Evaluating ATTILA, a cycle-accurate GPU simulator

Miguel Ángel Martínez del Amor

Department of Computer and Information Science
Norwegian University of Science and Technology (NTNU)
miguelan@stud.ntnu.no
Project supervisor: Lasse Natvig
lasse@idi.ntnu.no

26th January 2007

Abstract. As GPUs' technology grow more and more, new simulators are needed to help on the task of designing new architectures. Simulators have several advantages compared to an implementation directly in hardware, so they are very useful in GPU development. ATTILA is a new GPU simulator that was born to provide a framework for working with real OpenGL applications, simulating at the same time a GPU that has an architecture similar to the current designs of the major GPU vendors like NVIDIA or ATI. This architecture is highly configurable, and also ATTILA gives a lot of statistics and information about what occurred into the simulated GPU during the execution. This project will explain the main aspects of this simulator, with an evaluation of this tool that is young but powerful and helpful for researchers and developers of new GPUs.

1. Introduction

The main brain of computers is the CPU (Central Processing Unit), where the instructions are executed, manipulating by this way the data stored in memory. But with the new applications that demand even more resources, the CPU is becoming a bottleneck. There is a lot of works about how to improve the CPI (Cycles per Instruction) which is a good parameter for making comparisons between uniprocessor architectures, and the search for parallelism is the best way to do it. It is possible to parallelize the execution of each instruction by performing this execution in stages. In this manner, when an instruction is in the second stage, the next operation can start with the first stage, so an instruction can start before finish the last one. It is similar to the Henry Ford method for a car factory where the construction pipeline is divided in stages and one car can be only in each stage (never two in the same stage), now imagine that a car is an instruction and the factory is the CPU. It is easy to demonstrate that CPI value can achieve 1 with this method, which is the perfect and theoretical value although it was never achieved; these techniques and also all the stages that each instruction is divided in are called *pipeline*. If we try to use the parallelism between instructions (loops, threads …), it is possible to issue more than one instruction per cycle, so the CPI can be less than one. It can be done by using multiple pipelines for multiple instructions, and neither it was possible to achieve a perfect value less than 1. Superscalar and VLIW architectures are examples about this kind of exploit parallelism.

Moreover, thread and process parallelism are used by multiprocessor architectures as SMT (Simultaneous Multi Threading, several threads can be executed at the same time in the same processor), UMA (Uniform Memory Access, also called SMP, all processor share the main memory) and NUMA (Non Uniform Memory Access, each processor has a part of memory). But all of these kinds of parallelism have limits, and in some fields it has still some troubles to find them. Other problem that CPUs have is that they are for general purpose, so they must consider all the possibilities making more difficult the optimization of code execution. A solution for the last feature is to make others specific systems that can help the CPU to improve the execution time.

In computer graphics the CPU needs to delegate to the GPU the calculations of graphics instructions and data. GPU is a specific purpose processor that is optimized for working with graphics which are based on streaming (continuous source of data, like buffers or arrays of elements). There was many works trying to make a general purpose CPU that could work with stream data, the first one was on the 70's and was called Vector processor. Nowadays this idea is used in technologies like MMX, SSE or 3DNOW that help the CPU with graphics calculations. The key of the optimizations that the stream specific purpose architecture provides is based on the independence between the elements of a stream. In this manner, if a unit needs the result stream from a previous one, it does not have to wait until this previous unit finish with the entire stream, instead the unit can start to make calculations with one element of the stream at the same time that the previous unit sends the processed element. It means that there is an overlapping between instructions that share resources, and it is really powerful for working with streams. Graphic processors (GPU, Graphic Processor Unit) use these kind of architectures based on streaming processing, and these processors are placed outside the CPU for performing the complex graphic calculations and being the responsible of refreshing the screen with new frames in parallel with the execution of instructions on the CPU. The technologies mentioned before (MMX, SSE or 3DNOW) never replace a GPU; they just help the CPU and GPU to perform multimedia and graphic operations.

However, CPU and GPU have to cooperate if they want that the system shows good results with graphic operations. As it can be seen in figure 1.1 (next page), CPU and GPU are interconnected by the North Bridge, but the real communication is made by the system memory (DRAM). So the CPU puts data and code into the memory while the GPU takes this data, makes the operations and sends the information to the output display.

The most complex graphic calculations are about 3D graphics on real time (i.e. videogames), so newer GPUs are optimized for this kind of applications. The pipeline of 3D graphics is based on triangle and texture streaming, which means that the main goal is to transform 3D data (coordinates of triangle models, easy to understand for programmers) into pixels that are displayed on the 2D screen.

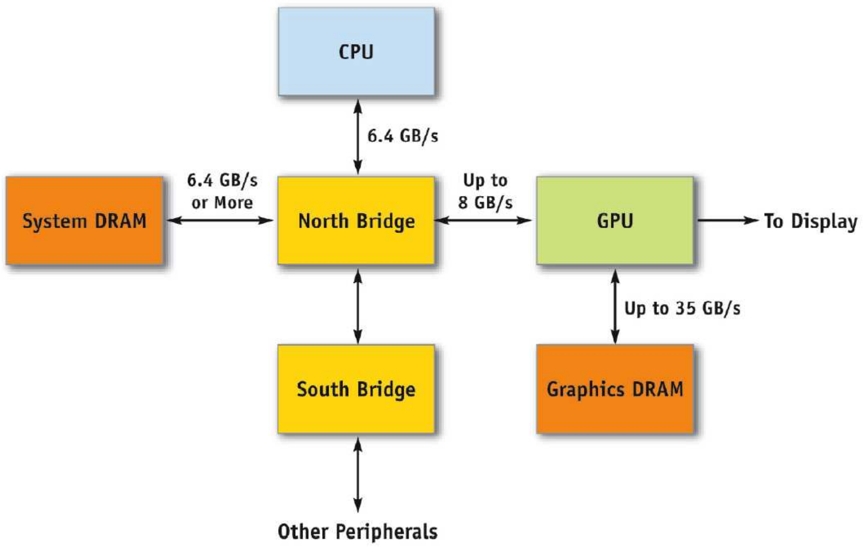


Figure 1.1 The place of the GPU into a typical computer system (from [3]). That is a Nvidia implementation for the GeForce architecture.

Figure 1.2 shows a general pipeline architecture for 3D data. The input of the GPU is 3D object coordinates (mainly from triangles) that go to the Geometry Stage where the 3D coordinates are translated to the space of 2D coordinates of the display. In the last stage, Rendering, the correct colour of each pixel is calculated in base of the associated colour, light hitting, textures and other effects as alpha (translucent) and fog of each Geometry stage's output (also called fragments). Furthermore, the Geometry stage can be divided into 2 stages. The first one is the Transform and Lighting stage which is the process of displaying a three-dimensional object onto a two-dimensional screen, providing lighting effects to the scene. The Rasterization (also called Triangle Setup in [1]) sets up and clip the triangle, and makes fragments for the next stages (Rendering). A fragment is not still a pixel (sometimes these two terms are not distinguished, like in Direct3D specification), it has attributes like colour and depth, and multiple fragments correspond to one pixel (i.e. when using blending or transparency functions). Application tasks (AI, camera, interaction …) and scene level task (collisions …) are always performed by the CPU (software), because these are general purpose programs.

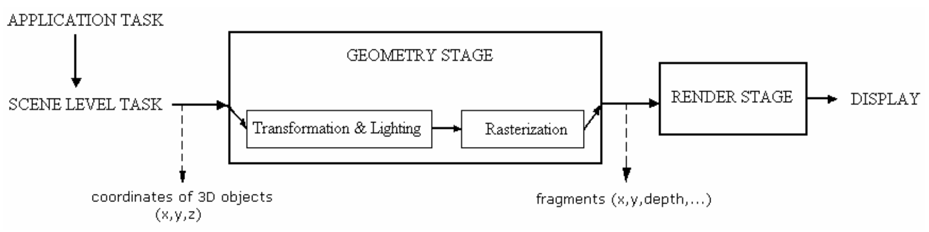

APPLICATION TASK
SCENE LEVEL TASK
GEOMETRY STAGE
Transformation & Lighting
Rasterization
RENDER STAGE
DISPLAY
coordinates of 3D objects (x,y,z)
fragments (x,y,depth,...)

Figure 1.2 Abstract architecture of the graphic pipeline.

Not all of these stages were implemented on the first Graphic Card that just displayed the lines into the screen, so the other stages were made as a software process by the CPU. With the evolution of transistors and hardware, it was possible to implement the pipeline into the GPU; meanwhile the CPU just works with the application tasks and scene level physics, giving to the GPU triangle coordinates from the results.

The most complex stage and the bottleneck of GPU is the Rendering Stage (it calculates the colour and position of pixels). At the beginning GPUs were able to output only one pixel in several cycles. Because of this bottleneck, CPUs were able to output more triangles than the GPU could handle. Two solutions were taken: pipelining and parallelism. Nowadays a GPU has about 16 parallel pipelined rendering processors, it was very easy because rendering 3D graphics is a repetitive task and it is possible to do with different pixels at the same time. With these changes, the CPU and the access to 3D data into the memory is becoming the bottleneck of graphics rendering, so the best solution for this is that the GPU takes more workload than the CPU, in this case, place the stage Transform and lighting into the GPU and improve the access of the GPU to the memory (i.e. PCI Express).

The pipeline with all the graphics stages implemented on hardware was called fixed function pipeline. Although it was faster, the implementation of all the pipeline on hardware had a constant behaviour, that is, it always performs the same operations with 3D data. Consequently, this pipeline was not flexible for graphic programmers in case of, for example, changing the API or making a new kind of operation with the vertices.

