Estimating Mobile Memory Requirements and Rendering Time for Remote Execution of the Graphics Pipeline

Kutty Banerjee, Fan Wu and Emmanuel Agu

Abstract

Mobile devices have limited processing power, memory and battery power. Remote execution, wherein part or entire graphics pipeline is offloaded to a powerful surrogate server, is an attractive solution for low end mobile devices such as PDAs and cell phones that lack floating point units or GPUs. We have found that remote execution of floating-point-intensive pipeline stages such as transform and geometry operations can produce speedups of up to 10 times for a low-end mobile device. We introduce generalized pipeline-splitting, a paradigm whereby 15 sub-stages of the graphics pipeline are instrumented with networking code such that they can run either on a mobile client or a surrogate server. To validate our concepts, we create Remote Mesa (RMesa). As a foundation for deciding which stages of the pipeline would benefit from remote execution, this paper presents analytical models for the overall rendering time, memory requirements and roundtrip network delay incurred by RMesa.

Categories and Subject Descriptors (according to ACM CCS): I.3.2 [Computer Graphics]: Remote systems, distributed/networked graphics

1. Introduction: Remote Mesa (RMesa)

Mobile devices have limited processing power, memory and battery power. Remote Execution provides mobile hosts the with a mechanism to offload rendering operations to a powerful surrogate server in order to increase the complexity of scenes that mobile hosts can render, improve overall rendering speed and under some conditions, improve battery use.

Remote Mesa (RMesa) is a modified OpenGL implementation that enables a mobile client to remotely execute certain stages (and sub-stages) of the graphics pipeline on a remote surrogate server. Specifically, networking code is placed between 15 OpenGL stages and sub-stages such that partially rendered vertices, contextual information and relevant matrices (modelview, projection, etc) can be transferred from a mobile client to a surrogate server for execution. The server executes the transferred stage(s) before returning results back to the mobile client. AnyGL [5] has previously split the graphics rendering pipeline into four nodes on a cluster.

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 assymetry 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 GPUs 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 1 is a sample of our results that shows the speedup in the overall rendering time for a PDA when the modelview stage is rendered on a server, over a Wireless LAN.

RMesa is the remote execution module of our Mobile Adaptive Distributed Graphics Framework (MADGRAF) [2] in which a powerful server can assist a resource-deprived mobile client in rendering demanding graphics scenes. A MADGRAF server may also automatically simplify or compress requested meshes, or convert them to billboards clouds.

The actual decision on where each of RMesa's stages should be rendered is determined by a centralized server-side MADGRAF module called the 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*

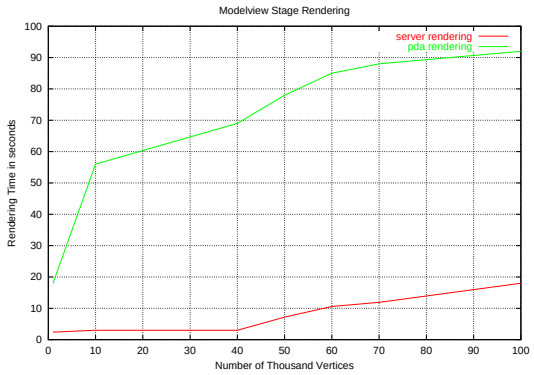

Figure 1: *PDA rendering time with modelview on server*

client and server machines (2) *Memory requirements of a given mesh* (3) *Network overhead incurred in transmitting data between client and server* that 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 in [4] to encompass remote execution in RMesa, the different rendering capabilities of the the client and server, considers memory requirements of mobile clients and network overhead incurred during remote execution. The rest of this paper is as follows. Section 2 discusses our model for predicting the rendering time of RMesa, section 3 discusses our model to predict the memory consumption of individual stages and section 4 analyzes the roundtrip network delay incurred during remote execution and section 5 is our conclusion.

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 2.1 discusses the equation used to predict the execution time whereas 2.2 discusses how the individual parameters in the equation are measured or obtained.

2.1. Measured Parameters

Our model for the total rendering time of a given client-server mapping of RMesa builds on the equation 1, proposed in [4] and models the rendering time, T of a single machine software implementation of OpenGL as

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

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 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.

