and (likewise for b,c,d,e,f,g), then

$$P_a + P_b + P_c = A - P_e - P_f - P_g = P_{threads} \tag{4.1}$$

As a concrete example of P_e, consider that the I/O request is a read request to the hard disk for a block of size 2KB. The specification sheets tell us that the energy estimate for a 1KB read operation is *'k'* units. Therefore $P_e = 2 * k$.

Figure 4.2 shows an example of overlapped CPU and I/O operations. Thus, at the end of this pass, we have been able to separate the I/O energy consumption from the CPU energy consumption and also get a list of the energy consumed by individual I/O devices.

Pass Two - CPU Thread Level Accounting

In this pass, we will isolate the energy consumed by different threads individually. This pass takes as input the energy.log and cpu.log files from the energy tracking stage. It parses the energy.log file for two consecutive energy, time intervals. It obtains a list of all thread ids from cpu.log that were run in this time range. For example, if in a time range threads with IDs 1211, 2032 and 5101 were run 5, 6 and 2 times respectively. Also the energy consumption in this time range was 50 units. Therefore, the energy consumed by the thread with id 1211 is

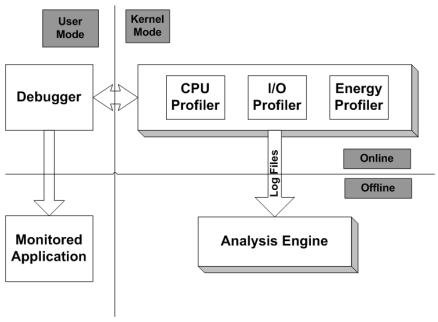


Figure 4.3: PowerSpy System Architecture

$$P_{1211} = 50 * (5/5 + 6 + 2) \tag{4.2}$$

The underlying assumption behind equation 4.2 being that CPU level power consumption is directly proportional to the time (or number of cycles) spent in running the thread.

4.4 PowerSpy System Architecture

In this section we describe our implementation of PowerSpy on the Windows operating system. Windows was chosen due to its overwhelming popularity on multiple mobile and ubiquitous computing devices. PowerSpy is targeted to work for Windows 2000, XP and Windows Server 2003. The components are as shown in figure 4.3.

As shown in the figure, some of the components execute in the user mode whereas the others execute in the kernel mode. Similarly, some components are operated online whereas the Analysis Engine currently operates offline but will operate online in future.

4.4.1 Debugger

The Debugger is responsible for initiating the application program that is to be monitored. It is responsible for keeping track of the different threads that the application spawns out. It maintains an in-core database of these threads and communicates

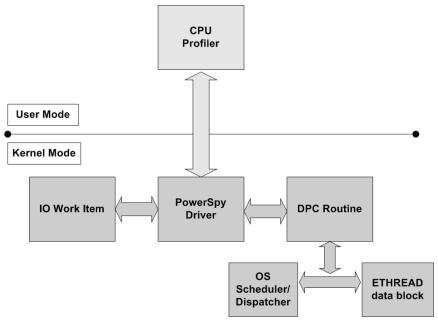


Figure 4.4: CPU Profiler Functional Diagram

them to the CPU Profiler. The use of a debugger process in tracking thread activity is important since under the Win32 platform, a debugger process that initiates another process has complete access over the debuggee's memory address space. Also, each time that the application spawns out a thread, the debugger is notified.

4.4.2 CPU Profiler

The CPU Profiler keeps track of the thread that was being run by the Operating System, each time a context switch occurs. If the thread being run matches any of the threads in the in-core database maintained by the Debugger, then it adds a flag and outputs the same to the "cpu.log" file.

The CPU Profiler communicates with the kernel mode components as shown in figure 4.4. The PowerSpy driver is a Windows kernel mode legacy driver [24]. Its purpose is to install a Deferred Procedure Call (DPC), which is a routine that monitors the Windows Thread Scheduler also called as *The Dispatcher Object* and is part of the Windows Executive [22]. It is important to note that since DPC objects run at the same privilege as the Windows Dispatcher, it is not controlled or scheduled by the Dispatcher and as such is in a position to monitor the activity of the Dispatcher. The DPC routine is executed once every 10 milliseconds which is the normal quantum of all threads in Windows 2000. However, a context switch may occur before this time period because of thread level preemption. The "EThread" data block is an undocumented data structure maintained for each running thread by the Windows Scheduler [22].

Thus, the DPC routine gets a snapshot of the Windows scheduler. It also queues an I/O work item, which executes at a lower privilege than the DPC (since it is bad practice for long operations to be performed at exalted privileges). The I/O Work Item stores the system timestamp, and the thread ID which is the thread ID monitored by the DPC at the instance that the DPC was executed. In other words, the DPC captures the information but does not write it to the "cpu.log" file since file operations may take a long time. Rather it passes this operation to the I/O Work Item which writes to the "cpu.log" file. Figure 4.5 shows the sampling process. $T_1 to T_6$ are the sampled time intervals where the current thread being run is stored. Also the remnant battery power is stored.

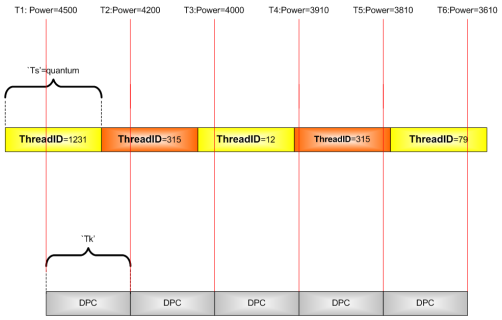


Figure 4.5: CPU Profiler Timeline

4.4.3 Energy Profiler

The energy profiler captures the amount of energy lost by the system during the time that the application is being monitored, and can be implemented either in hardware or software. As part of PowerSpy package, we provide a software solution. However, hardware measurements can easily be used instead with our existing framework.

In the hardware solutions, a digital multimeter is used to measure the instantaneous power drawn by the entire system at a high sampling frequency. PowerScope uses a hardware solution. These power values are then recorded along with their time stamps. In the software solution, we use system calls to measure the amount of battery energy remaining at a sampling frequency and record the same along with

the time stamps. We now briefly discuss the pros and cons of hardware and software energy profiling.

The advantage of the hardware method is that since the energy sampling is done independent of the system being monitored, the sampling frequency can be very high and is independent of any OS activity. Also, the instantaneous power values are more accurate than their software counterparts. The software version is limited by the sampling rate at which the operating system gets updates from the actual battery, and suffers from the drawback that at very high sampling rates, the battery device may not be able to report changes in the remnant power accurately. In other words, if we sample the battery at 10 milliseconds, then considering 10 such sampled values, it may happen that the remnant battery is the same across all 10 values. However, in reality, it may be the case that the remaining battery energy has changed but is not being updated at that frequency.

For our purposes, since we wanted to develop a software module that could be conveniently integrated into our MADGRAF system [2], in which clients are mobile. In such a scenario, the deciding factor was the disadvantage that the hardware method involves external equipment and is not easily portable.

PowerSpy uses the software solution for querying battery levels. The I/O Work Item regularly queries the battery device using IOCTL_QUERY_BATTERY_STATUS ioctl. Thus, the I/O Work Item generates an IRP with the above ioctl and issues it to the battery class device driver. The battery device driver returns the remaining battery power in milli Watt Hour units.

4.4.4 I/O Profiler

The I/O profiler keeps track of all I/O requests send to the various devices connected to the system. Under the Windows Operating System, devices are sent asynchronous messages called I/O Request Packets (IRPs) [24]. So, we need a way of capturing all IRPs sent forth to the different devices. In order to do this technically there are two options. One is to write a upper filter driver [24] for all devices. However, writing a filter driver for all devices connected to the system is not a very scalable or feasible

solution. The other solution is to read undocumented structures inside the OS kernel such as reading the ETHREAD data structure in the "ntoskrnl.exe" file.(contains the xp kernel)

We have made use of a third party tool, *IRP Tracker Utility* [30] which writes all IRPs issued by the system along with their time stamps and status in "io.log" file. The time stamp used by IRP Tracker is in the hour:minute:seconds:milliseconds format. However, the time stamp used by the CPU Profiler and the Energy Profiler is in the form of a 64 bit value representing the number of 100-nanosecond intervals since January 1, 1601 . In order to convert the IRP Tracker's time format to that of the PowerSpy profilers' format, we used the Win32 system call SystemTimeToFileTime.

4.5 Results

In this section we present the results of experiments using PowerSpy to profile the power usage of a wide range of applications including the Outlook Express Windows mail reader, Mozilla web browser, Visual Studio IDE compiler, VRML browser and Microsoft Media player. All profiled applications were available as unmodified executables.

The laptop used for the test is a Dell Inspiron 8500 with Hitachi disk and WaveLan Wireless Card. The only devices that we profile are the disk and the wireless network card . However, the principle of I/O profiling and filtering using specification sheets can also be extended to other devices.

Also, the disk and the wireless cards have been brought into a known power state, i.e., the D0 state This was done by doing a dummy read of both the devices from user mode ensuring that the devices are in a maximum power consuming state.