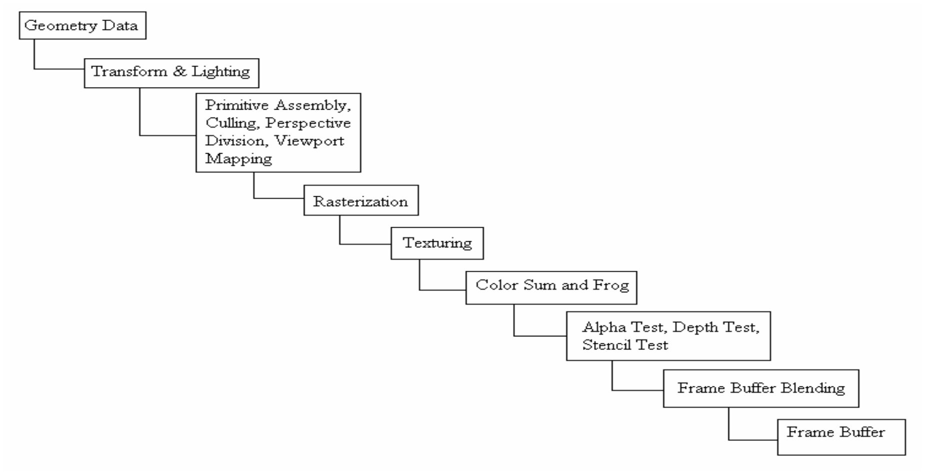
The solution (and nowadays the most used method) was to develop a programmable pipeline where the GPU can execute small programs called *shaders* over the vertices and fragments. Figure 1.3 (next page) shows a general view of this new programmable graphic pipeline. A shader is a piece of code that program certain parts of the graphics pipeline, and there are two types: vertex shaders (replace Transform and Lighting stage) and fragment shaders (replace the texturing, colour sum and fog). These shader codes use low-level languages, such as ARB_Vertex_Program, ARB_Fragment_Program and Direct3D 9 Shading languages. However there are high-level languages with compilers that make easy the complex programming task. Some of the high-level languages are Cg[1] and GLSlang[2].

There are a widely amount of work research for future GPUs. Now it is possible to execute general purpose code using a GPU as a transformation to streams (i.e. GPGPU project in www.gpgpu.org), but sometimes it is not more efficient than a CPU because there are some limitations as said in [2]. Some new compilers, like the BrookGPU project [10], try to help with the task of making general purpose applications for GPU, but they are still in development. Other future work is to implement unified shader units on the GPU, that is, join vertex and fragments units in a general one. Vertex and fragment shaders are so similar and their union can be performed relatively easy. It provides better performance in some cases; for example,

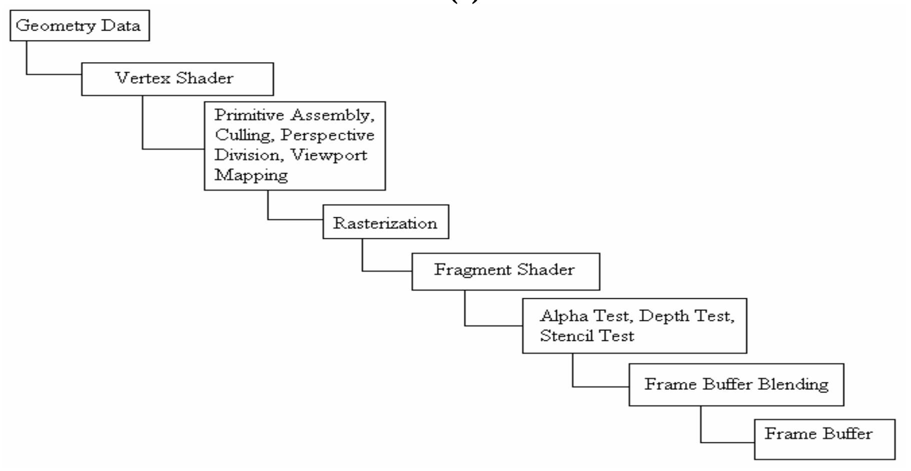
[1] Cg is a high-level shading language developed by NVIDIA. Homepage is at http://developer.nvidia.com/page/cg_main.html.

[2] GLSlang is a high-level shading language that will be introduced with OpenGL 2.0. Homepage is at http://developer.3dlabs.com/documents/glslmanpage_index.htm.

when it is needed more vertex shaders than fragment units and then, it is possible to share resources because of the use of unified shader units. Future shading languages will support this novel architecture, like Shader Model 4.0 in Direct3D and GLSlang in OpenGL 2.0.



(a)


Geometry Data
Vertex Shader
Primitive Assembly, Culling, Perspective Division, Viewport Mapping
Rasterization
Fragment Shader
Alpha Test, Depth Test, Stencil Test
Frame Buffer Blending
Frame Buffer

(b)

Figure 1.3 Overview of the GPU pipeline. (a) A fixed function pipeline. (b) A programmable function pipeline, with vertex and fragment shader units.

As it can be seen, the task of developing a novel architecture for GPU purpose is hard, like for CPU purpose is. Simulators are a good tool that can help to GPU developers to investigate new technologies without a physical implementation. A new simulator was done by the department of computer architecture of polytechnic university of Catalonia, with almost all the code public. The name of the project and also of the simulator is ATTILA. This simulator is cycle accurate and execution driven, and provides a wide set of statistics that permits the researcher to analyse the new architecture. The urge that this simulator can achieve for the development and test of new GPU architectures has to be considered, that is the reason because this report makes an evaluation of this tool.

In section 2 it will be commented some previous related works. Section 3 will explain in detail the simulator: architectures that ATTILA can simulate, an introduction to the source code and statistics that it provides. Section 4 will make an evaluation of the simulator, executing testprograms of OpenGL and a discussion of the weak and strong sides of ATTILA. Finally, section 5 gives a general conclusion of the report and comments some possible work to continue in a not so far future.

At the end of this paper there are also four appendices. Appendix A contains a small dictionary of abbreviations and terms that can help to the understanding of this paper. So if one does not know the meaning of a word, just look for it in Appendix A. Appendix B and C contains more information and details about the simulator. The former (B) has the collection of statistics that ATTILA provides; the latter (C) shows an introduction to the configuration of ATTILA by parameters. As a bonus for this paper, Appendix D comments how ATTILA can improve the simulation time by parallelizing some code.

2. Antecedents and related work

The previous works about ATTILA are the presentations and works made by the authors, which are Victor Moya del Barrio, Carlos González, Jordi Roca, Chema Solís and Agustín Fernández from the department of computer architecture of the Polytechnic University of Catalonia (Spain), and Roger Espasa from Intel in Barcelona. These works can be found on [8]. "An End to End, Highly Detailed Simulator for the ATILA GPU Microarchitecture" [5] and "ATTILA: A cycle-Level Execution-Driven Simulator for Modern GPU Architectures" [6] explain the simulator on detail (the most used for this paper). [11], [12] and [13] are works that uses ATTILA as the basis for their experiments (includes a little explanation of the simulator).

But ATTILA is not the only choice for simulate GPU architectures. There are previous frameworks like QSilver (see [14] or the home page http://qsilver.cs.virginia.edu/, work from the University of Virginia), SM (inside the project Sh, which is a metaprogramming language for programmable GPUs (see home page at http://libsh.org/index.html), one interesting paper that talks about SM is [15]) and one developed by NVIDIA, NVEmulate (of course it can simulate only NVIDIA architectures, in special, GeForce Series, but support GLSlang and a wide set of OpenGL, the webpage is at http://developer.nvidia.com/object/nvemulate.html).

3. ATTILA in detail

3.1 GPU simulated architecture

The architectures that ATTILA can simulate are based on a common pipeline that can support real 3D graphic data going through its stages. This baseline pipeline uses techniques developed from the main GPUs manufacturers such as ATI and NVIDIA, and others from many researchers at universities; it is based on the OpenGL specification, in the manner that D3D (Direct3D) is not supported yet (but it is being in development). By this way, the pipeline is ready to work with real data like modern GPUs do, but for simplicity, some stages was not implemented such as the fog and alpha stages (they are simulated by fragment shader programs). This section makes an overview of the baseline pipeline that ATTILA can simulate with an explanation of the general architecture simulated (to know how to configure it with parameters, see Appendix C) and a short explanation of the implementation on code (as a beginning for configure a whole new pipeline).

3.1.1 Architectures based on the baseline pipeline

For a good understanding, on figure 3.1 (next page) there is a scheme of the simulated pipeline where the red lines (or lines without arrows) are control wires and the data go through the black wires (or lines with arrows). It is supposed that between two connected stages by data wires there is a buffer where the first stage stores its outputs and the next takes its inputs. It implies that each stage run independently in terms of synchronization from the others, in other words, each stage can work with different latencies without waiting to the previous stage. Below the pipeline there are details of the boxes ROPz (Z and Stencil tests), Shader and ROPc (color write), and following the scheme there is an explanation of each stage of the pipeline, that is, the task that each box has.

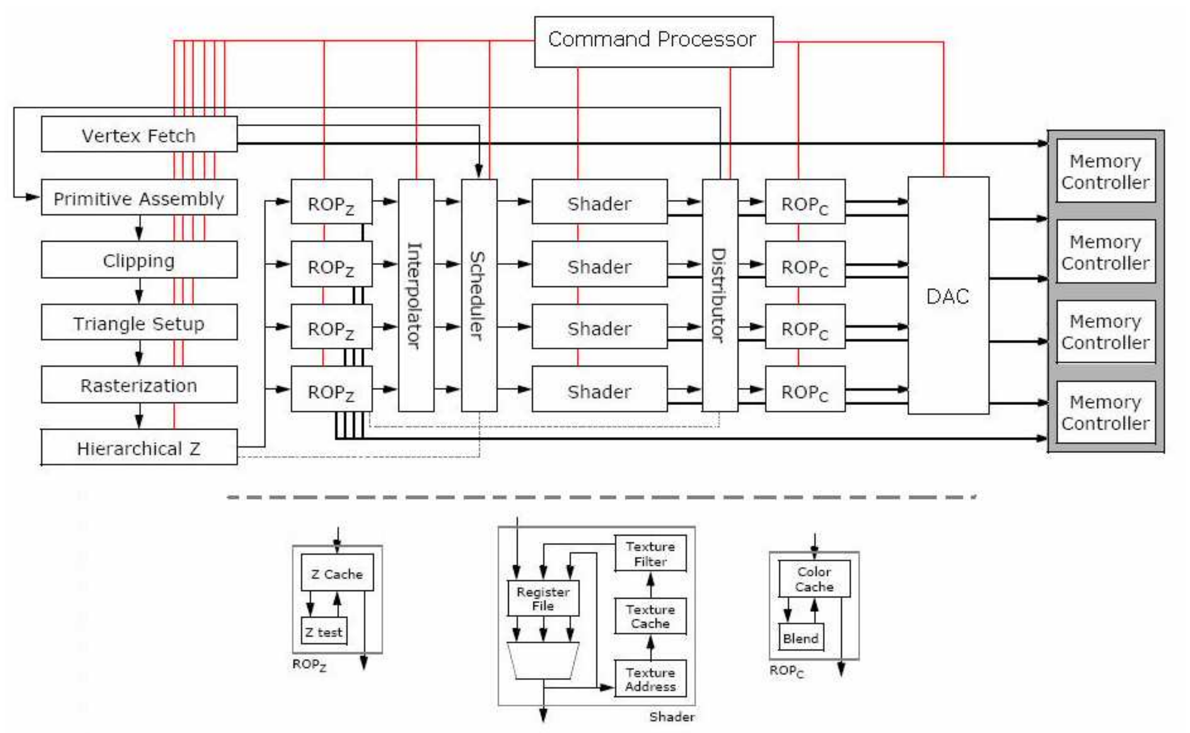


Figure 3.1 Scheme of the ATTILA supported pipeline (from [6]).

