Eurographics Symposium on Rendering 2003
Per Christensen and Daniel Cohen-Or (Editors)

Rendering Time Estimation for Real-Time Rendering

Michael Wimmer[†] and Peter Wonka[‡]

[†]Vienna University of Technology, Vienna, Austria
[‡]Georgia Institute of Technology, Atlanta, USA

Abstract

This paper addresses the problem of estimating the rendering time for a real-time simulation. We study different factors that contribute to the rendering time in order to develop a framework for rendering time estimation. Given a viewpoint (or view cell) and a list of potentially visible objects, we propose several algorithms that can give reasonable upper limits for the rendering time on consumer hardware. This paper also discusses several implementation issues and design choices that are necessary to make the rendering time predictable. Finally, we lay out two extensions to current rendering hardware which would allow implementing a system with constant frame rates.

Categories and Subject Descriptors (according to ACM CCS): J.7.6 [Computer Applications]: Real time I.3.1 [Computer Graphics]: Hardware Architecture I.3.3 [Computer Graphics]: Picture/Image Generation - Display algorithms I.3.7 [Computer Graphics]: Three-Dimensional Graphics and Realism - Virtual reality

1. Introduction

The quality of a real-time rendering application is determined by several aspects, including impressive models, visually attractive effects like shadows, reflections and shading, etc. In this paper we take a look at another component that largely contributes to high-quality graphics simulation as sought for in computer games, trade shows, and driving simulators. It is a factor that has often been overlooked in current products: fluent and continuous motion. For the impression of fluent motion, it is necessary to render at a fixed high frame-rate, mainly determined by the refresh rate of the display device (typically 60 Hz or more). Failing to do so results in distracting and visually displeasing artifacts like ghosting (Figure 7) and jerks[7]. Furthermore, the predictability of frame times is crucial for the ability to schedule certain events and to coordinate input with simulation and display. Especially due to the use of hardware command buffers, the time span between the issuing of a rendering command and its actual execution can easily be over one frame. If the rendering time of the frame is not known in advance and frame times have a high variance, the apparent moving speed will change constantly (see Figure 6). This is most visible when the viewer is rotating.

To obtain a system with a fixed frame rate, it is necessary to guarantee that the rendering time does not exceed a certain time limit. One building block of such a system is a prediction of the rendering time for a given frame. While previous work[5, 1] showed how to build a real-time rendering system with guaranteed frame rates when such a prediction function is given, the actual prediction function used for the rendering time estimation should be studied in greater detail.

In this paper, we undertake a more in-depth study of rendering time in order to show which simple heuristics can be used for this purpose and how successful they are. Among others, we present an approach based on sampling rendering times, and an improved mathematical heuristics based on a cost function. An important part of this work is the proposal of two hardware extensions—a time-stamping function and a conditional branch mechanism—which make rendering time estimations more robust. We show how to implement a soft real-time system providing fixed frame rates using these extensions. We also skirt practical issues and design choices encountered when implementing a real-time rendering system, and give hints and examples for those. In the long run, we aim to use the results of this paper to construct a soft real-time system with bounded frame times. Rendering time estimation can also be used to calculate efficient placement of impostors in complex scenes.

First, we set out by defining the rendering time function.

As the most general form we propose

$$t = RT(SG, RA, HW, ST),$$

where SG is a scene graph, RA is the rendering action used for traversal, HW is the hardware, and ST is the current state of the hardware, software and the operating system. While this form is general enough to incorporate all important effects that influence the rendering time, it is complicated to use in practice. Therefore, we will use a simpler form, where the scene graph is an ordered set of objects $\mathbf{X} = (x_1, ..., x_n)$, with given geometry and attributes $x_i = (g_i, a_i)$. Furthermore, we assume that the rendering action is implicitly defined by the attributes of the objects. We thus obtain the following form:

$$t = RT(\mathbf{X}, HW, ST)$$

This formulation of the rendering time function is the basis for the framework which we will use to discuss different aspects of the rendering time estimation problem.

The rest of the paper is organized as follows. Section 3 explains the functionality of current rendering hardware, section 4 describes our framework to estimate rendering time and explains the different tasks and factors that contribute to the rendering time and how to estimate them. In section 5 we describe the crucial parts of the rendering time estimation in greater detail, and in section 6 we propose two hardware extensions necessary for a soft real-time system. Sections 2, 7 and 8 present previous work, results and conclusions.

2. Previous Work

Funkhouser and Séquin[5] demonstrate a real-time rendering system that bounds frame times through appropriate level-of-detail (LOD) selection. They assign a cost and a benefit to each LOD for each object. The optimization tries to select for each object such a LOD that the sum of the benefit values is maximized, given a maximum allowable cost. The cost metric used is equivalent to the following rendering time estimation:

$$RT(x) = max(c_1 * \#polys(x) + c_2 * \#v(x), c_3 * \#pix(x))$$

where x is the LOD in consideration, $\#polys$ is the number of polygons and $\#v$ is the number of vertices of the object, and $\#pix$ is the number of pixels in the projection. The parameters c_1, c_2 and c_3 are constants that are determined experimentally by rendering several sample objects.

While vertex and polygon counts are both reasonable estimates for geometric complexity, a better match for the way modern GPUs actually behave is the number of *actually transformed vertices*, i.e., the number of vertices that are not taken from the post-transform vertex cache used by such GPUs, but actually fetched and transformed. Hoppe uses this number as a basis for a cost function used for the creation of efficient triangle strips[8]. While Hoppe additionally considers the index transfer for indexed triangle strips, due to the small size of indices this factor plays only a limited role for rendering time estimation. See Figure 1 for an illustration of the different concepts.