4.5.1 Outlook Express

PowerSpy was used to profile the execution of the Outlook Express mail reader and the results are as shown in figure 4.6. For the above experiment, Outlook Express was started and mails were checked with the *Send and Receive* facility for a single

Thread ID	Energy Consumed (mWh)
1329	15
133	12
291	1
9031	1
1100	1
3002	1

Device	Energy Consumed(mWh)
\Device \tcp	51
\Device\DR0	22

Figure 4.6: Outlook Express Energy Profile

POP3 mail account. The individual thread level power consumption *does not include the power consumed by the Operating System internal operations.*

Referring to figure 4.6, the power consumed by the I/O device '\Device\tcp' is higher than that consumed by any of the individual threads (power consumed by the threads as shown in figure 4.6 is purely due to CPU operations). The device '\Device\DR0' represents I/O access to the disk, the power consumed by which is also substantial. Thus, a major chunk of the power consumed by Outlook Express is due to network and disk access.

4.5.2 Microsoft Visual Studio

The power profiling performed on 'Microsoft Visual Studio' revealed its power consumption details as shown in figure 4.7. At the time of profiling, a sample VC++ project was opened, rebuilt and closed. As can be seen, the power consumed by the hard disk namely, \Device\DR0 is somewhat high indicating that the application tends to be more aggressive in its use of the disk, probably due to extensive the file read/write, swapping and disk writes that are typical of the compilation of projects with a large number of files.

Thread ID	Energy Consumed (mWh)
1122	9
3651	6
782	11
297	31
202	12
203	5

Device	Energy Consumed (mWH)
\Device\DR0	55
\Device \tcp	0

Figure 4.7: Microsoft Visual Studio results

4.5.3 Mozilla

Mozilla web browser was tested for its power consumption and the results are described in figure 4.8. A sample site at opengl.org was opened and refreshed and the power profile results were tabulated. As is evident from the fig 4.8, the power consumed by the tcp device is somewhat high. The disk I/O power consumption is comparatively low.

4.5.4 Media Player

The results of Windows Media Player profiled for power is given in figure 4.9. The experiment carried out with Windows Player was playing an online movie trailer. The URL of the site with the trailer was entered into Media Player and the movie started. It is very apparent, that the bulk of operation is carried out by the network device. The power consumption details reveal the same that is, \Device\tcp has a higher share in the power consumption list.

4.5.5 OpenVRML Browser

An open source VRML browser was used to test the power consumed by VRML files of various sizes. Figure 4.10 provides details about the power consumption by the

Thread ID	**Energy Consumed (mWh)**
123	121
124	32
125	42
313	51
417	1
1299	1

Device	**Energy Consumed (mWh)**
\Device\DR0	152
\Device\tcp	430

Figure 4.8: Mozilla web browser energy profile

Thread ID	**Energy Consumed (mWh)**
2379	402
3412	98
786	22
352	1
1102	1
1471	1
1472	1
1481	2
1296	2
1926	1
1106	2
1291	1

Device	**Energy Consumed (mWh)**
\Device\DR0	165
\Device\tcp	850

Figure 4.9: Windows Media Player results

Thread ID	Energy Consumed (mWh)
1321	52

Device	Power Consumed (mWh)
\Device\DR0	42

Figure 4.10: VRML Browser (10,000 vertices)

browser when opening a VRML file with 10k vertices. The complexity of rendering a given VRML scene file was roughly proportional to the number of vertices in the scene description. The power consumed by the individual thread is relatively high. The initial loading and parsing of the VRML file is CPU intensive and that explains the high power consumption of a CPU work oriented thread. We also profiled 1000-vertex and 100K-vertex VRML files and found that in all cases, the amount of power consumed by the individual CPU work oriented thread is far higher than that consumed by any of the I/O devices. As the size of the VRML file gets larger, the amount of power consumed by the CPU thread in parsing and loading the file increases.

4.6 Summary

We have presented PowerSpy, a software tool for fine-grained power profiling on the Windows operating system. We present results of the power consumption of five diverse applications.

Our profiled applications consumed the most power on networked data transmission, where applicable. The Outlook Express mail reader, Media player and Mozilla web browser were all in this category. Applications such as Visual Studio that did not have significant networking activity expended power on disk I/O. Finally, the VRML browser performed many CPU operations to render a single graphics scene, which was reflected in its energy usage.

PowerSpy also helped us understand the inherent limitations involved in Remote Execution since remote executing a stage involves network data transmission. By and large, this power consumption was higher than that by the local execution of the stages.

Chapter 5

Intelligraph

MADGRAF is a Distributed Graphics Environment in which a powerful surrogate server aids the rendering process in a weak mobile client. It does so by using either Remote Execution or by using the Transcoder Based Approach.

In Remote Execution, individual stages of the graphics pipeline are executed between the client and server in an interleaved manner thereby reducing the client side load. The server being able to execute the stages faster than the client, results in a performance gain at the client side despite the network overhead incurred in data transmission to and from between the client and the server. The disadvantage being this performance gain depends upon individual stages and the size of the graphics file being rendered. For example, small sized files do not gain much using this approach as the network overhead dominates the time gained by a faster server side execution. Also stages like Modelview and DepthTest tend to gain higher than the others. The advantage being that the final image displayed is at the same object space resolution as the original. The Remote Execution component inside MADGRAF is *RMesa*.

As part of the Transcoder Based Approach, the initial file to be rendered is reduced in quality by using techniques such as simplification. Thus, an initial file containing 1 million vertices could be simplified to one containing a thousand vertices. This simplification occurs at the server side. Since, the rendering time of graphics images is directly proportional to the number of vertices contained therein, this reduction in size is almost a guaranteed execution speedup process. However, this increased speed

comes for a price, namely, poorer image quality.

Intelligraph allows the MADGRAF system to choose between the two dynamically. As an example, if the system starts up with using Remote Execution, after some time it could decide that despite best efforts, the execution time is increasing. In which case, it could decide to simplify the image by δ and then continue with Remote Execution. This decision making process is a two stage process. When MADGRAF starts, Intelligraph answers the following question:

What is the optimal Stage Mapping to be used for Remote Execution?

Later when MADGRAF is up and running, Intelligraph answers the following question:

Based upon the performance of the system, is it necessary to introduce the Transcoder Based Approach for a while before switching back to Remote Execution?

The remainder of the chapter is organized as such. In section 5.1 we discuss the performance metrics that are considered by Intelligraph in the decision making process. These metrics are gathered by the Environment Monitor on the MADGRAF client. These metrics are further classified as static metrics, dynamic metrics and transient metrics. Section 5.4 talks about the decision making process. The decision making process uses the metrics from section 5.1 and they are further classified as static decisions and dynamic decisions depending upon the metrics they make use of. Section 5.2 discusses the implementation details along with the design undertaken to implement Intelligraph.

5.1 Metrics

In this section we discuss the various metrics considered by Intelligraph during the decision making process and also elaborate on the importance of such metrics in the overall execution of a graphics system. The metrics discussed in this section are classified as follows:

1. *Static Metrics:* Metrics that do not change in their value during the entire execution period of MADGRAF on a client-server machine pair. E.g., processor speed, total RAM size.

2. *Dynamic Metrics:* Metrics that regularly change in their value and therefore need to be measured periodically. These values can increase or decrease in value. E.g., memory available, cpu load.

3. *Transient Metrics:* Metrics that either monotonically increasing or decreasing, for example, remnant battery power. This value does not fluctuate like the CPU Load (of course unless the system gets plugged in to AC supply).

5.1.1 Static Metrics

The static metrics include *Processor Speed* and *Memory Size* on a given machine. These parameters are currently used to identify a machine. Thus from Intelligraph's perspective, two clients with the same processor speed and memory size are equivalent. These parameters are also used to identify server machines. The role of these parameters is to identify the extent of asymmetry between the client and server processing powers.

5.1.2 Dynamic Metrics

These metrics are measured on the client machine periodically since they may increase or decrease in value over time. They include

1. *CPU Load*: Percentage of time spent by the processor in the user and kernel modes for a given sample. The sample period is defined by the runtime system and is defined as the period after which the System Monitor samples each of the individual dynamic metrics. On the client machine, in addition to MADGRAF, there are other applications being executed. The Operating System also has its own set of services operating in kernel mode. This parameter gives us a good idea of the number of threads that are competing with the MADGRAF application. For higher load values, remote execution is an obvious solution.

2. *Available Memory*: We measure memory availability as the number of memory pages available. 3D graphics models are typically notorious for memory requirement. This parameter indicates the number of free pages in the physical memory

that are yet to be allocated. The list of free pages available is maintained by the Operating System. As this value starts diminishing, the OS typically starts swapping out physical memory pages to secondary storage thereby incurring the disk latency overhead. It is worth noting that by remote executing stages of the graphics pipeline, we are freeing up memory which would otherwise have been used on the client machine. For example, if the Modelview stage consumes 10MB of data. Executing this stage on the server implies that this 10MB would be consumed on the server machine and not on the client machine.

