Adaptive CPU Scheduling to Conserve Energy in Real-Time Mobile Graphics Applications

Fan Wu, Emmanuel Agu, and Clifford Lindsay

Worcester Polytechnic Institute, Worcester, MA 01609

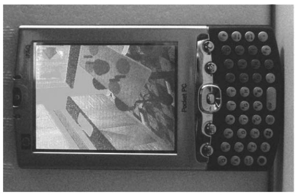
Abstract. Graphics rendering on mobile devices is severely restricted by available battery energy. The frame rate of real-time graphics applications fluctuates due to continual changes in the LoD, visibility and distance of scene objects, user interactivity, complexity of lighting and animation, and many other factors. Such frame rate spikes waste precious battery energy. We introduce an adaptive CPU scheduler that predicts the applications workload from frame to frame and allocates just enough CPU cycles to render the scene at a target rate of 25 FPS. Since the applications workload needs to be re-estimated whenever the scenes LoD changes, we integrate our CPU scheduler with LoD management. To further save energy, we try to render scenes at the lowest LoD at which the user does not see visual artifacts on a given screen. Our integrated Energy-efficient Adaptive Real-time Rendering (EARR) heuristic reduces energy consumption by up to 60% while maintaining acceptable image quality at interactive frame rates.

Keywords: Energy conservation, Multiresolution rendering,Real-time rendering, CPU scheduling.

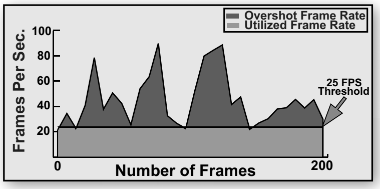
1 Introduction

Battery-powered mobile devices, ranging from laptops to cell phones, have become popular for running 3D graphics applications that were previously developed exclusively for desktop computers or gaming consoles. Mobile devices now feature more processing power, programmable graphics hardware and increased screen resolution. Emerging mobile graphics applications include multiplayer games, mobile telesurgery and animations. Mobile graphics applications offer a new commercial opportunity especially considering the total number of mobile devices sold annually far exceeds the number of personal computers sold. The mobile gaming industry already reports revenues in excess of $2.6 billion worldwide annually, and is expected to exceed $11 billion in 2010 [1].

The most limiting resource on a mobile device is its short battery life. While mobile CPU speed, memory and disk space have grown exponentially over the years, battery capacity has only increased 3-fold in the past decade. Consequently, the mobile user is frequently forced to interrupt their mobile graphics experience to recharge dead batteries.



(a) Screenshot on a HP iPaq Pocket PC



(b) Application running at high real-time frame rate

Fig. 1. Screenshot of HP Ipaq & Example Application Framerate

Application-directed energy saving techniques have previously been proposed to reduce the energy usage of non-graphics mobile applications. Our main contribution in this paper is the introduction of application-directed energy saving techniques to make mobile graphics applications more energy-efficient. The main idea of our work is that energy can be saved by scheduling less CPU timeslices or lower the CPU's clock speed (Dynamic Voltage and Frequency Scaling (DVFS)) for mobile applications during periods when its requirements are reduced.

In order to vary the CPU timeslices alloted to a mobile application, we need to accurately predict its workload from frame to frame. Workload prediction is a difficult problem since the workload of real-time graphics applications depends on several time-varying factors, such as user interactivity level, the current Level-of-Detail (LoD) of scene meshes and mip-mapped textures, visibility and distance of scene models, and the complexity of animation and lighting. Without dynamically changing the application's CPU allotment to correspond to its needs, the mobile application's frame rate fluctuates whenever there is a significant change in scene LoD, animation complexity, or other factors that affect its workload. Such spikes above 25-30 Frames Per Second (FPS) drain the mobile device's battery and increased energy consumption by up to 70% in our measurements (see figure 1b). We propose an accurate method to predict the mobile application's workload and determine what fraction of the CPU's cycles it should be alloted to maintain a frame rate of 25 FPS. As the application's workload changes, we update its CPU allotment at time intervals determined by a windowing scheme that is sensitive to applications with fast-changing workloads and prudent for applications with slow-changing workloads. Our adaptive CPU scheduling scheme dampens frame rate oscillations and saves energy.