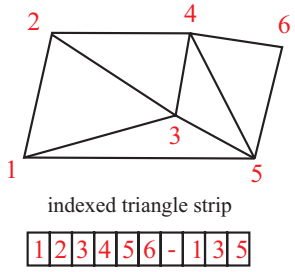

Figure 1: *This figure illustrates different attributes of an indexed triangle strip. The strip consists of 6 indices, then a restart denoted by - and three more vertices. The number of polygons is 5, the number of indices is 9, the number of vertices is three times the number of polygons (15) and the number of actually transformed vertices is 6, i.e., each vertex is only transformed once, as they all fit into the vertex cache.*

Aliaga and Lastra[1] construct a view-cell based real-time rendering system. They sample the view space with a large number of viewpoints, where each sample viewpoint defines a view cell around it. For each cell, they select objects for direct rendering according to a cost-benefit function. The remaining objects are replaced by layered depth images. The cost metric is based on the number of triangles and ignores the influence of rasterization. The proposed rendering time estimation is therefore

$$RT(x) = c_1 * \#tris(x)$$

where c_1 is determined by the triangle rate of the given hardware. The accuracy of these estimations will be compared with our estimations in section 7.

The problem of maintaining a specified frame rate has also been addressed in the Performer system[13], however based on a reactive LOD selection system. However, bounded frame rates can only be guaranteed using a predictive mechanism. Regan and Pose[12] demonstrated a system capable of maintaining fixed frame rates with a special-purpose display controller for head-mounted displays based on just-in-time image composition.

3. Rendering Hardware Overview

This research focuses on consumer hardware, namely a PC with a state-of-the-art graphics card (current examples are NVIDIA's GeForce or ATI's Radeon products). Consumer hardware is not naturally geared towards the construction of

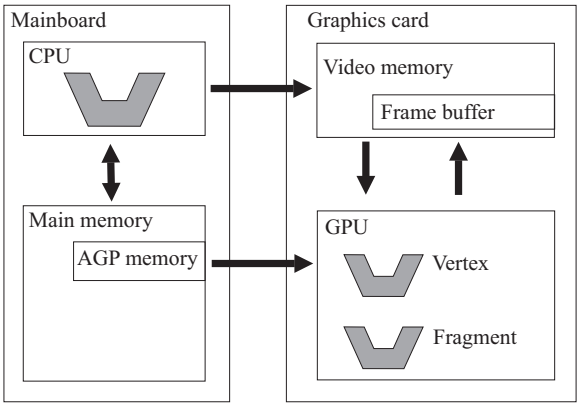

Figure 2: *This figure shows an overview of the graphics architecture. The arrows indicate the most important flows of data.*

a real-time rendering system, because hardly any tools are available to give a hard time limit for the execution of a given set of rendering commands. However, these systems are very wide spread and used by many interactive rendering applications. Therefore, it seems worthwhile to provide a best-effort estimation of the execution time, knowing that a hard real-time rendering system cannot be achieved. So we aim at the construction of a soft real-time rendering system working with statistical guarantees for the rendering time.

In the following we give a functional overview of the rendering system (see Figure 2). The rendering process uses the CPU and the GPU in parallel. The application is executed on the CPU and sends commands to the GPU via the graphics driver. We will use a typical frame of a simple rendering engine to illustrate this functionality (see also Figure 4). On the *CPU* side, the application issues a clear screen (clr) command to initialize the frame buffer. Then the application traverses all objects $x_i = (g_i, a_i)$. For each object, the application has to set the state of the graphics hardware according to the attributes a_i and then send the geometry g_i. The driver sends the commands to a *command buffer* (FIFO). The GPU reads commands from this buffer and executes them. State commands change the state of the pipeline according to the attributes of the primitives. Geometry is sent down the pipeline starting with the vertex processing unit, after which it is rasterized into fragments that in turn go to the fragment processing unit. The primitives, mainly indexed triangle strips, consist of vertex data and index data. For efficiency, both vertex and index data are either stored in AGP memory or in video (=graphics card) memory, both of which are directly accessible by the GPU.

4. The rendering time estimation framework

In this section we will propose a framework for analyzing the rendering time. The main idea is to assume a subdivision of the rendering process into a number of conceptually independent tasks which can be estimated separately. The remaining interdependencies that are not captured by our model can usually be subsumed under a so-called system task. Note that the test system used to obtain the empirical results shown in this section is described in section 7.

4.1. The refined rendering time estimation function

The refined rendering time estimation function RT which is used for the discussion in this section is made up of estimations ET for four major components (tasks),

- system tasks (ET_{system}),
- CPU tasks (ET^{CPU})
- idle time ($ET^{CPU}_{idle}, ET^{GPU}_{idle}$), and
- GPU tasks (ET^{GPU}),

in the following way:

$$RT = ET_{system} + max(ET^{CPU}, ET^{GPU})$$

with

$$ET^{CPU} = ET^{CPU}_{nr} + ET^{CPU}_{r} + ET^{CPU}_{mm} + ET^{CPU}_{idle}$$

and

$$ET^{GPU} = ET^{GPU}_{fs} + ET^{GPU}_{r} + ET^{GPU}_{mm} + ET^{GPU}_{idle}.$$

Here, the indices nr denote non-rendering code, fs frame setup, r rendering code, mm memory management, and $idle$ is idle time.

4.2. System tasks

The operating system and other applications use time for tasks like network services, indexing services, file system, memory managers etc., some of which cannot be totally eliminated even during the execution of a higher-priority process. Further, the graphics driver itself might schedule certain tasks like optimization routines, which are not documented and cannot be predicted. This time is denoted as ET_{system} in our framework, and is the main reason for variations in frame times for a single view.

To understand its influence on the rendering time, we conducted the following experiment. In a test scene we selected a number of individual viewpoints and measured the rendering time for 10,000 subsequent renderings for each viewpoint. All non-critical services were turned off (including indexing and other applications with file access), and the rendering thread was set to a higher priority, thus basically eliminating all non-critical operating system activity. Synchronization with the vertical retrace was also turned off in order to eliminate any dependency on the physical display device. We expected the rendering time variations to resemble a lognormal distribution, which would be typical for a

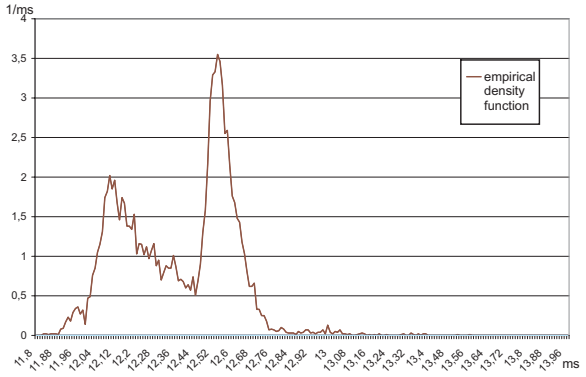

Figure 3: *This graph shows an example for an empirical density function of the rendering time derived through 10,000 renderings from the same viewpoint.*

process involving completion times (this has been observed, for example, in quality control engineering and traffic flow theory[6]). However, there are variations that follow a quite peculiar pattern. Figure 3 shows the empirical probability density function for one of the chosen viewpoints. It does not resemble any of the well-known distribution function.