3. *Interactivity*: The graphics pipeline is executed each time a frame is redrawn on the screen typically in reaction to user activity. Interactivity measures the rate at which the entire graphics pipeline is executed. Interactive applications like games have a high Interactivity value since the objects in the screen are constantly moved around forcing redrawing (read as executing the graphics pipeline). At higher values of interactivity, simplifying the graphics image is an obvious solution as simplification is a very good approach to speeding up the rendering process. Since during simplification, the number of vertices in the original file is decreased, this results in lesser processing job consuming lesser time.

5.1.3 Transient Metrics

These metrics are similar to the Dynamic metrics except that the change in them is monotonically increasing/decreasing such as the *Remnant Battery Power*. This is a monotonically decreasing function on a mobile device running unplugged (from AC supply).

5.2 System Architecture

In this section we will briefly go through some of the governing design principles for Intelligraph and how the individual components inside Intelligraph adhere to the same.

```
SomeClass *anObject_=FactoryClass->getSomeClass();
anObject_->invokeMethod();
```

Figure 5.1: On the client machine, SomeClass is instantiated. However the actual object implementation is created on the server machine by the factory method. The client gets a proxy and uses it just like it was the actual object. All operations finally execute on the server machine and the results are sent back to the client.

5.2.1 Design Principles

Intelligraph is based on a client-server architecture. The client side environment monitor tracks and reports the system performance whereas the server takes the bulk of decisions based on the reports. However, we have also thought about having multiple servers for a single client such that if a single client fails, then there are other servers it can connect to. Also, a single server can have multiple clients connecting to it. In short, we have a multi-client, multi-server architecture. 5.2.1 and 5.2.1 discusses this aspect in further details.

Distributed Object Based Middleware

We have used an object based Middleware similar to the CORBA architecture, namely, Ice[16]. The advantage of having an object middleware being that programming distributed applications becomes far simpler. One no longer has to provide tedious socket calls, message sequences and wait methods. The programming model is similar to that on a single machine. There are multiple objects and one can use them without being bothered about the fact that some of these objects are not present on the same machine. In short, the actual location of the implementation of each object is left until runtime and can be controlled by a separate service or a script file. This allows objects to be moved between multiple machines without the application that uses these objects breaking, stopping or even knowing about the same.

Figure 5.1 shows a sample instantiation of a server based object on the client side.

In short, Ice takes care of the bulk of networking in addition to providing features such as persistence, object migration, object replication. Furthermore, the server side implementation of the object is language neutral, meaning that the client side

code can be C++ while the server side implementation of the object can be in PHP, JAVA, C++, VB. Thus, Ice provides a very strong object oriented platform which is OS independent and programming language neutral.

Distributed File System

The client and server machines are required to access some common files, namely, the VRML files that the client displays. We decided to use the features provided by existing Distributed File Systems rather than developing features similar to these ourselves. The current implementation uses the NTFS file systems' features. Thus objects can interact with files without assuming the machine that these files are present on by using a mapped network drive common between all machines implementing the objects. Thus, if the client wishes to display a file that is present on the server machine, in the absence of a DFS, separate code would have been required to fetch the file and display it on the client machine. Also, having a DFS ensures that the client has to fetch only as much of the file as is required for display on screen. This feature can be useful for large sized VRML files where fetching the entire file before displaying it can produce a huge startup latency. Other operations such locking against multiple access, sharing control, client side caching are provided by the DFS. Currently, the MADGRAF system does not work in the absence of a DFS between the client and server machines.

5.2.2 Individual Components

The system architecture of Intelligraph is as shown in figure 5.2.

The Monitor components are responsible for monitoring different metrics at the client side. There are a list of such Monitor components that are maintained by the Observer component. When the monitored value inside a Monitor component exceeds a certain value, it goes into the Critical state. If this Critical state persists for a while then it goes into the Very Critical state. Depending upon the state of each of the Monitored components, the Observer invokes a Decision component. Thus the Observer component maintains a one-to-many association relationship with each

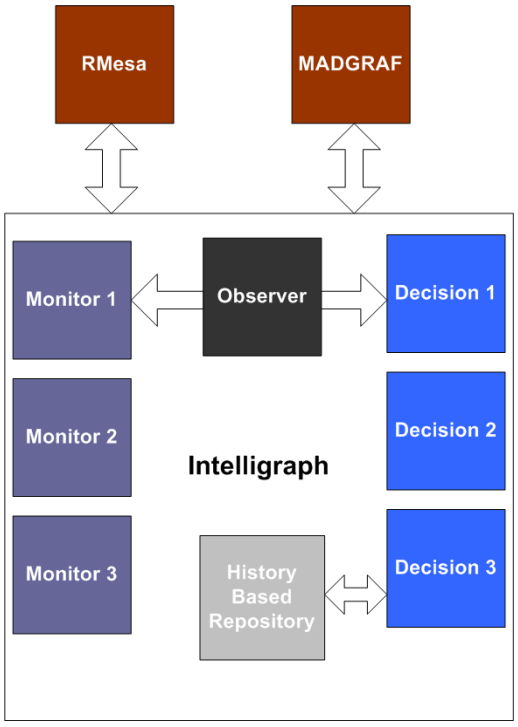


Figure 5.2: Intelligraph System Architecture

of Monitor and Decision components. This architecture allows us to plug-in newer Monitor components and mix-n-match them with either existing Decision components or with newer ones.

Each of the Decision components communicate with the *History Based Repository* that contains the results needed for making History Based decisions as discussed in 5.4.

Each component is comprised of one or more objects (Ice objects). Therefore, they can be implemented on either the client machine or the server machine, the actual decision to which is taken at runtime in a process called as *object deployment*. As part of a typical deployment, the Monitor and Observer objects are placed on the client machine whereas the Decision and History Based Repository objects are placed on the server machine.

5.3 Environment Monitor

The responsibility of the Environment Monitor is to provide the rest of the Intelligraph system with the client machine's runtime performance details. It measures a list of metrics on the client machine periodically and communicates the same to the rest of Intelligraph. It measures the following parameters:

1. *CPU Load*: The Win32 Performance Counter Objects[41] "Privileged Time" and "User Time" under the Processor namespace contain the CPU Load information.

2. *Memory Pages Available*: The Win32 Performance Counter Object "//Memory//Available MBytes" stores this information.

3. *Percentage of Battery Power remaining*: The "GetSystemPowerStatus" Win32 API call provides the percentage of battery power remaining in the system.

The Performance Counter Objects mentioned above can each be queried using the PDH library provided by the Win32 subsystem.

5.4 Decision Process

Intelligraph decides at runtime between Remote Execution and the Transcoder Based Approach. So for example, MADGRAF starts off using Remote Execution but in between, Intelligraph instructs it to use the Transcoder. The Transcoder performs the necessary steps such as modifying the original 3D graphics file to be displayed. Thereafter, the system moves on using Remote Execution. Intelligraph also controls the parameters required by the Transcoder. An example of a Transcoder is *Simplification*. Intelligraph controls the extent of simplification. For Remote Execution, Intelligraph controls stages and their location of execution. For example, it can instruct certain stages of the graphics rendering pipeline to be executed on the server while the other stages execute locally on the client machine. In short, Intelligraph controls the shuffling between Remote Execution and the Transcoder Based Approach and also controls the functioning of these operations.

1. *History Based Approach*: A system is executed previously under varying inputs and internal configurations. For each of the input, configuration permutation, the performance is noted down. The input to the system, the internal configuration and the performance is stored in the database. Later, at runtime, the database is searched for the internal configuration that yielded the optimal performance for a given input. This is similar to the notion of a lookup table. For input values not stored in the database, interpolation techniques can be made use of. For example, consider we are trying to decide at runtime what is the best stage map for Remote Execution from the perspective of power consumption. Previously, we executed each of the stages remotely and locally and noted down the power consumed in the process on a set of machines and stored the results in the database. At runtime, when faced with the decision making problem on a given machine, we simply scan the database for results pertaining to the given machine and choose the stage map that yields the least power consumption.

2. *Analytical Approach*: The system performance is modeled as a function of the input and the internal configuration. Later, at runtime, given the input, the configuration yielding the optimal performance is calculated mathematically.

In the *History Based Approach*, capturing the results for a vast number of input, configuration permutations is a challenge. For the optimal power consumption problem, if we are given 15 stages and the fact that each stage can be executed either locally or remotely, then we arrive at 2^{15} permutations for just a client-server pair. Similar permutations will exist for each client-server pair that needs to run the algorithm. Further, the veracity of the interpolation techniques for permutations not in the database can be a suspect. However, it has the advantage of simplicity especially for problems that are difficult to model analytically. On the other hand, the *Analytical Approach* does not require execution of the system with various permutations of input and internal configurations. However, it is difficult to create a model for all systems. Intelligraph uses a hybrid approach using both of the above approaches at different instances.

In addition to the above classification of decisions, Intelligraph's decisions can be classified as follows based on their time of invocation:

1. *Static Decisions*: Invoked when the MADGRAF system starts and answers the following question: What is the best internal configuration for Remote Execution to begin with? Thus, the static decision making process is executed when the MADGRAF system starts.