- **Command processor**: controls (red wires) the whole pipeline, buffers and the transactions of batches (set of vertices) from the system memory to the GPU memory, and receives the commands sent from the processor and forwards them to the pipeline.
- **Vertex fetch or Streamer**: requests vertex input attributes and converts them into internal format (4 component 32 bit float point vectors) for the vertex shader. It is supported both plain vertex attribute buffers and indexed vertex buffers. The former is a natural stream of vertices like arrays that can be stored in internal buffers implemented by a full associative cache. The latter is used for the reuse when a vertex is shared between two or more adjacent triangles by associating an index to each one. For example when the TRIANGLE_FAN mode in OpenGL is used and one vertex is shared between all the triangles defined. A cache is used to reuse these indexed vertices by using a connection with the output of the vertex shader (to store post-calculated shading vertices), and then, avoid repetitive calculations when using shared vertices.
- **Vertex shader**: is the programmable unit explained in the introduction. It takes a batch (a stream of vertices) processed by the Streamer and executes a vertex shader program (provided by the Command Processor) over them. The vertex shader unit was implemented using as reference the ARB_vertex_program specification, so for supporting Direct3D it is

needed also to change the shader implementation. However, not all the features of the ARB specification are supported, but the implemented ones are enough for simulating frames from novel games like Doom3 or Quake4.

The vertex shader units can be shared with the fragment shaders ones, that is, it can be used a unified shader model. In this way, there are mainly two simulator binaries: one that supports unified shader units and another one that supports separated shader units. Because vertex and fragment shaders are similar, it will be explained the whole shader unit as unified. So for the understanding of the separated model, it is enough to divide the part of vertex shader in one group of units and the part of fragment shaders in other.

- **Shader units**: This architecture supports unified shader units where, as mentioned so far, the units used for vertex and fragment shading are the same (it is optional because there is other version of the simulator for non-unified shader architecture). This unit has an ALU based on SIMD architecture of 4 component 32 bit float point vectors and scalar ALUs to support fragment shading.

  The shader on detail can be seen on the bottom of figure 3.1, where there are two separated parts, one with the ALU and register file, and another with the texture units. Vertex Shaders use only the ALU and register file, whereas Fragment Shaders use the whole unit (ALU, texture ALU and texture cache). As each shader unit can support the execution of several threads at once, they can execute vertex and fragment programs at the same time without interaction between them (thread information contains if it is a vertex or a fragment shader and the execution is performed only if there is free resources in the shader unit).

  A small cache is used to reduce the usage of the memory bandwidth for texture accesses and it is based on the texture cache architectures from Hakura and Gupta research [7], using a 256 byte cache line that can store 8x8 32 bits Texel tile, with 4 sets and 16 lines per set that implies a cache of 16 KBs. As said before, shader units are based on the ARB ISA (both ARB_Vertex_program and ARB_Fragment_program) that comprises an ALU and four registers files (as can be seen on figure 3.2 (next page)): for input attributes (Read Only), for output attributes (Write Only), temporal registers (Read/Write) and for constants (Read Only, with values configured just before the simulation). Each register is a 4 component 32 bits float point vector (the variable type of the ARB), in exception of the Temporal Bank that store 2 components 32 bits vectors. Furthermore, it was defined others 4 kinds of registers files: Address Register Bank (store integers), Condition Code Register (store an integer), Boolean Constant Bank (store

Booleans) and Texture Sampler (defines texture unit in usage). The number of registers per bank depends on the kind of shader and on the shader version, thus having five types that a shader unit can represent: Fragment or Vertex Shader based on model version 1.0, Fragment or Vertex Shader based on model version 2.0 and unified shader. For example, the number of registers for the constant bank is 96 in vertex shader 1.0 and 256 in the version 2.0.

One shader can fetch more than one instruction (configured as a parameter) and the shader architecture implements multithreading, hiding the latencies that instructions and texture memory access have. The required number of threads depends on the kind of shader: 12 threads are enough for vertex shader, but fragment shader needs up to 112 threads because the leverage of texture accesses' large latencies. Also the number of physical instruction registers (instruction memory size) varies, from 96 to 448 in that order.

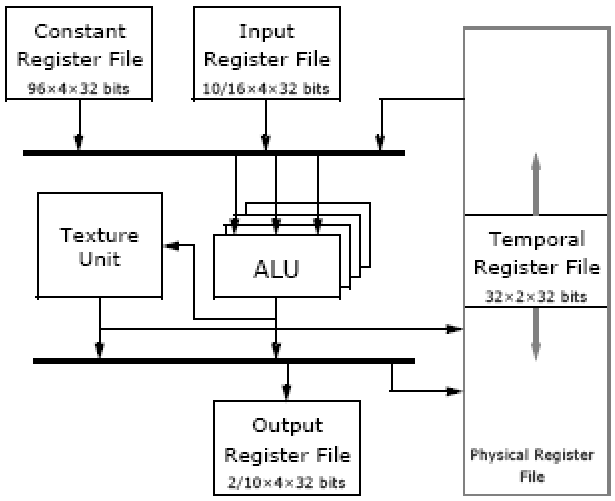

Constant Register File 96×4×32 bits
Input Register File 10/16×4×32 bits
Texture Unit
ALU
Temporal Register File 32×2×32 bits
Output Register File 2/10×4×32 bits
Physical Register File

Figure 3.2 Unified Shader model based on the ARB specification (from [6]).

- **Primitive Assembly**: assembles the vertices from the vertex shader output into rendering primitives. It can be seen on figure 3.1, below the Vertex Fetch box. It supports five kinds of OpenGL triangle primitives: lists, fans, strips, quad lists, and quad strips, but it can not support lines or points yet what implies that the primitives points, lines and lines strip will be ignored and not showed into the output. This unit uses a small memory to store the last vertices and assemble the new object (triangle), and also it uses some specific registers that controls this queue.
- **Clipper unit**: selects the triangles that are completely or partially inside

the view frustum[3] volume. It can be seen on figure 3.1, below the Primitive Assembly box, called as Clipping. It uses a buffer for the clipped triangles and several registers as clippedTriangles (store the count of triangles clipped) and frustumClip that works as a flag.

- **Rasterization**: makes fragments from the culled triangles. It comprises the Triangle Setup, Fragment Generator, Hierarchical Z Test and the Interpolator. However, as seen in the pipeline from the above scheme (figure 3.1), Triangle Setup, Hierarchical Z, Z and Stencil tests and Interpolator stages are outside the Rasterization stage, but actually they are all together in the same box on the source code. This stage is represented in figure 3.1 as a box called Rasterization, placed below the Triangle Setup box.
    - **Triangle Setup**: calculates several equations in order to rasterize the primitives and do interpolation, these are, the triangle edge equations and depth interpolation equation.
    - **Fragment Generator**: uses an iterative algorithm for transforming the area of a triangle projected on the viewport to fragments. Each fragment will represent a pixel, but it is not a pixel yet, for example it has some attributes as 2D screen coordinates, value of the three edge equations of the respective triangle, a cull flag and the depth (for Z tests).

      Fragments are generated in a set called tile that is useful for achieving performance with the access to the memory using the locality behavior and also for the implementation of the Z stages (Hierarchical Z buffer and Z compression). The tile's size can be configured in 3 levels: the highest can be set to the fit of a memory page, the middle level to the fit of a single fragment processing unit in the pipeline and the smallest to the fit of a HZ block or framebuffer cache lines. The middle and smallest level are set to 8x8 fragments. Moreover, there are two different implementations of Fragment Generator, one based on the algorithm described for Neon architecture and the recursive rasterization algorithm described by McCool.
    - **Hierarchical Z**: removes non visible fragments at a very fast rate as 8x8 fragments per cycle in the default configuration. Non visible fragments can be the ones culled by the fragment generator or fragments outside the scissor window. This stage can be removed easily just changing a value of a configurator parameter (called DisableHZ). For achieving this fast rate, it is used a cache and an on chip buffer that needs just 256KB for resolutions of 4096x4096 with 8 bits of Z precision. After that, the fragment tiles are divided into fragment quads (2x2 fragments) for the next stages. As we can see on figure 3.1,

[3] View frustum: In 3D computer graphics, the **viewing frustum** or **view frustum** is the region of space in the modeled world that may appear on the screen; it is the field of view of the notional camera. (Source: www.wikipedia.org).

there are two ways where the outputs of this stage can go: to the Z and Stencil tests or to the Shaders. The former is the default option; the latter is for supporting the OpenGL standard when using alpha and fog modes, where it is supposed that the alpha testing is before the Z and Stencil tests but after fragment shading. The problem is that alpha and fog calculations are made by a shader, so Shader Units (running as Fragment Shaders) can be seen as the Alpha and Fog stages not implemented. Then, when alpha and/or fog are used, the output of the HZ (Hierarchical Z) goes to the alpha unit (that is, the discontinuous line to Shader units) and the outputs of Shader units come back to the Z and Stencil tests.