Since the application's workload changes and should be re-estimated whenever LoDs are switched, we have coupled our CPU scheduler with the application's LoD management scheme. When switching scene LoD, we minimized energy consumption by selecting the lowest LoD at which the user does not see visual artifacts, also known as the Point of Imperceptibility (PoI) [2]. Although our primary goal was to minimize the mobile application's energy consumption, we also ensured that the frame rates and visual quality of the rendered LoD were acceptable. In summary, our integrated EARR (Energy-efficient Adaptive

Real-time Rendering) heuristic minimizes energy consumption by i) selecting the lowest LoD that yields acceptable visual realism, ii) scheduling just enough CPU timeslices to maintain real-time frame rates (25 FPS). EARR also switches scene LoD to compensate for workload changes caused by animation, lighting, user interactivity and other factors outside our control. To the best of our knowledge, this is the first work to use CPU scheduling to save energy in mobile graphics. Our results on animated test scenes show that CPU scheduling reduced energy consumption by up to 60% while maintaining real time frame rates and acceptable image realism. The rest of the paper is organized as follows. Section 2 presents related work and background, Section 3 through Section 5 describes our proposed EARR heuristic, Section 6 describes our experimental results. and section 7 presents conclusions and future work.

2 Related Work and Background

Application-Directed Energy Management: This class of energy management schemes uses Dynamic Voltage or Frequency Scaling (DVFS) [3,6] or intelligently reduces the application's output quality to conserve energy [4,5]. For instance, energy can be saved by intelligently reducing video quality [5] or document quality [4]. In DVFS, energy is conserved by dynamically reducing the processor's speed or voltage, without degrading the application's quality. GRACE-OS [3] proposes a DVFS framework for multimedia applications, which probabilistically predicts the CPU requirements of multimedia applications in order to guide CPU speed settings. Chameleon [6] proposes CPU scheduling policies for soft real-time (multimedia), interactive (word processor) and batch(compiler) applications. To the best of our knowledge, adaptive CPU scheduling to conserve energy has not previously been applied to graphics applications.

LoD management to maintain real-time frame rates: Funkhouser and Sequin [7], and Gobetti [9] both describe systems that bound rendering frame rates by selecting the apprioprate LoD. While Funkhouser and Sequin used discrete LoDs, Gobetti extended their work by using multiresolution representations of geometry. Wimmer and Wonka [10] propose estimating the upper limit of rendering times. The Performer system maintains a specified rendering speed [12], by switching LoDs whenever application frame rate changes. Tack et al [11] describe a model for predicting rendering times on mobile devices.

Background: In order to minimize the amount of energy consumed our technique uses a combination of Adaptive CPU scheduling, LoD management using Wavelets, and the Point of Imperceptibility (PoI) metric. Our goal during scheduling is to assign the lowest fraction of CPU time slices that runs the application at the selected LoD as close to 25 FPS as possible. To further refine our ability to adjust scene complexity in order conserve energy, we can adjust Wavelet levels for two situations. First, we minimize energy usage by continuously choosing the lowest mesh resolutions that is visually acceptable. Second, we can switch to lower mesh LoDs is to compensate when animation, lighting and other scene elements not in our control and require more CPU cycles. In order to quantify the distortion caused by mesh simplification we use the perceptual metric proposed by Wu *et al* [2] that generates a PoI LoD of a mesh. This metric factors in the effects of lighting, shading and texturing on the perceptibility of simplification artifacts and exploits knowledge of how human perception works to pinpoints the lowest LoD at which mesh simplification artifacts are imperceptible on mobile displays of different resolutions.

3 Our EARR Heuristic

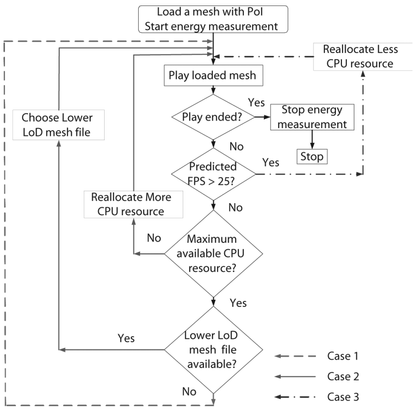
The accurate prediction of the application's CPU requirements in advance is important in our work because allotting CPU cycles after the application's demands increase causes jerkiness or frame rate to drop. In complex real-time graphics applications, it is difficult to accurately model and predict all factors that affect the observed frame rates. Using a statistical workload predicting model, we developed an heuristic that is both efficient to compute and accurate. Through a series of steps described below, our EARR heuristic compares the predicted with the actual frame rate and adaptively adjusts future predictions, mesh LoDs, and CPU resource allocations to minimize energy consumption.

