

Modeling The GPU

(CS6282, Semester 2 AY 2006-2007)

BY ASHWIN NANJAPPA

Email: ashwin@comp.nus.edu.sg

Web: <http://www.comp.nus.edu.sg/~ashwinna/>

April 18, 2007



1. Motivation

Most consumer computing devices of all form factors (workstations, laptops, cellphones) have dedicated hardware called the *Graphics Processing Unit (GPU)* that deals with 2D/3D graphics. Today not only is the speed and power of GPUs rivaling that of the CPU, but its parallel nature is lending it to unique *GPGPU (General-Purpose Computation on GPU)* [1] applications in areas as diverse as physics and mathematics. Also, the GPU has recently undergone a radical change in its fundamental architecture.

Surprisingly, there seem to be no analytical models for the GPU found in computing literature. Here, we've attempted to create a simple model for the GPU. The model is shown to approximate well to both the pipeline and unified architectures. We analyze the model to validate it and then conduct tests to compare with the predictions of the model.

2. Introduction

The GPU is the hardware that is used to render graphics on displays. It takes as input graphics *primitives* (points, lines, triangles, colors) and outputs the processed *fragments* (pixels) to a window or the full display.

There have been 3 distinct stages of development in GPU architecture:

1. Fixed function pipeline
2. Programmable shader pipeline
3. Unified programmable shaders

2.1 Fixed Function Pipeline

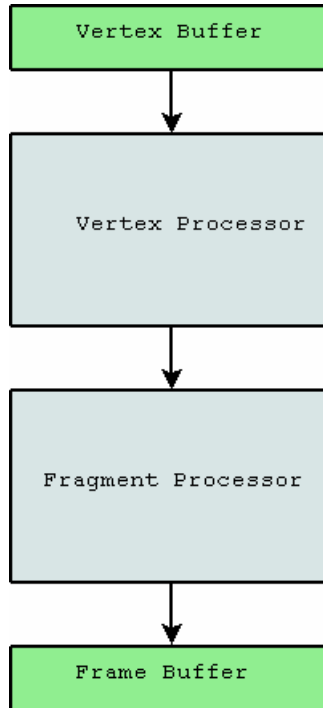


Figure 1: Fixed function pipeline

The *fixed function pipeline* architecture came with the early GPU (not to be confused with the humble 2D/3D raster graphics card, its predecessor). It had separate vertex and fragment shader units. But, the functionality of these shaders was limited to what was designed into the hardware or firmware. The user could not extend or program the shaders. The *NVIDIA GeForce 256* [2] was the first such GPU.

The fixed function pipeline GPU is now dead and gone. From here onwards we concern ourselves with the latter two types of GPU.

2.2 Programmable Shader Pipeline

The *programmable shader pipeline* is similar in design to the fixed function pipeline. It is the mainstay of most of the GPUs today. Its vertex and fragment shaders can be loaded with programs written in GPU assembly or higher level shader languages like *Cg* [3] or *GLSL* [4]. The *NVIDIA GeForce 3* [5] was the first GPU that could be programmed by the user.

In the programmable shader pipeline, every primitive enters at the top, undergoes vertex processing, followed by fragment processing and is output as a fragment (pixel) onto the display. Typically, a single primitive generates a large number of fragments. Hence, the number of fragment shaders is typically much larger than that of vertex shaders.

2.3 Unified Programmable Shaders

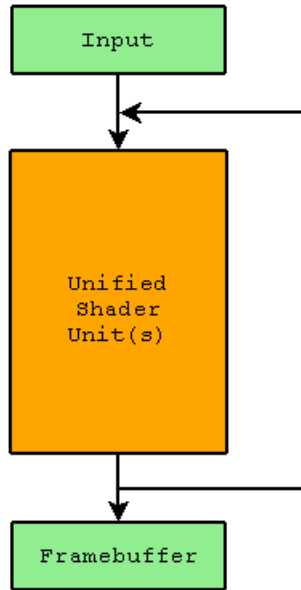


Figure 2: Unified programmable shader(s)

The GPU architecture has undergone major surgery in the last year with a shift towards an *unified shader architecture* by both NVIDIA and ATI. The recently released *NVIDIA GeForce 8800* [6] is the first such GPU available to the public. In this new design, the traditional pipeline of 2 or more stages with limited functionality shader units is abandoned in favour of a bunch of generic programmable shaders to which output from other such shaders can be fed back.