    - **Z and Stencil tests**: makes a test of fragments using a buffer for Stencil with 8 bits per element and a buffer for Z with 24 bits. These tests are based on the OpenGL specification, where Stencil culls the fragments if they have the right value into the buffer (configured by the OpenGL code), and Z buffer makes a strict test with the depth value of each fragment. The Z buffer is stored on the GPU memory, so a cache and a compression algorithm are placed to reduce bandwidth usage: the Z cache has a size of 16KB, with 16 lines, 256 Bytes per line and 4 associatively; and the compression algorithm achieves ratios of 1:2 or 1:4 without lose information. This stage is also called early Z because its use before the shader unit. This stage is represented by the ROPz box in figure 3.1, just after the Hierarchical Z stage.
    - **Interpolator unit**: interpolates the attributes of the fragment that passes the Z and Stencil tests across the primitive being rendered, what is something necessary for fragment shader operations. This stage uses the perspective corrected linear interpolation algorithm described on the OpenGL API for varying variables on the OpenGL Shading language.

- **Fragment Shader**: when unified shader model is selected, it is used the whole shader unit explained before, but if it is working with a non unified shader model, the Fragment Shader unit are also the same of the whole unified shader unit while the Vertex Shader unit are just the Shader Unit without the texture components (in the implementation code there is only one shader unit that implements the unified model).
- **Color Write Unit**: updates the framebuffer, where the outputs frames are stored waiting for being drawn into the screen. The architecture of this unit is similar to the Z and Stencil tests Unit, where the framebuffer is stored into the GPU memory and then, it is used a cache with the same configuration of the Z cache. Also it is supported a fast clear operation of the buffer. This stage is represented as the ROPc box in figure 3.1.
- **DAC**: is the unit in charge of dumping the framebuffer into the output. In real GPUs the output is the screen, in the simulator the output is a file .ppm (see 3.3).

- **Memory Controller**: makes an interface with the GPU memory and system memory. The access to the GPU memory is based on the GDDR3 standard using a width of 64 bytes per 4 cycles; however the effective bandwidth is 64 bytes per cycle with 4 channels used on the baseline pipeline. On the other hand, the access to the system memory can be simulated with AGP or PCI Express x16 bus (two channels for read and write).

In summary, the units of the pipeline that access to the memory are: CommandProcessor (it reads the commands that the CPU sends to the GPU), Streamer (it fetches and loads the vertices sent from the CPU), ZSTencilTest (the ZBuffer is stored on memory), ColorWrite (the framebuffer is stored on memory), DAC (it dumps the final frame to a file (simulation) or to the video memory), and TextureUnit (it access to the texture memory map).

The pipeline explained in this section is the baseline pipeline that the simulator supports. It is possible to configure the pipeline by parameters, for example, in number of units, latencies or configuration of caches and memory (see Appendix C for further information), but none of these parameters implies a depth or novel change to the pipeline, and if it is needed to change the pipeline in the manner of include a new rasterization algorithm (for example the Direct 3D rasterizer implementation) or place a new stage on the pipeline it is sure that the simulator's source code must be modified.

3.1.2 Architectures not based on the baseline pipeline: Introduction to the source code

In order to simulate a new architecture that differs from the baseline pipeline and the pipeline changed by the parameters supported by the simulator, it is needed to make a new code for that. ATTILA was also implemented taking the idea of future modifications in mind. The code is object-oriented (using the C++ language) and implements a model of boxes and signals that provides an abstraction level helpful for developers. Figure 3.3 (next page) represents the reduced class diagram where it can be seen the three main classes: Box, Signal and Statistics.

Statistics is an abstract class that is based on keeping events using the method inc(). The NumericStatistics class is defined to store the number of times of one or more events. All the instances of Statistics are managed by other class called StatisticsManager, where the Statistics objects are stored, dumps theirs values into the output file in one cycle (by the method clock()) and search by name into the set. Each box (and also the caches) can use several Statistics instances for measuring information. For further information about the Statistics that ATTILA supports, see section 3.2 and Appendix B.

Signal is a class that models a buffer with a configurable latency; it means that when data is wrote on the buffer it is not possible to read it some cycles later. It is common that each box has Signal instances for input and output. Usually there are only two boxes related with one signal, one that write data (this signal is an output for it) and another that read data (this signal is an input for it). The class SignalBinder manages all Signal instances on the system, where it is possible to dump all the Signal states or also it is possible to search a Signal by name.

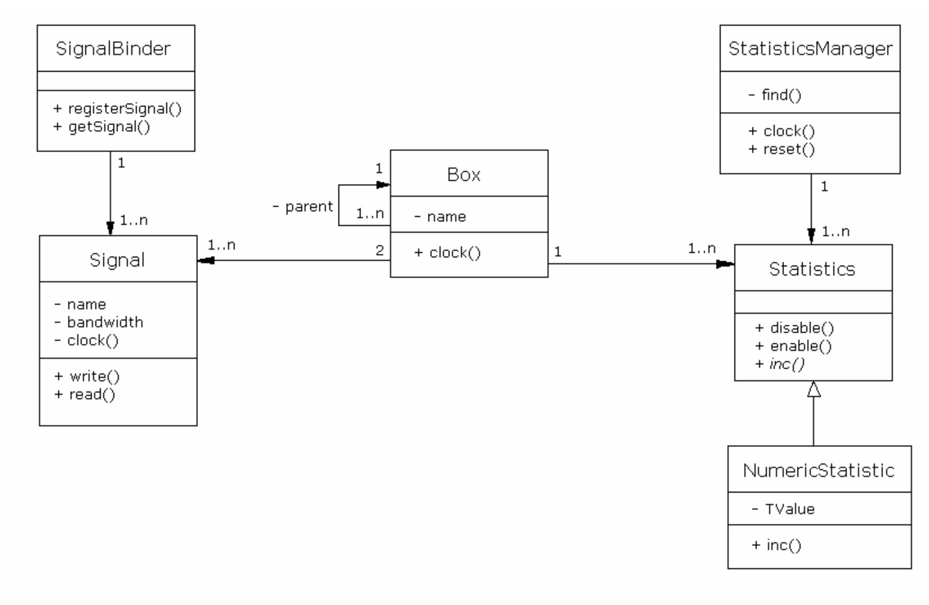
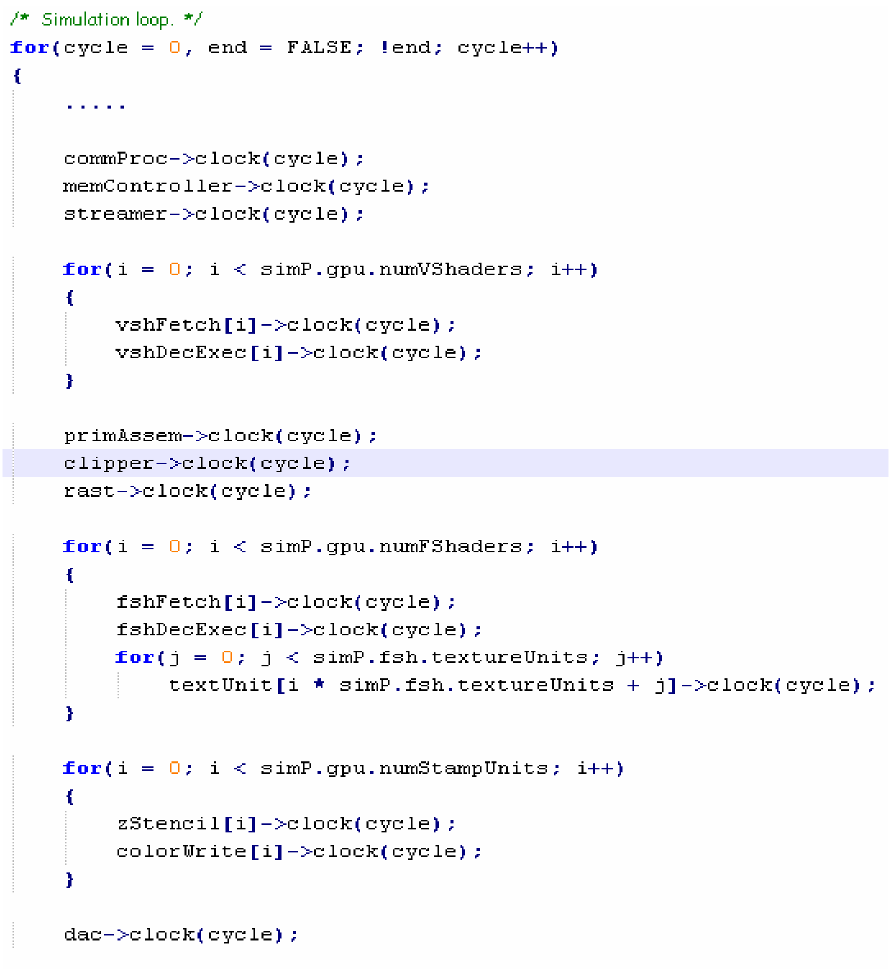


Figure 3.3 Abstract class diagram of ATTILA simulator.

The stages and other functions of the pipeline are implemented as subclasses of Box. A Box object has always input and output signals and many Statistics objects, thus, it is possible to say that these three classes (Box, Signal and Statistics) are the ingredients for a new pipeline.

A new functionality in the architecture can be implemented as a Box subclass, without varying so much the rest of ATTILA's code. Therefore, the skeleton of the pipeline is defined when the signals are associated to each box. Every Signal object has a name that acts as identifier, and each box that wants to use a signal have to know the associated name. By this way, two connected boxes have to be agree on the name of the share signal, hence the writer box that has the signal as output has to call the method newOutputSignal (in Box class), and the reader box the newInputSignal. These functions register a signal into the SignalBinder object on the first call, but on the next, if the signal is already registered it returns the respective Signal object.