One option is to assume that the variation of this distribution represents the influence of system tasks. For example, the bimodal nature of the distribution suggests that there is a regular system task (e.g., the thread scheduler, related to the thread quantum) which is executed approximately every 2 to 3 frames and takes about 0.5 ms. The variation of the distribution also depends on the total execution time of the frame. Since we aim for a system with fixed frame times, we estimate the time of the system tasks as a constant c representing the maximum deviation of the minimum rendering time for a given confidence interval and target rendering time: $ET_{system,confidence} = c$. The constant c is determined by calculating the width of the given confidence interval with respect to the empirical probability distribution function of a test measurement. Although the distribution can vary greatly near the mean value, we found the extremal values and therefore the estimated constant to be quite consistent. For our test system, we used a confidence interval of 99%, from which we calculated c as 1.52ms.

4.3. CPU tasks

The CPU is responsible for several tasks. First, there is application code that does not directly contribute to rendering, like artificial intelligence, animation, collision detection, sound and networking (ET_{nr}^{CPU}). Second, there are rendering-related tasks like scene-graph traversal, view-frustum culling and dynamic state sorting (ET_{r}^{CPU}). Third, there are issues of memory management like the recalculation of dynamic vertex data and texture downloads (ET_{mm}^{CPU}). Fourth, there is the idle time when the GPU is too busy to accept further commands (ET_{idle}^{CPU}).

4.3.1. CPU memory management

Texture memory management. If not all textures for a given scene fit into video memory (and can therefore not be downloaded before rendering), a memory management routine is required which selects for each frame which textures need to be downloaded. For efficiency reasons[2] and for predictability, this should not be left to the graphics driver. The texture management time $ET_{mm,tex}^{CPU}$ is given by the sum of download times for all textures selected for download for a given frame.

Geometry memory management. Similarly, *static geometry* needs to be managed by a geometry management routine if not all the static geometry in a scene fits into memory directly accessible by the GPU (video memory and AGP memory). The geometry management time is $ET_{mm,geom}^{CPU} = c * g$, where g is the amount of memory for the geometry scheduled for download in the current frame, and c is the memory copying speed. In the case of indexed geometry, indices are transferred either during the API call (in unextended OpenGL) or are managed together with the vertex data. *Dynamic geometry*, i.e. animated meshes where vertex positions change, should be written directly to AGP memory upon generation in order to avoid double copies. For a comparison of geometry download methods and their influence on the rendering time see also section 4.5.3.

4.4. Idle time

Ideally, CPU and GPU run in parallel. However, sometimes either the CPU or the GPU sits idle while waiting for the other part to supply data or complete a task. This (undesirable) time is called idle time and occurs in the *graphics driver*, which runs as a part of the user application on the CPU. The issue of idle time arises when the graphics driver writes commands for the graphics card to the command buffer and this buffer is either full or empty:

- When the buffer is full, the driver blocks the application and returns only when the buffer can be accessed again. This usually means that the CPU is supplying graphics commands fast enough and could be used to do more complex non-rendering calculations.
- When the buffer is empty, the GPU is starved for rendering commands (this situation is therefore also known as back-end starvation). The CPU is not supplying graphics commands fast enough and the GPU sits idle, leaving some of its potential unused.

A rendering application which makes best use of the available graphics hardware should strive for maximum parallelism and avoid backend starvation (so that $ET_{idle}^{GPU} = 0$). If this cannot be achieved because the non-rendering code

is causing the starvation and cannot be optimized, the unused time in the GPU can still be used to execute more complex shaders or draw more complex geometry without affecting the rendering time. In the following, we therefore assume a balanced system, where the GPU is always busy ($ET_{idle}^{GPU} = 0$) and the CPU might be occasionally idle. In our current implementation, the engine performs non-rendering code for the next frame when the GPU is still busy drawing the current frame. This is illustrated in Figure 4.

Note that some graphics commands require other buffers apart from the command buffer which can also become full and lead to idle time. The most important example are immediate-mode geometry commands in OpenGL, where the driver accumulates vertices in AGP or video memory buffers. These buffers are however usually quite small, so that the driver has to stall the CPU on most rendering commands and practically all parallelism between CPU and GPU is lost. Note that these commands are also undesirable because of the overhead imposed by the large number of API calls required. Geometry should therefore only be transferred under application control using extensions[4, 10], which also has the advantage that static geometry need not be sent every frame (see section 4.3.1).

4.5. GPU tasks

The rendering tasks on the GPU typically constitute the most important factors for the rendering time. We can identify the following tasks[11]:

Per-frame calculations. At the beginning of each frame, the frame buffer has to be cleared, and at the end of each frame, the back buffer has to be swapped with the front buffer. In a real-time setting, the swap should be synchronized with the vertical retrace of the screen in order to avoid tearing artifacts. While actual buffer swap times are negligible, clear times and buffer copy times (for windowed applications where buffer swapping is not possible) need to be taken into account in the estimation.

Per-primitive-group calculations. State setup including texture bind, material setup, vertex and fragment shader bind and shader parameters. The speed of this stage is determined by the number and type of state changes in a frame (with changes in vertex and fragment programs usually being the most expensive, followed by texture changes).

Per-primitive calculations are not fully developed in current graphics hardware. One example on current hardware is the adaptive refinement of triangles based on vertex normals[9].

