Time-Critical Rendering and the

Temporal Behavior of Graphics Systems

by

George Kyriazis

A Thesis Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the Degree of

MASTER OF ENGINEERING

Approved:

____

Michael J. Wozny
Thesis Advisor

Rensselaer Polytechnic Institute
Troy, New York

May 1993

Table of Contents

List of Figures

Acknowledgments

I would first like to thank everybody in the Rensselaer Design Research Center's Visual Technology Program for the "best of times" and the "worst of times". Special thanks fo Prof. Wozny for providing me with an excellent environment for this work. Extra thanks to Robert "Bob" O'Bara for finding 10 millions ways of playing "Devil's Advocate", for his infinite wisdom in scheduling "early-morning" meetings and for continuously finding ways to improve this thesis. Major thanks to Jim Miller for some long productive discussions, for eating my "bagel"s and for peacefully wondering why things don't compile while I am out to lunch. Thanks also go to all my office-mates (ccw from my location): Jan "Mr. Surface" Bøhn, Steve Rock, Mike "where's my thesis" Roubik, Jim Miller and George "XView" Petras.

I would also like to thank all the people at Sun that have made this work possible: Salim Abi-Ezzi for the inspiration and support for this work, Srikanth Subramaniam, Leon Shirman, Russ Pflughaupt for his infinite XGL wisdom, and Zoran Popovic for introducing me to emacs and teaching me the existence of the C-h key. I would like to emphasize my thanks to Salim for everything that he has done for me, including the unforgettable summer of '91 at Sun Microsystems.

I'd like to finally thank all my friends and enemies for keeping me sane during this period. Special thoughts go to the person who very generously decided that my education is worth more than it costs, and to my parents for developing the engineering spirit in me, even though neither of them have any such background (how did you do it anyway?), and for realizing it's best for me if they stay with no children at home.

Abstract

Time-critical rendering is the act of trying to render images on the screen under temporal constraints, usually imposed by the human observer. This problem is very common in visual simulation and virtual environment applications, but a complete and unified solution does not exist yet.

In this thesis, we first develop an abstract model of the graphics pipeline. Then we give a brief description of a time-critical renderer and how it fits in the above model. The time-critical renderer will require the use of a temporal model of the graphics pipeline in order to produce a visually correct result under the given temporal constraints.

The above time model is then explained in detail. We begin by explaining how this model is utilized, and then describe the development of the tests that were used to evaluate that model for various graphics architectures.

From the results obtained from the above tests, we then discuss characteristics of the various architectures tested, and provide feedback on the requirements that time-critical rendering imposes on future graphics architectures.

1. Introduction

We are interested in the fields of rendering and interactive computer graphics. Within that scope, we are attacking the problem of time-critical rendering, that is rendering under a temporal constraint.

1.1. The Big Picture

Psychology tells us that the process of learning and understanding is tightly coupled with the ability of the brain to relate and form concepts [BHN87]. Vision is a very critical sense to humans, and therefore a large percentage of the human understanding and interaction is done by visual communication. The use of images to convey information has been used throughout history from static images like paintings or illustrations, to movies that also convey the concept of time to the human brain. Computer graphics represents the final frontier to address such interaction in the continuous quest for conceptual realism.

Computer graphics, (CG) as an applied topic, started in the early 1960's [Sut63]. Since then, it has been acting as an "enabling" technology in various science, engineering and art disciplines. Today, fundamental theories and concepts have been laid out, and CG has obtained different names depending on its involvement with various disciplines (scientific visualization, visual simulation, or virtual environments, to name a few). Additionally, CG applications can be divided into interactive and non-interactive (batch). CAD systems and modelers can be classified as interactive, and ray-tracing can be classified as non-interactive. Since we are interested in the temporal behavior of the human/machine interface, we will be looking at interactive computer graphics.

In addition to computer-generated interactive graphics, other electronic technologies (mostly video) are also trying to become more interactive, for example, Interactive Television (ITV, [Lu93][Cla92]), video-disk based electronic games and portable computing. Computer graphics is also being used in interactive simulation and simulation steering applications, where the user is trying to get a better feeling of the simulation results through interaction with the actual data.

The problem with today's graphics systems is that they are focused more on the data representation and visual attributes of the geometry, and have neglected temporal behavior, which ends up being much more inconsistent from system to system than visual behavior. The programmer's interface to PHIGS [PHI88][RO93] is a perfect example; almost all of the API is reserved by the get/set routines for the visual representation/attributes and the state of the internal graphics database. We can see its temporal unpredictability by trying, for example, to rotate a cube using two different implementations of PHIGS on two different machines. One may be too slow to realize that the cube is rotating, the other one may be too fast to realize that there is a cube in the screen.

1.2. Related / Previous Work

Related work can be traced in the fields of real-time visual simulation, multi-media systems, computational geometry, surface modeling and real-time systems.

Visual simulation is an area in which most of the related work took place. High-end flight simulators have tried for many years to provide a real-time temporal feedback for complex visual environments. Most of the references in that field, though, are very difficult to find, because of the military applications of flight simulation [Mon90], and up to recent years have only been applied to expensive and special flight-simulation hardware. Simulation sickness, which is one of the best examples of physiological side-effects of misleading visual and temporal cues related to visual simulation, is also described in [Mon90]. Work related very closely to flight simulation is also described in [ZPMW92], that shows a network-oriented visual simulation system used in battle simulation. Interactive frame rates, culling of out-of-view geometry and level-of-detail representation switching[1] are utilized. In terms of non-military work, [ARB90] and [FST92] consider the case of interactive walkthroughs of buildings. Here, room partitioning and walls are used extensively to partition the database for culling during traversal. Again, level-of-detail is utilized and interactive frame update rates are achieved. Finally, the Performer toolkit by SGI [Sil92d] provides a framework for visual simulation that includes all of the features of the above references. It also claims to be able to keep up a constant frame rate. But it is left to the programmer, to provide geometry that is renderable in the requested amount of time.

All of the above work utilizes extensive data partitioning and computational geometry algorithms. [PS85] provides an excellent review of the basic algorithms. Additionally, work done in accelerating ray-object intersections in ray-tracing have and will be used more in the future of interactive graphics. Such work, which also utilizes computational geometry, is mostly related to data partitioning in order to decrease that amount of data that is to be considered for rendering at each frame. A very comprehensive summary can found in [Gla89] and [AK89]. Algorithms like those will have to be used extensively inside the time-critical renderer.

[1] The act of switching between different representations (of various levels of complexity) of the same object, usually as a function of the distance from the viewpoint. You can then use a more detailed representation when the object is close and a less detailed one when the object is far away.

In terms of rendering algorithms, [Abi89] has shown an algorithm for the adaptive tesselation of NURBS based on the relationship between the surface and the viewpoint. The tesselation criterion in the above algorithm can be extended so that tesselation levels are not only adjusted based on the view, but also on time allotted for rendering the surface in question.

1.3. What's missing

The most basic aspect of time-critical or real-time systems that is missing from all of the above work is treatment of time as an integral part of the system. Even in the cases where interactive update rates are required, time constraints are treated as either being satisfied automatically, or built on top of an inherently non-temporal system.

It is the purpose of this paper to treat time as an integral part of the graphics system. By knowing the temporal behavior of the different graphics operations, we can better solve time-critical problems.

It is important to note the significance of time in interactive systems. When an application is a collection of several interacting systems (the user being one of them), it is important for each of the systems to be able to do its work in relation with the rest of the systems. The user, for example, being a part of the system, has several constraints attached to him. An example would be that the frame rate has to be above a certain limit for him to perceive temporal continuity and avoid simulation sickness. Another example is that of a scientist viewing a simulation of a highly varying phenomenon, with the restriction that he always wants to keep a constant frame update rate. A possible non-constant update rate may be interpreted as a temporal behavior of the data itself, instead of an artifact of the rendering system. By letting each system know about temporal relations with the rest of its world, we are allowing it to control its temporal behavior so that interactivity is sustained.

Graphics systems to date have not dealt with time in such a way, because the whole structure of the system has to be re-thought to include time; each section of the system has to have time incorporated to its behavior. Usually the temporal aspect of the system is not given that much thought because of its complexity and high level of interaction with the rest of the system, and is ignored from being a basic requirement.

What is needed is a rendering system that can also supply to the users a temporal interface to the rendering process. This is a very important issue, since the average programmer who wants to make his program time-conscious, does not need to deal with all the trouble of knowing the temporal behavior of the graphics system he is working on, and then incorporating it to his own application.

In this thesis we are trying to lay out the major requirements behind such a time-critical renderer, and the development, implementation and results of a time model of the graphics system that is used by the renderer to provide us with predictable screen update rates. This work will provide the necessary tools for the creation of the time-critical renderer himself, which is not going to be a subject of this paper.

1.4. Roadmap of the Thesis

The remainder of this thesis will follow the following format: In Chapter 2 we develop an abstract model of the graphics system. Then, in Chapter 3 we briefly discuss the requirements for a time-critical renderer and elaborate on the rendering time model, its creation and its utilization. In Chapter 4 we describe the tests that were needed for the model's creation and in Chapter 5 we discuss the results obtained.

2. Overall structure of the system

In order to state the requirements behind a time-critical renderer and explain the rendering time model, we first have to describe an abstraction of the graphics system. This will help the creation of a framework for further discussion.

2.1. Formal Definitions

In order to provide an abstraction of the time-critical rendering problem, we will view the rendering process as a mapping. Traditionally, the application completely specifies how some piece of geometry is being rendered. This process can be thought of as a one-to-one mapping of the user's specification of what is to be rendered, to the visual result actually being displayed:

$$R_{classic}: \ \mathbf{S} \rightarrow \mathbf{V} \tag{2.1}$$

where

$R_{classic}$ is the traditional (classic) rendering process,

$\mathbf{S}$ is the user's specification of what is to be rendered (the scene description), and

$\mathbf{V}$ is the visual result

Note that there is no explicit feedback to the user of the renderer of the actual time that the rendering process took.

In solving the time-critical rendering problem, we would like to have the flexibility to control $\mathbf{t_a}$, the actual time that the rendering process took, so that it matches a given requested time $\mathbf{t_r}$ for the specified specification $\mathbf{S}$. In many cases, by just adding the restriction that $\mathbf{t_r}$ has to be satisfied, overconstrains the problem; the rendering system can only satisfy $\mathbf{t_r}$ if $\mathbf{t_a} < \mathbf{t_r}$; in that case, it can simply sleep for the remaining $\mathbf{t_r}$-$\mathbf{t_a}$. We want to create a set of different geometric representations for each object specified, that render in different amounts of time. Then, the renderer controls **V**, the visual result, by specifying alternate representations, so that $\mathbf{t_a}$ can vary. The rendering process can then be treated as a one-to-many mapping of the specification and time required, to the visual result and actual time:

$$R_{TC} : \ (\mathbf{S}, \mathbf{t_r}) \rightarrow (\mathbf{V}, \mathbf{t_a}) \tag{2.2}$$

where

$\mathbf{R_{TC}}$ is the visualization process of the time-critical renderer.

$\mathbf{t_r}$ is the requested time to render **S**.

The mapping process $\mathbf{R_{TC}}$ will, from now on, be called the *time-critical renderer* (TCR).

In traditional graphics systems, a specification maps one-to-one to a visual result, as stated above; there is no way of loosely interpreting what the user meant. In our case, to obtain the necessary flexible mapping, we need a specification **S**, that is ill-defined; we call this the "high-level" or "abstract" description of the renderable information. It is the responsibility of the TCR to choose an appropriate geometric interpretation **G** of **S** such that $\mathbf{t_a}$ approaches $\mathbf{t_r}$. For example, the user may specify that he wants a "chair" at a certain position in space, with a certain color and other characteristics, without specifying an exact description of the geometry of the chair. The TCR will then decide, depending on how much time is allowed for the chair to render itself, and other factors, the exact geometric description that fits the constraints.

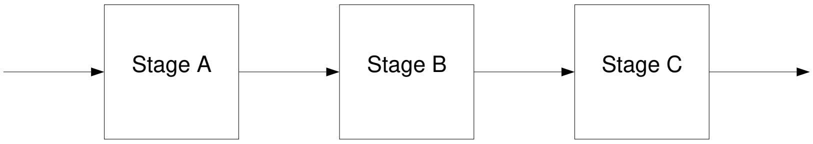

Figure 2.1 A sample pipeline containing several stages.

It is important to emphasize that it is highly unlikely to be able to create a TCR that can satisfy $\mathtt{t_r} = \mathtt{t_a}$ exactly. In order to be able to accurately predict the time that a specified piece of geometry will take to render, the renderer must have a time model for itself. Today's systems (both software and hardware) are deficient in this respect, because of their high level of complexity. The evaluation of a fairly complete time model of a such a complex system could take a big fraction of the execution time of the actual process, and would therefore be of no use.

For example, let's examine some current rendering engines: The SGI engines [Sil92c] are mostly implemented in hardware. They contain several pipelined stages, each one of which contains processing units operating in parallel. Each of the pipeline stages are connected using queues. The temporal behavior of each one of the pipeline stages is very well defined in terms of their inputs, but the behavior of the system as a whole is poorly defined. As shown in the example pipeline of Figure 2.1, the behavior of stage C, is defined as a function of the output of stage B, but not as a function of the inputs at the beginning of the pipeline. We can't really "estimate" with precision the inputs to stage C because that's the reason we have stages A and B there in the first place. The best we can do is provide a statistical estimate of the throughput and latency.

In the case of Sun's GT accelerator, the situation is similar, except that some of the early stages in the pipeline are implemented using general purpose processors (Intel i860's to be exact). Fully programmable processors allow for more complex algorithms. This

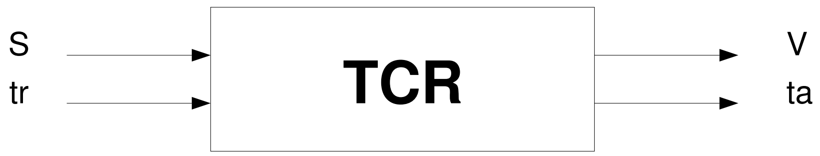

Figure 2.2 An open-loop control system. There is no feedback from t_a to t_r.

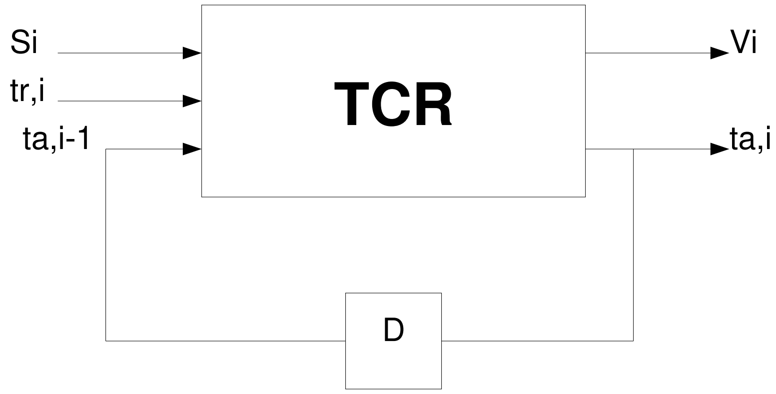

Figure 2.3 A closed-loop control system.

causes the temporal behavior of these stages of the pipeline to be much less clearly defined (because of the ill-defined temporal behavior of average-case complex algorithms), which adds an additional level of complexity creating a time model for this architecture.

There are additional parameters that affect $\mathbf{t_a}$, such as system load, the state of the system, and of course the design of the computer system itself (processor type and speed, speed and type of communication with I/O devices, etc.)

By making certain assumptions and approximations about the behavior of the pipeline stages, though, we can create a rough model of the time behavior of the system.

The above description implicitly assumes the use of an open-loop control system (as in Figure 2.2). We can also use a closed-loop control system (see Figure 2.3), represented by the mapping

$$R_{TC}: \left(\mathbf{S_i}, \mathbf{t_{r,i}}, \mathbf{t_{a,i-1}}\right) \rightarrow \left(\mathbf{V_i}, \mathbf{t_{a,i}}\right) \tag{2.3}$$

where i signifies the iteration or frame number.

The delay $\mathbf{D}$ of the feedback is very closely related to the frame time. The effect of $\mathbf{D}$ could be variable if the frame time is not kept fairly constant. Here we will assume that $\mathbf{D}$ is kept constant across frames, since we are hoping that we will be able to keep up with the requested frame rate.

2.2. Coherency assumptions

From Equation 2.3 and Figure 2.3, we can observe that $\mathbf{S_i}$ and $\mathbf{t_{r,i}}$ are inputs to the system, while $\mathbf{V_i}$ and $\mathbf{t_{a,i}}$ are outputs. There are certain assumptions that can be made about both inputs and outputs that help in simplification. These assumptions are related to "coherency" properties; the fact that the parameter in question does not have a big variation with respect to time.