Also this order can be seen easily again on the synchronization of the stages, that is, the execution of one cycle by each stage. The execution of each stage is done sequentially using the method clock() that Box class has; firstly one stage executes a cycle, and when this stage finish, the next one can execute its cycle. This order has to respect the organization defined by the connections with signals between the boxes, if not, the simulation will crash because the data processed in one stage can not go to the next, in exception of parallel stages or just forwarded stages (forward the information but does not change any data). The sequential execution of the stages is done by the simulation loop which structure is showed in figure 3.4. This portion of code shows a non-unified architecture where several vertex and fragment shader units are simulated by arrays of Box objects.



```
/* Simulation loop. */
for(cycle = 0, end = FALSE; !end; cycle++)
{
    .....

    commProc->clock(cycle);
    memController->clock(cycle);
    streamer->clock(cycle);

    for(i = 0; i < simP.gpu.numVShaders; i++)
    {
        vshFetch[i]->clock(cycle);
        vshDecExec[i]->clock(cycle);
    }

    primAssem->clock(cycle);
    clipper->clock(cycle);
    rast->clock(cycle);

    for(i = 0; i < simP.gpu.numFShaders; i++)
    {
        fshFetch[i]->clock(cycle);
        fshDecExec[i]->clock(cycle);
        for(j = 0; j < simP.fsh.textureUnits; j++)
            textUnit[i * simP.fsh.textureUnits + j]->clock(cycle);
    }

    for(i = 0; i < simP.gpu.numStampUnits; i++)
    {
        zStencil[i]->clock(cycle);
        colorWrite[i]->clock(cycle);
    }

    dac->clock(cycle);
```

Figure 3.4 Main loop of the execution of the simulator.

3.2 Statistics and measures provided

The most important component of ATTILA is, undoubtedly, the wide kind of statistics that it provides. It is essential because statistics permit to analyze the results of the simulations, in other words, the main goal of ATTILA is to simulate a GPU architecture, test it with graphic data from real applications and analyze the tests using the collection of several statistics about the simulated architecture. ATTILA provides up to 204 different statistic variables that measure all the components of the pipeline.

When a simulation finishes, the statistics are stored into an output file called stats.csv. It can be opened with Microsoft Excel or compatible (i.e. Open Office Calc or an editor as Notepad) and has a structure of columns: each column represents the values of each statistic variable, and each row represents the range of cycles showed into the first column. A statistic variable is a counter that characterizes the number of times an event happened inside a range of cycles. Figure 3.5 shows the first column of stats.csv (the range of cycles) and the statistic variable BlockedThreads. For example, the first value of BlockedThreads, 42048, means that between the cycles 0 and 9999, there were 42048 blocked threads.

Cycles	BlockedThreads_ShF-FS0
0..9999	42048
10000..19999	55032
20000..29999	10632
30000..39999	17472
40000..49999	13824
50000..59999	53952
60000..69999	46848
70000..79999	26112
80000..89999	32640
90000..99999	19200
100000..109999	21696
110000..119999	19848
120000..129999	28920
130000..139999	0

Figure 3.5. Example of columns from file stats.csv.

The format of a variable's name is like this:

what_unit-id

Where "what" means what is measured (on the last example, BlockedThreads), "unit" means the unit where the measure is applied (ShF is Fragment Shader) and "id" means the instance of the unit measured (FS0 is the Fragment Shader with id 0).

In Appendix B there are a summary of the statistic variables and units that ATTILA can measure, from hits and misses of the several caches to culled fragments. It can be showed that there are a wide range of variables that measure the behavior of the simulated architecture. As explained in section 3.1.2, it is also possible to implement your own statistic variables in case you need something else.

3.3 ATTILA's framework

One of the purposes of the ATTILA project was to execute real applications with the simulator, as a real GPU does in the real life. To achieve this, the authors developed a framework that permits the simulator to do tests with real OpenGL applications (a Direct3D framework is in construction). This framework has 4 stages and it is composed of 6 elements, as seen on figure 3.6.

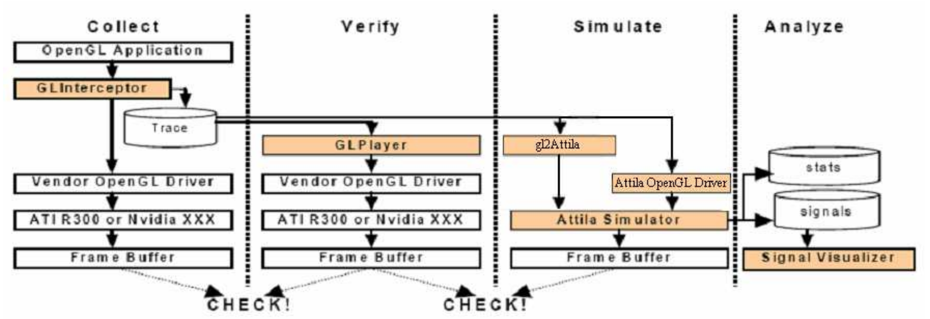


Figure 3.6. Adaptation of the diagram of OpenGL ATTILA's framework from [6]

The first stage is called *Collect*, and it is the step where the data from the real applications is captured as a trace file. The element used here is GLInterceptor that is composed in others two elements: opengl32.dll and GLIConfig.ini. The latter is the configuration file for GLInterceptor, and the former is a library that will be used by the real application as the real OpenGL library. This library dumps all the OpenGL API calls into a trace file, and also forwards the calls to the real library, permitting to continue with the application's execution. The way to use it is just to copy the two files commented above to the same folder of the application's binary. See that the library is a .dll file, which means that it can be used only in a Windows platform. Only the Simulate and Analyze stages can be done with Linux platforms. After the execution of the application, three files will be created: tracefile.txt (that has the API function calls and parameters, for example, glBegin(GL_QUADS)), BufferDescriptors.dat (that keeps the buffers pointed by a pointer parameter (the value of the pointer is now an identifier that index in the Buffer Descriptors file) and MemoryRegions.dat (that stores the data passed as arrays in a number of OpenGL API calls (glTexImage2D for example)).

The second stage is called *Verify*, and it is where a verification of the trace file is performed. It ought to be done to verify the correctness of the traces before the execution with the simulator. The element GLPlayer is used in this stage and it permits to run the trace calls into the real OpenGL driver again. Then, it is possible to see if the trace was well done. GLPlayer has two elements: GLPlayer.exe and GLPConfig.ini. The former is the program that executes the trace and the latter is the configuration file.

The third stage is called *Simulate*. Here there are a distinction between Linux, Cygwin and Mingw32, and Visual Basic environments. If a Visual Basic system is used, the ATTILA OpenGL Driver (gllib.a) will be utilized, if not, the gl2attila tool has to be used. Both of them have the same behaviour: translate the OpenGL calls from the trace into AGP transactions for ATTILA. The driver permits the simulator to execute the trace directly, while gl2attila is a program that translates the OpenGL trace to an AGP transaction trace file that the simulator can execute. It is recommended to place the simulator (unified or non-unified version) and configuration file (renamed to bGPU.ini) into the same folder of the traces (or vice versa, place the trace files into the simulator folder) or changes the configuration file.

The forth stage is called *Analyze*, and it is where the simulator outputs are analyzed. The output files that the simulator provides in the third stage involve the frame dumped (framex.pnm), the statistics (stats.csv, explained on section 3.2) and the debug file (Signals in figure 3.6, that corresponds to the file signaltrace.txt, the documentation does not explained yet how to use it). This is the stage where the developers or researchers have to take the information from the simulator and make their conclusions.

4. Evaluation of the simulator

The next tests was run in a bounds of a system that has these features: processor Intel Pentium M750 at 1,86Ghz, 2MB of Cache L2, 1024 MB of memory with technology DDR2 and a graphic card ATI MOBILITY RADEON X600SE with 256 MB HyperMemory (128 MB shared and 128 MB dedicated). The environment used was Cygwin with a Windows XP SP2 Operating System. In every simulation, the ATTILA version was the one from 13/01/07 (17/01/07 for Prey traces) and the configuration file was ATTILA-rei-580.ini (see folder confs).

4.1 Installation of ATTILA

ATTILA simulator is a non-commercial simulator. This product is distributed by the authors, and nowadays it can be found in the ATTILA’s log [8]. There are mainly two distributions: with just the binaries (plus GLPlayer and GLInterceptor) and with only the source code of the simulator (no GLPlayer neither GLInterceptor included). This section will explain how to use and compile the simulator distribution built on October of 2006. This version had some problems when compiling, nowadays fixed by updates from December of 2006, explained at the end of the section too.

The installation and usage of the binaries is very simple, just decompress the file and then use the folder according to the system used, either Linux (folder “binpack/linux”) or Windows (folders “binpack/cygwin”, “binpack/mingw32” or “binpack/vs2005”). Each folder has the binaries bGPU and bGPU-Uni, the former is the binary that corresponds to the implementation of a GPU with non-unified shaders, and the latter corresponds to unified shader architecture. But ATTILA’s framework has more elements than the simulator, like GLInterceptor and GLPlayer that are used for capturing the OpenGL calls in a trace and replay them, respectively (more detail on the next section). The files for these two elements are placed in the folder “binpack/vs2005” that limits the use of them only in a Windows system (there are not files for Linux yet). The explanation of the parameters and way of use are in the file USAGE.txt, for each element provided for the ATTILA´s framework.

The installation of the source code is a bit more complex, but not difficult. The most important thing is to be sure that we are using an operating system and environment compatible with the simulator, that is: Windows with Cygwin software installed, Windows with Mingw32 software installed, Windows with Visual Studio 2005 installed or Linux using a gcc version 3.X (not later), all of them running in a 32 bit system. If the simulator is tried to be compiled into another environment, it is very probable that it can not. The problem of this low portability is due to the pre-compiled file gllib.a, provided to simulate an OpenGL driver for the simulated graphic card. The code of this library is not public available, so the compilation of the simulator is restricted to the different versions of gllib.a found on the folder "lib". The way of choosing one is to change the name of the right file according to the system used to gllib.a, in the same directory "lib".