2. *Dynamic Decisions*: Invoked periodically and answers the following questions: Should the Transcoder Based Approach be used momentarily before switching back to Remote Execution? The dynamic decision making process is invoked periodically while the MADGRAF system executes.

Irrespective of the above two decisions, Intelligraph requires input data in order to make them. Static Decisions require Static Monitored Data as discussed in 5.1.1 whereas Dynamic Decisions require Dynamic Monitored Data as discussed in 5.1.2.

Keeping the above two classifications in mind, we shall cover the Static Decision making process in 5.4.1 and the Dynamic Decision making process in 5.4.2.

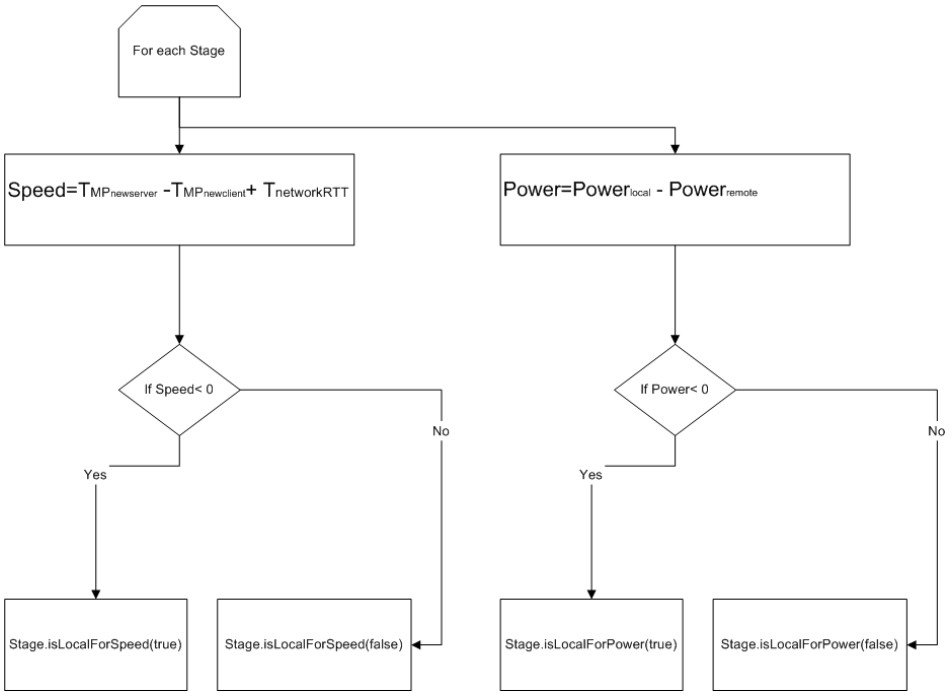


Figure 5.3: Intelligraph Static Decision Making Process

5.4.1 Static Decisions

In this stage, we attempt to answer the following two questions: *What is the optimal Stage Map for Remote Execution using RMesa?* Stage Maps for RMesa can be chosen optimized for Power or for Rendering Speed. We discuss the decisions for each of the optimizations as follows. The diagrammatic representation of the logic is as shown in figure 5.3 whereas figure 5.4 shows a code snippet of the static decision making process

Optimized for Rendering Speed

For each stage, its rendering time on a given machine, (machine characterized as explained in 5.1.1) is priorly calculated and stored in the database. The rendering time for a given stage is the time taken to execute that stage with only 1 input vertex. (as explained in equation 3.1. We extend the work in equation 3.1 to develop a T_{MP} for every stage of the graphics pipeline. The original T_{MP} for instance includes the

```
    vector<Monitor*>::const_iterator ptr_;
    for(ptr_=_monitorList.begin();ptr_!=_monitorList.end();ptr_++)
    {
      if(((*ptr_)->isPreviousTrip())&&((*ptr_)->isTrip()))
      {
iSMTranscoderPrx transcoder_=_factory->createTranscoder();
_numVertices=transcoder_->decision(_numVertices);

_viewer->reduceVertices(_numVertices);
_numVertices=_viewer->getNumberVertices();
return;
      }
    }

  if(_cpuLoad->isTrip()||_memory->isTrip())
  {
    STAGELIST::iterator sPtr_;
    for(sPtr_=_stages.begin();sPtr_!=_stages.end();sPtr_++)
    {
      if((*sPtr_)->isLocalTime())
      {
cout<<"successfully shipped a stage to the server"<<endl;
(*sPtr_)->setLocalTime(false);
break;
      }
    }
  }

  if(_power->isTrip())
  {
    STAGELIST::iterator sPtr_;
    for(sPtr_=_stages.begin();sPtr_!=_stages.end();sPtr_++)
    {
      (*sPtr_)->setLocalTime((*sPtr_)->isLocalPower());
    }
  }
```

Figure 5.4: Decision Making Code Snippet

rendering time of 1 vertex for the modelview, clipping, projection and perspective divide. However, the modified $T_{MP_{new}}$ includes the rendering time of 1 vertex for only one stage. For each stage, decision must be taken whether to execute it locally or remotely. Thus, the database must contain a $T_{MP_{new_{client}}}$ for the client as well as a $T_{MP_{new_{server}}}$ for the server machines.

The stage is chosen for remote execution only if

$$T_{MP_{new_{server}}} < T_{MP_{new_{client}}} + T_{networkRTT} \tag{5.1}$$

where: $T_{networkRTT}$: is the round trip delay incurred in transmitting the vertices and the context information as discussed in 3.4.

The above approach is a combination of the History Based Approach and the Analytical Approach since we have stored the results obtained by modeling the execution time of the individual stages of the graphics pipeline. This process of optimizing for rendering speed is shown in the left part of the flowchart in figure 5.3

Optimized for Power

For each stage 'A', we measure the power consumed by the entire application when rendered locally using PowerSpy[3]. Later, we measure the power consumed by the application when stage 'A' is rendered remotely. The local power is subtracted from the remote power. This result is stored in the database for every client machine (as characterized in 5.1.1). If this result is positive, then remote executing stage 'A' consumes lesser power. Otherwise, stage 'A' is chosen for local execution. This process of optimizing for power consumption is shown in the right part of the flowchart in figure 5.3.

5.4.2 Dynamic Decisions

In this stage, we attempt to answer the following questions: *Given the current performance of the system, do we need to invoke the Transcoder Based Approach? Do we need to reconsider the Stage Map used by RMesa?*

Action	CPU Load	Memory	Power
A	Critical	X	Normal
A	Critical	Critical	Normal
A	X	Critical	Normal
B	X	X	C
C	Very Critical	X	X
C	X	Very Critical	X
C	X	X	Very Critical

Table 5.1: Dynamic Decision Making of Intelligraph. The "X" indicates an OR condition between Critical and Normal. So the third entry means that if Memory is Critical and Power is Normal and if CPULoad is either Critical OR Normal, then action A is initiated.

Action	Description
A	Among the locally rendered stages, pick the one that has the highest $T_{MP_{new_{client}}}$ and render it remotely
B	For all stages, choose the Stage Map that yields the least power consumption
C	Invoke the Transcoder

Table 5.2: Explanation of Actions triggered by Intelligraph's Dynamic Decision Making

The metrics used for making the decisions, namely, the dynamic metrics and the transient metrics are discussed in 5.1.2 and 5.1.3 respectively. Intelligraph is informed by the System Monitor when each of the dynamic metrics reaches an alarming state. Each metric can be in the *Normal*, *Critical*, *Very Critical* state. The "Remnant Battery Power" parameter does not have the Very Critical State. Below we shall see how Intelligraph responds to the different states of the dynamic metrics:

CPU Load

Critical: Searches for the stage that is rendered locally and consumes the maximum time $T_{MP_{new_{client}}}$ and offloads it for remote execution. The reasoning being that the stage that consumes the maximum time rendered locally also contributes to the high CPU Load, therefore rendering it remotely will help decrease the current load.
Very Critical: Invokes the Transcoder Based Approach whereby at present, the 3D

graphics image is simplified by 10%.

Remnant Battery Power

Critical: For all the stages, choose the stage map which provides the optimum power consumption as discussed in 5.4.1 even at the cost of rendering time.

Remnant Memory Pages

The response is identical to that for CPU Load. A stage of the graphics pipeline that consumes a lot of time for execution also inherently consumes higher memory. Since the time taken by a stage is linearly proportional to the number of vertices input to the particular stage.

Periodically, each of the Monitored values discussed above are sampled and their states checked. Depending upon their state, actions as indicated in table 5.1 are taken. Table 5.2 describes each of the actions referred to in table 5.1 in details. As is apparent, the *Very Critical* condition has a higher priority in the action sequence, meaning a check to see if if any of the monitored values are in the *Very Critical* state is carried out first.

5.4.3 Decision Making Summary

Below is a summary of the actions taken by Intelligraph.

1. For the list of stages in the graphics rendering process, apply the *Optimization for Speed* and *Optimization for Power* as explained in 5.3. At the end of this step, each stage has an internal flag set indicating whether it must execute locally/remotely and if doing so would be optimal from the rendering perspective/power consumption perspective.