Per-vertex calculations can be further broken down into index lookup (if an indexed rendering primitive is used) vertex fetching from video or AGP memory, and execution of the vertex shader. The time spent for these calculations is determined by the complexity of the vertex shader and the number of actually transformed vertices (see section 2), which can be determined by doing a FIFO-cache simulation for the geometry to be estimated. Note that one way to minimize the number of actually transformed vertices needed for a given geometry is to use a vertex-cache aware triangle stripper[3]. Note also that the vertex cache can only work for indexed primitives and when geometry is stored in AGP or video memory, therefore non-indexed primitives should only be used for geometry containing no shared vertices.

Triangle setup is the interface between per-vertex and per-fragment calculations. Triangle setup is usually not a bottleneck in current GPUs.

Per-fragment calculations or rasterization. These calculations are done by the fragment shader and subsequent stages. Examples include texture mapping, multi-texturing, environment mapping, shadow mapping etc. The speed of this stage is determined by the complexity of the fragment shader, but also by the efficiency of early fragment z-tests present in newer cards, and texture memory coherence.

In the following, we discuss some other factors influencing rendering time, including bottlenecks, rendering order and the type of memory used.

4.5.1. The myth of the single bottleneck in an application

A common misunderstanding with regard to the rendering pipeline is that the speed of an application is defined by a single bottleneck. While this is true for each particular point in time, the bottleneck can change several times even during the rendering of one single object. Due to the small size of post-transform vertex caches, the fragment and the geometry stage can work in parallel only for one particular triangle size, which depends on the complexity of the vertex and the fragment shader. Triangles larger than this "optimal triangle" usually stall the pipeline, whereas smaller triangles will cause the fragment stage to sit idle.

While for some far-away objects the rendering time is determined only by the geometry stage, most objects consist of several triangles larger than the optimal triangle and several ones that are smaller. An example is shown in Table 1, where neither vertex nor fragment shader can be made more complex for free (as should be the case for a single bottleneck). This effect needs to be incorporated in rendering time estimation heuristics. Our results indicate that a sum of fragment- and geometry terms might be better suited to estimate rendering time than taking the maximum of such terms[5]. A new heuristic based on this observation is introduced in section 5.3.

4.5.2. Rendering order

The rendering order can influence rendering time in two ways: First by the number of state changes required, which suggests that objects should be sorted by their rendering

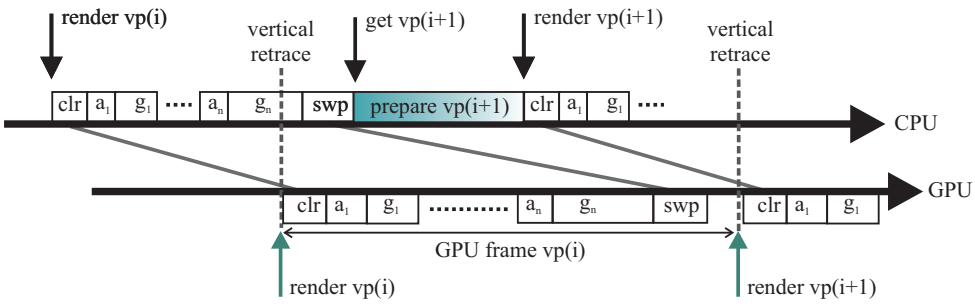

Figure 4: *This figure shows how CPU and GPU work in parallel. When the GPU starts rendering the current frame ($vp(i)$), the CPU has already written most of the rendering commands for this frame to the command buffer. When the CPU is finished with the current frame, it can prepare the next frame ($vp(i+1)$), starting with CPU-intensive non-rendering tasks and memory management, while the GPU is still busy rendering the current frame.*

	simple pixel	complex pixel	difference
simple vertex	4.969	7.35	2.381
complex vertex	9.859	11.856	1.997
difference	4.89	4.506	

Table 1: *This table shows the rendering time for one viewpoint in the terrain scene. If the complexity of the vertex shader is increased, the rendering time increases (independent of the fragment shader complexity). If the complexity of the fragment shader is increased the rendering time increases as well (independent of the vertex shader complexity). This shows the lack of parallelism between the fragment and geometry stages.*

modes. Second by the effect of pixel occlusion culling, which suggests that geometry should be rendered front to back in order to reduce the amount of fragments that actually need to be shaded.

Table 2 illustrates that mode sorting improves rendering time especially in CPU-limited cases because state changes are CPU intensive. The other test scenes do not profit from mode sorting because they either contain no state changes (terrain scene), or for other reasons (forest scene, no improvement although 194 state changes are reduced to 2), in which case dynamic mode sorting even increases rendering time.

Table 3 shows the effect of distance sorting in a strongly fill-limited setting, as compared to normal rendering (no sorting).

4.5.3. GPU memory management

There are several possible choices in which to send geometry to the graphics hardware, including whether to use

Sort order:	no sort	matrix/ texture	texture/ matrix	presorted text./mat.
texture changes	984	449	211	211
material changes	639	525	110	110
alpha changes	186	28	12	12
cpu limited:				
GPU time in ms	9.188	7.725	8.539	7.07
geometry limited:				
GPU time in ms	14.634	14.328	14.138	14.138

Table 2: *Examples for mode sorting in the city test scene in a CPU-limited and a geometry-limited setting (with a more complex vertex shader). The column headers indicate the sort order used (e.g., in the second column, first by transformation matrix, then by texture). In the presorted case, there is no CPU overhead.*

	normal	f2b	b2f
GPU time in ms	10.7	9.143	12.552

Table 3: *Distance sorting (front-to-back and back-to-front) in the city scene in a fill-limited setting.*

indexed or non-indexed primitives, which type of memory to use (AGP or video), which memory layout to use (interleaved/non-interleaved vertex formats) and which routine to use for the memory copy. The graphics hardware and the driver are usually poorly documented and the optimal choice can only be found by intensive testing.