In terms of inputs, we can define:

- *Model coherency, C_m*: $|\mathrm{S}_\mathbf{i} - \mathrm{S}_\mathbf{i-1}| \leq \varepsilon$. Temporally adjacent model specifications differ only a small amount in terms of data they contain. As discussed above, $\mathbf{S}$ is the input description of the model.
- *Model-Time coherency, C_{mt}*: $\left|\mathbf{t}_{\mathbf{r},\mathbf{i}} - \mathbf{t}_{\mathbf{r},\mathbf{i-1}}\right| \leq \varepsilon$. Adjacent frames are supposed to be rendered in about the same amount of time. Note that $\mathbf{t_r}$ is the time that is left for rendering (per frame) after the simulation that is required for the given frame is performed.
- *Load coherency, C_l*: $|\mathbf{l_i} - \mathbf{l_{i-1}}| \leq \varepsilon$. The average system load over a frame is almost the same across adjacent frames.

where ε is an arbitrary small constant, not necessarily the same in all definitions.

In terms of outputs, we can now define:

- *Real-Time coherency, C_{rt}*: $\left|\mathbf{t_{a,i}} - \mathbf{t_{a,i-1}}\right| \leq \varepsilon$. Adjacent frames are actually rendered in about the same amount of time.
- *Perceptual coherency, C_p*: $|\mathbf{P_i} - \mathbf{P_{i-1}}| \leq \varepsilon$. Adjacent frames produce about the same perceptual information to the user.
- *Rendering (geometry) coherency, C_g*: $|\mathbf{G_i} - \mathbf{G_{i-1}}| \leq \varepsilon$. Adjacent frames contain about the same renderable geometric information to be processed. Notice that geometry is actually an "intermediate" product, since it is not directly accessible to the user.

As an example, if C_m is satisfied, then we would like C_p to be satisfied (model did not change, therefore the user's perception of the model did not change). This could probably be done by keeping **G** in sync with **S**. Note that this is only possible if C_l, as well as C_{mt} are satisfied.

In order to provide a framework under which the time-critical rendering problem is to be solved, we have to understand the components of the problem in detail. Let's examine the input specifications first in detail:

C_m depends on what data the user wants to visualize. In most CAD applications, relatively static walkthroughs and virtual environment applications, where the user usually either orbits the data in order to get a better understanding of its three-dimensional nature, or gradually changes from one environment to another, C_m will be satisfied most of the time. In scientific visualization applications and relatively dynamic virtual environment applications, on the other hand, where there is a good chance that the whole data set changes from one frame to the next, C_m will probably be not satisfied.

C_{mt} means that:

1. The user specified a constant frame time, which is usually the case.
2. The simulation takes about the same time in adjacent frames.

The second assumption is arguable, since we can't guarantee that the application simulation code will fit the time constraints. However, that's the best we can do now, since we can't enforce time constraints to the application at this time. SGI Performer [Sil92d] is also making the same assumption.

C_l is extremely difficult to predict and is also discontinuous. Things affecting system load include but are not limited to:

- System and user background processes.
- Peculiarities of the window system.
- Interaction of the main processor with the graphics and other input/output subsystems.
- Unpredictable interaction of the user with the system.

Here, we are taking the load $\mathbf{l}$ to mean the average system load over the period in which rendering for one frame is to be performed. It is a reasonable assumption that C_l is not generally satisfied; however, there are time periods though, that the load is fairly constant. We should be able to take advantage of these periods where C_l is satisfied, and produce fairly constant frame rates. It is also conceivable that the renderer's perception of $\mathbf{l}$ may be used as an input to schedule process priority at the operating system level. This is an area where real-time operating systems can be extremely useful [HKNW][Sil91].

From the above descriptions of the concurrency definitions, we can make the following assumptions in order to make the problem more tractable:

1. C_m is satisfied, in other words, the model stays relatively constant across frames. This assumption will help us define a way to predict the rendering time for a slowly changing model (for example, a robot moving in the screen), that we can then extend to a model with high variation when C_m is not satisfied.
2. C_l is satisfied. The frame time is assumed to be small enough that the load can be treated as constant between adjacent frames.
3. C_{mt} is not satisfied. This is done because of the reasons stated above. If C_{mt} is satisfied we have a trivial problem; we are basically rendering a static model.

The above assumptions do not allow us to work with data that is continuously changing by large amounts from frame to frame. For example, in scientific visualization applications, or in building walkthroughs in situations where you enter a new room and suddenly many new objects come into view.

As an example to further understand C_{mt}, let's consider the following: We are given a certain amount of time to render a scene description, and we fully utilized computing resources in order to obtain a visually correct result. Now, suppose that the application's simulation process took more time than usual for a given frame or the system load changed, and that we must meet the time limit we are given. We, as the renderer, then, have to alter the produced geometry, $\mathbf{G}$, in order to meet the time criterion. Note that in this case, C_m was satisfied, but C_{mt} or C_l were not. Since we altered the geometry displayed in the screen to meet the time criterion, we observe that we satisfied C_{rt}, but probably violated C_p. This may potentially produce a visually displeasing result, since the violation of C_p did not come from a violation in C_m. The way the geometric description of a model should be altered, then, under time constraints, to handle graceful visual change, is a very complex geometric, perceptual and psychological problem. This is a basic task of the TCR itself, and is not addressed in this thesis.

Note that **G** is a hidden implementation and C_g is really a hidden specifications, even though the geometry is what actually gets drawn on the screen. In most cases the user does not really care about the implementation details of what gets rendered (the geometry itself). All he cares is that what he sees is pretty “close” to what he asked. This difference between what should be drawn and what actually gets drawn may be interpreted by the user’s visual and perceptual systems as something inherent in the data, while it was merely an artifact of the rendering system. In the reverse case, a jump should have been perceived by the user, but the system could not accurately convey it. Jaggies or aliasing is an equivalent spatial effect in current graphics environments.

In the same way that spatial aliasing or discontinuities may be irritating to the user, temporal discontinuities could have the same effect. Consider the case of time itself being the data that needs to perceived by the user: If a scientist is interactively viewing a simulation of a highly varying phenomenon, he may interpret a non-constant update rate as the temporal behavior of the data itself, while it was merely an artifact of the rendering process! As it has already been stated, it is highly unlikely that C_m will be satisfied in those situations. In addition, C_p **has** to be violated, then, in order to produce the discontinuities inherent in the model. Since we are not dealing with these cases right now, we should just keep the above example in mind, because a time-critical rendering system will have to deal with them in the future.

2.3. Constraint Specification and the TCR

The above discussion tells us that there is a need for a specification language that notifies the TCR of the aspects of the rendering process that are important to the user. Some of the questions that the TCR will probably need to ask are:

1. How important are certain primitives with respect to others?
2. If I can't render everything in the alloted time, should I sacrifice primitives (not draw them at all), or extend the frame time?
3. If the frame rate is to be extended, by how much?
4. How can I alter the displayed geometry in the current frame, if I know the behavior of the input model in the near future?

Therefore, the TCR is trying to find the best (optimal) visual representation of the input model, while being restricted by constraints such as the ones stated above. We are making the assumption that the human is not a hard real-time system (where any violation of a constraint is considered a failure), but a "soft" one; ie. a small violation of the frame-time is tolerated most of the time, provided that it is not commonplace. There are mainly two reasons why this is a valid assumption:

1. The human perceptual system is flexible and can usually "forgive" small inconsistencies, provided they don't happen often.
2. Screen updates are done at a multiple of the frame time (usually 1/60 sec). If we violate a frame deadline we can then:
    a. Display what we have rendered so far. This assumes that less important geometry is rendered last.
    b. Extend our allotted time by one frame time. Because of case (1), this may not have a displeasing result if it is not done often.

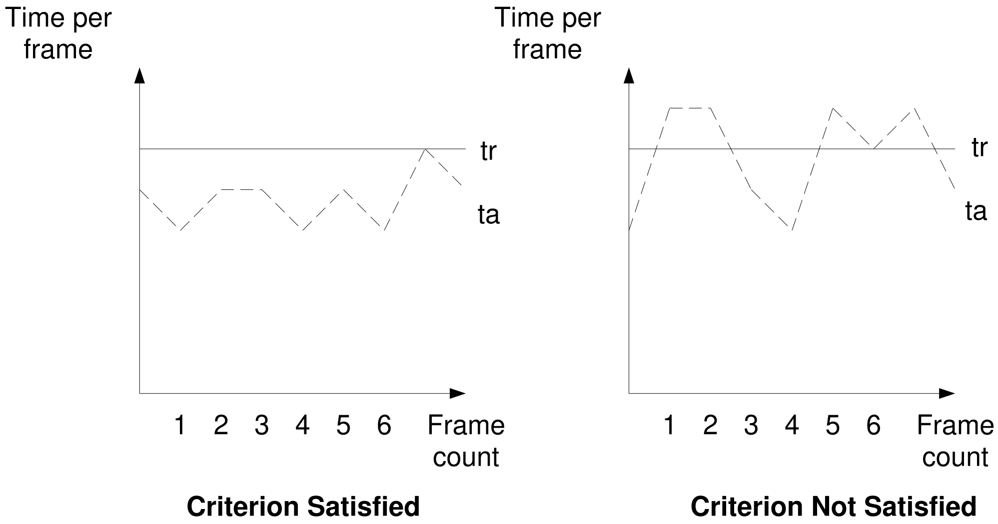

Figure 2.4 Example showing compatibility between $\mathbf{t_r}$ and $\mathbf{t_a}$

We can then restate the time-critical rendering problem as a constrained optimization problem with soft constraints. As stated in the introduction, the internals of the TCR and the specification of constraints are not going to be described in this paper.

2.4. Introduction to the Rendering Time Model

As we have seen, the input model description is mostly time-varying and discontinuous. The system load is also time varying. Our objective is to control $\mathbf{t_a}$ so that it matches $\mathbf{t_r}$, under all these time-varying conditions. As an example, the compatibility criterion may be that we should never exceed a given frame time, or $\mathbf{t_a} < \mathbf{t_r}$, as shown in Figure 2.4

Even with a closed-loop control system, it is not obvious how to create a specification-to-geometry mapping to track $\mathbf{t_r}$. As an attempt to solve this problem, we introduce the Rendering Time Model (RTM). The RTM produces $\mathbf{t_p}$ (predicted time), which is an open-loop estimate or $\mathbf{t_a}$ given the input model. The TCR will use the RTM to produce an initial estimate for the rendering time, as shown Figure 2.5. This estimate together with any additional information that the TCR may have (e.g. constraints, estimates and

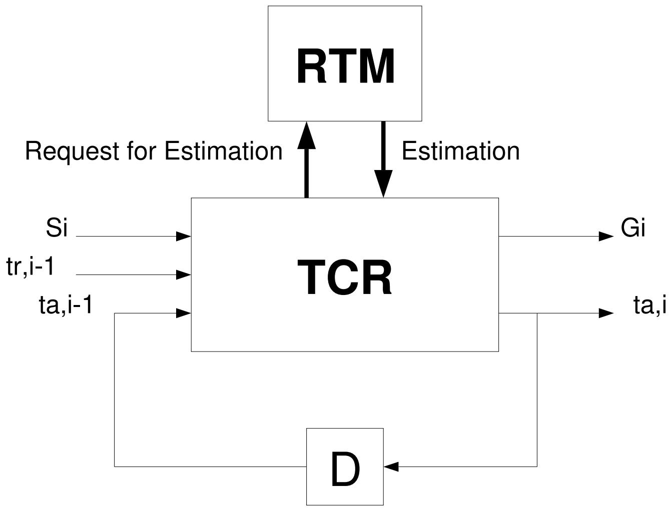

Figure 2.5 The high-level structure of the TCR and the RTM

feedback from previous frames, etc.) will produce a geometric representation of the scene that has a better chance of being rendered at the requested amount of time, than just the open-loop or closed-loop model.

In Chapter 3, we will describe the considerations behind the creation and operation of the RTM.

3. The Rendering Time Model (RTM)

This chapter treats, in detail, the RTM and it's relationship with the rest of the graphics system. First we will briefly discuss the graphics environments in which the RTM and the TCR can be implemented in, and then will describe how the RTM is implemented in such environments and explain in detail all the experiments that were needed for its creation.

3.1. Graphics Environment Requirements

Our implementation of the RTM and the TCR can be built around any object-oriented graphics system that fits the requirements that are outlined in this section.

There is a class called *RenderingDevice* that communicates in a system-dependent way to the underlying immediate mode procedural graphics library (like GL[Sil92a] from SGI or XGL[Sun91] from Sun). Low-level *Primitive*s, like *Polyline*s, *Polygon*s or *QuadMesh*es (which are the basic implementation of all parametric surfaces with a domain of dimension 2 and a range of dimension 3), communicate with the *RenderingDevice* to render themselves. Higher level *Primitive*s are able to express themselves in terms of other *Primitive*s (either low-level or high-level), and they render themselves in the screen in that way. As it can be seen from Figure 3.1, each primitive understands a method called **draw()**. When a low-level primitive gets this method, it notifies the interactive device that it needs to be drawn. When a high-level primitive gets this method, it delegates it to its implementation, and the implementation then performs the decomposition and rendering appropriately. In that way, the platform and graphics library dependent details of how a primitive gets drawn are hidden in the *RenderingDevice*, which makes portability of the system across platforms easier for the designers of the renderer.

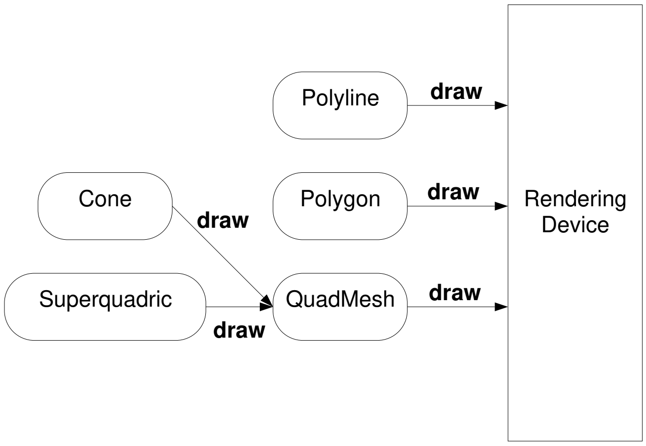

Figure 3.1 How primitives draw themselves

From the same figure, we can see that there is a one-to-one correspondence between the low-level primitives and the rendering device; there is a slot in the device for each low-level primitive. Also, observe that since higher-level primitives are rendered in terms of low-level primitives, the temporal behavior of the high-level primitives (like *Cone* or *Superquadric*) mostly depends on the behavior of their low-level implementations (*QuadMesh* in this example), provided that either the decomposition time is negligible or that decomposition is done as a pre-processing step.

Another *Primitive* that is utilized during rendering is a *Container*. A container's purpose is to delegate the **draw** method to all of its constituent objects in a way that is characterized by that type of container. For example, a *Bag* may contain a *Transformation*, a set of *Attribute*s and a list of *Primitives*. Its job is to update the current transformation matrix and attribute state of the *RenderingDevice*, and then delegate the **draw** method to all of its *Primitive*s.

3.2. Description of the RTM operation

In this section, we will describe how the interface for the RTM looks, and how the

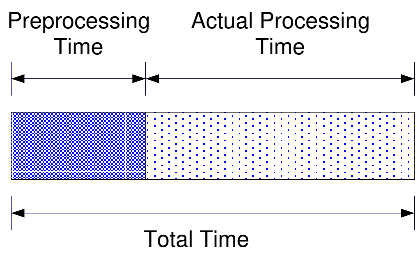

Figure 3.2 Total rendering time subdivision of high-level primitives into preprocessing time and actual processing time.

TCR can use this information while rendering a frame.

3.2.1. Definitions

First, it is necessary to make several definitions.

Traversal is defined to be the process by which a *Primitive* (including a *Container*), is inquired for its data for estimation or rendering purposes.

Estimation can be performed for either spatial or temporal purposes:

- *Spatial* estimation is required when we want to produce an approximation of some spatial feature of the traversed entity, ie. obtain the bounding volume extents.
- *Temporal* estimation is required whenever we want to produce an approximation of the rendering time of the traversed entity.

Rendering is always performed in order to create an approximation of the visual representation of the traversed object.

For rendering, we divide the total time required for rendering of high-level primitives into

1. *Preprocessing* time, which is the time that it takes for a high-level primitive to convert itself into low-level primitives
2. *Rendering* time, which is the time that the low-level primitive takes to render itself.