2. The system starts executing using the optimized for speed stage mapping.

3. Hereafter, at the end of a fixed interval (set in the runtime), the following values are monitored, namely, *CPU Load, Memory, Remaining Battery Power.*

4. If any of the monitored values are *Very Critical*, then the transcoder is invoked.

5. The remaining combination of monitor value state and reaction is as shown in 5.1.

6. Lastly, the graphics file is rendered and the steps from 3 to 6 are repeated.

5.5 Summary

In order to aid the weak mobile host render complex 3D graphics images, as part of MADGRAF, we propose two alternative approaches, "Remote Execution " and' 'The Transcoder Based Approach". Remote Execution distributes the graphics rendering work between multiple machines but the fidelity of the final image displayed is not compromised upon. However, in the Transcoder Based Approach, the image to be displayed is reduced in quality and then displayed on the mobile device. Intelligraph is the intelligence agency of MADGRAF which chooses between "Remote Execution" or "The Transcoder Based Approach" at runtime. This decision is based upon the runtime performance evaluation of MADGRAF and is manifested as a two stage process. First, when the MADGRAF system starts and thereafter periodically while the MADGRAF system executes, Intelligraph decides between the two above mentioned alternative approaches. In short, Intelligraph allows MADGRAF to intelligently combine and use Remote Execution with the Transcoder Based Approach.

Chapter 6

Conclusions

We will conclude by sharing our experiences in developing MADGRAF and also the performance of the MADGRAF system as a whole with the different components communicating with one another.

6.1 Development Experiences

6.1.1 VRML Browser overlaying

In MADGRAF, we have the flexibility to choose a third party VRML Browser. However, having this flexibility means that we are forced to use the black box approach and therefore not expect to change browser source. One of the areas where the browser source would ideally require modification is in image refresh after transcoder operations.

Consider for example, that the client browser is displaying a VRML file *bunny.wrl* containing five thousand vertices. In the middle of operations, Intelligraph decides to simplify this file to a file containing two thousand vertices. This simplification operation takes place on the server machine. The important question is between the time that the simplification request goes to the server, the server actually simplifies the file and the client receives this file there is a lag, so what must be displayed during this lag? Our approach is to continue displaying the old file. But once the

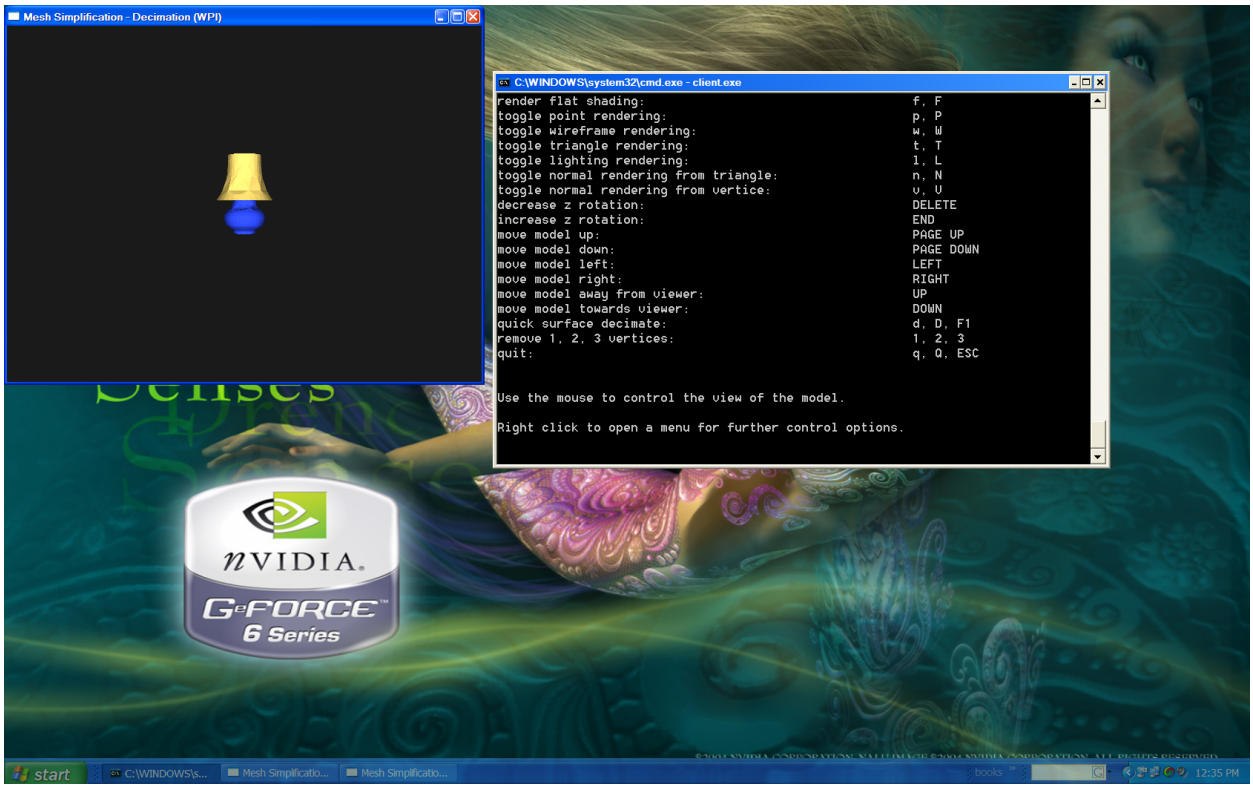


Figure 6.1: MADGRAF Client Browser Window Overlaying

new simplified file arrives, what must happen? Since we have not modified the source code of the browser, therefore we cannot just seamlessly make the browser application stop displaying the file it is currently displaying and render another file on the same window and same rendering context. In fact, the one feasible solution is to reopen another window display the new file in this window and close the older window. However, starting a new browser and closing the old browser involves a lag which can spoil the realtime nature of graphics applications.

Therefore, we overlay the current window with a new window. Thus the new window matches in size and position (on the desktop) the old window. Once this new window is ready, the old window is closed. From our experience, to the user, the process of opening a new window and closing the older window was not visible. Figure 6.1 shows the transition point as is evident from the taskbar below, that there are actually two browsers overlayed on top of one another.

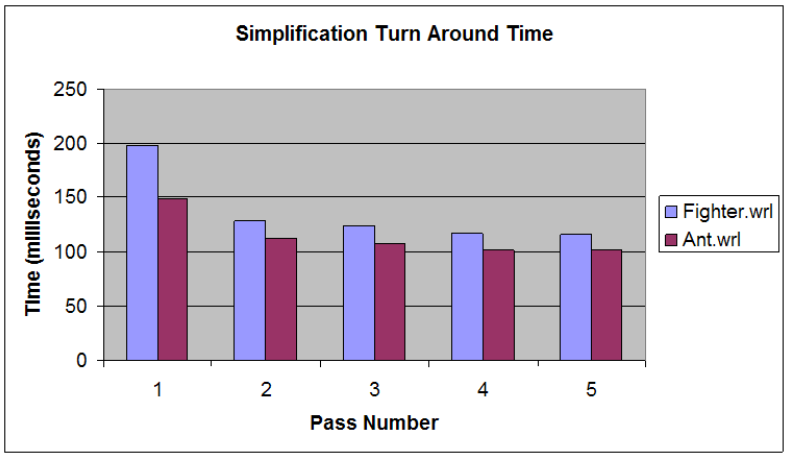


Figure 6.2: Turn Around Time for Simplification

6.1.2 Ice Runtime Performance

MADGRAF is a distributed graphics framework. Generally, in graphics applications, the system latency expected is low, meaning that the API, libraries are typically highly optimized sometimes by avoiding unnecessary levels of abstraction. The networking in MADGRAF is implemented using a Distributed Object Based Middleware, namely, Ice. Before the completion of the development process, we did have some apprehensions about the overhead incurred by the Ice runtime in invoking method calls on remote objects. A place where this latency can show up is during the transcoder operations. The client has a proxy to a Transcoder object stored on the server machine. The client invokes methods on the Transcoder object and the corresponding methods get invoked on the server object while the results get fetched back. From the traditional socket programming perspective, consider this to be the case when the client requests the server for a file. The server performs some operations on the file and returns the file back to the client.

However, we found out that the latencies expected above did not occur and that the Ice runtime is optimized for high performance system level applications.

6.2 MADGRAF System Performance

We executed the MADGRAF system displaying two different VRML files and noted down the transcoder operation time. The transcoder operation in this case was a simplification operation, simplifying the original file by 50%

Figure 6.2 shows the turn around time involved in the simplification operation. The VRML files *fighter.wrl* and *ant.wrl* contain 332 and 486 vertices respectively. The turn around time involves the time involved in the client requesting a simplification operation, simplifying the VRML file at the server, returning the simplified file back to the client and also the runtime overhead involved as a result of using the distributed middleware object based system.