The fastest type of memory is generally video memory, but it only comes in limited quantities, and streaming ge-

	rendering		copying	
	[ms]	[tv/sec]	[ms]	[MB/sec]
Test1: terrain video memory				
independent	4.36	19.73	1.7	793
parallel	4.59	18.75	1.9	733
Test2: terrain AGP memory				
independent	4.95	17.38	1.4	952
parallel	6.27	13.72	3.3	420
Test3: terrain AGP memory, interleaved				
interleaved AGP				
independent	4.41	19.52	1.4	952
parallel	5.04	17.06	2.9	481
Test4: city video memory CPU limited				
independent	9.28	11.878	4.1	793
parallel	14.167	7.78	5.1	644

Table 4: *The table shows rendering and memory copy times in two cases: independent shows the times for rendering and copying alone; parallel shows the slowdown when rendering and copying are done concurrently and compete for the same memory (tv/sec are the actually transformed vertices per second).*

ometry directly into video memory during rendering might reduce bandwidth for texture accesses. Table 4 shows that rendering from and copying to video memory can be done in parallel without the two influencing each other significantly. This means that geometry downloads to video memory don't increase the overall rendering time if there is enough CPU time left. However, when rendering is already CPU limited (Test4), the total rendering time increases by the full time needed for the copy. Therefore, dynamic geometry transfers should be avoided in this case and as much geometry as possible stored statically in video memory. Test2 shows that in the case of simple vertex shaders, using AGP memory can slow down the GPU due to the limited bandwidth of the AGP bus, especially when geometry is also copied to AGP memory in parallel. This has to be considered when working with animated meshes. An interesting observation (Test3) is that the geometry layout can influence the rendering speed so that terrain rendering with interleaved geometry from AGP memory is almost as fast as rendering from video memory. Note also that these results are close to the maximum transformation capability of the used rendering hardware.

5. Methods for rendering time estimation

In this section, we give several heuristics which can be used to calculate the rendering time estimation function. These heuristics are then compared and evaluated in section 7.

To obtain an estimation for the rendering time, we have to choose a method in a spectrum that is spanned by the extremes of *measuring* and *calculating*. We propose three basic methods: one sampling method that is mainly defined by measurements (and gives RT directly), another hybrid method that is a tradeoff between sampling and heuristic calculations and a third method that uses a heuristic function based on the number of the actually transformed vertices and rendered pixels (the latter two methods estimate ET_r^{GPU}, i.e., the other terms of RT have to be estimated separately as described in section 4).

5.1. View-cell sampling

The proposed sampling method works for a view-cell based system, where a potentially visible set (PVS) is stored for each view cell. For each view cell we discretize the set of view directions, randomly generate n views around each discretized direction and measure the rendering time for each view. The maximum rendering time of the n sample views is used as an estimation for the total rendering time RT of the direction and the view cell under consideration.

5.2. Per-object sampling

The hybrid method estimates the rendering time of a set of objects by adding the rendering time estimations of the individual objects. The assumption is that when two sets of objects are rendered in combination ($\mathbf{X}_1 \oplus \mathbf{X}_2$), the rendering time is at most linear with respect to the rendering times of the original sets $\mathbf{X}_1$ and $\mathbf{X}_2$.

$$ET_r^{GPU}(\mathbf{X}_1 \oplus \mathbf{X}_2) \leq ET_r^{GPU}(\mathbf{X}_1) + ET_r^{GPU}(\mathbf{X}_2)$$

To estimate the rendering time of a single object, we parameterize the rendering time estimation function by three angles $ET_r^{GPU}(x) = ET_r^{GPU}(x, \alpha, \gamma, \phi)$ (see Figure 5 for a 2D view). The angle α is the angle between the two supporting lines on the bounding sphere. This angle (which is related to the solid angle) is an estimate for the size of the screen projection. The angles γ and ϕ (for elevation) describe from which direction the object is viewed. In a preprocess, we sample this function using a regular sampling scheme and store the values in a lookup table together with the object. As the angle α becomes smaller, the rendering time will be geometry limited and not depend on the viewing parameters any more. We use this observation to prune unnecessary test measurements. This rendering time estimation can be used in two ways:

Per-viewpoint estimation: For an online estimation, the

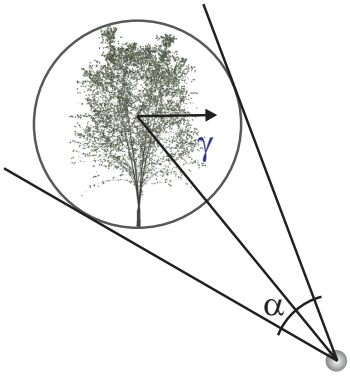

Figure 5: *This figure shows precalculated per-object sampling. The rendering time estimation is parameterized with three angles, two of those are shown in this 2D figure.*

rendering time is looked up in the table stored with the object with respect to the current viewpoint and view direction. Care has to be taken not to make the estimation overly conservative: For objects straddling the view frustum, the angle alpha has to be clipped to the view frustum. This is especially important for nearby objects since they tend to cover a larger screen area, i.e., they usually contain several large rasterization-limited triangles which add significantly to the estimated time.

Per-view-cell estimation: For a view cell, the rendering time estimation is more involved. As in view-cell sampling (section 5.1), we discretize the set of possible viewing directions from a view cell. For each discretized direction, a conservative estimate of the rendering time is calculated separately in the following way: We seek for each object the point on the view-cell boundary where the object bounding sphere appears largest in the viewing frustum. This point lies either on a boundary vertex of the view cell, or on a boundary face such that the view frustum given by the viewing direction is tangent on the object bounding sphere. The rendering time estimation associated with this point is then looked up as in the per-viewpoint estimation, and added to the estimated rendering time for this viewing direction. Finally, the rendering time estimation for the whole view cell is calculated as the maximum of the rendering time estimations from the discretized viewing directions.

5.3. Mathematical heuristics

As the last model, we compare several mathematical heuristics for the rendering time estimation ET_r^{GPU}. Note that in previous work, the first 3 heuristics shown here were used alone, without regard for the other components of *RT* described in this paper. We propose to the presented heuristics within the complete rendering time estimation framework, which allows taking into account effects like texture and geometry management etc.

The first heuristic **H1** is the *triangle count*[1]. The assumption on which this heuristic is based is that the ratio of actually transformed vertices to triangles is uniform over the whole scene, and that the rendering time is determined by the geometry stage.

$$ET_r^{GPU}(x) = c * \#tris$$

where $\#tris$ is the triangle count of an object, and c is the triangle rate for a given hardware.