At the start of the heuristic, all meshes are rendered at their PoI LoD. As the mesh moves during an animation, EARR reallocates CPU resources using the workload predicting model and the CPU scheduling policy. There are three cases to which our heuristic is required to adjust the application parameters, each requiring different actions. If we let d denote the current LoD of a mesh and d_p denote its PoI LoD. Let f denote the frame rate at which that scene is currently being rendered. The three cases are as follows:

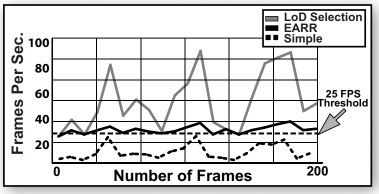
Case 1: the predicted frame rate drops such that $f_i < 25$, current LoD $i =$ minimum LoD possible, and 100% of CPU cycles already alloted to this task: In this case, we are at the limits of the parameters under our control (minimizing LoD and maximizing CPU cycles). We conclude that the mobile's resources are not enough to render the scene at 25 FPS and we cannot rectify the situation. In such a scenario, we simply choose the minimum possible LoD and set the CPU cycles to a maximum and achieve the highest frame rate possible.

Case 2: the predicted frame rate drops such that $f_i < 25$, current LoD $i =$ PoI, d_p: In this case, the heuristic will allocate more CPU resources to increase the rendering frame rate. If the frame rate is still less than 25 FPS, the heuristic will then choose a lower LoD level to increase the frame rate to 25 FPS and allocate the optimal fraction of CPU cycles, C_{opt} accordingly. We note that in this case, to achieve 25 FPS, we are forced to use an LoD below the mesh PoI, which will cause visual artifacts.

Case 3: the predicted frame rate increases such that $f_i >> 25$, current LoD i = PoI, d_p: EARR continues to use the PoI LoD but tries to save energy by reducing the percentage of CPU timeslices scheduled for our application to the minimum required in order to maintain a frame rate of 25 FPS. Figure 2a is the flow chart of EARR heuristic.



(a) EARR Heuristic Flow Chat



(b) Frame Rates at check points along animation path

Fig. 2. Heuristic and Frame rate

In the following sections, Section 4 describes our workload predicting model and Section 5 describes the CPU scheduling policy.

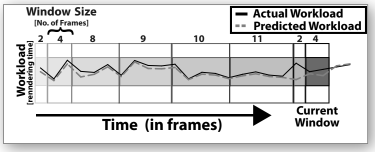
4 Workload Predicting Model

The workload predicting model predicts what fraction of available CPU timeslices should be alloted to our mobile application in order to sustain a target frame rate of 25 FPS. To minimize energy consumption, the goal of the CPU scheduler is to allocate *just enough* CPU cycles to finish rendering each frame *just before* its deadline expires. Our target frame rate of 25 FPS yields a deadline of 40 ms for each frame to complete rendering. The optimal (fewest) CPU resources C_{opt} to meet our task's deadline can be expressed as:

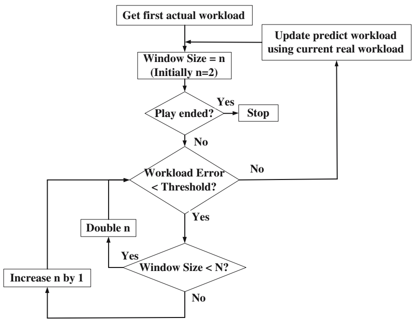
$$C_{opt} = \frac{\tau}{r_{max}} \times C_{max} \tag{1}$$

where C_{max} is the maximum available allotment of the processor's timeslices, C_{opt} is a reduced allotment of CPU timeslices generated by our algorithm, which just meets the frame's deadline. r_{max} is the rendering time of a mesh if all available processor cycles are alloted to our application and τ is the deadline for the frame. As mentioned above, $\tau = 40$ ms. We apply our workload predictor as follows. At runtime, given a frame rendering deadline, τ, we use equation 1 to calculate the optimal CPU processor allocation, C_{opt}. We then use our pre-generated statistics to estimate the meshes LoD that corresponds to C_{opt}.