Appendix A

Inside MADGRAF - The Build Process

In this document, we are going to address the following three issues, namely,

- *The Build Process of MADGRAF*
- *Source Code Explanatation*
- *Easy Steps to extending MADGRAF*

A.1 The Build Process

MADGRAF is a cross-platform application developed from ground up. Platforms on which the build process has been tested so far include Linux (2.6 kernel), Windows XP,2000, FreeBSD. The MADGRAF build is a command line operation and does not require any explicit IDE to be used for the same. However, one may continue to use their IDE of choice and build MADGRAF. In this section, we will address the following issues:

- *Third Party Libraries Required*
- *Build Environment Variables*

- *The actual build process*

The MADGRAF build directory can be found under the madgraf account at */madgraf/current/*.

A.1.1 Third Party Libraries Required

MADGRAF requires the following third party libraries:

- *wxWidgets*[38]: Cross platform system level calls, directory and file I/O operations. For example, *wxExecute* is a cross platform object based API used to spawn a new process. This is a wrapper around the the UNIX based *execvp*. Consider this to be a replacement of the JDK for JAVA programmers. For example, to query the list of files in a given directory, it would require one to do a ReadFile under win32 and a readdir under the UNIX world with widely varying syntaxes. Instead of having to write wrappers/ containers for such as this, wxWidgets takes care of the functionality/portability issue. In the future, the RegEx and threading library that comes with wxWidgets, can be used. The threading library is consistent with the POSIX thread model.

- *Ice*[16]: Distributed Object Middleware, developed by Michi Henning. It is an improvement of the CORBA object model. To begin with, consider Ice to be a replacement of RPCs that maintain states (since objects maintain state of the variables they contain).[39] We will cover Ice and CORBA in details in later sections.

- *Polyreduce*: An open sourced VRML browser. It also implements a simplification algorithm. Its usage is therefore for displaying VRML files, implementing simplification.

- *GZip*: Used for compression of data transferred between the client and the server machines.

- *CMake*[37]: Cross Platform build process. This is used for building MADGRAF in an OS independent manner.

```
PATH=$PATH:$HOME/bin/cmake/linux-x86/bin:$HOME/progs/CORBA/Ice-2.1.0/bin
LD_LIBRARY_PATH=$HOME/bin/wx/linux/lib:$HOME/progs/CORBA/Ice-2.1.0/lib
```

Figure A.1: Linux Environment Variables setup to capture coordination with Third Party Components using the BASH shell

A.1.2 Build Environment Variables

The presence of third party components in the build process means that the build process must know where in the file system, these components are present. We make use of environment variables to achieve the same under both Windows and Linux. This section explains the various environment variables that will have to be affected if MADGRAF is ported to a new machine. (These variables have already been setup on the CS file system, therefore any linux machine mounting the same should not require explicit changes). Environment variables are required to inform the compiler during the build process as to where it can find the include files for the third party libraries. During link time and execution time, they need to be able to inform the linker and the loader, the path to the library files. In short, there are compile time variables and link (load) time variables. We will address this separately for Windows and Linux.

A.1.3 Linux Environment Variables

The LD_LIBRARY_PATH must be set to the path containing the library files for Ice and wxWidgets. This informs the loader the path it has to search for these libraries. This is assuming that one is operating as non root and cannot do an ‘ldconfig‘.

The PATH variable must contain the path to the binaries of CMake, GZip and Ice. Figure A.1 shows a sample variable instantiation under Linux.

The include and lib paths for wxWidgets are handled in the CMake build file as will be discussed later.

A.1.4 Windows Environment Variables

For the windows platform, the LIB and INCLUDE variables need to be set as above. However, the dll files for Ice, wxWidgets need to be copied to the system32 directory.

A.1.5 The Actual Build Process

MADGRAF uses the cmake build process. A quick read of the cmake tutorial[37] would be helpful. CMake is a cross platform replacement of the GNU Build Process. A Build process is different from regular makefile based systems in the sense, that in large sized projects, makefiles and individual component dependencies are automated and not handwritten as they span various directories and are prone to mistake when hand written. The CMake build process automates the process of actually generating Makefiles both for Linux and Windows based systems. The "nmake" utility is the answer to the traditional "make" utility under Linux.

To build the existing code base, step into the *project* subdirectory under the MADGRAF root build directory. Step into either of LINUX, Win32 directories depending upon the platform required. Thereafter, merely typing *make* (or *nmake* if on the Win32 platform) is sufficient to build the existing system. The resultant binaries are available under the bin directory whereas the static library files are made available under the lib directory as shown in the figure A.2.

Sample CMake Build File

Under each of the project/OS directories, there is a *CMakeLists.txt* that controls the build rules for each of the different source level components. A sample CMakeLists.txt file is shown in the figure A.3.

In figure A.3, the ADD_LIBRARY line is used to add an object file named machine built from the files machine.cpp and machine.h under the directory pointed to by the INTELLIGRAPH variable. Likewise, the ADD_EXECUTABLE line is used to construct a binary from input object files. The TARGET_LINK_LIBRARIES is used to link the binary with a set of libraries. These individual lines are used by the CMake build process to autogenerate the Makefiles efficiently for the underlying platform.

```
MADGRAF
 -repository
 -src
 -bin
  +win32
  +linux
   -test
 -lib
  +win32
  +linux
 -project
  +win32
  +linux
```

Figure A.2: Build Directory Structure of MADGRAF

```
ADD_LIBRARY(machine ${INTELLIGRAPH}/machine.cpp ${INTELLIGRAPH}/machine.h
           ${INTELLIGRAPH}/imachine.h)
ADD_LIBRARY(stage ${INTELLIGRAPH}/stage.cpp ${INTELLIGRAPH}/stage.h)
............
ADD_EXECUTABLE(${TEST}/smpower ${INTELLIGRAPH}/testsmpower.cpp)
TARGET_LINK_LIBRARIES(${TEST}/smpower smpower ismpower stage dbstage istage
                      machine dbmachine imachine smpower dbpower Ice IceUtil)
```

Figure A.3: CMakeLists.txt

A.2 Source Code Explanation

In this section we will identidy the key components present in the MADGRAF code base and for each component, we will study the classes and the corresponding source files in details. MADGRAF makes use of a distributed file system (NTFS drive mapping) and the NFS (under LINUX) transparently for file transfer and to present the clients with a uniform view of the VRML file repository.

A.2.1 Component Identification

The major components involved in MADGRAF are as follows:

1. *Intelligraph*
2. *Transcoder*
3. *MetaFile*
4. *VRMLViewer*
5. *Database*
6. *Ice Controller*

Intelligraph

The Intelligent Decision making engine of MADGRAF. All source files are under the *ROOT/src/intelligraph* directory. Any directory mentioned is a subdirectory to it. The key classes, their responsibilities along with the source files containing them are enumerated below:

- *iMachine*: An interface declared in *imachine.h*, used to capture the information about a machine. Implementations of this interface include *Machine in machine.cpp* and *iMachinePrx in imachine.cpp*. Machine is the server side implementation of this class whereas clients wishing to model machine information use the proxy to this class. Makes use of the proxy pattern.[21] For example,

inorder to represent a client machine with processor speed 2 GHz and memory size 1 GB, a Machine object is created on the server and its proxy held by the client. This object can later be supplied to Intelligrap to represent the client machine's processing power.

- *iSMPower*: An interface declared in *ismpower.h*, used to capture the decision making process for optimizing stage mapping wrt Power. This interface is implemented by *SMPower in smpower.cpp* and *iSMPowerPrx in ismpower.cpp*. The client side Intelligraph requiring to invoke this decision object instantiates a SMPower object on the server machine and holds a proxy to this object, namely, iSMPowerPrx. Operations on iSMPowerPrx have a one-to-one mapping with the operations on the SMPower class.

- *iSMTime*: An interface declared in *ismtime.h*, used to capture the decision making process for optimizing the frames per second on the client side. This interface is implemented by *SMTime in smtime.cpp* and *iSMTimePrx in ismtime.cpp*.

- *iStage*: An interface declared in *istage.h*, used to model a Stage in Remote Execution. A Stage models information about the machine it is executed on and whether it is executed locally or remotely. Thus one stage is executed with two Machine objects, one of which is the server Machine (remote) and the other one the client Machine (local). This interface is implemented by *Stage in stage.cpp* and *iStagePrx in istage.cpp*.

Transcoder

This component is reponsible for performing the transcoder operations on the MADGRAF 3D graphics files. It can be found under the ROOT/src/transcoder directory. The *iTranscoder* interface captures the basic operations that can be performed by any Transcoder, namely, *encode* and *decode*. For example, the *SimplificationRatio class in simplificationratio.cpp* and the *GZip class in gzip.cpp* implement the *iTranscoder* interface. In case of the *SimplificationRatio* class, the encode operation takes in an

input file name, simplifies the file by a certain ratio (provided during object construction) and returns the simplified filename. The decode operation is a simple noop and is maintained for consistency. For the *GZip* class, the encode operation compresses the file and transfers it to the client. The decode operation decompresses the file.

MetaFile

This component is responsible for storing information about the server side File System. In particular, it is responsible for being able to query the list of available VRML files on the server, tackle the differences in naming conventions for Windows and Linux based files systems and initiate the file copying operations on the Distributed File System used by MADGRAF. It contains a single class, MetaFile in the file ROOT/src/metafile/metafile.cpp.