The second heuristic **H2** is the actually transformed vertex count[8].

$$ET_r^{GPU}(x) = c * tv$$

where tv is the number of actually transformed vertices and c is the vertex rate for a given hardware. This heuristic reflects the geometry processing stage more accurately than **H1**, but still neglects the influence of rasterization on rendering time.

A more complete heuristic (**H3**) is Funkhouser's cost function[5] with

$$ET_r^{GPU}(x) = max(c_1 * \#polys(x) + c_2 * \#v(x), c_3 * \#pix(x))$$

where x is the object under consideration, $\#polys$ is the number of polygons of the object, $\#v$ is the number of vertices of the object and $\#pix$ is the number of pixels in the projection.

As was discussed in section 4.5.1, the bottleneck in a rendering pipeline can shift several times even when rendering a single object. The *maximum* of geometry and rasterization terms as used in **H3** is actually a lower bound for the actual rendering time, whereas the *sum* of the two terms constitutes the upper bound, and is therefore a more conservative estimation. The experiments in section 4.5.1 also suggest that in practice, the actual rendering time often tends towards the sum. Furthermore, the factor which determines geometry transformation time is the number of actually transformed vertices and not just the number of vertices or polygons. Based on these two observations, we propose a new rendering time estimation heuristic **H4** which improves upon the previous ones:

$$ET_r^{GPU}(x) = c_1 * \#tv(x) + c_2 * \#pix(x)$$

The heuristics presented here will be compared in section 7. However, none of the rendering time estimation functions shown in this section is sufficient to build a soft real-time system, either because timing results are not accurate enough with current timing methods (for the sampling approaches) or because estimated rendering times can sometimes be exceeded in practice even if the prediction is very

accurate (for the mathematical heuristics). In the next section, we will propose hardware extensions to overcome these problems.

6. Hardware extensions for a soft real-time system

In this section, we introduce two hardware extensions that make it possible to implement a soft real-time system. The extensions deal with two problems encountered when using rendering time estimation: the timing accuracy problem and the estimation accuracy problem.

6.1. The timing accuracy problem

All the heuristics presented in section 5—especially the per-object sampling method—rely to some extent on the ability to measure the rendering time for specific objects. While it is relatively easy to measure the time taken by a specific CPU task, it is very difficult to obtain such measurements for GPU-related tasks. CPUs provide an accurate time-stamping mechanism via an instruction that returns the current CPU clock cycle counter. Inserted before and after a number of instructions, the difference of the counters can be used to calculate the time required for executing the instructions.

The GPU, however, is a separate processing unit, and any timing mechanism implemented on the CPU will either give only information about the interaction with the command buffer, or include significant overhead if the GPU is explicitly synchronized before each timing instruction (e.g., via the OpenGL command `Finish`).

6.2. The time-stamping extension

Due to the large uncertainties that such inaccurate timing can introduce in the rendering time estimation, it seems useful to implement timing directly on the GPU and to extend current hardware with a *time-stamping function*, which can be used for acquiring the accurate measurements needed to set up the rendering time estimation functions in section 5 (either for sampling or for calibrating one of the heuristic formulae).

This function should operate similarly to the OpenGL occlusion-culling extension: A time-stamp token is inserted into the command buffer, and when this token is processed by the GPU, a time stamp representing the current GPU time (e.g., a GPU clock cycle counter) is stored in some memory location and can be requested by an asynchronous command. Due to the long pipelines present in current GPUs, there are two possible locations for setting the time stamp: (1) at the beginning of the pipeline (when the token is retrieved from the command buffer), and (2) at the end of the pipeline, which is the more accurate solution. The difference between two such time stamps can then be used to calculate the time required for the GPU to render an object much in the same way as for a CPU to execute some instructions. The communication paths between the backend of the GPU and the CPU necessary for such an extension are already in place (they are used to transmit pixel counters used in occlusion queries).

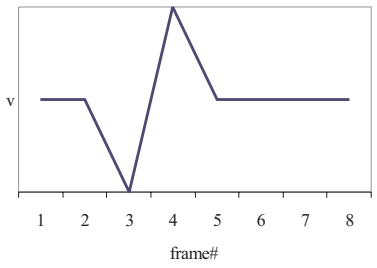

Figure 6: *This diagram shows the strong variations in perceived movement speed during a frame skip (frame 2 is repeated).*

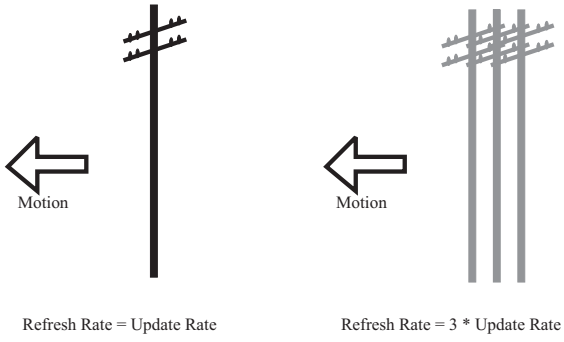

Figure 7: *Ghosting artifact when the frame rate doesn't match the screen refresh rate (recreated after an image from [7]). Such artifacts are especially visible on sharp edges in the images, and when the viewer is rotating.*

6.3. The estimation accuracy problem

Funkhouser[5] notes that for the metric **H3**, the actual rendering time does not deviate more than 10% from the predicted rendering time in 95% of frames. However, this means that in a real-time setting assuming a frame rate of 60 Hz, there would be up to 3 skipped frames per second. Figure 6 shows the strong variations in perceived movement speed caused by a single frame skip. In the accompanying video, we show that even one skipped frame every several seconds is unacceptable if a moderately smooth walkthrough is desired. The video also shows that switching to a lower frame rate (such as 30 Hz) for a longer duration is not acceptable either, due to ghosting artifacts which appear when the frame buffer is not updated at each screen refresh (see Figure 7).

For a quality real-time application we aim for an estimation accuracy higher than 99.9%, or a maximum of 1-2 frame

skips per minute. Even the metric **H4** or the per-object sampling method using the time-stamping extension (both introduced in this paper) will not be able to provide such an accurate estimation. We therefore propose a hardware extension that can "fix" estimation errors during the rendering process.