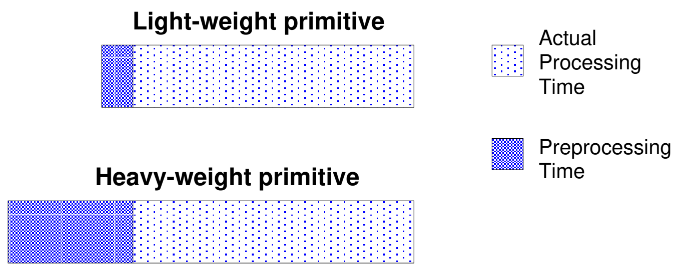

Figure 3.3 Light-weight vs. heavy-weight high-level primitives.

A pictorial representation can be seen in Figure 3.2.

We also divide all high-level primitives into two categories:

1. *Light-weight*, that are easily expressible in terms of low-level primitives, and require little or no extra preprocessing during rendering.
2. *Heavy-weight*, that require significant additional processing during spatial rendering. NURBS that adjust tessellation based on the view are a perfect example [Abi89].

Figure 3.3 shows the difference between light-weight and heavy-weight primitives.

All of the preprocessing time is spent in the main processor(s), while the actual processing time can be spent in either the main processor or attached graphics subsystems. The theory of the RTM that is described here deals with both light-weight and heavy-weight primitives. The current implementation, though, does not deal with preprocessing separately, and is therefore only operational with light-weight primitives. The reason is the following: Any user of the system can write his/her own high-level primitive and attach it to the system. In order to provide accurate temporal rendering of a new primitive, that primitive must be able to provide an estimation of its rendering time. If this primitive is a heavy-weight primitive, then the user must specify this estimate. At this time, it is not clear how the user can be forced to provide an accurate time estimation. That limits our implementation to only light-weight high-level primitives.

3.2.2. Structure of an RTM inquiry

The RTM is inquired during the estimation traversal of a primitive. In order to explain the RTM inquiry, we first have to look how the TCR will use that information.

The TCR contains, internally, a list of primitives that it has to render or perform a temporal estimation of, for each frame. These two sets may not be the same. The rendering list contains the primitives that are about to be drawn to the screen in the current frame. The temporal estimation list contains the primitives that we want to inquire an estimate for their rendering time for the current frame.

A reason why there are elements in the spatial list that are not in the temporal list may be that the TCR may make the assumption that the time estimate for a certain primitive did not change, so there is no reason to inquire the RTM for that primitive for the current frame. In the reverse case (elements in the temporal list do not exist in the spatial list), it may be that the TCR wants to get an idea of the rendering time of primitives that are not visible yet, but will be pretty soon.

The TCR will, usually, first estimate the time for the primitives in its temporal estimation list (to get an estimate of the time), and then it will render the primitives in its rendering list.

Because all of our high-level primitive are light-weight, almost all of the time spent during a rendering occurs at the low-level primitive level (refer to Figure 3.1). Since all of the device dependencies are kept inside *RenderingDevice*, and since the temporal behavior of a low-level primitive is implementation dependent, it is perfectly reasonable to place the RTM inside *RenderingDevice*. This makes the design of the system very

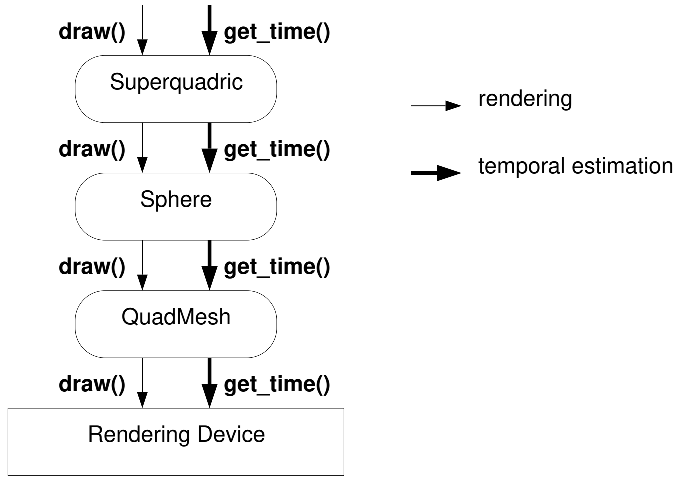

Figure 3.4 Rendering and temporal estimation to an rendering device.

clean, since there is no visual difference between a rendering and a temporal estimation of any *Primitive*, as it can be seen in Figure 3.4. A *Superquadric*, under special cases, creates an internal *Sphere* for efficiency. The sphere, without any knowledge to the superquadric, contains an internal *QuadMesh* for rendering purposes. When someone asks for either a temporal estimation or rendering of a superquadric, it delegates the request to the internal sphere, which, in turn, delegates to the quadmesh, which in turn sends the appropriate request to the device.

In all light-weight primitives, the device performs all of the time estimation using the RTM. When heavy-weight primitives are to be rendered, the main structure of the system stays the same, except that the intermediate primitives operate on the time that they got from their delegated primitive, to add the estimate for themselves, as shown in Figure 3.5. In this example, a NURBS in delegating the **get_time()** method to its quadmesh. The quadmesh returns the estimation that the device returned to it, but the NURBS has to update that information with its internal estimation of its rendering time. There are two things that we should note though:

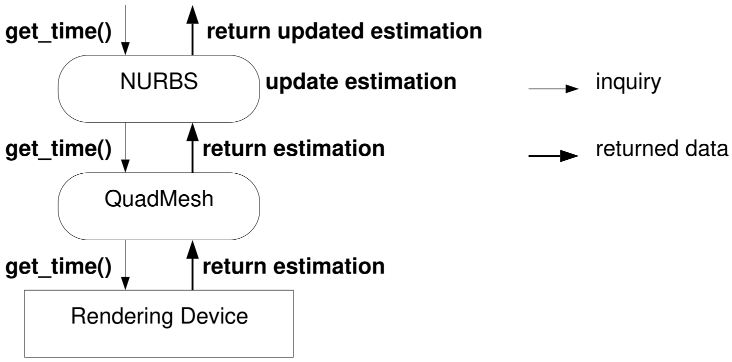

Figure 3.5 Temporal estimation for heavy-weight primitives.

1. The additional time that the NURBS incorporated into the time estimation is part of the pre-processing time. As it was discussed above, all preprocessing is performed in the main processor(s), which means that the time estimation will differ from machine to machine. This implies that an database should be used to store the temporal behavior of the high-level primitives for different architectures, on a per-class basis. The homogeneity of rendering that existed in light-weight primitives is destroyed, though.
2. In the future, a NURBS primitive could be incorporated to the Device. In this case, that preprocessing time will have to migrate as actual processing time to the Device. This could be tricky.

These are mainly the reasons why heavy-weight primitives are not discussed in this paper.

3.3. Formation of the RTM

In the previous section, it was discussed how an inquiry to the RTM is handled. Now we will describe the contents of the inquiry itself, and how the *RenderingDevice* is performing the actual estimation.

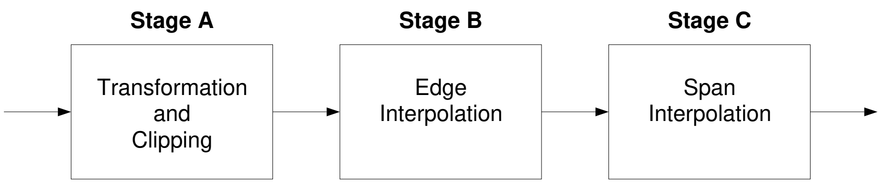

Figure 3.6 The graphics pipeline.

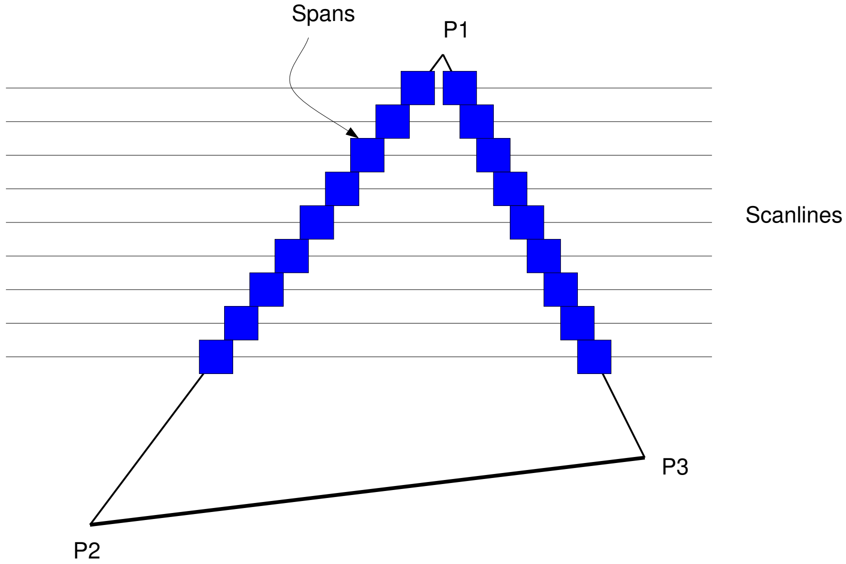

Figure 3.7 Edge interpolation

3.3.1. Time subdivision

It has been observed ([Dee92][Fis92][Pfl92][Sil92b]) that the rendering time of hardware graphics accelerators can be subdivided into several parts. To understand this subdivision, we have to first understand the internal structure of the engines themselves. Almost all of the hardware graphics engines to date operate mostly in a pipelined manner, as shown in Figure 3.6. Stage A of the pipeline ("Transformation and Clipping") is the most floating-point intensive part; it converts polygon descriptions in floating-point 3–D coordinates to integer 2–D pixel coordinates together with Z values. The output of stage A is usually in the form of triangles or quads. For illustration purposes, let's assume that stage A produces just triangles. Stage B takes in triangle vertex coordinates and produces the endpoints of the spans, as shown in Figure 3.7. This is basically a scan-conversion

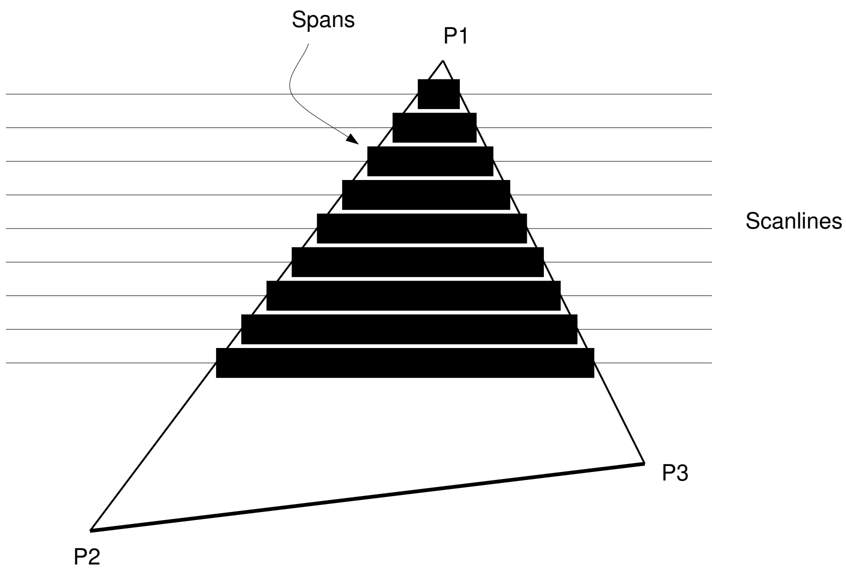

Figure 3.8 Span Interpolation

of the edges of the input triangle. The output of the stage B is, then, a collection of spans for the triangle. Stage C, finally, calculates (horizontally in this case) all the pixels in the spans that is got from stage B, as shown in Figure 3.8. This produces all the pixels to be drawn for the input triangle. Usually, the time per pixel for edge interpolation is bigger than the time for pixel interpolation. It must be noted that this type of interpolation also occurs for the R, G and B color components, if gouraud shading is enabled, and for the z-values if z-buffer style hidden surface elimination is performed.

From the above discussion we can see that we associate different times for each different section in the graphics pipeline. We can, then, call *modeling time* the time associated with the modeling (transformation and clipping) of the primitive, *edge interpolation* time the time associated with the interpolation of the edges into spans, and *span interpolation* time the time associated with the calculation of the pixels in the spans themselves.

For the purposes of the RTM, we let t_m to be *modeling time*, and t_p (pixelation time) to be the combined *edge interpolation* and *span interpolation* time. The reason why edge and span interpolation times are bundled into one number is that

- For the purposes of the RTM, it is pretty difficult to differentiate the two.
- In some graphics architectures, these two operations are not clearly separated into distinct pipeline stages, so differentiating them would be wrong.

We call the combined pixelation and modeling time, t_t (total time).

Different graphics architectures are built differently. For non-pipelined architectures, for example, $t_t = t_p + t_m$, which means that there is no overlap in the execution of the modeling and pixelation (the two operations are performed sequentially). For pipelined architectures, though, we have that $t_t = \max(t_p, t_m)$ [HB84] if we assume that transport delay is insignificant. This means that there is a different "combination" function that produces t_t, depending on the architecture, which means that apart from the constituent times t_m and t_p, we also have to keep track of the combination function that produces t_t from these two values.

When the RTM is inquired to produce an estimate of the rendering time of a primitive, it does not return an estimation of the total time t_t, but of the constituent times t_m and t_p. The TCR, depending on the architecture, may need the separate times and the combination function, to estimate what is the best representation of a primitive for the current architecture.

3.3.2. Creating the RTM

In this section, we will describe the several tests that are performed in order to produce the RTM. The primitives that were tested for the RTM are:

1. *Polyline*: A piecewise linear curve, or a set of line segments connected beginning-to-end.
2. *MultiPolygon*: A collection of disjoint polygons with no common vertices and same number of vertices for each constituent polygon. Here, each polygon is a triangle.
3. *TriangleStrip*: The set of triangles such that for each $\mathbf{i}$, $\mathbf{V_i}$, $\mathbf{V_{i-1}}$ and $\mathbf{V_{i-2}}$ form a triangle, given $\mathbf{V}$ to be a list of vertices.
4. *QuadMesh*: A set of 4 vertex polygons (quads) organized in a 2–D mesh.

All of the above primitives were chosen because they are either implemented in the underlying immediate mode library, or they can be efficiently and easily implemented.

For each type of primitive we will perform tests that accumulate results for both t_m and t_p. We expect that t_p will be the same for all primitives of the same "dimensionality", ie. all area defining primitives (*Multipolygon*s, *TriangleStrip*s and *QuadMesh*es) will have the same t_p, and all length defining primitives (*Polylines* or edges of surface primitives with edges turned on) will have the same t_p. We also expect that t_m may vary from primitive type to primitive type, since different algorithms may have to be used for different primitive types.

3.3.3. Evaluation of t_p

In this section we will describe the tests designed for the evaluation of t_p. Attributes to be tested were chosen based on our initial assumptions of how t_p is affected, and then by refining those assumptions. These assumptions are:

- The rasterizers contain edge processors and span processors. Each one of those performs a different function, and tests should be able to separate their timings.

- Span processors are oriented a certain way. There will be a situation where the span processors will have no work, while the edge processors will have a lot of work. This happens in the case of “sliver” triangles: Triangles with very small area, but large perimeter. Tests should show when this happens.

The list of parameters tested is definitely not exhaustive. The tests shown below are designed so that they produce results based on the above assumptions:

1. Time per pixel, no color interpolation between vertices (constant color per polygon).
2. Time per pixel, with color interpolation between vertices.
3. Time to calculate one span (edge walking for one step), with no color interpolation.
4. Time to calculate one span, with color interpolation.
5. Ratio between horizontal and vertical slivers, no color interpolation.
6. Ratio between horizontal and vertical slivers, with color interpolation.

The first two tests measure what is usually called the polygon “fill-rate”: The throughput of the rasterizer (span interpolator). This number is rarely given by vendors. We are trying to get the measurement by drawing a polygon where:

- The time spent in the span interpolator is small with respect to the time spent in the edge interpolator.
- The time spent transforming points is small with respect to the time drawing pixels.

This is achieved by drawing one large polygon several times. Specifically, the test was performed by drawing a large primitive (300 x 600 = 180000 pixels) 1000 times and measuring the elapsed time. The whole test was performed 10 times. The average time was then divided by 1000 (number of primitives) and then by 180000 (number of pixels per primitive) to come up with a time per pixel. The area of the primitive was chosen to be big enough so that we are definitely render-bound (see Section 3.3.5) and the effect of the rendering of multiple primitives is minimized. Additionally, the number of primitives rendered was large enough to provide a good estimate keeping in mind the (possibly) low resolution of the system timer.