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 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. [4] 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. 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 3.1, we discuss the different types of memory used by an application while in 3.2, we discuss the memory etimation policy for each stage of the graphics pipeline. Further in 3.3 we compare the results from 3.2 with those obtained from actual measurements.

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.

3.2. 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.

3.2.1. 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:

$$Memory_{Geom} = M_{GC} + I_{pv} * V + I_{pcv} * V_C \quad (2)$$

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:

$$Memory_{Light} = M_{LC} + V * GLSize \quad (3)$$

$$Memory_{Normal} = M_{NC} + V \quad (4)$$

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.

3.2.2. 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.

$$Memory_{accum} = W * H * sizeof(GLaccum) \quad (5)$$

$$Memory_{alpha} = W * H * sizeof(GLchan) \quad (6)$$

$$Memory_{stencil} = W * H * sizeof(GLstencil) \quad (7)$$

$$Memory_{depth} = W * H * maxDepth \quad (8)$$

$$Memory_{color} = W * H * maxColor \quad (9)$$

where *maxDepth* = either 2 bytes or 4 bytes depending upon the platform and *maxColor* =1,2 or 4 depending upon whether stereo and back buffers are used. M_{GC} = 23132 bytes and I_{pv} = 52 bytes. I_{pcv}, *maxDepth*, *maxColor* = 4 bytes. M_{LC} = 352, *GLSize* = 16 and M_{NC} = 32.

3.3. Model Validation by Actual Memory Measurement

In order to validate our model developed in 3.2, we compare the predicted results with the actual results obtained by

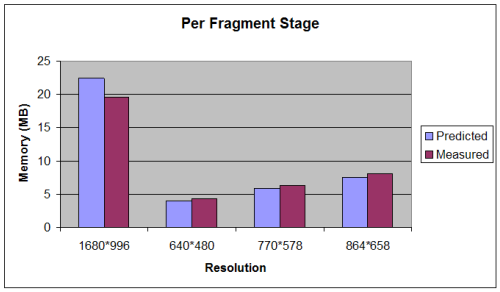

Figure 2: *Predicted Vs. Measured Memory Allocations*

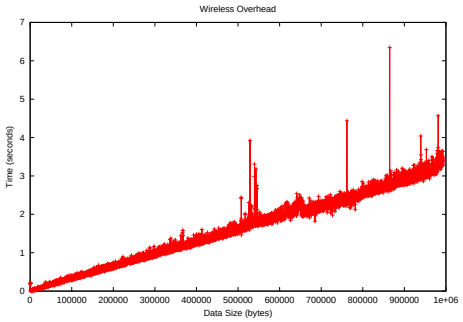

Figure 3: *802.11b Wireless LAN Round Trip Time*

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 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 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 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.

4. Network Overhead

In this section, we estimate the network overhead involved in sending data from the mobile host to the server. Figure 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 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.

5. Conclusion and Future Work

We have presented a model that allows a server to estimate the rendering time and memory consumption of Remote Mesa client. Battery power is currently one of the most limiting resources on a mobile device. Currently the model does not include a model to estimate the energy consumption by individual stages of the pipeline. Though we have presented a work PowerSpy[3] which measures the amount of energy consumed by different threads and I/O devices attached to the system, it cannot not predict energy consumption. In the future, we hope to develop a model to predict power usage on the mobile device.

References

[1] Kutty Banerjee, Emmanuel Agu, *Remote Execution for 3D Graphics in Mobile Devices*, in Proc. IEEE WirelessCom 2005. 1

[2] Emmanuel Agu *et al*, *A Middleware Architecture for Mobile 3D Graphics*, in Proc. IEEE Mobile Distributed Computing Workshop, June 2005. 1

[3] K. Banerjee, E. Agu. *PowerSpy: Fine-Grained Software Energy Profiling for Mobile Devices*, in Proc. IEEE WirelessCom 2005. 4

[4] Nicolaas Tack *et al*, *3D Graphics Rendering Time Modelling and Control for Mobile Terminals*, in Proc. 3D Web, 2004. 2, 3

[5] Yang J *et al*, *Design and implementation of a large-scale hybrid distributed graphics system*, in Proc. Eurographics PGV 2002, pp. 39-49. 1