6.4. The conditional branching extension

We propose using a *conditional branch* in graphics hardware to switch to a coarser LOD if the remaining frame time is not sufficient. Such a conditional branching extension consists of a start token containing a sequence of numbers $t_{r_1}, \ldots, t_{r_n}$, and a branch token. When the GPU encounters the start token, it compares each t_{r_i} with the time remaining until the next vertical retrace. If t_{r_j} is the first number that is smaller than this time, all commands in the command buffer up to the j-th encountered branch token are skipped. Then all commands until the next branch token are executed, and finally all commands until the n-th branch token are skipped again.

The values t_{r_i} should be set to the result of the rendering time estimation function for all remaining objects for this frame, including the current object at a specific level of detail i. Since any tasks done by the driver on the CPU side will have to be executed for all conditional branches, such branches should only contain rendering commands which refer to geometry already stored in AGP or video memory, which can be accessed by the GPU directly.

6.5. Soft real-time system

Using the two proposed extensions, a soft real-time system can be implemented. The time-stamping extension guarantees accurate timings for the rendering time estimation function. Based on this function, appropriate LODs are selected for each object in each frame in such a way that the total frame time is not exceeded (there are several ways how to do that, but the LOD selection process is not topic of this paper). To guarantee that no frame is skipped even if the rendering time estimation fails, some objects (starting with those that (1) are already at a certain distance from the viewer, to reduce popping, and (2) still have some geometric complexity so that their rendering time is not negligible) are accompanied with a small number of coarser LODs, which are automatically selected by the graphics hardware if the remaining time is not enough to render the predetermined LOD. Note, however, that such a system is still a soft real-time system because frame skips can still occasionally occur (e.g., due to unforeseeable stalls caused by the operating system), but they will be reduced to a negligible number.

In order to guarantee that no geometry has to be transferred over the bus for objects that are not actually rendered, all LODs for all objects could be stored in GPU-accessible memory at the beginning of the walkthrough, if there is sufficient memory. In general, the geometric data for the LODs will be managed along with other geometric data as explained in section 4.3.1. We do not expect the additional data required by the LODs to be a significant burden on the bandwidth between CPU and GPU, since not all LODs for each object, but only a small number of additional LODs will be used. In an analogy to mip-mapping, the memory required for the lower LODs should not exceed the main model—this holds true for both discrete LODs (where successive levels should differ by a significant number of triangles) and progressive LODs (where lower levels are part of the higher levels).

7. Implementation and results

7.1. Test setup and models

The empirical values for all tables in this paper were obtained on an Athlon XP1900+ with an NVIDIA GeForce3 graphics card. The graphics API was OpenGL[15] and the operating system Windows 2000. All timings were taken by synchronizing the pipeline (using the `glFinish` instruction) and reading the processor clock cycle counter both before and after rendering the object to be timed. The object was actually rendered several times between the two readings in order to minimize the influence of the synchronization overhead on the timing.

We will shortly describe the models that we used for the measurements in the paper (see also Table 5). The first scene is a model of a city (see Figure 8), our main benchmark. A PVS is calculated for each cell in a regular grid of 300x300 $10m^2$ view cells[14]. The second model is a textured terrain from another city (see Figure 9). The third model is a forest scene (see Figure 10) consisting of 157 trees with varying complexity. Each object in each scene consists of one or several lists of indexed triangle strips, where each strip can have a different texture. Visibility and view-frustum culling works on the object level. The ratio of actually transformed vertices per triangle shows how well the scene is adapted to exploiting the vertex cache. State switches and triangles per state switch indicate the traversal overhead on the CPU and the GPU.

7.2. Comparing the per-viewpoint estimation methods

To compare the quality of a prediction function, we conducted the following test. We recorded a path through the test scenes and compared the prediction function for each viewpoint with the actual frame time. We compare the time for all four mathematical heuristics (h1–h4) and per-object sampling (pos). The constants in the mathematical heuristics were determined using linear regression for 10,000 test measurements of different objects.

For each frame we calculate the squared difference between estimated and measured frame time and take the average of these values as a criterion for the quality of the render-

value	city	terrain	forest
#objects	5609	1	157
#triangles	3,453,465	86,400	579,644
#tv	6,419,303	86,000	1,028,978
#tv / tri	1.86	0.99	1.78
#textures	88	1	2
#state switches (sw)	14267	1	314

Table 5: *Some descriptive values for the test scenes. #objects is the number of objects in the scene that are handled by the occlusion-culling or view-frustum culling algorithms. #tv is the number of actually transformed vertices.*

	h1	h2	h3	**h4**	**pos**
city	3.27	0.7	1.12	**0.26**	1.41
forest	30.35	25.5	20.8	20.5	**5.41**

Table 6: *Average squared errors for per-viewpoint estimations (ms^2). The newly introduced heuristics h4 and pos (per-object sampling) show improvements over the previous ones.*

ing time estimation (Table 6). We use the city and the forest scene for this test.

In the city scene we can see that the proposed per-object sampling heuristic provides reasonable results, but due to the timing inaccuracy problem it is inferior to all heuristics except for the triangle count. The forest scene is more challenging to predict and here per-object sampling performs much better than the other algorithms. The newly proposed heuristic h4 based on adding geometry and rasterization terms performs better than the previous mathematical heuristics in both scenes, and is significantly better than all other heuristics in the city scene.

7.3. Comparing the per-view-cell estimation methods

For the city scene, we precalculated a PVS for a regular view-cell subdivision. Then we recorded a 1000 frame walkthrough for the per-view-cell sampling method and the per-object sampling method. Per-view-cell sampling underestimated 2 frames and resulted in an average squared error of 0.64. The per-object sampling method underestimated no frame, but due to the timing accuracy problem the average squared error was increased to 2.33.