The second two tests measure the throughput of the edge interpolator. This number is not given by vendors, but it is important. The reason why this is, can be demonstrated by the following two cases:

1. A certain area on the screen is taken up by a single polygon.
2. The same area is taken up by many small polygons.

The latter case is going to take longer to render, because of two reasons:

1. We reached a point where $t_m > t_p$ (if the combination type is $\max$), or t_m is a significant fraction of t_p (if combination type is add). This reason will be explained in detail later.
2. There are many more edges present in the latter case, and since edge interpolation takes a significant amount of time, the second case took longer to render.

In order to perform these two tests, lots of sliver-like triangles were drawn so that:

- The time spent in the span interpolators is small with respect to the time spent in the edge interpolators.
- Again, the time spent transforming points is small with respect to the time drawing pixels or calculating spans.

This was done by dividing a large area into many sliver triangles pointing in the orientation of the span interpolators. Specifically, the total primitive area was 500 x 600 = 300000 pixels. This area was divided among 1000 sliver triangles. A total of 100 primitives were tested and the result timed. The whole test was, again, repeated 10 times, and the average was divided by the 100 (number of primitives), then by 1000 (number of constituent triangles/primitive) and finally by the long dimension of each sliver triangle. The result is an estimation of the time for the evaluation of each span. Again the area of the primitive and the number of primitives tested were chosen to be big enough to provide an accurate timing.

Finally, the last two tests are trying to estimate which way (horizontal or vertical) are the edge interpolators oriented in the graphics architecture. Two tests are performed: One drawing slivers in the horizontal direction, and the second one drawing slivers in the vertical direction. The size of the primitives was the same as in the previous two tests. The ratio between the two times is then registered.

For the case of the *Polyline* primitive, there is no “fill-rate” that has to estimated. At this time, the only test performed for testing polyline pixel speed, is as follows: Polylines consisting of 2000 vertices and with each line segment being 580 pixels, were drawn at random orientations. Each primitive was drawn 100 times and timed. The result was averaged over 10 times and then divided by 1999 (number of line segments) and then by 580 (pixels per line segment), providing an estimate for the time per pixel.

3.3.4. Evaluation of t_m

The attributes of t_m that need to be tested, were taken mostly because of their effect on the visual representation of the object on the screen and knowledge of the underlying algorithms. The assumptions made where that t_m:

- Is not affected by the orientation of the object (provided that the number of vertices visible is constant).
- Is heavily affected by the number of points associated with the primitive.
- Is affected by the information attached to each vertex (color per vertex does not affect t_m, but normal per vertex does). This assumption is based on the knowledge of the underlying algorithm: Both the normal and the vertex have to be transformed; the color does not have to.
- Is affected by lighting, and lighting is performed in a homogeneous way, ie. the number of directional lights does not affect the overhead for positional lights.

Based on the above observations, tests are performed to evaluate several characteristics. The list of characteristics is, but no means, an exhaustive list of all the parameters that affects t_m. It is just a small list of parameters that can be used for our proof-of-concept implementation of the TCR. These parameters are:

1. Time per point, no lighting performed (no normals transformed, precalculated color applied on a per primitive basis).
2. Time per point, lighting performed once per primitive (flat shading).
3. Time per point, lighting performed once per vertex (gouraud shading).
4. Time per directional light, using flat shading.
5. Time per directional light, using gouraud shading.
6. Time per positional light, using flat shading.
7. Time per positional light, using gouraud shading.
8. Overhead to perform lighting, using flat shading.

Primitive	Size of Primtive
QuadMesh	100 x 100 = 10000 points
TriangleStrip	10000 points
MultiPolygon	1000 triangles = 3000 points
Polyline	10000 points

Figure 3.9 Complexity of each primitive tested.

9. Overhead to perform lighting, using gouraud shading.

It is worth noting that in the above tests it is implied that the lighting model can be applied once per vertex, but color interpolation between vertices may not occur. Color interpolation is a characteristic that affects t_p, and not t_m.

In order to make accurate measurements from the above tests, we need to minimize t_p. This is done by minimizing the area and the perimeter of the primitives. The tests for this case then draw polygons with almost no area. For example, for the case of the *QuadMesh* primitive, an array of 100 by 100 points occupied only 1 pixel in the screen.

For the *Polyline* primitive, not all of the above tests were performed. Many graphics libraries do not support lighting of polylines. Therefore, only the first test was performed: Time per point with no lighting.

The table in Figure 3.9 shows the number of points for each tested primitive of each class. Before anything is rendered, the graphics library was setup to reflect the test conditions (e.g.: flat/gouraud shading, number of positional or directional lights). Then, a cached version of the primitive was evaluated. Finally, we come to the time estimation stage: Each primitive was rendered 100 times and the result timed. This whole procedure was repeated 10 times. The average time resulted was divided by 100 (the

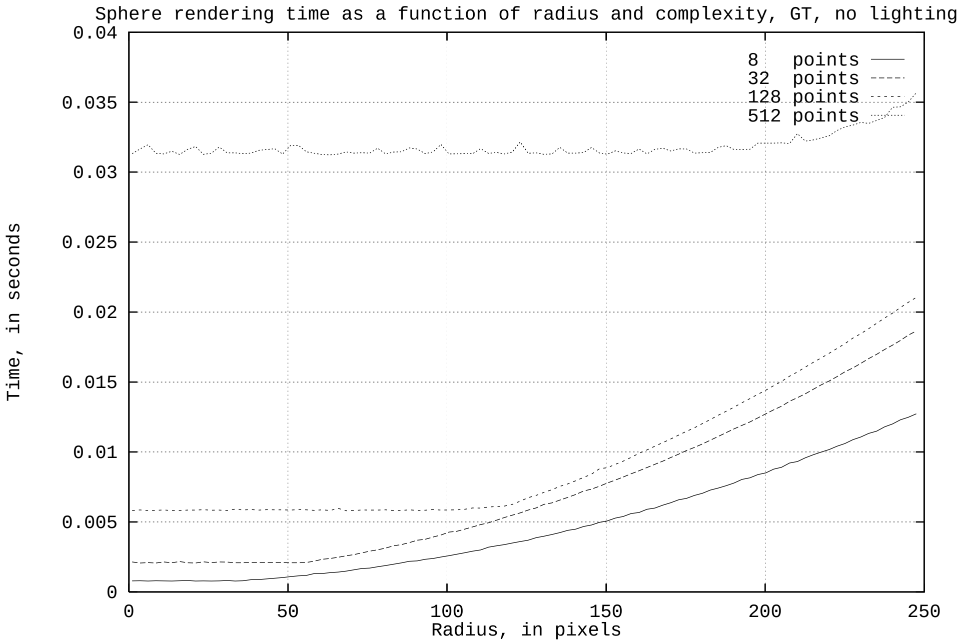

Figure 3.10 Sphere rendering time as a function of radius and complexity, GT, no lighting.

number of primitives) and then by the number of points for each primitive. The result is an estimation of the time per point for the primitive in question.

The results from these tests are going to be shown in the next chapter.

3.3.5. Combination of t_m and t_p.

As we have noticed in the previous section, it may take a different amount of time to draw the same number of pixels on the screen. One of the reasons stated, was that depending on the number of points, t_m may prove to be bigger than t_p.

A demonstration of this behavior will also make us better understand the meaning of the combination type for t_m and t_p. A test was run in two different machines: A Sun GT accelerator, which is a heavily pipelined machine, and a Sun GS accelerator, which is not pipelined too much. Further details on the processor type in each case are not

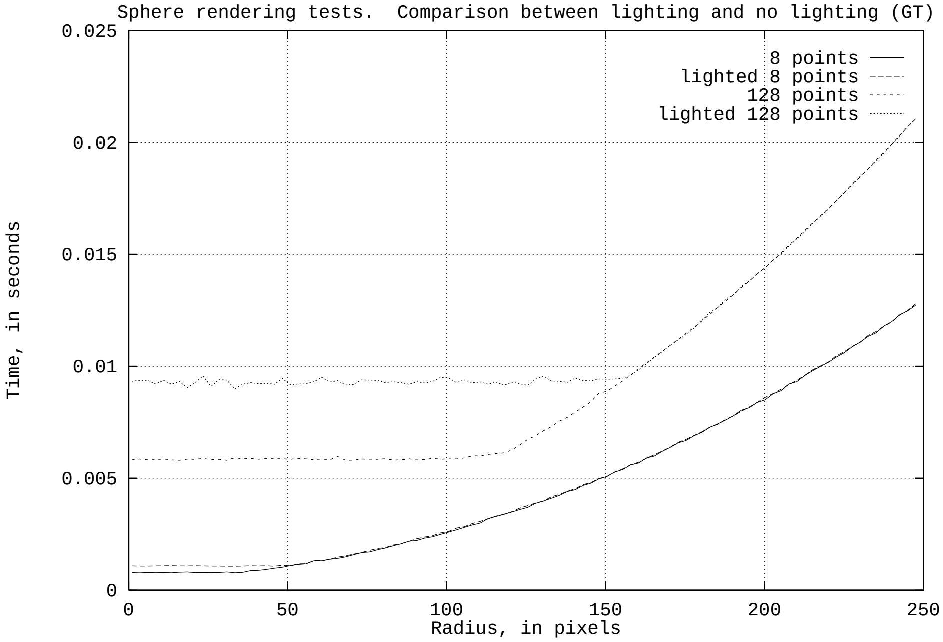

Figure 3.11 Sphere rendering tests. Comparison between lighting and no lighting on the GT

necessary for this example. Spheres of different levels of detail (level of tesselation) and different diameter were drawn and the results plotted. We first show the results on the GT accelerator, which has a $\max$ combination type. Each line in Figure 3.10 represents increasing radius (and therefore area) for a certain level of tesselation. As you can see, for each of the 4 different levels of tesselation, there is a “break” that corresponds to the point where $t_m = t_p$. To the left of the “break”, where $t_m > t_p$, total rendering time remains constant no matter what the radius of the sphere is. To the right of the break, where $t_m < t_p$, total rendering time is a function of the radius. If a primitive has $t_m > t_p$, we call that primitive to be **model-bound** or **transformation-bound**. If a primitive has $t_m < t_p$, then we call this primitive **render-bound**.

Figure 3.11 shows how the break point moves if the data to be processed per point

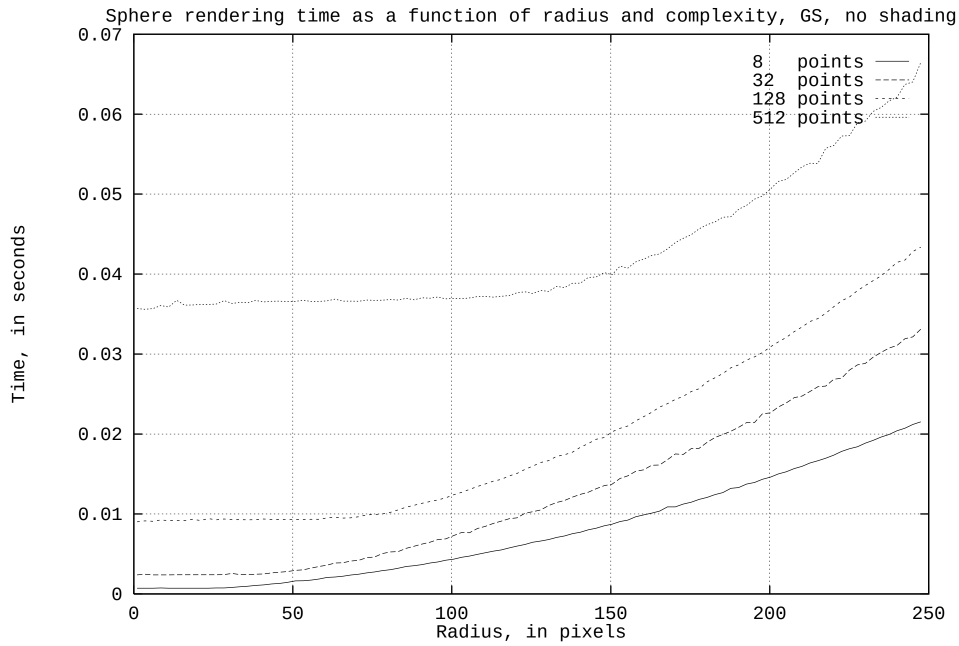

Figure 3.12 Sphere rendering time as a function of radius and complexity, GT, no lighting.

increases. The graphs that are shown in the figure can be divided into two groups of two plots each. Each group contains the same level of complexity (plotted against increasing radius), but one plots has lighting turned off, and the other one has lighting turned on. Because of the $\max$ combination type, t_m only affects the total time before the break point. After the break point, the total time is equal to t_p, and it is the same for both plots in a group.

For the case of the GS accelerator, which has an add combination type, the results, which are shown in Figure 3.12 and 3.13 are slightly different. As it can be more clearly seen in Figure 3.13, the two plots corresponding to the same level of complexity do not merge to the same render-bound curve, and the transition between **model-bound** and **render-bound** behavior is smoother, because of a blending between the two behaviors.

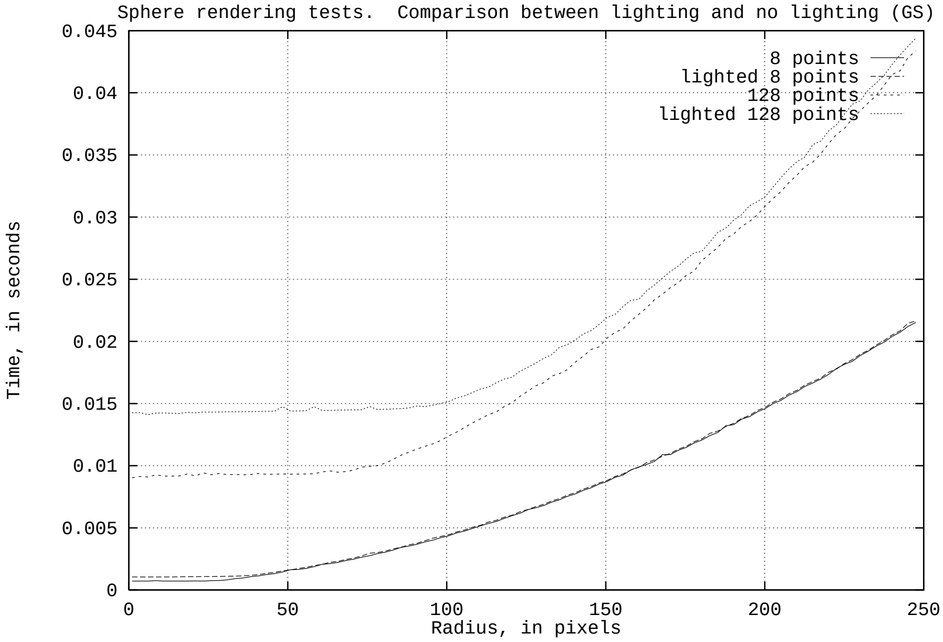

Figure 3.13 Sphere rendering tests. Comparison between lighting and no lighting on the GS.

We can still define, though, a "conceptual" break point, where, for example, $t_m = \alpha t_p$, where α is a fraction that distinguishes between mostly model-bounded vs. mostly render-bound behavior.

As it can also be seen from all of the above graphs for both architectures, different levels of complexity under the same radius restriction, produce plots that have different t_p in the render-bound region. This is the effect of edge-interpolation to t_p.

When performing tests, the combination type for each architecture is assumed to be known. We don't try to estimate it.

In the next chapter the results from the tests on the various architectures are going to be shown.

4. Results

In this chapter, we will show the results obtained by the RTM for the different types of graphics architectures together with what we can learn from them, and finally see how this can be used in a sample implementation of a TCR.

4.1. RTM Results

Currently, a trimmed-down RTM produced with the tests described in the previous chapter is operational for the following graphics architectures:

1. Sun's GS accelerator on a Sparc2.
2. Sun's GS+ accelerator on a 1–processor Sparc10.
3. Sun's GT accelerator on a Sparc2.
4. Sun's GT accelerator on a 1–processor Sparc10.
5. Evans & Sutherland Freedom ES3100 on a 1–processor Sparc10.
6. Evans & Sutherland Freedom ES3100 with one geometry processor board enabled (instead of two) on a 1–processor Sparc10. This configuration has been named ES3050.
7. SGI's Indigo Elan with an R3000 main processor.
8. SGI's Indigo2 Extreme with an R4000 main processor.
9. SGI's Reality Engine with 2 Raster Managers and 4 R3000 main processors.