A complex scene typically contains multiple objects at different LoDs. The workload for rendering the scene depends not only on the total number of scene triangles, but also on their visibility, which varies as the camera and objects move. Thousands of triangles might be visible from some camera positions,



(a) Window Size Updates

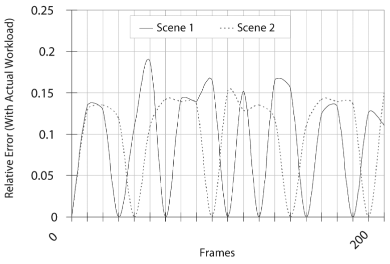

Get first actual workload
Update predict workload using current real workload
Window Size = n (Initially n=2)
Play ended?
Yes
Stop
No
Workload Error < Threshold?
No
Yes
Double n
Yes
Window Size < N?
No
Increase n by 1

(b) Flow Chart for Workload Predicting Model (N=8)

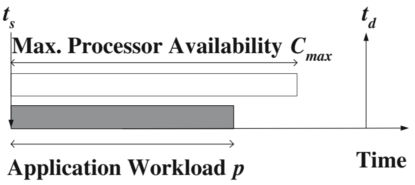
Fig. 3. Window Size Update and Flow Chart

whereas just a few may be seen from others. We used the *eye-to-object* visibility algorithm described in [13] to determine the set of visible objects to be rendered in each frame and calculated the entire scene's workload as the sum of all visible objects where each object's workload is determined by the method of Funkhouser and Sequin [7], which suggested that the number of triangles in a mesh is a good predictor of its rendering time. To characterize the accuracy of the method, relative error between measured rendering times and predicted rendering times produced by our workload predictor for a single model, was calculated for various LoDs. The results corroborates the results of Funkhouser and Sequin [7] since all relative errors are less than 4%. The eye-to-object visibility algorithm culls away large portions of a model that are occluded from the observer's viewpoint, and thereby improves the accuracy of workload estimation significantly. In the eye-to-object algorithm, the scene can be subdivided into cells, and the model partitioned into sets of polygons attached to each cell. After subdivision, cell-to-cell and eye-to-cell visibility are computed for each cell of subdivision. This algorithm has been previously used to accelerate a range of graphics computations, including ray-tracing and object-space animation.

Thus far, our predictor focused on the workload for rendering one frame of a scene. Next, we consider changes in application's workload over time. Since the application workload changes only slightly from one frame to the next (milliseconds), we maintain the current predicted workload for n future frames, where n is called the *window size* and is varied depending on how quickly the workload is changing. The choice of n affects the performance of our algorithm. If n is too small, then we update predicted workload too often incurring large computation overheads. If it n is too large, then the system may not be sensitive enough to fast-changing workloads and the error between the predicted and actual workload may become too high. Therefore, in our prediction model, the window size (n) is adaptively varied at run-time. Figure 3a shows our window size updates method, which is inspired by the Transmission Control Protocol (TCP) in networking. Initially, we calculate and update our predicted workload every two



(a) Accuracy of Workload Predicting Model



(b) Processor Availability Vs. Workload

Fig. 4. Accuracy of Workload Predicting model and Processor Availability

frames ($n = 2$). Every time the error between the predicted and actual workload is smaller than a pre-determined threshold, the window size is doubled. We continue to double the window size until $n = 8$, beyond which the window size is increased by 1 every time the observed error falls within an acceptable threshold. Whenever the workload prediction error exceeds the threshold, we reset the window size to 2 and set predicted workload to the value of the currently observed workload value. Figure 3b is a flow chart of the workload predicting model.

The adaptive workload predictor estimates the workload of each frame at full processor speed, from which we can estimate the CPU timeslices required to render a frame at our target frame rate. We tested our technique with two scenes provided by the Benchmark for Animated RayTracing(BART) [8], The results are shown in figure 4a. It can be observed that the relative errors are bounded in 0.18.

5 CPU Scheduling Policy

Our CPU scheduler runs a three-phase, which are workload estimation, estimating processor availability and determining processor resource allocation. We now formalize our CPU scheduling algorithm. For each real-time task T, let us denote its start time by t_s and its deadline as t_d. Let C_{max} denote the maximum fraction of CPU timeslices that are currently available for running applications. It is important to note that without the intervention of our scheduling algorithm, all tasks will run with 100% allocation of all available CPU timeslices, C_{max}. The fraction of CPU timeslices required by T will be denoted by p. We note that the execution time of the task T is inversely proportional to p. In summary, a feasible schedule of the task guarantees that the task T receives at least a fraction, A, of the maximum available CPU cycles such that it receives $A * C_{max}$ CPU cycles before its deadline, where $A \leq 1$. Given the application workload p, the maximum processor availability C_{max} and interactivity deadline t_d, as shown in figure 4b, our allocation policies fall into two distinct cases.