7.4. Hardware extension

In the accompanying video, we simulated the conditional branching extension by randomly selecting some objects for

Figure 8: *A snapshot of the city scene.*

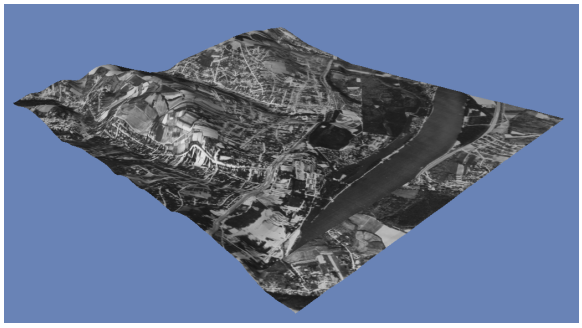

Figure 9: *A snapshot of the terrain scene.*

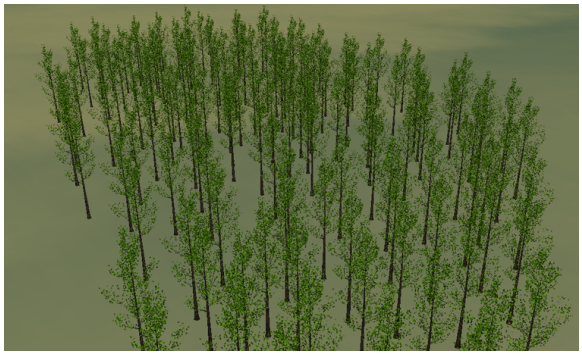

Figure 10: *This figure shows a snapshot of the forest scene. Note that each tree consists of 6654 actually transformed vertices.*

some frames which receive a different LOD than the designated one, as would be the case if the rendering time estimation were incorrect. This effect is compared to the conventional frame-skip effect.

7.5. Discussion

The results obtained during our tests, especially for the per-viewpoint estimation methods, make it difficult to give a unique recommendation on which rendering time estimation to use. For example, the per-object sampling method (pos) gives better results than the mathematical heuristics in the forest scene, but is not so well suited for the city scene. The reason for this are the timing inaccuracies discussed in section 6.1. The individual objects in the forest scene consist of a much higher number of primitives than in the city scene, which reduces the influence of the inaccuracies and makes the estimations of the individual objects more precise. The mathematical heuristics, on the other hand, do not perform so well on the forest scene because this scene contains triangles of strongly varying screen projections, which influences the rendering time in ways that a single analytic formula is not able to capture.

Another factor which will influence the decision on a suitable rendering time estimation method is the involved computational effort. While the per-object sampling method can provide potentially superior results (see the forest scene), it also requires a costly sampling step for each individual object.

In summary, we recommend the per-object sampling method in cases when an exact estimation is critical and the mathematical heuristics fail. This is likely to happen for objects with many triangles of strongly varying screen projections. In all other cases, the newly proposed heuristic h4 presents an improvement over previous heuristics. However, when the hardware extensions proposed in this paper are available, the per-object sampling estimation will be significantly more attractive, if one is wiling to invest the effort in preprocessing.

8. Conclusions

In this paper, we introduced a framework for estimating the rendering time for both viewpoints and view cells in a real-time rendering system. We showed that previous work only captures certain aspects of the rendering time estimation and is only applicable under controlled circumstances. However, our aim is to achieve a system with a constant frame rate of at least 60 Hz in order to avoid annoying ghosting artifacts and frame skips. We propose several new rendering time estimation functions to be used in our framework, including one based on per-object sampling (pos), and one based on an improved mathematical heuristic (h4), and demonstrated their applicability in a walkthrough setting. While the new heuristics showed significant improvements over previous methods in some cases, we also observed that their effectiveness is hampered by limitations in current graphics hardware, which is not suited for constant frame rate systems. We therefore propose two hardware extensions that remedy this problem, one for accurate timing measurements, and one for conditional branches in the rendering pipeline.

References

1. Daniel G. Aliaga and Anselmo Lastra. Automatic image placement to provide a guaranteed frame rate. In *SIGGRAPH 99 Conference Proceedings*, pages 307–316, 1999. 1, 2, 8
2. John Carmack. Plan update 03/07/00, 2000. available at http://finger.planetquake.com/plan.asp?userid=johnc&id=14310. 4
3. NVIDIA Corporation. Nvidia developer website, nvtristrip 1.1, 2003. http://developer.nvidia.com/. 5
4. NVIDIA Corporation. Nvidia opengl specifications, vertex array range extension, 2003. available at http://developer.nvidia.com/. 5
5. Thomas A. Funkhouser and Carlo H. Séquin. Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In *SIGGRAPH 93 Conference Proceedings*, pages 247–254, 1993. 1, 2, 5, 8, 9
6. N. Gartner, C. Messer, and A. Rathi. Traffic flow theory. Technical report, Turner-Fairbank Highway Research Center, 1993. 4
7. James L. Helman. Architecture and performance of entertainment systems, appendix a. *ACM SIGGRAPH 95 Course Notes #6*, pages 1.19–1.32, 1995. 1, 9
8. Hugues Hoppe. Optimization of mesh locality for transparent vertex caching. In *SIGGRAPH 99 Conference Proceedings*, pages 269–276, 1999. 2, 8
9. ATI Technologies Inc. Truform – white paper. http://www.ati.com/technology/, 2001. 5
10. ATI Technologies Inc. Ati opengl extensions, vertex array object extension. http://www.ati.com/, 2003. 5
11. Kekoa Proudfoot, William R. Mark, Pat Hanrahan, and Svetoslav Tzvetkov. A Real-Time procedural shading system for programmable graphics hardware. In *SIGGRAPH 2001 Conference Proceedings*, pages 159–170, 2001. 5
12. Matthew Regan and Ronald Pose. Priority rendering with a virtual reality address recalculation pipeline. In *SIGGRAPH 94 Conference Proceedings*, pages 155–162, 1994. 2
13. John Rohlf and James Helman. IRIS performer: A high performance multiprocessing toolkit for real–Time 3D graphics. In Andrew Glassner, editor, *SIGGRAPH 94 Conference Proceedings*, pages 381–395, July 1994. 2
14. Peter Wonka, Michael Wimmer, and Dieter Schmalstieg. Visibility preprocessing with occluder fusion for urban walkthroughs. In *Rendering Techniques 2000*, pages 71–82, 2000. 10
15. M. Woo, J. Neider, T. Davis, and D. Shreiner. *OpenGL Programming Guide*. Addison Wesley, 1999. 10