Thus these shaders can handle the roles of both vertex and fragment processing by being loaded with the appropriate program. This architecture has led to an increase in the transistor count, but scales well to all kinds of loads. This architecture needs a *load manager* which dynamically decides on the allocation of the generic units to vertex and fragment processing duties. (In the NVIDIA 8800, the load manager is known as the *thread processor*.)

3. Specialization vs. Unification

Graphics hungry applications are not the only reason why the graphics community is now favoring an unified shader GPU. The reason also lies in the increasing complexity of the pipeline shaders and their low utilization on certain workloads.

The pipelined shader GPU performs efficiently on most workloads which have a good balance of vertex and fragment processing. However, a lot of the complex shader silicon on the GPU lies wasted on workloads which are skewed towards either vertex or fragment processing only. With the increase of such disparate workloads and their dynamics, the pipeline shader GPU finds its silicon not being put to use effectively.



Figure 3: Utilization of pipeline GPU on vertex-intensive workload.



Figure 4: Utilization of pipeline GPU on fragment-intensive workload.

The unified shader GPU on the other hand has an intelligent load manager which tries to distribute the various generic shaders based on the workloads. This results in a very efficient utilization of the GPU silicon.

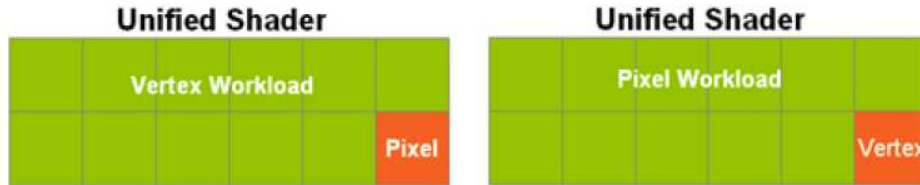


Figure 5: Unified architecture extracts maximum utilization from its generic shaders.

In the next section we characterize GPU workloads. In later sections, we model and experiment based on these workloads and try to squeeze out the defining characteristics that differentiate these two GPU architectures.

4. Workload Characterization

A GPU deals with various values such as vertices, fragments, color, depth and alpha. We restrict the scope of the workload to just vertices and fragments as these are directly related to the shader units under study.

4.1 Primitive Size

If P_v and P_f are the number of vertices and fragments in the workload, then they are related such that $P_v \leq P_f$. We further approximate this relation as

$$P_v \approx \frac{P_f}{m_p} \quad (1)$$

for an integer m_p .

4.2 Instruction Size

The amount of processing required per-vertex or per-fragment is also a factor of the workload. This can be characterized by the number of GPU assembly instructions that are executed for each vertex or fragment.

We restrict ourselves to workloads where the coefficient of variation of per-vertex and per-fragment instruction count is less. This is not an impractical restriction since a change in this count typically implies a change in the workload type. Also, with the GPU being a SIMD processor, all input primitives are applied to the same number of instructions on average.

It can be inferred that both the primitive size and instruction size play a part in determining the characteristic of a workload. In the sections that follow, I_v and I_f represent the *per-vertex* and *per-fragment instruction counts*.

5. Workload Types

In the modeling and experiments that follow, we are concerned with three kinds of workloads.

5.1 Equitable Workload

An *equitable workload* is defined as that in which the vertices and fragments are related by:

$$I_v \cdot P_v = I_f \cdot P_f$$

Using (1), this means:

$$I_v = m_p \cdot I_f \tag{2}$$

For further ease, we introduce new symbols P'_v and P'_f as follows:

$$P'_v = I_v \cdot P_v \text{ and } P'_f = I_f \cdot P_f$$

Thus,

$$P'_v = P'_f \tag{3}$$

5.2 Vertex-Intensive Workload

A *vertex-intensive workload* is defined as:

$$P'_v \gg P'_f \tag{4}$$

5.3 Fragment-Intensive Workload

A *fragment-intensive workload* is defined as:

$$P'_v \ll P'_f \tag{5}$$

6. An Analytical Model For The GPU

We model the GPU as an *open queueing network* composed of 2 M/M/1 *single-server queues* connected in series. The servers S_v and S_f represent a single vertex and fragment shader unit respectively with *service times* T_v and T_f . Only the service rate is of interest since it determines the arrival rates and the queue lengths.