The compilation of the simulator depends on the environment used: when Visual Basic 2005 just open the created project on the folder win32 and compile it with the options that Visual Basic provides; when Linux, Cygwin or Mingw32 just edit the file makefile.defs in the manner that the variable GPU3DHOME has as value the absolute address of the folder where the files are placed. The creation of the executable files is performed by makefile when a non Visual Basic environment is used. This makefile has several options that are used like:

```
make exec mode
```

make is the Unix instruction for executing a makefile. The parameter exec can be bGPU (compile only the non-unified shader simulator version), bGPU-uni (compile only the unified shader simulator version) or all (compile both versions). The parameter mode indicates the format of the final executable files that can be debug (the executable file can be debugged (use of the gcc option –g)), profiling (it is the simple version of the simulator), optimized (it is used the optimization options that gcc provides) or verbose (the simulator will give all the execution information in the standard output).

At the date of this current report there was a new version built on December of 2006 (see [8]) that help to compile the simulator with Linux in an easier and more portable way than the October version explained in this section. In this version, the library gllib.a is not used in Linux, Cygwin and Mingw32 anymore, where there were portability problems in special with Linux environments. The way to proceed to compile is the same than before, in exception of choosing any gllib library because now the job of this library is performed by a program called gl2attila (provided into the binaries distribution). The previous library translates the OpenGL API calls to AGP transactions, and now gl2attila translates the OpenGL API call trace into an AGP transaction trace file, which can be used directly as an input for the simulator.

4.2 Running OpenGL testprograms on ATTILA

4.2.1 Testprograms selected

The best way to understand ATTILA and one goal of this current report is to work with the simulator. As explained on section 3.3, ATTILA framework is prepared to work with applications that use the OpenGL API, which means that programs that use another API different to OpenGL as Direct3D or OpenGL ES (OpenGL for Embedded Systems) is not going to work. GLInterceptor is able to trace only OpenGL API calls by the substitution of the real OpenGL library with the provided library. This provided library (opengl32.dll, see section 3.3) cannot capture Direct3D or OpenGL ES because mainly two reasons: an application that uses OpenGL ES, for example, is not going to search and use the file opengl32.dll, and the libraries OpenGL and OpenGL ES do not have the same entries or format in several functions, and it implies that it does not work anymore with the current framework for OpenGL (it is needed to develop a new framework for this library (see the explanation of the author in [9]), for example, the one for Direct3D is under construction).

However, the task of choosing OpenGL testprograms is not easy. The problem is that GLInterceptor (GLPlayer and the simulator too) does not support calls that implements a complex behaviour (i.e., reading data buffers, handling 'objects' (like textures or vertex buffers)) or extended functions for ATI or NVIDIA GPUs. Section 3.1.1 shows that it is supported only ARB shader implementations (ARB_vertex_program and ARB_fragment_program), so the testprograms must have disabled options as GLSlang or shader objects. It happens because ATTILA only implements a minimum set of OpenGL API.

The variety of applications that can be traced and executed is very small, even more, the authors recommend using the applications that they tested before; other program out of these tests is likely going to fail. Also it is possible (and recommend by the authors too) to implement applications that uses just the OpenGL calls that ATTILA implements. The list of OpenGL functions supported was published in the simulator version of 13/1/2006.

The programs tested and recommended by the authors are: Volumetric Lighting II (by Humus (www.humus.ca)), Unreal Tournament 2004[4] (only tested with NVIDIA cards), Doom3, Quake4, Prey and Chronicles of Riddick. Each game needs a configuration that allows to ATTILA trace and work with them. Also can be possible to work with games using engines based on Doom3's engine[5] (Quake4 and Prey are), but it is not sure because not all was tested. Even more, the traces captured depend on the GPU that the system has due to the no implementation of specific OpenGL extensions (for ATI or NVIDIA). In this case, it was tried to capture traces from the Unreal Tournament 2004 Demo, but it was not possible because, as said before, the system uses an ATI GPU or maybe the usage of the incorrect demo version (that the authors did not try). Moreover, in this work, several games out of these tests was tried to be traced and simulated with ATTILA. Some of them were: Angeles (San Angeles Observation OpenGL ES version example by Jetro Lauha (http://iki.fi/jetro), an OpenGL ES application translated to OpenGL and well traced by GLInterceptor, but it uses a mode called colorMaterialMode and it is not supported in the simulator; traces from the OpenGL ES version was not possible to capture), HalfLife (the game crashes after the execution and the simulation is not possible), Quake3 (similar effect than with HalfLife) and GoogleEarth (the application crash before been executed, so it is not possible to take traces). Even more, several OpenGL ES applications were tested and it was not possible to capture it, as said before. It was tried to change the file libGLES_CM.dll (the OpenGL ES library) with the opengl32.dll (the OpenGL

[4] Unreal Tournament 2004 has to be configured with the following options in the configuration file (system/UT2004.ini): activate the use of OpenGL driver (OpenGLDrv.OpenGLRenderDevice) and not Direct3D, activate rendered vertex buffer and disable vertex shader.

[5] A game that use the Doom3 engine like Quake4, Prey or the same Doom3 has to be configured with the configuration file with the next options: disable two sided stencil, index buffers and copyToTexture, and enable arb2 render path and vertex buffers.

library provided with ATTILA), and of course it crashed because several call entries was not found.

The chosen testprograms was Doom3 (with the trace file provided in the log page), Prey and Volumetric Lighting II. They was tested by the authors and show how to work with pre-captured traces (Doom3), a well-known program that works perfectly with ATTILA (Volumetric Lighting II) and a game that does not work wholly with ATTILA (Prey). Another reason because they was chosen is the real graphic data that they provide (in the other hand we can make our programs, but they don't provide the reality and power than a professional application offers to ATTILA).

4.2.2 Execution and collection of data

The way to capture traces, simulate and collect the statistics is the same as explained on section 3.3 for ATTILA's framework. In this section it will be presented some examples that show how ATTILA works and some aspects to take in account when using the simulator.

Finally, figure 4.1 shows a summary of what was able to do and what not with the testprograms selected and many others, with the simulator version on 17/1/2007. This last version can simulate traces from Prey with uniform shader version (before there was an infinite loop with shaders and it never finished the simulation for frames from the game, but yes from the loading screen). The times showed are approximately, that is, it is the mean of two simulations, but it can vary from different Operating Systems and computers.

	Volumetric Lighting II	**Prey**	**Doom3**	**Quake3**	**Angeles (OGLES version)**
Traces captured	OK	OK	OK (downloaded)	OK	FAILED
Gl2attila translation	OK	OK	OK	FAILED	FAILED
Playable by GLPlayer	FAILED	FAILED	OK	FAILED	FAILED
Simulation with uniform shaders	OK	OK	OK	FAILED	FAILED
Simulation with non-uniform shaders	OK	FAILED	FAILED	FAILED	FAILED
Time of simulation	12m 53sec 1st frame	24m 51sec 800th frame	13m 22sec 200th frame	FAILED	FAILED

Figure 4.1. Summary of what ATTILA can do and what not (version of 17/1/2007).

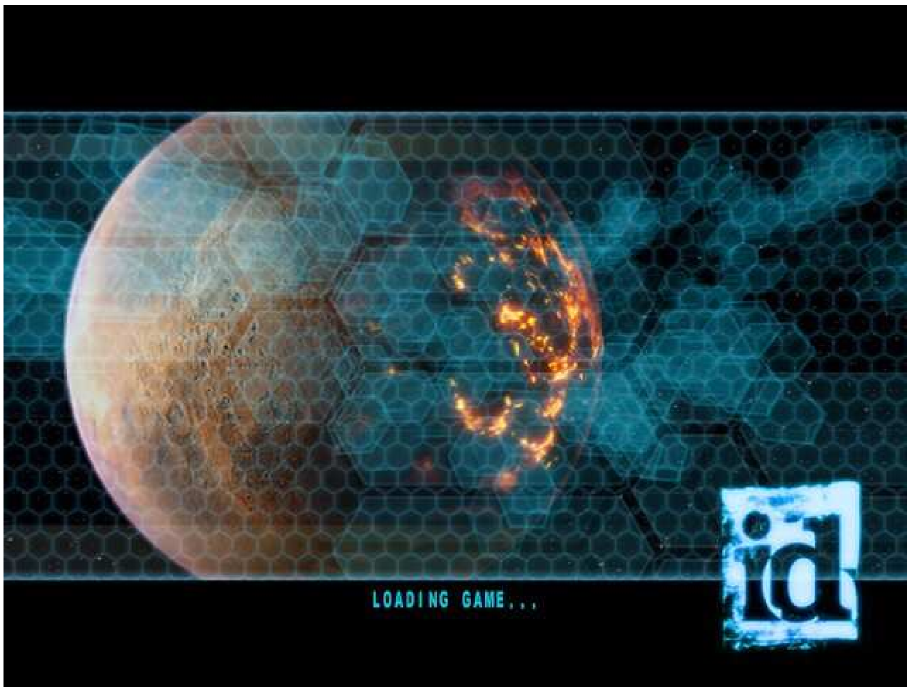
4.2.2.1 Captures from testprograms

GLInterceptor has to be copied into the application binary folder (opengl32.dll and GLIconfig.ini) for the capture of traces from the testprograms (first stage of the framework). So for the Volumetric Lighting II (VLII), just copy these files into the folder VolumetricLightingII, for Prey into the main folder. The Doom3 traces were taken from [8]; it was created and posted by the authors, so just decompress these files into a folder to work with them.