VRMLViewer

This class captured in ROOT/src/vrmlviewer/vrmlviewer.cpp is a wrapper around the actual VRMLViewer used by MADGRAF for display on the client machine. Currently, a default VRML Browser, namely, *polyreduce in algo.exe* is made use of which can be found under ROOT/bin/win32/algo.exe. During the construction of this object however, any other VRML Browser's complete pathname can be provided, in the absence of which, the default browser mentioned above is used.

Database

Intelligraph makes use of past performances for remote execution (NOT for the Transcoder Based Approach). A file based database system is implemented by ROOT/src/database/database.cpp. Each of the tables in the database, namely, Machine, Stage, Time, Power are accessed through individual classes, namely, *DBMachine, DBStage, DBTime and DBPower.*

Ice Controller

This is the core of the networking and distributed part of MADGRAF. Currently, we make use of an improved variant of CORBA, namely, Ice[16]. There are 3 kinds of objects (this applies to all classes that have been covered so far), namely, client side classes, server side classes and mixed classes. Client classes are those that are *implemented* entirely on the client machine. Likewise, server classes are those classes that are implemented entirely on the server machine. Thus, the client does not know about the existence of these classes and cannot directly invoke methods on them. However, the mixed classes are those that are implemented on the server (can be migrated between machines, persisted to secondary storage, run as object per thread or multiple objects per single thread) but can be accessed from the client using a proxy object. A Proxy object implements the same interface as the server side object but punts the method invocations to the server side object, thereby shielding the client from the actual location of the server object. The methodi invocation is similar to that in RPC. The underlying networking used for the data transfer can be controlled. I suggest a read of the Ice documentation for further details, also as a future improvement, consider using IcePack, IceBox and Freeze for stronger distributed object features.[16] The server side implementation of these mixed classes can be in Java, C#, VB, PHP, C++.

The source files can be found under the ROOT/src/intelligraph/temp directory. (As an aside, consider forgiving the unseemly names for the following classes :-))

hello: This interface is used to return the clients with the proxies for the server side objects. This makes use of the factory pattern[21]. This interface is implemented by *helloI class in helloI.cpp* and by *helloPrx class in hello.cpp*. The *helloI* class is the server side implementation of this interface whereas the *helloPrx* is the client side proxy. A client uses a configuration file such as that in ROOT/bin/linux/config (or ROOT/bin/win32/config) to inform the Ice Runtime about the network location of the different machines containing Ice objects. I haven't got the chance to test out multiple servers and replication of objects but such as this feature is available under Ice. This has the advantage that if one server is down, then another server can continue serving the client without any application stops. Also, wireless clients

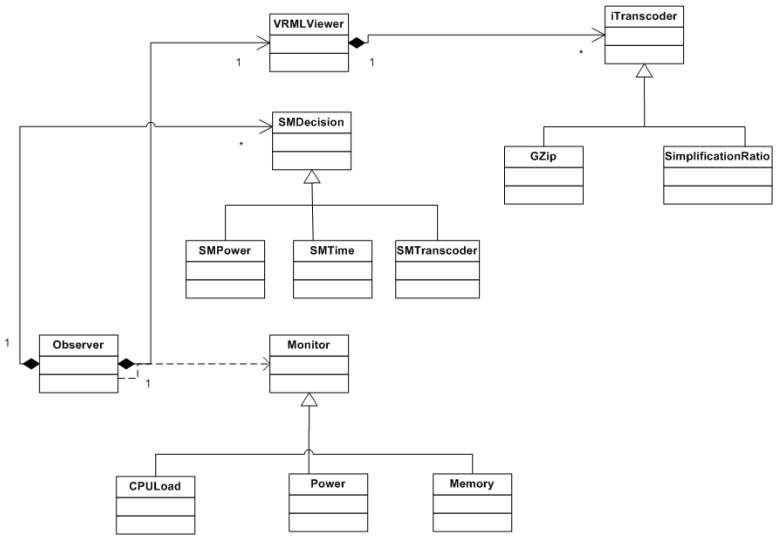


Figure A.4: Key Classes in MADGRAF and their association

can choose between multiple servers depending upon known proximities. (Alrite, I am guessing here:-)) As an example, consider that the client wishes to get a proxy to a mixed class such as *GZip* class. It uses its *helloPrx* proxy object to request a proxy to the *iTranscoderPrx* object which is a transcoder proxy. Observer the factory methods in the helloI.h file each returning proxy objects. This class can be specialized to override the factory methods.

Figure A.4 shows a high level relationship between some of the key classes.

A.3 Extending MADGRAF

This section concerns with explaining how the present functionality of the MADGRAF system can be augmented. We will discuss sequential steps to be taken during coding to implement changes to the following classes:

- *Transcoder*
- *Monitor*
- *Decision Making*

```
ADD_LIBRARY(newtranscoder ${TRANSCODER}/newtranscoder.cpp
                          ${TRANSCODER}/newtranscoder.h)
...
TARGET_LINK_LIBRARIES(server ..... newtranscoder)
```

Figure A.5: Changes to CMakeLists.txt for adding a new transcoder

```
/** Into the header file*/
Madgraf::iTranscoderPrx createDummyTranscoder(const Ice::Current&)
.........
/**In the cpp file*/
Madgraf::iTranscoderPrx first::helloI::createDummyTranscoder(
const Ice::Current &_system_)
{
Madgraf::iTranscoderPtr object_=new Madgraf::Dummy();
return Madgraf::iTranscoderPrx::uncheckedCast(_system_.adapter->addWithUUID(
object_));
}
```

Figure A.6: Add these lines to helloI.h and helloI.cpp respectively

A.3.1 Extending the Transcoder

Following are the steps to be undertaken to add a new Transcoder. An example of such would be if one wanted to implement a new simplification algorithm and add it into the current MADGRAF system.

1. Under the "ROOT/ src/ transcoder/ " directory, add two new files, namely, newtranscoder.h and newtranscoder.cpp. (Replace newtranscoder with an appropriate name for the transcoder). The new files contain the class implementing the Simplification algorithm. This class must extend the interface, *iTranscoder*.

2. In the "ROOT/ project/ linux/ CMakeLists.txt" build file, add a line as shown in figure A.5, the ADD_LIBRARY command.

3. Change the "ROOT/ src/ intelligraph/ temp/ first.ice" file and add the following line of text
   Madgraf::iTranscoder createDummyTranscoder()*. The .ice file is an interface file for Ice. The createDummyTranscoder method is the factory method that returns an object of the newly created Transcoder.

4. Step into the "ROOT/ src/ intelligraph/ temp" directory and type the following: *slice2cpp first.ice*. This compiles the interface file to generate automated C++ code. This concept is called as language mapping in CORBA parlance.

5. Change the ROOT/ src/ intelligraph/ temp/ helloI.h and *ROOT/ src/ intelligarph/ temp/ helloI.cpp* files to contain the 2 lines shown in figure A.6. However, in the *helloI.h* file , don't forget to include the header file of the newly created transcoder, for example, in this case, *newtranscoder.h*.

6. In the "ROOT/ project/ linux/ CMakeLists.txt" file, hunt for the line containing the ADD_EXECUTABLE statement for *server* and add the *dummy* transcoder library to it as shown in figure A.5.

7. In the "ROOT/ src/ vrmlviewer/ vrmlviewer.cpp" file, change the *VRMLViewer :: reduceVertices* method to contain the following line of text in place of any other similar line present. *Madgraf::iTranscoderPrx transcoder_=_factory -> createDummyTranscoder();*

Thats it!! Congratulations, you have just added a new transcoder into the MADGRAF system without having to bother about CORBA/ Ice objects, networking or any other subtleties.

A.3.2 Extending the List of Monitor Objects

The steps involved in the extension process is as follows:

1. This is a fairly simple process. The new monitor object must extend the interface *Monitor*. The new monitor must also provide a body to the *bool getValue(void)* method. This method does the job of monitoring whatever the

```
Monitor *newmonitor_=new NewMonitor();
_monitorList.push_back(newmonitor_);
```

Figure A.7: Add these lines to Observer.cpp Observer::Observer method

Monitor object is supposed to monitor. It returns true if the monitored value is beyond a threshold and false otherwise. Two successive true returns is called as a double trip whereas one such is called as trip. Thus, a Monitor object can be in either of the following states:

(a) *Normal*

(b) *Trip*

(c) *Double Trip*

2. The intented architecture being that depending upon the different states of the various Monitor objects, decisions can be taken inside Observer::decision ("ROOT/src/intelligraph/clientcontrol/observer.cpp"). Once the new Monitor class is implemented, it must be instantiated and registered to a list of Monitor objects maintained by the Observer object. Refer A.4 for the relation between the two classes.

   The new monitor object can be added to the Observer object in its constructor as shown in figure

Thats it!! Thereafter, every sampled period, the Observer class will run each of the Monitor objects to check for its state and then invoke decision based on that.