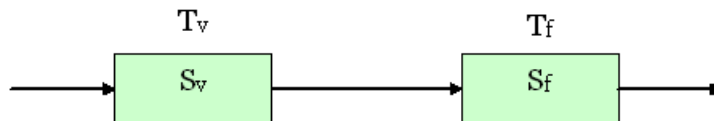


Figure 6: A model for the GPU.

If the GPU has n_v and n_f number of vertex and fragment shader units, the above model can be extended to represent it by scaling the service times by $\frac{1}{n_v}$ and $\frac{1}{n_f}$, i.e.,

$$T'_v = \frac{T_v}{n_v} \text{ and } T'_f = \frac{T_f}{n_f}. \quad (6)$$

6.1 Model For Programmable Pipeline GPU

The programmable pipeline GPU can be modeled by the above open queueing network without any changes.

Case 1

To handle an equitable workload efficiently, the programmable pipeline model must have service times given by:

$$\frac{T'_v}{T'_f} = \frac{P'_f}{P'_v}. \quad (7)$$

Since n_v and n_f of a programmable pipeline GPU are fixed, let us fix the service times of the programmable pipeline model at its equitable workload values given in (7). Let these be denoted as \hat{T}'_v and \hat{T}'_f . From (7) and (3) it follows that,

$$\hat{T}'_v = \hat{T}'_f \quad (8)$$

Case 2

When fed with a vertex-intensive workload, the programmable pipeline model isn't efficient. This is because, from (4) and (7), it can be seen that for efficient handling the model needs

$$T'_v \ll T'_f \quad (9)$$

But, the service times in this model are fixed at the equitable values from (8), so it is inefficient. Furthermore, we can infer from (9) that some fragment shader units will lie idle for this workload.

Case 3

Similar to case 2, the model isn't efficient for a fragment-intensive workload since,

$$T'_v \gg T'_f \quad (10)$$

6.2 Model For Unified Shader Model

The unified shader GPU is a bit trickier to model. If n_u is the number of shader units in the GPU, each with service time of T_u , then it can be represented by a *partly open queueing system* [7] as shown below:

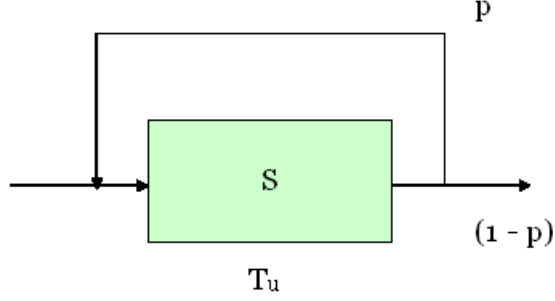


Figure 7: A possible model for the unified shader GPU.

The *exit probability* $(1 - p)$ of this model is given by:

$$(1 - p) = \frac{P'_f}{P'_v + P'_f} \quad (11)$$

Since this model is intuitively similar to the operation of the unified shader GPU, it could be a good *average value model*. However, we abandon it since it turns the existing deterministic system into a probabilistic one.

Instead we modify the open queueing system presented in Fig. 6 with suitable service times to model the unified shader GPU. The intelligent load manager in the GPU decides upon the values of n_v and n_f to suitably balance the workload while maintaining the relation:

$$n_u = n_v + n_f \quad (12)$$

If such a load distribution is possible, then the service times of S_v and S_f are given by:

$$T'_v = \frac{T_u}{\binom{n_u}{n_v}} \text{ and } T'_f = \frac{T_u}{\binom{n_u}{n_f}} \quad (13)$$

It is easy to see that for any kind of workload, the following is true:

$$\frac{n_v}{n_f} \approx \frac{P'_v}{P'_f}$$

That is, the ratio of the allocated vertex shaders to fragments shaders should match the ratio of the vertex and fragment workload.

So, it follows that

$$\frac{n_v}{n_v + n_f} = \frac{P_v}{P_v + P_f}$$

Using (12),

$$n_v = \frac{P_v \cdot n_u}{P_v + P_f} \quad (14)$$

Similarly, we can obtain,

$$n_f = \frac{P_f \cdot n_u}{P_v + P_f} \quad (15)$$

Using (14) and (15) in (13), we obtain the service times for our model,

$$T'_v = \frac{T_u \cdot P_v}{P_v + P_f} \text{ and } T'_f = \frac{T_u \cdot P_f}{P_v + P_f} \quad (16)$$

Thus we've derived the service times for the servers in the unified shader model. Next, we apply this to the different kinds of workloads and see how it behaves.