$$A = \frac{p}{min(C_{max}, t_d - t_s)} \tag{2}$$

$$C_{opt} = \begin{cases} C_{max} & : \quad C_{max} < \hat{p} \\ min(C_{max} \times \frac{\hat{p}}{min(C_{max}, t_d - t_s)}, C_{max}) & : \quad otherwise \end{cases} \tag{3}$$

Case 1: If $C_{max} < p$, then the application's demand for CPU timeslices exceeds CPU availability. In this case, the CPU scheduler cannot meet the task's deadline while using the current mesh LoD. Our scheduling algorithm shall allot all available CPU timeslices to the task and also reduce mesh LoD to lower the offered workload p.

Case 2: If $t_s + p < t_d$, the task can complete before its deadline. If all available CPU resources are alloted to this task, the rendering speed achieved is larger than 25 FPS. In this case, the algorithm reduces the fraction of CPU timeslices alloted such that the demanded workload p is just adequate to complete the task before its deadline. The percentage of CPU resources alloted is calculated in equation 2.

In the equation 2, the deadline $t_d - t_s$ is known. We chose $t_d - t_s$ as 40 ms, p is determined by using our workload predictor. The maximum CPU resources currently available, C_{max} can be monitored by our resource adapter. Given an estimated demanded workload, $\hat{p}$ and the maximum processor availability, C_{max}, the optimal CPU resource allocation, C_{opt}, is computed in equation 3.

6 Experiment and Results

In this section we describe performance of the EARR heuristic on both a laptop and a PDA. The laptop used was a Windows Vista Lenovo T61 laptop equipped with an Intel Core 2 Duo 2.1GHz processor and 3GB RAM. The PDA is a Windows CE HP iPAQ Pocket PC h4300 with a 400 MHz Intel XScale processor and 64MB RAM. We repeated all experiments eight times, eliminated the minimum and maximum values before averaging all others. We animated a pre-determined animation path in the kitchen scene provided by the Benchmark for Animated RayTracing (BART) [8]. We ran three sets of experiments using the BART kitchen scene, applying three levels of adaptations: (1)**Simple: No LoD switching, no adaptive CPU scheduling;** (2)**LoD Selection: LoD switching, no adaptive CPU scheduling;** (3)**Our EARR heuristic: LoD selection with adaptive CPU scheduling.**

Measuring the exact energy consumption of the CPU alone is a fairly difficult problem. To measure the energy consumption of the CPU independent of our experiments we subtracted the base idle energy consumption to give our applications power usage and multiplied power usage by execution time to arrive at energy consumption. In our experiment, the base power consumed by the laptop in idle mode was 12.58W. During our experiments, we set 20 check points along the animation path of the mesh. Figure 2b is a plot of measured frame rates at these check points while testing the 3 different adaptation levels in section 6.

In the experiments called *"simple"*, without switching of scene LoD causes the observed rendering speed to be generally low and non-uniform, as shown by black dashed line of figure 2b. The straight dashed line is the target minimum frame rate of 25 FPS. Without LoD selection in the simple experiment, the target frame rate of 25 FPS is not achieved.

In the experiments called *"LoD selection"* adaptation level, the objects do not show visual artifacts due to LoD reduction and the application frame rate is always above 25 FPS. Even though the frame rate is much faster than the *"simple"* test, the frame rate still fluctuates a lot. Moreover, since no CPU scheduling is used, 100% of all available CPU cycles are always alloted (C_{max}) to the application, and at many points during the experiment, the scene rendered much faster than (overshoots) 25 FPS. The blue dashed line of Figure 2b illustrates this point, that CPU cycles are wasted when rendering the scene beyond 25 FPS. At frame 20 and frame 120, the frame rate drops, this heuristic compensates by choosing a lower LoD to render, thus causing the frame rate to go up. However, this lower LoD will show some visual artifacts since it is below the PoI LoD. At frame 40 and frame 170, the available CPU resource is enough to maintain a frame rate greater than 25 FPS, we then switch the LoD of meshes to their PoI.