Out of all the primitives tested, the favored primitive by most users is *QuadMesh*. The reason why this primitive is chosen is simple: Parametric surfaces are defined with a domain in $\mathbb{R}^2$ and a range in $\mathbb{R}^3$ ($S : \mathbb{R}^2 \rightarrow \mathbb{R}^3$). In a 3–dimensional graphics system, the range criterion is always satisfied (since we are drawing in a 3–dimensional space), and the domain criterion is best satisfied by using a *QuadMesh*.

We will only show some characteristic results in this section. The full set of results is included in the Appendices. The results shown here will be for the *TriangleStrip* primitive, because it is usually the primitive of choice for the graphics engines, and for the *Polyline* primitive, because it is the only non-area primitive in our tests.

The triangle strip has one additional feature: The number of polygons modelled is extremely close to the number of points that it contains. Our peak results in terms of points/sec transformed (with no lighting performed) for triangle strips, should, therefore, be very close to the peak rated polygon performance of the machine.

One may wonder why we show tests results for the *TriangleStrip* when the most used primitive is *QuadMesh*. This was done because the *QuadMesh* primitive is implemented using *TriangleStrip*s. Our implementation of the tests was done using two graphics libraries: XGL from Sun and GL from SGI. XGL already has a quadmesh primitive defined, but that primitive is internally split into triangle strips and then rendered [Pfl92]. GL does not have a separate quadmesh primitive, so we implement it ourselves, as a set of triangle strips. Additionally, Evans & Sutherland does not seem to have the *QuadMesh* primitive implemented in hardware at the time of the test (performance is more than an order or magnitude less than expected), so the results are deceiving for that architecture.

The tests shown below, then, are classified as triangle strip t_p, triangle strip t_m, and polyline tests.

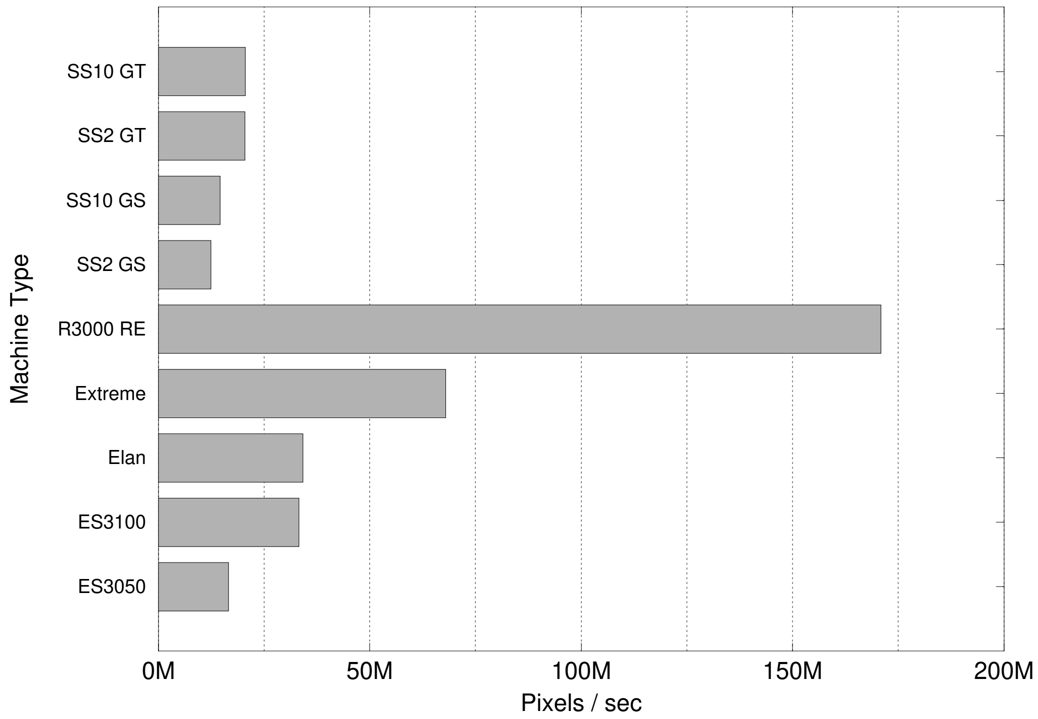

Figure 4.1 Pixel throughput across different machines.

4.1.1. Triangle Strip t_p Results

Each of the architectures tested had the same performance using color interpolated and non-color interpolated triangles. Figure 4.1 shown the results obtained. We can observe the following from the plots:

1. The GT figures using different main processors are exactly the same. This makes sense, since the main processor is not involved in scan conversion.
2. The GS figures on the Sparc10 are a bit better than on the Sparc2, because the Sparc10 accelerator is an "improved" version. So, what we stated previously in the case of the GT accelerator is not violated.
3. The ES3100 had double the performance of the ES3050, as expected.

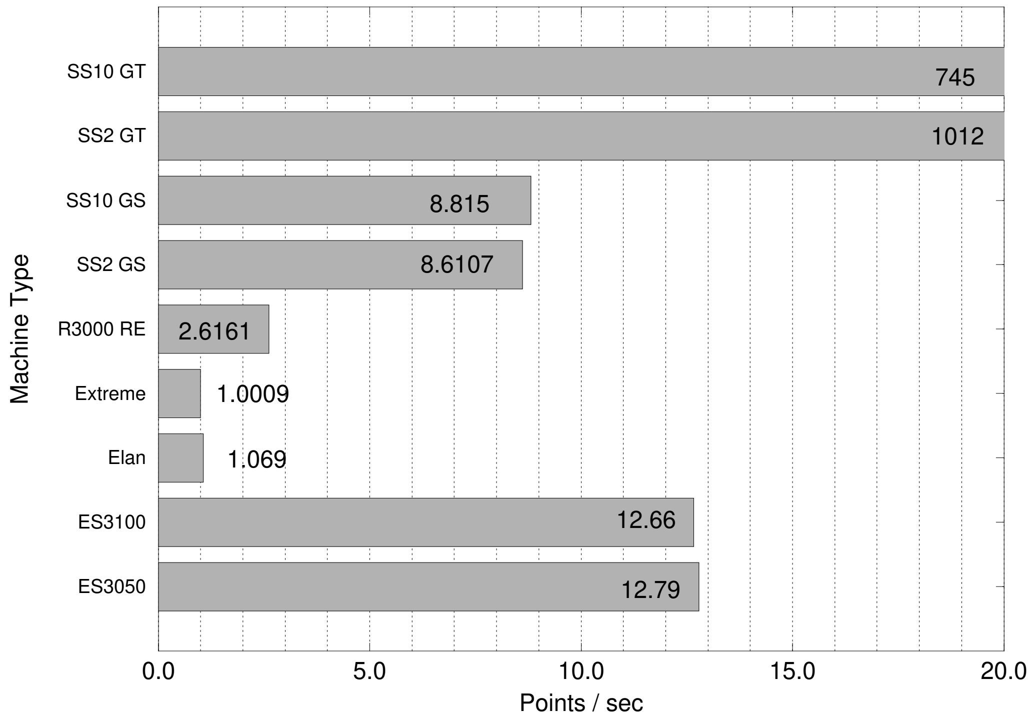

Figure 4.2 Sliver ratio among different architectures, using color interpolation between vertices.

4. The SGI machines had the highest polygon fill performance, with the Reality Engine being the fastest, by far.

Another important aspect of t_p (as discussed in Section 3.3.5) is the ratio between horizontal and vertical slivers. From the results shown in Figure 4.2, we can make the following observations:

1. Each set of architectures has similar behavior: The E&S machines are grouped together, the SGI machines are grouped together, the GS architectures are also grouped together, and the GT (being a different architecture) is by its own.
2. The GT shows an unusually high sliver ratio; a couple of order of magnitude bigger than the rest.

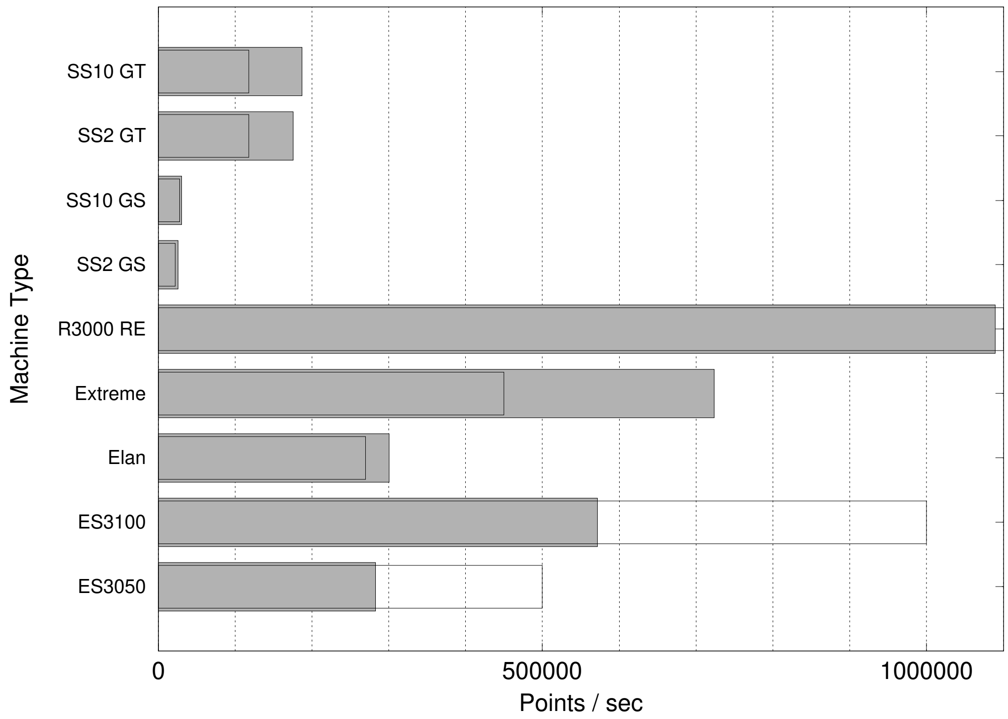

Figure 4.3 Time per point, no lighting. Grey boxes is actual performance in points transformed/sec, transparent boxes is the rated performance in terms of polygons/sec, assuming one point maps to one triangle.

3. The E&S machines, even though they have different pixel throughput (as shown before), they have very similar sliver ratios.
4. Finally, the SGI machines seem to have the best sliver ratio of all. It is interesting to note that the ratio on the Extreme (1.0009) is extremely close to 1.

4.1.2. Triangle Strip t_m Results

Time per point, no lighting (Figure 4.3). Our results for the E&S point performance was much lower than the machine rated polygon performance. In all of the other architectures, our performance was higher than the rated one.

The "higher than expected" performance does not mean that we are getting a higher performance than the one the machine is rated as. We should not forget that the rated performances are for polygons of a certain size (usually either 50 pixel or 100 pixel gouraud-shaded non-lighted meshed triangles). If the machine has a high pixel fill-rate figure, then our point performance is going to be very close to the rated polygon performance. If the machine has a relatively low pixel fill-rate figure, then our point performance is going to be higher than the rated polygon performance, because the machines were probably render-bound when rated.

This can be easily seen from the fact that in the case of the Reality Engine, we got very close (but still undershot) the rated polygon performance. This is because of the extremely high throughput of the rasterizers of the Reality Engine. All of the other architectures, because of their inferior rasterizer performance, produced performance figures that where higher than rated.

We can see from the results obtained, that the polygon performance of a graphics engine can be, in most cases, approximated by our estimation of the point performance of the machine corrected by the performance of the rasterizer.

Time per directional or positional light (Figure 4.4 and 4.5). We should note that normally, a positional light takes about three times the amount of processing than a directional light, for a triangle. The reason is that even for a triangle with the same normal on all of its vertices, the lighting calculation for a directional light need only be performed once, while it needs to be performed three times for a positional light.

Here, a smaller number means better performance. Looking at both of the graphs, we can observe that:

1. The ES3100 was, again, twice the speed of the ES3050, as expected.

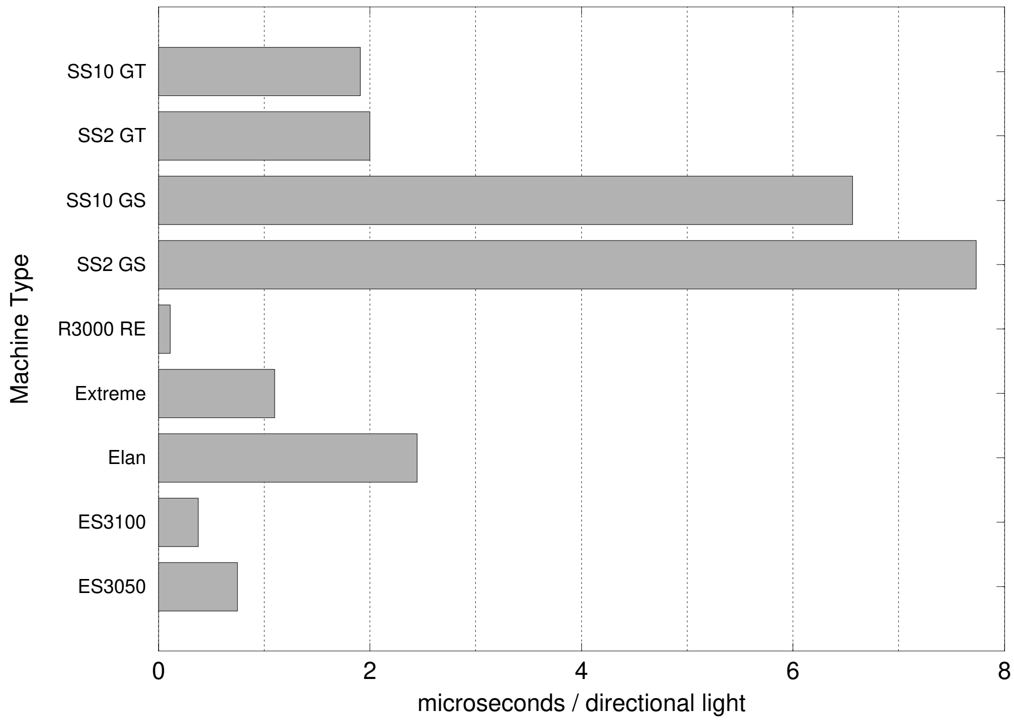

Figure 4.4 Time per directional light.

2. The Sparc 10 machines were slightly faster than the Sparc 2 machines.
3. The Reality Engine was much better than any other machine.
4. Only the GT had the triple overhead for positional lights. All the other architectures had less than three times the overhead. This implies that almost all architectures are performing some kind of parallel processing to accelerate positional lighting.

4.1.3. Polyline Results

Two sets of tests were performed for polylines. The results show performance figures in terms of points transformed per second and in terms of pixels drawn per second.

Time per pixel (Figure 4.6). Again, as expected, the E&S architecture managed to be 100% scalable, and the Reality Engine managed to, again, be the fastest. If we

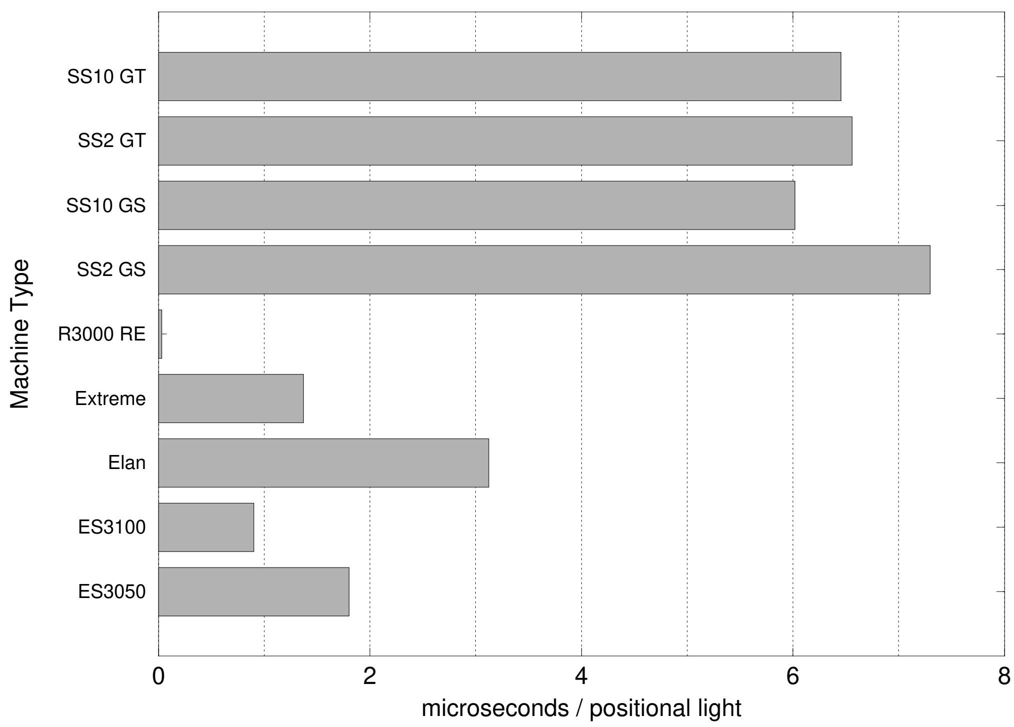

Figure 4.5 Time per positional light.

compare Figure 4.1 and Figure 4.6, we can see that the ES3100 is slower than the Elan and the Extreme drawing polygons, but it is faster drawing lines!