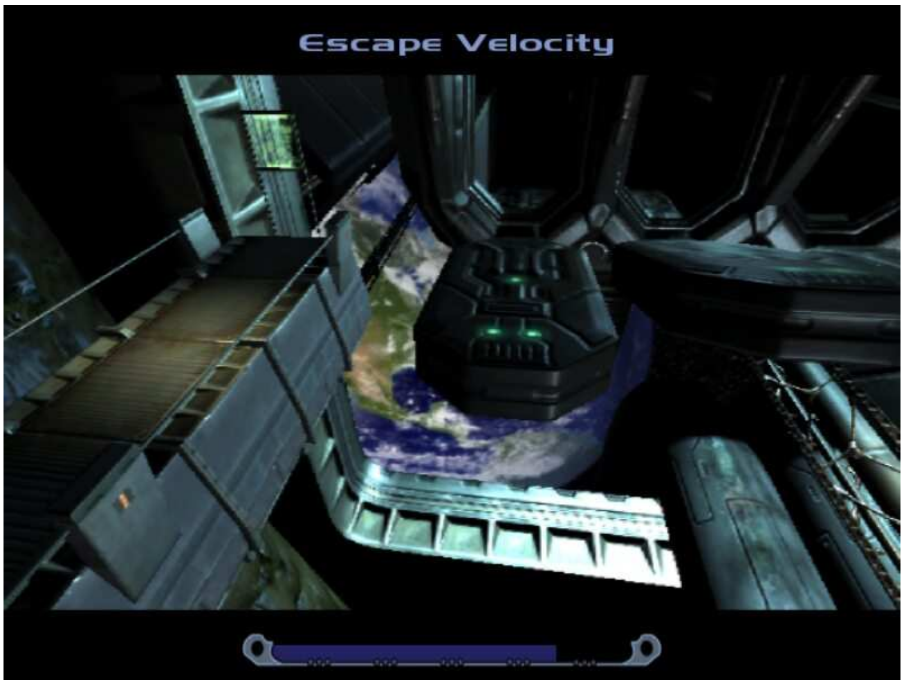
Once the files from GLInterceptor have been copied, it is possible to configure this tool by opening the file GLIconfig.ini and change some options like lastFrame (how many frames to capture, consider that 25 frames correspond to one second) or outputFile (the default name is tracefile.txt). Altering the parameter startFrame (from which frame start to capture) is not recommended because it can have some problems later with the simulation, so GLInterceptor does not support very well a hot start (however, the simulator does). If the application needs a special configuration (see 4.1.1), try to configure it before executing the program. VLII does not need any special configuration, and Prey needs to add and change some options (the options for a game that uses the Doom3's engine). When everything is configured, play a bit and GLInterceptor will work automatically.

Note that if the startFrame parameter is configured to 1 (recommended), then GLInterceptor will start to capture frames from the beginning of the first usage of OpenGL. In most games, menu and loading screens are implemented by OpenGL. It means that GLInterceptor will trace both menu and loading screens, as seen on figure 4.2. This figure shows that the first frames from a trace always corresponds to this kind of screens that not correspond to the rendering part of the game (the interactive and real game, not screens, videos, presentations or menu), which is the most interesting part to simulate. In Doom3 traces (4.2a) there are a loading screen capture, whereas in Prey traces a presentation, the menu (4.2b) and loading screen (4.2c) was captured. So be fast when loading the game for making the traces, if not, maybe you will not have time to start the game.

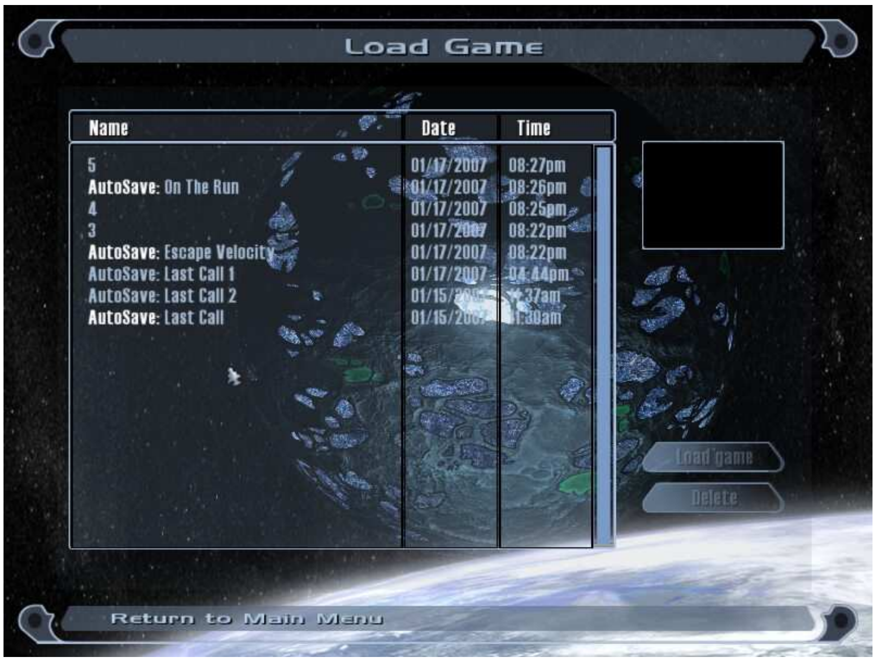
Angeles is an application with two versions, one for OpenGL and other for OpenGL ES. As show in figure 4.1, GLInterceptor was not able to capture traces from this application on its version using OpenGL ES (the reason is in section 4.2.1). But GLInterceptor was able to take a trace from OpenGL version, but as seen after that, it was not possible to make the translation with gl2attila. The trace from Quake3 also was captured, but this game always finished with an error panic when using GLInterceptor.



(a)



(b)



Name	Date	Time
5	01/17/2007	08:27pm
AutoSave: On The Run	01/17/2007	08:26pm
4	01/17/2007	08:25pm
3	01/17/2007	08:22pm
AutoSave: Escape Velocity	01/17/2007	08:22pm
AutoSave: Last Call 1	01/17/2007	04:44pm
AutoSave: Last Call 2	01/15/[illegible]	[illegible]:37am
AutoSave: Last Call	01/15/[illegible]	[illegible]:30am

(c)

Figure 4.2. When capturing traces from the first frame, GLInterceptor will dump into the trace also the frames corresponding to loading screens and menus. In this work there was captured (a) a loading screen from Doom3, (b) a loading screen from Prey and (c) loading menu interaction from Prey.

4.2.2.2 Replay of the captured traces

After finishing the application, some files are created. These files are the traces, and their names are tracefile.txt, MemoryRegions.dat and BufferDescriptors (if not other names were chosen), as explained on section 3.3. From this point, only the trace files are necessary (not game's files), so they can be moved to other folder (always the three together) or keep them in the same folder and copy GLPlayer and the simulator to it.

Then, it is possible to go to the second stage of the framework, Verification. Configure GLPlayer by the file GLPconfig.ini (maybe the most important is the name of the input file), and execute GLPlayer. In this work, GLPlayer did not work correctly with VLII and Prey traces: it did not show any frame and even more, it delete the content of the trace files (for example, the trace from Prey that is about 200 MB, after the execution of GLPlayer the trace became to only 1KB). The traces that GLPlayer can play are from Doom3 (downloaded directly from the authors, not captured with GLInterceptor in this work).

Figure 4.3 illustrates an example of GLPlayer. It is just a window that shows information like the number of frame, the resolution used and the current frame. The speed of this play can be modified with the configuration file, but by default it goes a bit more slowly, because the most important is to be able to see the frames and capture them for making a posterior check with the simulated ones.

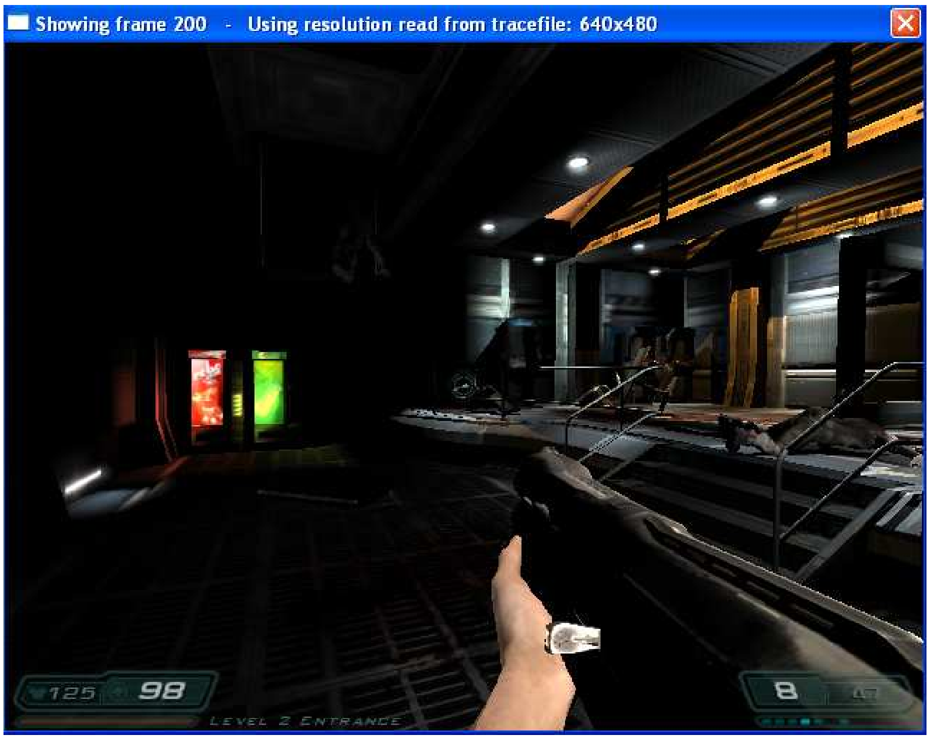

Showing frame 200 - Using resolution read from tracefile: 640x480
125
98
LEVEL 2 ENTRANCE
8

Figure 4.3. GLPlayer showing the frame 200 from Doom3 trace file. Volumetric Lighting 2 and Prey could not be played.