A.3.3 Extending the Decision Making Objects

Following are the steps to be undertaken to add a new Transcoder. An example of such would be if one wanted to implement a new simplification algorithm and add it into the current MADGRAF system.

```
ADD_LIBRARY(newdecision ${TRANSCODER}/newdecision.cpp
                        ${TRANSCODER}/newdecision.h)
...
TARGET_LINK_LIBRARIES(server ..... newdecision)
```

Figure A.8: Changes to CMakeLists.txt for adding a new Decision Maker

```
  /** Into the header file*/
  Intelligraph::iSMTranscoderPrx createNewDecision(const Ice::Current&)
  .........
  /**In the cpp file*/
  Intelligraph::iSMTranscoderPrx first::helloI::createNewDecision(
  const Ice::Current &_system_)
  {
Intelligraph::iSMTranscoderPtr object_=new NewTranscoder();
return Madgraf::iTranscoderPrx::uncheckedCast(_system_.adapter->addWithUUID(
object_));
  }
```

Figure A.9: Add these lines to helloI.h and helloI.cpp respectively

1. Under the "ROOT/ src/ intelligraph/ " directory, add two new files, namely, newdecision.h and newdecision.cpp. (Replace newdecision with an appropriate name for the Decision Maker). The new files contain the class (NewDecision) implementing the Decision Algorithm. This class must extend the interface, *iSMTranscoder*.

2. In the "ROOT/ project/ linux/ CMakeLists.txt" build file, add a line as shown in figure A.8, the ADD_LIBRARY command.

3. Change the "ROOT/ src/ intelligraph/ temp/ first.ice" file and add the following line of text
Intelligraph::iTranscoder createNewDecision()*. The .ice file is an interface file for Ice. The createNewDecision method is the factory method that returns an object of the newly created Decision Maker.

4. Step into the "ROOT/ src/ intelligraph/ temp" directory and type the following: *slice2cpp first.ice*. This compiles the interface file to generate automated C++ code. This concept is called as language mapping in CORBA parlance.

5. Change the "ROOT/ src/ intelligraph/ temp/ helloI.h" and "ROOT/ src/ intelligarph/ temp/ helloI.cpp" files to contain the 2 lines shown in figure A.9. However, in the *helloI.h* file , don't forget to include the header file of the newly created transcoder, for example, in this case, *newdecision.h*.

6. In the "ROOT/ project/ linux/ CMakeLists.txt" file, hunt for the line containing the ADD_EXECUTABLE statement for *server* and add the *newdecision* transcoder library to it as shown in figure A.8.

7. In the "ROOT/ src/ intelligraph/ clientcontrol/ observer.cpp" file, change the *Observer::decision* method to contain the following line of text in place of any other similar line present. *Intelligraph::iTranscoderPrx transcoder_=_factory -> createNewDecision();*

Thats it!! Congratulations, you have just added a new Decision Maker into the MADGRAF system without having to bother about CORBA/ Ice objects, networking

or any other subtleties.

Appendix B

PowerSpy Tutorial

Below is the set of steps required to executed PowerSpy:

1. Open a terminal window and enter *co*. This should take you to the correct directory.
2. sc start powerspy
3. Run IrpTracker. From the GUI, select the devices : \Device\tcpip and \Device\disk
4. *debugger ApplicationName*
5. Stop Irptracker and save its output log to some known place and remember it.
6. *offline \powerspyfile.dat IRPTrackerOutput*

The output of 6 contains the thread level power consumption followed by the Device Level Power Consumption.

Bibliography

[1] Kutty S Banerjee, Emmanuel Agu. *Remote Execution for 3D Graphics on Mobile Devices* in Proc. IEEE WirelessCom 2005 (to appear).

[2] Emmanuel Agu, Kutty Banerjee, Diane Kramer, Shirish Nilekar, Oleg Rekutin. *A Middleware Architecture for Mobile 3D Graphics*, in Proc. IEEE Workshop on Mobile Distributed Computing (MDC), June 2005, (to appear).

[3] Kutty S Banerjee, Emmanuel Agu. *PowerSpy: Fine-Grained Software Power Profiling for Mobile Devices*, in Proc. IEEE WirelessCom 2005 (to appear).

[4] Garland M and Heckbert P, "Surface Simplification using Quadric Error Metrics", in Proc. ACM SIGGRAPH 1997, pp. 209-216

[5] Decoret X, Durand F, Sillion F and Dorsey J, "Billboard Clouds for Extreme Model Simplification", in Proc. ACM SIGGRAPH 2003

[6] Deering M, Geometry Compression, in Proc. ACM SIGGRAPH 1995, pp. 13-20

[7] Tulika Mitra, Tzi-cker Chiueh. Implementation and Evaluation of the ParallelMesa Library.*International Conference on Parallel and Distributed Systems, December 14 - 16, 1998, Taiwan* .

[8] Jason Flinn, Dushyanth Narayanan and M. Satyanarayanan. Self-Tuned Remote Execution for Pervasive Computing. *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS-VIII), Schloss Elmau, Germany, May 2001.*

[9] Greg Humphreys, Mike Houston, Ren NG, Randall Frank, Sean Ahern, Peter D. Kirchner, James T.Klosowski. Chromium: A Stream-Processing Framework for Interactive Rendering on Clusters. *ACM Transactions on Graphics (TOG), Volume 21 , Issue 3 (July 2002), Proceedings of ACM SIGGRAPH 2002 Graphics hardware, pages 693 - 702, 2002.*

[10] Greg Humphreys, Matthew Eldridge, Ian Buck, Gordon Stoll, Matthew Everett Pat Hanrahan. WireGL: A Scalable Graphics System for Clusters. *Proceedings of SIGGRAPH 2001.*

[11] Decoret X, Durand F, Sillion F and Dorsey J, "*Billboard Clouds for Extreme Model Simplification*", in Proc. ACM SIGGRAPH 2003.

[12] Luebke, D.P, "*A Developer's Survey of Polygonal Simplification Algorithms*", IEEE CG&A, May/June 2000, pp 24-34

[13] Nicolaas Tack *et al*, *3D Graphics Rendering Time Modelling and Control for Mobile Terminals*, in Proc. 3D Web, 2004.

[14] Yang J *et al*, *Design and implementation of a large-scale hybrid distributed graphics system*, in Proc. Eurographics PGV 2002, pp. 39-49.

[15] Narayanan D, FLinn J, and Satyanarayanan M, "*Using History to Improve Mobile Application Adaptation*, in Proc. 3rd Workshop on Mobile Computing Systems and Applications (WMCSA) 2000.

[16] Michi Henning. *A New Approach to Object Oriented Middleware.* IEEE Internet Computing, January-February 2004, pp 66-75.

[17] M Pietrik, *An In-Depth Look into the Win32 Portable Executable File Format.* Inside Windows, MSDN Magazine. 2003 Microsoft Corporation.

[18] J Flinn, M Satyanaranan, *PowerScope: A Tool for Profiling the Energy Usage of Mobile Applications.* Second IEEE WMCSA 1999, New Orleans, LA.

[19] J R. Lorch, A J Smith, *Apple Macintosh's Energy Consumption.* IEEE Micro, vol. 18(6) 1998.

[20] T Li, L K John, *Run-time Modelling and Estimation of Operating System Power Consumption.* ACM SIGMETRICS '03, June 10-14, 2003, San Diego, CA.

[21] *Design Patterns: Elements of Reusable Object-Oriented Software.* Erich Gamma et al.,

[22] M Russinovich, D Solomon, *Inside Microsoft Windows 2000,* 3rd Edition, Microsoft Press, 2000.

[23] W Oney, *Programming the Microsoft Windows Driver Model,* Microsoft Press, 1999

[24] J Lozano, A Baker, *The Windows 2000 Device Driver Book* 2nd Edition, Prentice Hall, 2000

[25] J Robbins, *Debugging Applications,* Microsoft, 2000.

[26] Mark Segal, Kurt Akeley. *The OpenGL Graphics System: A Specification. Version 2.0*

[27] John L. Hennessy, David A. Patterson. *Computer Architecture, 3rd Edition A Quantitative Approach.* ISBN:1558605967 Morgan Kaufmann Published May 2002.

[28] OpenGL(R) Reference Manual: The Official Reference Document to OpenGL, Version 1.2 (3rd Edition). *Addison-Wesley Publishing Company.*

[29] Mark Segal, Kurt Akeley. The Design of the OpenGL Graphics Interface.

[30] *IRP Tracker Utility, V1.3.* Open Systems Resources.

[31] OpenGL website, *www.opengl.org*

[32] Microsoft DirectX Website, *http://www.microsoft.com/windows/directx/*

[33] AC3D 3D Authoring Toolkit, *www.ac3d.org.*

[34] Mesa3D Webite, *www.mesa3d.org*

[35] OpenVRML VRML Browser, *www.openvrml.org*

[36] Windump Website, *windump.polito.it*

[37] *www.cmake.org*

[38] *www.wxwidgets.org*

[39] CORBA Website, *www.omg.org*

[40] Brian Paul. *www.mesa3d.org.*

[41] *www.msdn.com*