Case 1

At equitable workload, using (3), (16) reduces to:

$$T'_v = \frac{T_u}{2} \text{ and } T'_f = \frac{T_u}{2} \quad (17)$$

Intuitively this means that the number of shaders are split equally between vertex and fragment processing tasks.

Case 2

With a vertex-intensive workload, using (4), (16) reduces to:

$$T'_v = T_u \left\lceil \frac{P_v}{P_v + P_f} \right\rceil \text{ and } T'_f = T_u \left\lfloor \frac{P_f}{P_v + P_f} \right\rfloor \quad (18)$$

The $\lceil \cdot \rceil$ and $\lfloor \cdot \rfloor$ functions are applied by the load balancer to divide the n_u shader units in an integral fashion while giving *priority* to units that will act as vertex shader units.

Case 3

Similar to case 2 above, with a fragment-intensive workload, using (5), (16) derives:

$$T'_v = T_u \left\lfloor \frac{P_v}{P_v + P_f} \right\rfloor \text{ and } T'_f = T_u \left\lceil \frac{P_f}{P_v + P_f} \right\rceil$$

With these derivations, the GPU in both its avatars has been completely modeled.

7. Experiments

To analytically validate the GPU models, some simple benchmarking experiments were conducted.

7.1 Hardware

Two workstations with similar CPU and memory capabilities were chosen.

NVIDIA 6600

Workstation 1 was fitted with a NVIDIA 6600 graphics card. This GPU is based on the programmable pipeline design. It has 3 vertex shaders and 8 fragment shaders. The core clock of the GPU is 500 MHz.

NVIDIA 8800

Workstation 2 was equipped with a NVIDIA 8800 graphics card. This is the first GPU based on the unified shader design. It has 128 stream processors which can do both vertex and fragment processing. The core clock is 1.35 GHz.

7.2 Software

The tests were written in C++ using the *OpenGL library* for graphics and the Cg shader language for vertex and shader programs. The code was compiled using the *Visual C++ Express Edition* compiler suite.

The GPU sections of the code were timed using the *high-performance timers* [8] available in `windows.h`.

7.3 Tests

Three kinds of tests were used to mimic the three kinds of workloads.

7.3.1 Vertex-Intensive Test

In this test, a large number of quads were drawn. The quad vertex positions were chosen at random to reduce chances of vertex caching by the GPU. The number of quads (and thus vertices) was chosen high enough to keep the GPU loaded well beyond its bounds. Additionally, a lighting function was applied on each vertex to increase the instruction count (and thus time spent) on each vertex. The fragment processing was kept simple with no texturing.

7.3.2 Fragment-Intensive Test

Here too a large number of quads were drawn. A texture of size 512x512 with randomly generated colors was mapped to each quad. The number of quads was not high. But, the quads were drawn in random shapes to increase the texture coordinate generation load on the fragment unit. To further increase fragment load, the fragment shader program accessed an increasing number of surrounding neighbour texels. This was done to load the texture cache and increase the per-fragment instruction count.

7.3.3 Equitable Workload Test

A combination of the above two tests was used. Both the number of quads, their shapes and the texture mapped into them was high but balanced.