Time per point (Figure 4.7). Again, we observe that the E&S machines have managed to double performance between the two models, which is expected. It is surprising, though, that the Reality Engine did not perform as well as expected (it is rated at 1.2 million vectors per second). The Extreme, though, managed to be very close to its rated performance (1.2 million vectors per second).

4.2. A sample TCR implementation

A proof-of-concept implementation of a time-critical renderer has helped us to make sure that the RTMs we have obtained can actually be useful in a real-life application.

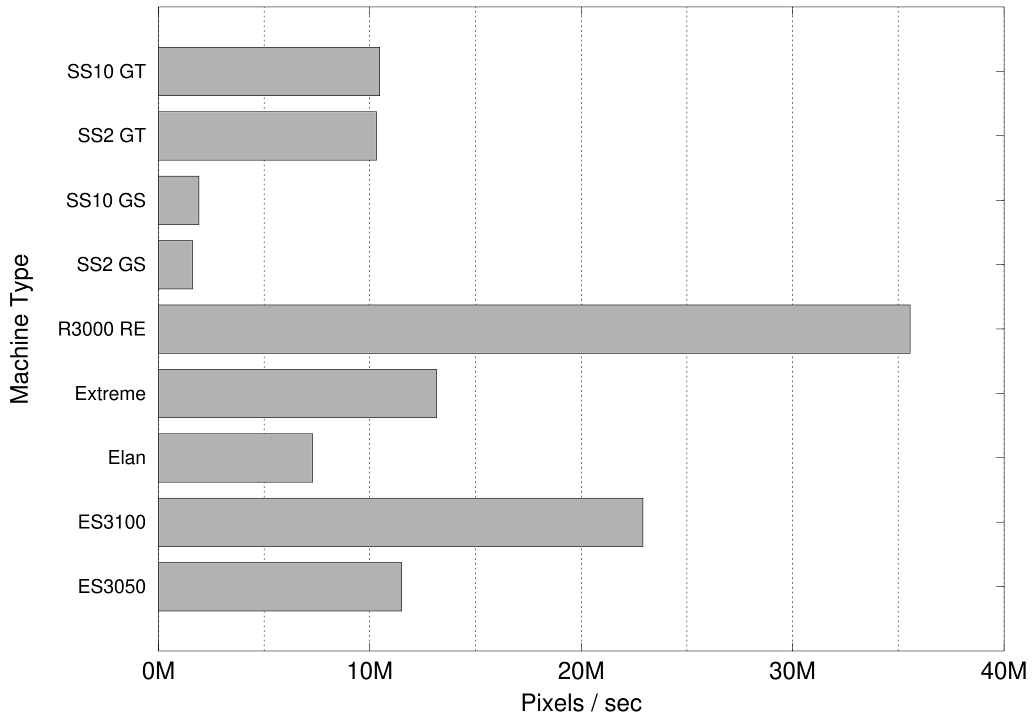

Figure 4.6 Polyline performance in pixels drawn per second.

This TCR also utilizes feedback from frame *i-1* to estimate the representation of each object for frame *i*.

Since our system is object-oriented, we will first describe the main "actors" in our system before we go into the algorithmic details of the TCR.

- A graphics database is composed out of *Container*s and *Primitives* organized in a hierarchical manner. Containers are the internal nodes, and Primitives are the leaf nodes. For this implementation of the TCR, the only kind of primitive is the *Multiple Representation* node. This node represents a single object and it contains multiple level of detail representations for the object. Other systems [Sil92d] call these nodes LOD (Level Of Detail) nodes.

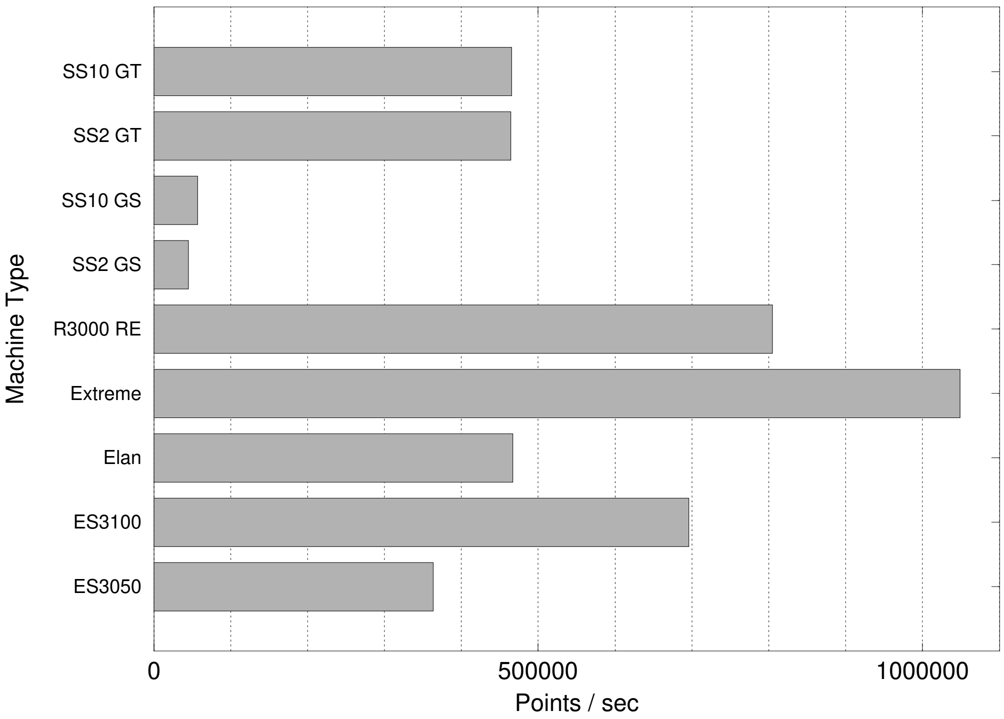

Figure 4.7 Polyline performance in points transformed per second.

- A *Scene* contains the list of objects that are supposed to be rendered at each frame. Scene is also responsible for the traversal of the object database at each frame. There are multiple types of "Scenes" in our system that provide us with different traversal algorithms of the graphics database.

At the beginning of the first frame, we set the scale factor S to 1. The algorithm that the TCR is executing at each frame is:

1. Do all the frame initialization operations. Here we clear the screen, start up the timer that evaluates how long this frame took, set up the view for the current frame, and call any additional callbacks that they user may have requested to be performed before the actual rendering of the scene starts. Let T_f be the requested frame time.

2. Divide the frame time among the primitives that require rendering. In the current implementation all objects are treated equally, but that may not be the case in the future.
3. Based on the time allocated per primitive and S, estimate the suitable representation for each object. Let estimated time T_{est} be the sum of the estimation times of all the representations of the objects chosen in Step 2. Since the object representations are not continuous, T_{est} will almost never be equal to T_f.
4. Draw each object using the estimated representation.
5. Recalculate S from T_{est} and T_f.

In the above algorithm description, the scale factor S is the feedback from the previous frame as described in Section 2.1 and Figure 2.3. Its job is to bias the RTM estimations for the future frames based on how much we were off in our estimation in the current frame.

Also note that the estimations calculated in Step 2 may be partially cached. The estimate for t_r can be cached when the view did not change, and therefore the visual representation of the geometry did not change. The estimate for t_m can be cached when the state of the system hasn't changed (ie. number of lights in the scene, type of shading, etc.). In the current implementation of the TCR, caching of RTM results is not performed.

An example program was written to test the experimental TCR. The simulation tested consists of several objects (cones, spheres, cylinder and toroids) moving around aimlessly in a unit cube of toroidal topology. The number of objects of each type can be varied either explicitly or automatically w.r.t. time. The frame rate and the range of representations for each object type can be varied interactively. We can choose to use or ignore the feedback scale factor S to improve the estimation of the RTM. We can also choose to use the time estimate based on only the modeling, modeling and area estimation, and finally modeling, area and span estimation. A sample screendump of the running program is shown in Figure 4.8

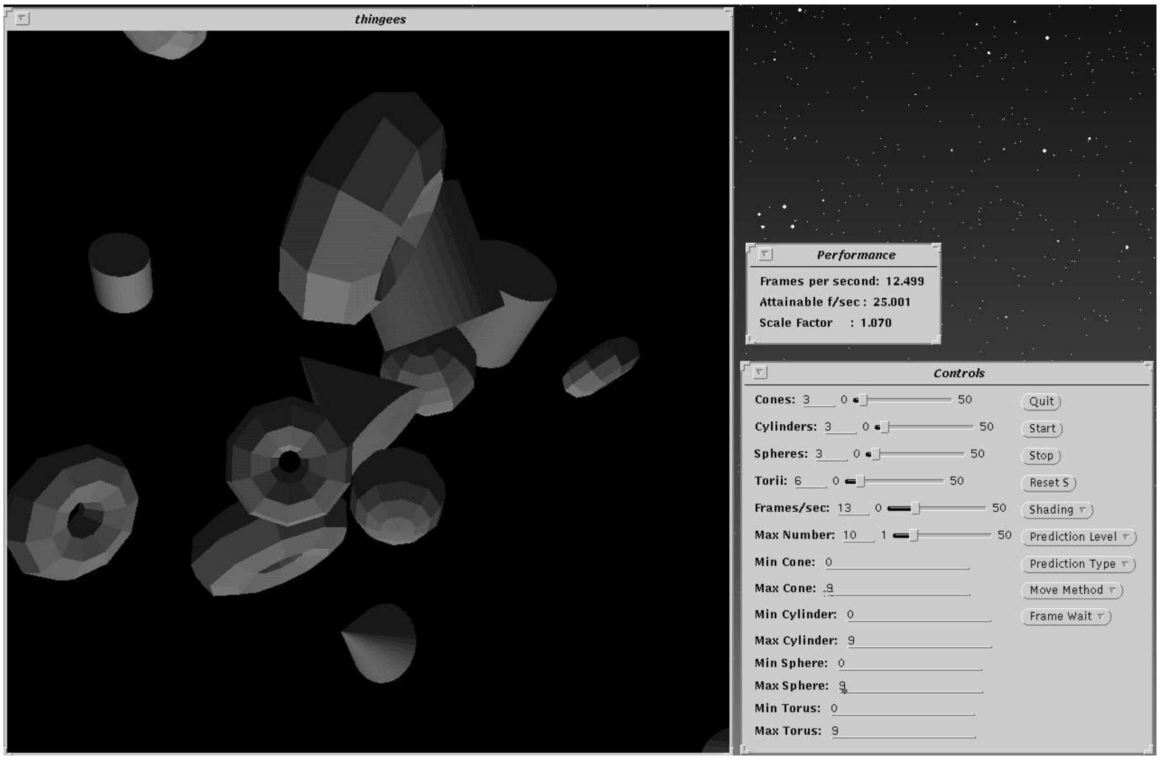

Figure 4.8 Screendump of the TCR program in operation.

Our results can be summarized below:

1. We are able to keep up with the frame rate under a large frame rate range.
2. *S* has proven very useful for correcting constant drifts of the RTM estimation, usually caused by system load and miscellaneous system overhead.
3. The amount of variation of *S* during the course of relatively constant system load was not big. *S* "biased" the RTM estimations correctly when a second instance of the TCR application was run at the same time and therefore altered the system load.

4. t_p has proven troublesome for the estimation of total time. It produces "ringing" of the representations between frames. Ringing is the effect of choosing alternating representations in adjacent frames. This effect is visually very displeasing and is unacceptable. Further work needs to be done on how to utilize t_p in a TCR.
5. t_m has been proven extremely useful. The TCR works best when using just the estimation for t_m.

5. Discussion

Here, we first discuss the outcome of the t_p and t_m tests, followed by how the break point and the model-bound / render-bound behavior affects the TCR.

5.1. Explanation of t_p and t_m results

Looking at the t_p results, we should note that t_p is not a function of the main processor. This means that if the scenes in your application are render-bound, getting a faster main processor will not improve performance.

The E&S machines produced interesting results. Out of the two machines tested, the ES3050 contains one DSP board (with two DSP processors per board), while the ES3100 contains two DSP boards. The addition of the extra DSP board boosted all of the performance figures by a factor of two, as expected. DSP processors accelerate floating point arithmetic and are used in the transformation and clipping stages of the graphics pipeline. In the case of the E&S machines, each board must also have some additional hardware that also accelerates pixel operations.

SGI machines were the ones with the fastest polygon fill performance. The Reality Engine is a special case, since it has more than twice the performance of the second fastest architecture, the Extreme, which is also an SGI machine, and more than five times the performance of the fastest non-SGI machine, the ES3100.

In terms of sliver ratio, we see that, again, SGI had the most regular performance than any of the other architectures. The sliver ratio in those machines is lower than in any of the other ones. The behavior that the GT showed , though, (sliver ratio a couple of orders of magnitude larger than any other architecture) is totally unacceptable for time-critical applications. This behavior basically means that the rendering time of almost any object (since most objects have a preferred edge direction) depends heavily of its orientation. It would very difficult for the TCR to try to estimate such behavior: Each object must be able to give a relatively accurate estimation of where its polygons are pointing. With some objects (NURBS, for example), it requires a lot of processing power to do that within the frame time. The E&S machines had about the same sliver ratio, which shows that the additional DSP processors did not affect the sliver ratio.

There may be some cases where the rasterizing back-end of the graphics processor is not really divided into edge and span interpolators, or these two sections are highly balanced. In these cases, the sliver ratio may not be that high. But even then, the span interpolation evaluation tests (time per span) are still going to be useful, since they give us an general estimate of the overhead per unit length of the perimeter of a polygon.

It is interesting to note that in most architectures, t_p is not affected by turning on color interpolation between vertices. Common sense dictates that color interpolation will cost additional time, but apparently most architectures contain optimized and/or pipelined interpolation hardware that does not cost extra time. This is a reason why graphics hardware vendors provide figures in Gouraud-shaded polygons: Color interpolation does not decrease performance.

As it can be easily seen, the evaluation of the time model for t_p is quite involved. Let's recall that the necessary requirements in order to estimate t_p are polygon area, polygon perimeter and polygon orientation. An additional problem, then, that is imposed on the TCR is that an estimate of the area, perimeter and orientation of **all** the polygons in an object is required for an accurate estimate of t_p. It is unreasonable to assume that the TCR can provide such estimates at frame rate.

In terms of t_m, there isn't too much elaboration we can do, except that the accelerators behaved as expected; t_m is the most temporally predictable section of the graphics pipeline. It seems that the separation of the total time in base time per point, overhead for lighting and then overhead for each additional light, matches the internal behavior of the graphics engines themselves.

5.2. Extending the RTM

The RTM described in this thesis is certainly far from complete. There are a lot more tests that need to be run in order to provide an RTM that can produce a time estimate for a certain primitive using the complete set of attributes that affect that primitive. In order to make the RTM complete, then, tests should:

1. Cover all of the primitives built in the graphics system. This includes things like NURBS, text, etc.
2. Cover all the attributes that affect the rendering of a certain primitive.

It should be noted, though, that an exhaustive search over all the attribute space is not needed. The temporal behavior of a certain set of attributes will probably be decomposable from the rest. From the tests we have run so far, the following two situations are perfect examples of tests that are decomposable:

1. All the tests that affect t_p are separate from all the tests that affect t_m.
2. The effect of the number of directional lights in t_m is not affected by the number of positional lights.

A detailed temporal representation of each primitive and primitive attribute is, of course, best left to the vendors themselves.

All of the tests performed so far have tested the time required for a certain primitive to render. They have not tested the amount of temporal overhead that a attribute causes to the system just by being set, even if it is not used. For example, switching attributes between each primitive is known to be costly. Graphics systems today assume temporal causality; attribute settings affect the primitives drawn after the attribute has been set. This is a one-to-one mapping from traditional and possibly old-fashioned graphics hardware and may not be a valid assumption for future multi-processing architectures, for example. The user of the renderer, though, should not have to care that the order in which he renders affects the frame time. The rendering system should have to deal with the order of primitives issue after it looks at the whole scene description.