In comparison to the other two experiments, the frame rate of *"EARR heuristic"* is more uniform with less fluctuations, as shown by red solid line of figure 2b. As in the "LoD selection" heuristic, at frame 20 and frame 120, the frame rate drops. EARR heuristic first tries to increase alloted CPU timeslices while using the PoI LoD. Since the frame rate continues to drop, the EARR heuristic selects a lower LoD and runs the CPU scheduler algorithm, which reduces the CPU resources alloted to 52% of the maximum available. The energy is saved while the application frame rate of 25 FPS is maintained.

In the experiments run on the laptop, the "simple heuristic" is used with the objects at their original LoD, the frame rate is only 13.54 FPS. However when EARR heuristic is used with multiple objects at their PoI LoDs, the frame rate is maintained at 25 FPS and never goes above 29 FPS. Therefore on average the target frame rate of 25 FPS is maintained. Figure 1a shows a screenshot of our applications on PDA. Our results show the LoD selection heuristic saves 27.4% of the energy, while EARR heuristic saves 62.3% of the energy consumption.

7 Conclusion and Future Work

We have presented our EARR heuristic that minimizes energy consumption while maintaining acceptable rendering speed and image quality. Our proposed EARR heuristic uses a workload predictor to adaptively predict frame rendering times and a dynamic CPU scheduler to save energy used by mobile 3D applications. Our experimental results show that our EARR heuristic generated more uniform frame rates than other strategies and successfully found the best rendering parameters that minimized mobile resource usage. Our experiments demonstrated energy savings of about 60%. In future, we shall extend our work in the following ways. *1) Improve energy saving by integrating Dynamic Voltage Scaling (DVS)* *and Dynamic Frequency Scaling (DFS).* We expect our heuristic will yield further savings after integrating DVS or DFS. *2) Improve PoI by integrating eye's gaze pattern* Eye's gaze pattern is another important factor affecting human visual perception. With cues about the eye's gaze pattern, we can increase the LoD of objects that user focuses on while reducing the LoD of objects outside of the focus area. In this way, even more rendering costs can be saved. *3) Accurately measuring CPU energy usage.* We currently estimate CPU energy usage using a subtractive technique described in section 6, which can be improved in accuracy. We plan to develop more accurate methods to more accurately measure CPU energy consumption on mobile devices.

References

1. Mobile Games Indus. Worth US $11.2B by 2010 (2005), `http://www.3g.co.uk/PR/May2005/1459.htm`
2. Wu, F., Agu, E., Lindsay, C.: Pareto-Based Perceptual Metric for Imperceptible Simplification on Mobile Displays. In: Proc. Eurographics 2007 (2007)
3. Yuan, W., Nahrstedt, K.: Practical voltage scaling for mobile multimedia device. In: Proc. of ACM MM 2004, pp. 924–931 (2004)
4. Flinn, J., de Lara, E., Satyanarayanan, M., Wallach, D., Zwaenepoel, W.: Reducing the energy usage of office applications. In: Proc. of Middleware 2001 (2001)
5. Tamai, M., Sun, T., Yasumoto, K., Shibata, N., Ito, M.: Energy-aware video streaming with QoS control for portable computing devices. In: Proc. of ACM NOSSDAV 2004, pp. 68–73 (2004)
6. Liu, X., Shenoy, P., Corner, M.: Chameleon: Application level power management with performance isolation. In: Proc. ACM MM 2005 (2005)
7. Funkhouser, T., Sequin, C.: Adaptive display algorithm for interactive frame rates during visualization of complex virtual environments. In: Proc. of ACM SIGGRAPH 1993, pp. 247–254 (1993)
8. Lext, J., Assarsson, U., Moller, T.: A Benchmark for Animated Ray Tracing. IEEE Computer Graphics and Applications 21(2), 22–31 (2001)
9. Gobbeti, E., Bouvier, E.: Time-Critical Multiresolution Scene Rendering. In: Proc. of IEEE Visualizatoin, pp. 123–130 (1999)
10. Winmmer, M., Wonka, P.: Rendering time estimation for Real-Time Rendering. In: Proc. of the Eurographics Symposium on Rendering, pp. 118–129 (2003)
11. Tack, N., Moran, F., Lafruit, G., Lauwereins, R.: 3D Rendering Time Modeling and Control for Mobile Terminals. In: Proc. of ACM Web3D Synposium, pp. 109–117 (2004)
12. Rohlf, J., Helman, J.: IRIS Perfromer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. In: Proc. ACM SIGGRAPH, pp. 381–395 (1994)
13. Teller S.: Visibility Computations in Densely Occluded Polyhedral Environments. Ph.D. thesis (1992)