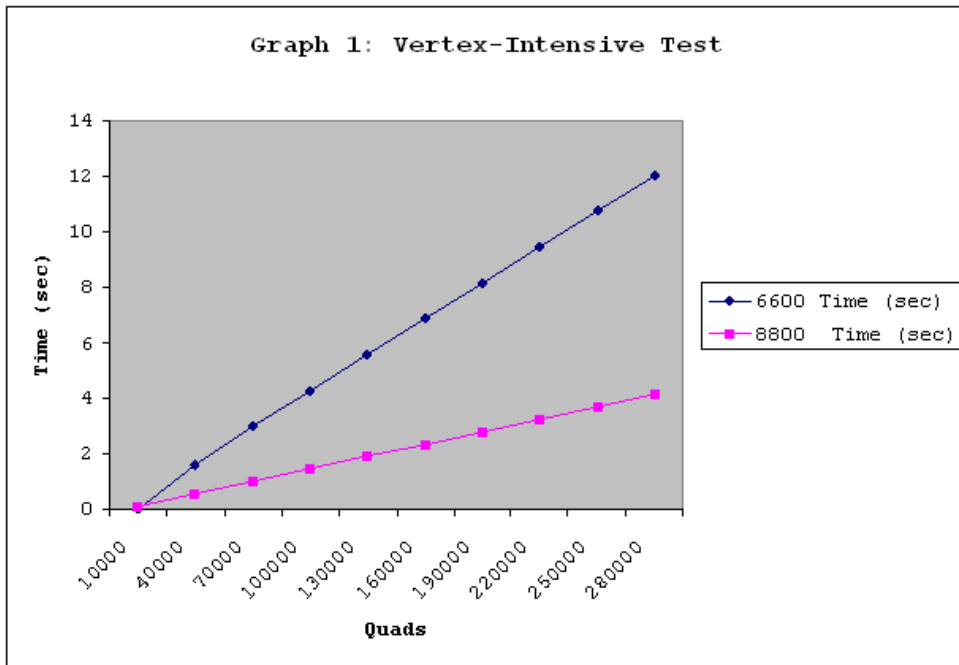
7.4 Results

The results obtained for the above 3 tests on the NVIDIA 6600 and 8800 are shown below:

7.4.1 Vertex-Intensive Test

	6600	8800
Quads	Time (sec)	Time (sec)
10000	0.0280741	0.132772
40000	1.60704	0.583104
70000	3.01708	1.04385
100000	4.29007	1.46368
130000	5.57167	1.93498
160000	6.86678	2.35821
190000	8.16422	2.80207
220000	9.43364	3.24959
250000	10.7326	3.69662
280000	12.0103	4.13487

Table 1: Vertex-Intensive Test

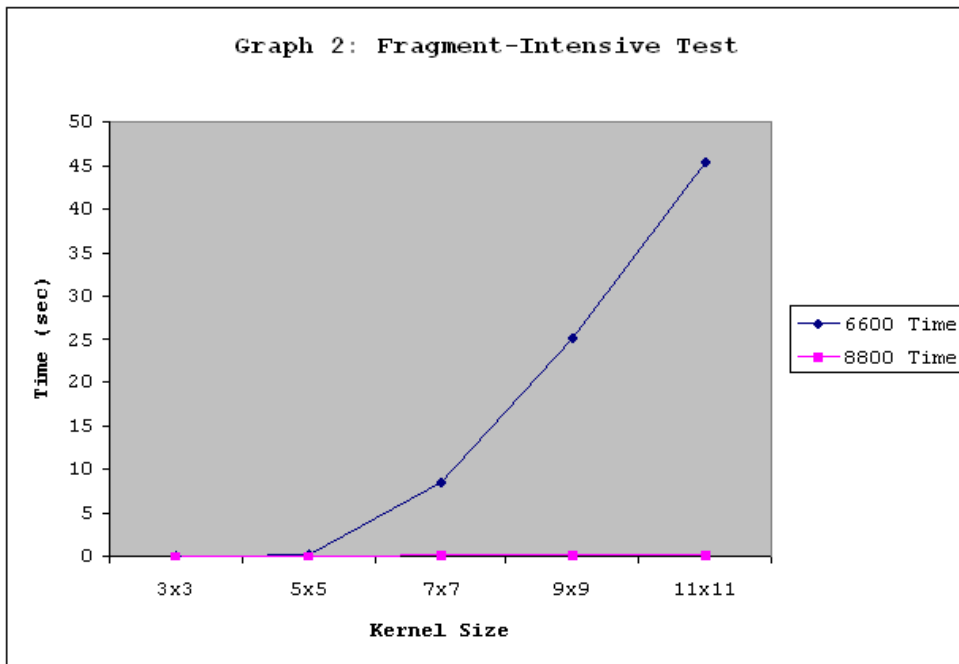


The results are a bit disappointing for our model. The programmable pipeline 6600 performs with the same linear growth as the unified shader 8800. For example, notice how the time doubles for both the GPUs between 100K and 220K quads.