This "pre-processing" of the data by the renderer has been looked at before in graphics systems like PHIGS [PHI88], and it can also be seen in the form of data partitioning in all visual simulation references listed in the introduction. In the general case, it involves altering as much as possible the data given to the renderer, in order to provide better performance. The ideas behind the TCR fit perfectly this concept, since the user is supposed to give the renderer a "vague" description of what he wants displayed, and then the system will use that information to produce the final result. If, for example, the user specifies that he would like certain parameters to be "free" or alterable by the TCR, then the renderer will pre-process the data in such a way that switching between the implementations that are allowed is optimal for this primitive. Since the renderer also knows the rest of the geometry displayed in the scene, it knows in what sequence w.r.t. to the rest of the scene definition it should render the current implementation of this primitive, so as to gain minimum overhead from the switching of attributes that was discussed in the previous paragraph.

5.3. Model-bound / Render-bound Break Point

How the above different implementations are chosen is very much related to the discussion about the break point and model-bound / render-bound behavior in Section 3.3.5. The fact that we have a separate estimation of t_p and t_m and therefore a way of calculating the position of the break point, can be as a criterion to switch implementations, as explained below.

First, let's assume we are in the model-bound region. A decrease in the level of detail will be beneficial, since it will move us to a curve of a lower level of complexity, and therefore decrease total rendering time. That decrease is very predictable, since we can estimate t_m with accuracy. A decrease of t_m will push our current position more towards the render-bound region.

On the other hand, if we already are in the render-bound region, a decrease of the level of detail will decrease the total time by an uncertain amount of time, because of the inaccurate estimation of t_p. The decrease will come from the decrease of the edge-interpolation time; the decrease of t_m does not count, since we are still in the render-bound region. That decrease will move us further into the render-bounded region. On the other hand, if we increase the level of complexity, then we can benefit because of two reasons:

1. The time overhead will be probably small, because it comes only by the increase of t_p because of the edge interpolation.

2. The visual effect will be significant, since the object probably takes a big area on the screen (since we are render bound) and therefore catches the attention of the user.
3. We will move closer to the break point, and therefore to the model-bound area of greater predictability.

This means that the TCR will probably want to be in the neighborhood of the break point in order to obtain best visual results and most predictable temporal behavior.

In order to make the job of the TCR easier, the creators of the graphics accelerators should try to:

1. Increase the performance of the rasterizers. This moves the break point more to the right, and makes the renderer be in a model-bound area most of the times.
2. Try to decrease the effect of edge-interpolation to pixelation time.
3. Provide us with rasterizers that are not that sensitive to the orientation of sliver polygons. Slivers make t_p very difficult to predict, which in turn has the effect of moving the break point around, making the job of the TCR harder.

It is interesting to note that all of the above items are trying to be solved in newer machines like the SGI Reality Engine or the Extreme. It looks like SGI has realized the need for greater predictability, and is designing systems with that philosophy in mind.

The TCR could compute this break point as a function of the level of complexity of the several implementations (levels of detail) of a certain object, in order to be able to know when it should switch between implementations. An common criterion that has been used by several people [ZPMW92][FST92][Mon90] to switch implementations is to make the detail level be inversely proportional to the distance from the viewpoint (the farther away, the less detail). Notice that this approach is incorporated in our way of switching representations based on the break point: If the area of an object increases, then our estimate of t_p for that object will also increase. Since we'd like our current rendering time to be as close as possible to the break point, we should switch to an implementation with a larger t_m (higher level of detail). This moves the break point to the right, and therefore closer to our current primitive area, since we were render bound because of the large value of t_p. Notice that our break point algorithm produces a quantitative way of creating and switching between different levels of detail that best utilizes the graphics pipeline, which is not provided by the heuristic algorithm used by the other references.

Additionally, we'd like the TCR to be able to switch implementation for a certain primitive "smoothly". In other words provide a "graceful" visual degradation path of the object quality as we decrease the time allotted for that object. This is not an easy problem. Performer [Sil92d] has chosen to visually blend the two representations by using transparency effects. This provides a smooth visual blend between implementations, but it does not provide a smooth temporal behavior, because both implementations have to be rendered during that blend. This is not favorable, since it violates C_g while C_m is satisfied. Performer, though, seems to be making the assumption that the accelerator is always going to be fast enough to render both implementations and still stay within the frame time.

5.4. Future Trends of Graphics Hardware

The above discussion brings us to another important issue. The generally acceptable philosophy today behind time-critical applications is to wait till the hardware becomes fast enough. This is an acceptable solution for us. Advances in rendering algorithms to accelerate rendering over the past several years have been minor compared to advances performed to accelerate raw performance of graphics hardware.

Additionally, temporal behavior of these graphics accelerators has not been very predictable. We see that changing, though, with recent architectures like the SGI Extreme and Reality Engine. In my opinion, future effort should be placed in more temporally predictable graphics systems (both in hardware and in software) together with an advancement in rendering techniques.

In addition to making temporally predictable systems, the designers should also provide a feedback of the time that a certain operation required. Spatial feedback is also necessary in some cases in order to avoid duplication of computing time and programming effort. For example, it would be useful to extract occlusion information from the Z-buffer, on a per-primitive or on a per-group-of-primitives basis, without having to calculate it separately. This could be used by a TCR to find out what objects are occluded and therefore to inhibit traversal of the part of the database that is not visible. The future looks very optimistic, in this aspect. Even today, we can see a trend of traditionally object-space algorithms in interactive graphics, like shadow generation, moving into hardware (on the Reality Engine, for example) and becoming image-based.

6. Conclusions and Future Work

6.1. Conclusions

In this thesis we attacked the problem of time-critical rendering and temporally predictable graphics systems. We started by describing a temporal model of the graphics pipeline. Then, requirements and a brief overview of a Time-Critical Renderer (TCR) were built on top of that temporal model. A Rendering Time Model (RTM) used by the TCR was then described in detail, together with its structure, creation, evaluation and use in order to estimate the rendering time of displayed geometry. The TCR will be using the RTM as an database of primitive-dependant temporal estimation.

This work has provided us with a lot of insight on how the temporal behavior of a rendering pipeline could be used by a TCR to render at predictable frame update rates. A simple TCR was also built to show that the RTM is indeed useful for time estimation.

Results show that temporal predictability of graphics systems is not an easy task. Designers should improve the temporal behavior of their systems, in order to make the job of the TCR easier.

6.2. Recommendations for Future Work

Certainly, there is a lot more work to be done in the development of a more complete and functional TCR; here we have just recognized the existence and barely touched the surface of the problem of time-critical rendering. Future work related to the TCR and the RTM include:

1. Evaluation of a more complete RTM for more primitives and attributes.

2. Estimation of overhead created by plain attribute changes, and prioritization of all attribute settings based on temporal overhead and visual significance. This requires incorporation of psychological effects of altering visual representations (Visual results created by what attributes seem to affect humans more? How much of a change in a certain attribute is tolerable, and under what circumstances?)
3. Implementation of the break-point switching algorithm for the TCR, use of the break-point for the automatic creation of multiple levels of detail for a certain object, and estimation of the algorithm's usefulness.
4. Restructuring, caching and optimization of the geometry so that the effect of attribute switching and miscellaneous graphics system overhead is minimized.
5. General-purpose spatial partitioning algorithms that work well for average-case scenes. How well do these algorithms incorporate with the ones produced in item 4?

Bibliography

[Abi89] Salim S. Abi-Ezzi. *The Graphical Processing of B-Splines in a Highly Dynamic Environment*. PhD thesis, Rensselaer Polytechnic Institute, May 1989.

[AK89] James Arvo and David Kirk. *An Introduction to Ray Tracing*, chapter 6, pages 201–262. Academic Press, 1989. A Survey of Ray Tracing Acceleration Techniques.

[ARB90] John M. Airey, John H. Rohlf, and Frederick P. Brooks. Towards image realism with interactive update rates in complex virtual building environments. *Computer Graphics*, 24(2):41–50, March 1990. Special Issue on 1990 Symbosium on Interactive 3D Graphics.

[BHN87] Ludy T. Jr. Benjamin, J. Roy Hopkins, and Jack R. Nation. *Psychology*. MacMillan, 1987.

[Cla92] Jim Clark. A TeleComputer. In *SIGGraph 1992 Conference Proceedings*, pages 19–23, 1992.

[Dee92] Michael Deering. Sun Microsystems, Inc. Private Communications, June 1992.

[Fis92] Sharon Fischler. Silicon Graphics Computer Systems. Private Communications, June 1992.

[FST92] Thomas A. Funkhouser, Carlo H. Sequin, and Seth J. Teller. Management of large amounts of data in interactive building walkthroughs. In *1992 Symbosium on Interactive 3D Graphics*, pages 11–20. University of California at Berkeley, 1992.

[Gla89] Andrew S. Glassner, editor. *An Introduction to Ray Tracing*. Academic Press, 1989.

[HB84] Kai Hwang and Fayé A. Briggs. *Computer Architecture and Parallel Processing*. McGraw-Hill, 1984.

[HKNW] James G. Hanko, Eugene M. Kuerner, J. Duane Northcutt, and Gerard A. Wall. Workstation support for time-critical applications. Technical report, Sun Microsystem Laboratories, Inc.

[Lu93] Cary Lu. Is ITV here to stay? *BYTE*, pages 139–145, February 1993.

[Mon90] Eric G. Monroe, editor. *1990 Image V Conference*. Image Society Incorporated, 1990.

[Pfl92] Russell Pflughaupt. Sun Microsystems, Inc. Private Communications, 1992.

[PHI88] PHIGS+ Committee. PHIGS+ Functional Description, Revision 3.0. *Computer Graphics*, 22(3):125–218, 1988.

[PS85] Franco P. Preparata and Michael Ian Shamos. *Computational Geometry, An Introduction*. Springer-Verlag, 1985.

[RO93] Michael Roubik and Robert O'Bara. PHIGS Interactor PHIGS Examiner. In *First Annual PHIGS User's Group Conference*, 1993.

[Sil91] Silicon Graphics Computer Systems. IRIX with REACT Technical Report, 1991. RTTR 6/91.

[Sil92a] Silicon Graphics Computer Systems. *Graphics Library Programming Guide*, 1992. Document Number 007-1210-040.

[Sil92b] Silicon Graphics Computer Systems. *Graphics Library Programming Tools and Techniques*, 1992. Document Number 007-1489-010.

[Sil92c] Silicon Graphics Computer Systems. IRIS Crimson Technical Report, 1992. ASD-IRISCRIM-TR(2/92).

[Sil92d] Silicon Graphics Computer Systems. *IRIS Performer Programming Guide*, 1992. Document Number 007-1680-010.

[Sun91] Sun Microsystems. *XGL 2.0 Reference Guide*, 1991. Part Number 800-5732-10.

[Sut63] Ivan E. Sutherland. Sketchpad: A Man-Machine Graphical Communication System. In *SJCC*, Spartan Books, Baltimore, MD, 1963.

[ZPMW92] Michael J. Zyda, David R. Pratt, James G. Monahan, and Kalin P. Wilson. NPSNET: Contructing a 3D virtual world. In *1992 Symbosium on Interactive 3D Graphics*, pages 147–156. Naval Postgraduate School, 1992.

Appendix A MultiPolygon Results

```
sec/pixel-nointerp      0.00000006007816466536
sec/pixel-interp        0.00000006012366071673

sec/point-none          0.00000340527890330122
sec/point-flat          0.00000345113903676975
sec/point-gou           0.00000341591362484905

sec/dirlight-flat       0.00000077196290249049
sec/dirlight-gou        0.00000075216212280793

sec/poslight-flat       0.00000132382012907328
sec/poslight-gou        0.00000197699591808487

ohead-lighting-flat     0.00000292826462100493
ohead-lighting-gou      0.00000163409845299611

combination-type        max

sec/span-nointerp       0.00000014303829232176
sec/span-interp         0.00000014275225623805

nointerp-h-v-ratio      12.93502807617187500000
interp-h-v-ratio        12.79462051391601562500
```

ES3050

```
sec/pixel-nointerp      0.00000003003959747616
sec/pixel-interp        0.00000003006806537087

sec/point-none          0.00000177455399352766
sec/point-flat          0.00000178179482190899
sec/point-gou           0.00000178459913513507

sec/dirlight-flat       0.00000036477840126281
sec/dirlight-gou        0.00000035760612604463

sec/poslight-flat       0.00000063456963061981
sec/poslight-gou        0.00000095341528094650

ohead-lighting-flat     0.00000147203320466360
ohead-lighting-gou      0.00000082382734944986

combination-type        max

sec/span-nointerp       0.00000007132477009009
sec/span-interp         0.00000007122121559178

nointerp-h-v-ratio      13.25118255615234375000
interp-h-v-ratio        12.66194248199462890625
```

ES3100

```
sec/pixel-nointerp      0.00000002924610242871
sec/pixel-interp        0.00000002905152918231

sec/point-none          0.00000226534302782966
sec/point-flat          0.00000379848529519222
sec/point-gou           0.00000401876832256676

sec/dirlight-flat       0.00000249460799750523
sec/dirlight-gou        0.00000251616324931092

sec/poslight-flat       0.00000314527574118983
sec/poslight-gou        0.00000317862100018829

ohead-lighting-flat     0.00000030741853151994
ohead-lighting-gou      0.00000033633250495768

combination-type        max

sec/span-nointerp       0.00000020289337498980
sec/span-interp         0.00000020348356599698

nointerp-h-v-ratio      1.07202482223510740000
interp-h-v-ratio        1.06184661388397220000
```

Elan

```
sec/pixel-nointerp      0.00000001471991328827
sec/pixel-interp        0.00000001471437638401

sec/point-none          0.00000106249342479714
sec/point-flat          0.00000148650008213735
sec/point-gou           0.00000164614289133169

sec/dirlight-flat       0.00000109077200249885
sec/dirlight-gou        0.00000108910035123699

sec/poslight-flat       0.00000136345840928698
sec/poslight-gou        0.00000136013295559678

ohead-lighting-flat     0.00000020784250409633
ohead-lighting-gou      0.00000020116499399592

combination-type        max

sec/span-nointerp       0.00000016348825226942
sec/span-interp         0.00000016527015134216

nointerp-h-v-ratio      1.00050878524780270000
interp-h-v-ratio        1.00090670585632320000
```

Extreme

sec/pixel-nointerp	0.00000000585053605562
sec/pixel-interp	0.00000000584960790917
sec/point-none	0.00000122788242151728
sec/point-flat	0.00000125947963169892
sec/point-gou	0.00000122330709473317
sec/dirlight-flat	0.00000000000000000000
sec/dirlight-gou	0.00000011874218586172
sec/poslight-flat	0.00000000221515006338
sec/poslight-gou	0.00000029582108140858
ohead-lighting-flat	0.00000000000000000000
ohead-lighting-gou	0.00000000000000000000
combination-type	max
sec/span-nointerp	0.00000002669114529397
sec/span-interp	0.00000003572273854502
nointerp-h-v-ratio	1.95516085624694820000
interp-h-v-ratio	2.61612367630004880000

Reality Engine

sec/pixel-nointerp	0.00000008062136203080
sec/pixel-interp	0.00000008036369081310
sec/point-none	0.00002425962702545803
sec/point-flat	0.00002427021536277607
sec/point-gou	0.00002428082552796695
sec/dirlight-flat	0.00000000000000000000
sec/dirlight-gou	0.00000000291038304567
sec/poslight-flat	0.00000183666998054832
sec/poslight-gou	0.00000000380532583222
ohead-lighting-flat	0.00001727099515846930
ohead-lighting-gou	0.00006742715777363628
combination-type	add
sec/span-nointerp	0.00000106303411939734
sec/span-interp	0.00000106318702819408
nointerp-h-v-ratio	8.61822223663330078125
interp-h-v-ratio	8.61073303222656250000

GS on a Sparc 2

sec/pixel-nointerp	0.00000006829361609562
sec/pixel-interp	0.00000006823478315710
sec/point-none	0.00002059564394585323
sec/point-flat	0.00002090850284730550
sec/point-gou	0.00002009677336900495
sec/dirlight-flat	0.00000000000000000000
sec/dirlight-gou	0.00000000067908939583
sec/poslight-flat	0.00000148453545989469
sec/poslight-gou	0.00000016574874450725
ohead-lighting-flat	0.00001410756703990046
ohead-lighting-gou	0.00005593881724053062
combination-type	add
sec/span-nointerp	0.00000089978885853270
sec/span-interp	0.00000096029282303789
nointerp-h-v-ratio	8.61208820343017578125
interp-h-v-ratio	8.81508064270019531250