7.4.2 Fragment-Intensive Test

Kernel Size	6600 Time	8800 Time
3x3	0.0116739	0.0665966
5x5	0.164006	0.0869022
7x7	8.6003	0.0972599
9x9	25.1679	0.126504
11x11	45.2463	0.166385

Table 2: Fragment-Intensive Test



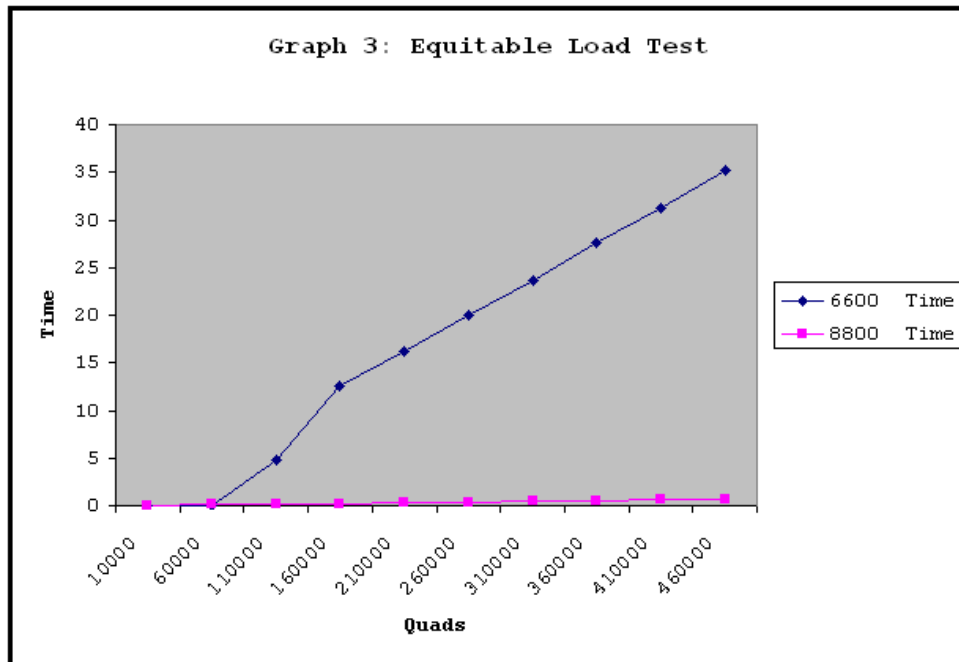
Though not discernable from the graph, from the values in the table it can be seen that the fragment-intensive test does prove the advantage of unified shaders. The values of the 2 GPUs don't match no matter what scaling value is used for the 6600 value. While the 8800 grows in a linear-like fashion, the 6600 times grow exponentially.

This validates the unified shader model since the linear-like growth implies that most of the shaders are being redirected to handle the fragment load.

7.4.3 Equitable Load Test

Quads	6600 Time	8800 Time
10000	0.00735508	0.00998158
60000	0.00588024	0.107402
110000	4.79863	0.172399
160000	12.5633	0.240269
210000	16.1632	0.320178
260000	19.939	0.396599
310000	23.7186	0.468015
360000	27.5236	0.53312
410000	31.2363	0.617171
460000	35.133	0.691092

Table 3: Equitable Load Test



Not very interesting since (as expected) both the 6600 and 8800 display matching linear growth. For example, notice how the values double between 210K and 410K quads.

8. Conclusion

A simple analytical queueing network model was developed for the GPU. Both the kinds of GPU architecture prevalent today were shown to be approximated well by this model. GPU workload was characterized and their effects on the model was explored. Experiments on the different workloads were conducted and found to match quite well with the expectations from the model. Thus the model was validated.

9. References

- [1] David Luebke, Mark Harris, Jens Kruger, Tim Purcell, Naga Govindaraju, Ian Buck, Cliff Woolley and Aaron Lefohn. GPGPU: general purpose computation on graphics hardware. *SIGGRAPH '04: ACM SIGGRAPH 2004 Course Notes*, 2004.
- [2] http://en.wikipedia.org/wiki/GeForce_256
- [3] William R. Mark, R. Steven Glanville, Kurt Akeley and Mark J. Kilgard. Cg: a system for programming graphics hardware in a C-like language. *SIGGRAPH '03: ACM SIGGRAPH 2003 Papers*, 2003.
- [4] Randi J. Rost. *OpenGL Shading Language*. Addison-Wesley Professional, 2004.
- [5] Erik Lindholm, Mark J. Kilgard and Henry Moreton. A user-programmable vertex engine. *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, 2001.
- [6] David Blythe. The Direct3D 10 system. *ACM Transactions on Graphics*, 2006.
- [7] Bianca Schroeder, Adam Wierman and Mor Harchol-Balter. Open Versus Closed: A Cautionary Tale.
- [8] <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winui/winui/windowsuserinterface/windowing/timers.asp>

□