GS+ on a Sparc 10

```
sec/pixel-nointerp      0.00000004889373173000
sec/pixel-interp        0.00000004891206017987

sec/point-none          0.00001562526631460059
sec/point-flat          0.00001562364013807382
sec/point-gou           0.00001562588840897661

sec/dirlight-flat       0.00000198682050722709
sec/dirlight-gou        0.00000198644988813612

sec/poslight-flat       0.00000644638384983409
sec/poslight-gou        0.00002053327079920564

ohead-lighting-flat     0.00000000000000000000
ohead-lighting-gou      0.00003493583790259436

combination-type        max

sec/span-nointerp       0.00000025150566784760
sec/span-interp         0.00000025150890792247

nointerp-h-v-ratio      845.58428955078125000000
interp-h-v-ratio        1012.10119628906250000000
```

GT on a Sparc 2

```
sec/pixel-nointerp      0.00000004860583047162
sec/pixel-interp        0.00000004862335600819

sec/point-none          0.00000533619004272623
sec/point-flat          0.00000500821352034109
sec/point-gou           0.00000488278237753548

sec/dirlight-flat       0.00000200998033506039
sec/dirlight-gou        0.00000195847928807780

sec/poslight-flat       0.00000646367288936744
sec/poslight-gou        0.00002052280615316704

ohead-lighting-flat     0.00000520024059369462
ohead-lighting-gou      0.00004332449680077843

combination-type        max

sec/span-nointerp       0.00000025130012204500
sec/span-interp         0.00000025135227588180

nointerp-h-v-ratio      1086.62561035156250000000
interp-h-v-ratio        745.21911621093750000000
```

GT on a Sparc 10

Appendix B TriangleStrip Results

```
sec/pixel-nointerp       0.00000006007816466536
sec/pixel-interp         0.00000006012366071673

sec/point-none           0.00000353729842572648
sec/point-flat           0.00000354469216290454
sec/point-gou            0.00000353429254573712

sec/dirlight-flat        0.00000074810577643802
sec/dirlight-gou         0.00000074954931505999

sec/poslight-flat        0.00000180237475433387
sec/poslight-gou         0.00000182697078798810

ohead-lighting-flat      0.00000401217948819976
ohead-lighting-gou       0.00000175699619830993

combination-type         max

sec/span-nointerp        0.00000014303829232176
sec/span-interp          0.00000014275225623805

nointerp-h-v-ratio       12.93502807617187500000
interp-h-v-ratio         12.79462051391601562500
```

ES3050

```
sec/pixel-nointerp       0.00000003003959747616
sec/pixel-interp         0.00000003006806537087

sec/point-none           0.00000174988520029729
sec/point-flat           0.00000174990873347269
sec/point-gou            0.00000176333060153411

sec/dirlight-flat        0.00000037723779655607
sec/dirlight-gou         0.00000037260460317157

sec/poslight-flat        0.00000090197289637217
sec/poslight-gou         0.00000090645215777840

ohead-lighting-flat      0.00000198176462617994
ohead-lighting-gou       0.00000085188526099955

combination-type         max

sec/span-nointerp        0.00000007132477009009
sec/span-interp          0.00000007122121559178

nointerp-h-v-ratio       13.25118255615234375000
interp-h-v-ratio         12.66194248199462890625
```

ES3100

sec/pixel-nointerp	0.00000002924610242871
sec/pixel-interp	0.00000002905152918231
sec/point-none	0.00000332868376062834
sec/point-flat	0.00000457904297945788
sec/point-gou	0.00000557627572561614
sec/dirlight-flat	0.00000244826992457092
sec/dirlight-gou	0.00000243600129579136
sec/poslight-flat	0.00000312537576974137
sec/poslight-gou	0.00000310664131575322
ohead-lighting-flat	0.00000044742046156898
ohead-lighting-gou	0.00000020104653231101
combination-type	max
sec/span-nointerp	0.00000020289337498980
sec/span-interp	0.00000020348356599698
nointerp-h-v-ratio	1.07202482223510740000
interp-h-v-ratio	1.06184661388397220000

Elan

```
sec/pixel-nointerp      0.00000001471991328827
sec/pixel-interp        0.00000001471437638401

sec/point-none          0.00000138245263769932
sec/point-flat          0.00000200493605007068
sec/point-gou           0.00000230432260650559

sec/dirlight-flat       0.00000109795894331910
sec/dirlight-gou        0.00000109639825041086

sec/poslight-flat       0.00000137250845000381
sec/poslight-gou        0.00000136983896936727

ohead-lighting-flat     0.00000021492905943887
ohead-lighting-gou      0.00000021989217202645

combination-type        max

sec/span-nointerp       0.00000016348825226942
sec/span-interp         0.00000016527015134216

nointerp-h-v-ratio      1.00050878524780270000
interp-h-v-ratio        1.00090670585632320000
```

Extreme

sec/pixel-nointerp	0.00000000585053605562
sec/pixel-interp	0.00000000584960790917
sec/point-none	0.00000091820476200155
sec/point-flat	0.00000146998729633196
sec/point-gou	0.00000144787952649494
sec/dirlight-flat	0.00000011116844689241
sec/dirlight-gou	0.00000017589407264040
sec/poslight-flat	0.00000003335973630669
sec/poslight-gou	0.00000002000115273404
ohead-lighting-flat	0.00000000000000000000
ohead-lighting-gou	0.00000025896929400915
combination-type	max
sec/span-nointerp	0.00000002669114529397
sec/span-interp	0.00000003572273854502
nointerp-h-v-ratio	1.95516085624694820000
interp-h-v-ratio	2.61612367630004880000

Reality Engine

```
sec/pixel-nointerp      0.00000008062136203080
sec/pixel-interp        0.00000008036369081310

sec/point-none          0.00003876144910464063
sec/point-flat          0.00003876637128996663
sec/point-gou           0.00003876365735777654

sec/dirlight-flat       0.00000773201008996693
sec/dirlight-gou        0.00000000282307155430

sec/poslight-flat       0.00000729853718439699
sec/poslight-gou        0.00000000000000000000

ohead-lighting-flat     0.00004240705675329082
ohead-lighting-gou      0.00020473878248594701

combination-type        add

sec/span-nointerp       0.00000106303411939734
sec/span-interp         0.00000106318702819408

nointerp-h-v-ratio      8.61822223663330078125
interp-h-v-ratio        8.61073303222656250000
```

GS on a Sparc 2

sec/pixel-nointerp	0.00000006829361609562
sec/pixel-interp	0.00000006823478315710
sec/point-none	0.00003277151699876413
sec/point-flat	0.00003277367068221793
sec/point-gou	0.00003289874803158455
sec/dirlight-flat	0.00000656235215501511
sec/dirlight-gou	0.00000001557540052488
sec/poslight-flat	0.00000601718556936248
sec/poslight-gou	0.00000000000000000000
ohead-lighting-flat	0.00003589478728827089
ohead-lighting-gou	0.00017145447782240808
combination-type	add
sec/span-nointerp	0.00000089978885853270
sec/span-interp	0.00000096029282303789
nointerp-h-v-ratio	8.61208820343017578125
interp-h-v-ratio	8.81508064270019531250

GS+ on a Sparc 10

```
sec/pixel-nointerp      0.00000004889373173000
sec/pixel-interp        0.00000004891206017987

sec/point-none          0.00000568845553061692
sec/point-flat          0.00000568159521208145
sec/point-gou           0.00000568562018088414

sec/dirlight-flat       0.00000199925239030563
sec/dirlight-gou        0.00000199816327040025

sec/poslight-flat       0.00000656116617392399
sec/poslight-gou        0.00003062694304389879

ohead-lighting-flat     0.00000201012971956516
ohead-lighting-gou      0.00005484548819367774

combination-type        max

sec/span-nointerp       0.00000025150566784760
sec/span-interp         0.00000025150890792247

nointerp-h-v-ratio      845.58428955078125000000
interp-h-v-ratio        1012.10119628906250000000
```

GT on a Sparc 2

sec/pixel-nointerp	0.00000004860583047162
sec/pixel-interp	0.00000004862335600819
sec/point-none	0.00000534878654434578
sec/point-flat	0.00000534617356606759
sec/point-gou	0.00000534629407411558
sec/dirlight-flat	0.00000190977584679786
sec/dirlight-gou	0.00000190571518032812
sec/poslight-flat	0.00000645320915282355
sec/poslight-gou	0.00003054888657061383
ohead-lighting-flat	0.00000206522781809326
ohead-lighting-gou	0.00005392796447267756
combination-type	max
sec/span-nointerp	0.00000025130012204500
sec/span-interp	0.00000025135227588180
nointerp-h-v-ratio	1086.62561035156250000000
interp-h-v-ratio	745.21911621093750000000

GT on a Sparc 10

Appendix C QuadMesh Results

```
sec/pixel-nointerp        0.00000006007816466536
sec/pixel-interp          0.00000006012366071673

sec/point-none            0.00005140862049302086
sec/point-flat            0.00005140517532709055
sec/point-gou             0.00005140721987118013

sec/dirlight-flat         0.00000160762283485383
sec/dirlight-gou          0.00000158568821007066

sec/poslight-flat         0.00000283702206615999
sec/poslight-gou          0.00000412322106058127

ohead-lighting-flat       0.00000124509097076952
ohead-lighting-gou        0.00000000000000000000

combination-type          max

sec/span-nointerp         0.00000014303829232176
sec/span-interp           0.00000014275225623805

nointerp-h-v-ratio        12.93502807617187500000
interp-h-v-ratio          12.79462051391601562500
```

ES3050

sec/pixel-nointerp	0.00000003003959747616
sec/pixel-interp	0.00000003006806537087
sec/point-none	0.00002567468982306309
sec/point-flat	0.00002567773481132463
sec/point-gou	0.00002567400770203676
sec/dirlight-flat	0.00000080217915865433
sec/dirlight-gou	0.00000079453883472524
sec/poslight-flat	0.00000141930161134951
sec/poslight-gou	0.00000206953154702205
ohead-lighting-flat	0.00000062344042817131
ohead-lighting-gou	0.00000000000000000000
combination-type	max
sec/span-nointerp	0.00000007132477009009
sec/span-interp	0.00000007122121559178
nointerp-h-v-ratio	13.25118255615234375000
interp-h-v-ratio	12.66194248199462890625

ES3100

sec/pixel-nointerp	0.00000002924610242871
sec/pixel-interp	0.00000002905152918231
sec/point-none	0.00000363404774361697
sec/point-flat	0.00000767703841120237
sec/point-gou	0.00000776956130721373
sec/dirlight-flat	0.00000227082978199178
sec/dirlight-gou	0.00000297859082820651
sec/poslight-flat	0.00000307537106891687
sec/poslight-gou	0.00000303545152746665
ohead-lighting-flat	0.00000000000000000000
ohead-lighting-gou	0.00000000000000000000
combination-type	max
sec/span-nointerp	0.00000020289337498980
sec/span-interp	0.00000020348356599698
nointerp-h-v-ratio	1.07202482223510740000
interp-h-v-ratio	1.06184661388397220000

Elan

sec/pixel-nointerp	0.00000001471991328827
sec/pixel-interp	0.00000001471437638401
sec/point-none	0.00000148841911595810
sec/point-flat	0.00000224119162339775
sec/point-gou	0.00000242963801611040
sec/dirlight-flat	0.00000108532583453780
sec/dirlight-gou	0.00000109103598333604
sec/poslight-flat	0.00000136984272103291
sec/poslight-gou	0.00000138126335968991
ohead-lighting-flat	0.00000000000000000000
ohead-lighting-gou	0.00000007188805284386
combination-type	max
sec/span-nointerp	0.00000016348825226942
sec/span-interp	0.00000016527015134216
nointerp-h-v-ratio	1.00050878524780270000
interp-h-v-ratio	1.00090670585632320000

Extreme

sec/pixel-nointerp	0.00000000585053605562
sec/pixel-interp	0.00000000584960790917
sec/point-none	0.00000107779862901225
sec/point-flat	0.00000134643096316722
sec/point-gou	0.00000176883588665078
sec/dirlight-flat	0.00000011060456017731
sec/dirlight-gou	0.00000033212077710232
sec/poslight-flat	0.00000007838408322414
sec/poslight-gou	0.00000027579676498135
ohead-lighting-flat	0.00000001513785718998
ohead-lighting-gou	0.00000000000000000000
combination-type	max
sec/span-nointerp	0.00000002669114529397
sec/span-interp	0.00000003572273854502
nointerp-h-v-ratio	1.95516085624694820000
interp-h-v-ratio	2.61612367630004880000

Reality Engine

sec/pixel-nointerp	0.00000008062136203080
sec/pixel-interp	0.00000008036369081310
sec/point-none	0.00004935747347190045
sec/point-flat	0.00004936086043016985
sec/point-gou	0.00004935419929097407
sec/dirlight-flat	0.00000000000000000000
sec/dirlight-gou	0.00000000138243194669
sec/poslight-flat	0.00000727085716789588
sec/poslight-gou	0.00000000402602973182
ohead-lighting-flat	0.00006914156256243587
ohead-lighting-gou	0.00020127386960666627
combination-type	add
sec/span-nointerp	0.00000106303411939734
sec/span-interp	0.00000106318702819408
nointerp-h-v-ratio	8.61822223663330078125
interp-h-v-ratio	8.61073303222656250000

GS on a Sparc 2

```
sec/pixel-nointerp      0.00000006829361609562
sec/pixel-interp        0.00000006823478315710

sec/point-none          0.00004136966890655458
sec/point-flat          0.00004133549373364076
sec/point-gou           0.00004143701153225265

sec/dirlight-flat       0.00000000000000000000
sec/dirlight-gou        0.00000000000000000000

sec/poslight-flat       0.00000594440871282131
sec/poslight-gou        0.00000001731192789123

ohead-lighting-flat     0.00005723942740587518
ohead-lighting-gou      0.00016750291979406029

combination-type        add

sec/span-nointerp       0.00000089978885853270
sec/span-interp         0.00000096029282303789

nointerp-h-v-ratio      8.61208820343017578125
interp-h-v-ratio        8.81508064270019531250
```

GS+ on a Sparc 10

```
sec/pixel-nointerp      0.00000004889373173000
sec/pixel-interp        0.00000004891206017987

sec/point-none          0.00000711524262442254
sec/point-flat          0.00000714121051714756
sec/point-gou           0.00000712716337147867

sec/dirlight-flat       0.00000203561353373516
sec/dirlight-gou        0.00000202542128135974

sec/poslight-flat       0.00000657089913147502
sec/poslight-gou        0.00003023053432116285

ohead-lighting-flat     0.00000251823075814173
ohead-lighting-gou      0.00005484401481226087

combination-type        max

sec/span-nointerp       0.00000025150566784760
sec/span-interp         0.00000025150890792247

nointerp-h-v-ratio      845.58428955078125000000
interp-h-v-ratio        1012.10119628906250000000
```

GT on a Sparc 2

sec/pixel-nointerp	0.00000004860583047162
sec/pixel-interp	0.00000004862335600819
sec/point-none	0.00000608805657975608
sec/point-flat	0.00000608745631325291
sec/point-gou	0.00000608759910392109
sec/dirlight-flat	0.00000202926844394824
sec/dirlight-gou	0.00000202026058104821
sec/poslight-flat	0.00000659578427075758
sec/poslight-gou	0.00003034470682905521
ohead-lighting-flat	0.00000151014501170721
ohead-lighting-gou	0.00005322217475622892
combination-type	max
sec/span-nointerp	0.00000025130012204500
sec/span-interp	0.00000025135227588180
nointerp-h-v-ratio	1086.62561035156250000000
interp-h-v-ratio	745.21911621093750000000

GT on a Sparc 10

Appendix D Polyline Results

sec/point	0.00000275304637398222
sec/pxl	0.00000008686247099376

ES3050

sec/point	0.00000143653574014024
sec/pxl	0.00000004361796257513

ES3100

sec/point	0.00000214050805880106
sec/pxl	0.00000013700615397738

Elan

sec/point	0.00000095374139164051
sec/pxl	0.00000007598521989394

Extreme

sec/point	0.00000124247480925987
sec/pxl	0.00000002811186838869

Reality Engine

sec/point	0.00002217965993622784
sec/pxl	0.00000061983570276425

GS on a Sparc 2

sec/point	0.00001757266909407917
sec/pxl	0.00000052270172545832

GS+ on a Sparc 10

sec/point	0.00000215364570976817
sec/pxl	0.00000009696411495952

GT on a Sparc 2

sec/point	0.00000214762894756859
sec/pxl	0.00000009547154178335

GT on a Sparc 10