4.2.2.3 Execution of the simulator with captured traces

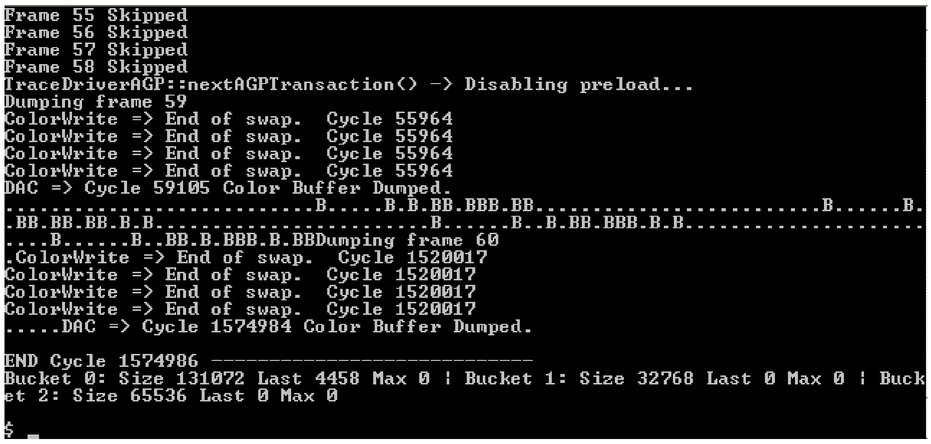
If the traces were checked but GLPlayer could not play them, it is still possible to go to the next stage, Simulation. In particular, GLPlayer was not able to play VLII and Prey traces, however, the simulator could simulate some frames correctly. For it, one version (Unified or non-Unified) of the simulator has to be selected, and copied together with one configuration file to the traces folder. Remember that if Linux, Cygwin or Mingw32 environment are used, then gl2attila has to be executed before the simulator (bGPU). Gl2attila also needs the same configuration file that the simulator will use (in this test, the example configuration file of ATTILA-rei-580.ini). In this work the simulator was executed within the bounds of Cygwin, so in every case gl2attila.exe was utilized by typing gl2attila.exe N, with N equals to the number of frames to translate (usually the whole trace, so equals to the number of frames captured). For example, ./gl2attila tracefile.txt 900 (translation used for Prey trace). It is possible to add other parameter after N, which means the starting frame (gl2attila.exe N M). It was tried but it did not work well when skipping frames in the simulation with ATTILA simulator. So it is recommended to capture and translate frames (gl2attila) starting always with the first frame, and then, it will be possible to skip and simulate frames different to the frame number 1.

The execution of the simulator is easy just typing the name of the binary (bGPU for non-unified version and bGPU-Uni for unified version), following the AGP trace translated by gl2attila (the common name is attila.tracefile.gz) and N M, two optional numbers that means the number of frames to simulate and the starting frame respectively. For example, ./bGPU-uni attila.tracefile.gz 1 800 (execution used for Prey, starting from the frame 800 and simulating only one frame, the frame 800th). The time of simulation of one frame depends on the application to simulate, which frame and the position of the frame (time elapsed for seeking the frame inside the trace depends of how many frames to skip for reaching the frame to simulate). As said before, a trace is captured from the first frame of the application; in a game it means that it will be captured the menu and loading screen too (see section 4.2.2.1). The simulation of a loading screen frame takes about 30 seconds for traces from Doom3, but a frame from the rendering part (interactive game) takes about 13 simulation's minutes.

Even more, the choice of one version is important. There are two versions, one for non-unified shaders and other for unified shaders. The version most tested by the authors and compatible is the unified one, whereas the non-unified is no longer tested because is based in a contemporary GPU but not as new as a unified architecture. That is the reason because the non-unified version works only with VLII, but not with Doom3 or Prey (it returns a simulator bug error after 12 minutes of simulation). As seen in figure 4.1, the non-unified version has problems with some applications; however, the unified version could simulate every well-translated trace file by gl2attila.

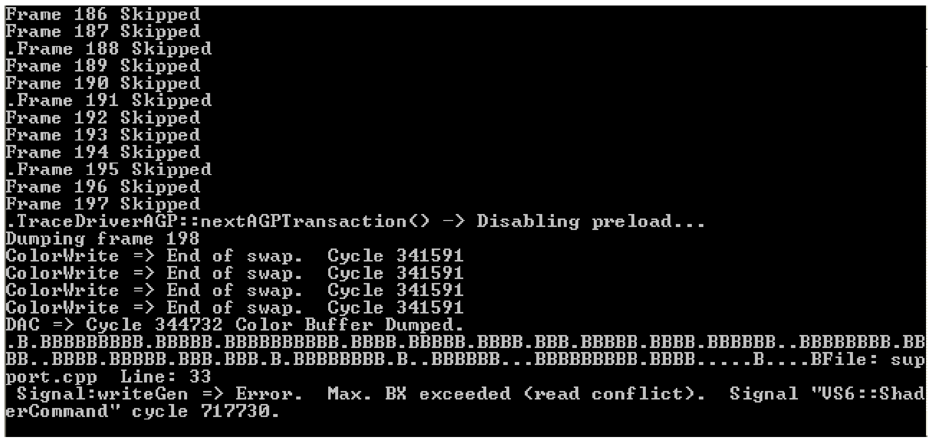
Figures 4.4b and 4.4c show the outputs from the non-unified simulator with Doom3 and Prey. They were not simulated correctly, but it was possible to have some statistics corresponding till the cycle 717730 for Doom3 and cycle 2996339 for Prey. Figure 4.4a shows how the finish of a well execution of the simulator is, in this case, using the non-unified version with VLII traces. Finally, figure 4.4d shows the error returned by gl2attila with the trace from Quake3. This trace contains an API call that ATTILA can not support, or maybe the trace is not well-finish because the bad exit that the game experiments when using GLInterceptor (it is not clear the reason).

As seen on the captured screens in figure 4.4, the simulator returns "B" and "." consecutively during the simulation. A "B" means that an OpenGL draw call (or batch) has been fully processed and a "." indicates that a number of cycles (10K by default) has passed.



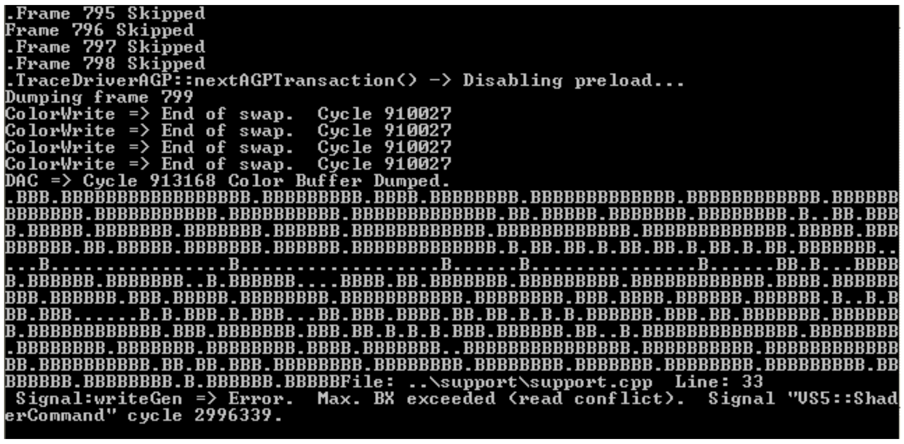
```
Frame 55 Skipped
Frame 56 Skipped
Frame 57 Skipped
Frame 58 Skipped
TraceDriverAGP::nextAGPTransaction() -> Disabling preload...
Dumping frame 59
ColorWrite => End of swap.  Cycle 55964
ColorWrite => End of swap.  Cycle 55964
ColorWrite => End of swap.  Cycle 55964
ColorWrite => End of swap.  Cycle 55964
DAC => Cycle 59105 Color Buffer Dumped.
...........................B.....B.B.BB.BBB.BB........................B......B.
.BB.BB.BB.B.B........................B......B..B.BB.BBB.B.B.....................
....B......B..BB.B.BBB.B.BBDumping frame 60
.ColorWrite => End of swap.  Cycle 1520017
ColorWrite => End of swap.  Cycle 1520017
ColorWrite => End of swap.  Cycle 1520017
ColorWrite => End of swap.  Cycle 1520017
.....DAC => Cycle 1574984 Color Buffer Dumped.

END Cycle 1574986 -----------------------------
Bucket 0: Size 131072 Last 4458 Max 0 | Bucket 1: Size 32768 Last 0 Max 0 | Buck
et 2: Size 65536 Last 0 Max 0

$ _
```

(a)



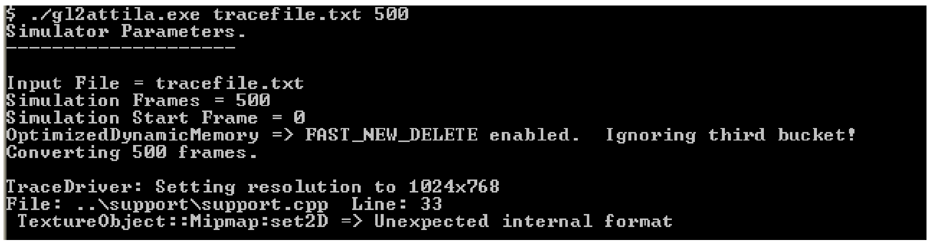
```
Frame 186 Skipped
Frame 187 Skipped
.Frame 188 Skipped
Frame 189 Skipped
Frame 190 Skipped
.Frame 191 Skipped
Frame 192 Skipped
Frame 193 Skipped
Frame 194 Skipped
.Frame 195 Skipped
Frame 196 Skipped
Frame 197 Skipped
.TraceDriverAGP::nextAGPTransaction() -> Disabling preload...
Dumping frame 198
ColorWrite => End of swap.  Cycle 341591
ColorWrite => End of swap.  Cycle 341591
ColorWrite => End of swap.  Cycle 341591
ColorWrite => End of swap.  Cycle 341591
DAC => Cycle 344732 Color Buffer Dumped.
.B.BBBBBBBBB.BBBBB.BBBBBBBBBB.BBBB.BBBBB.BBBB.BBB.BBBBB.BBBB.BBBBBB..BBBBBBBB.BB
BB..BBBB.BBBBB.BBB.BBB.B.BBBBBBBB.B..BBBBBB...BBBBBBBBB.BBBB.....B....BFile: sup
port.cpp  Line: 33
 Signal:writeGen => Error.  Max. BX exceeded (read conflict).  Signal "VS6::Shad
erCommand" cycle 717730.
```

(b)



```
.Frame 795 Skipped
Frame 796 Skipped
.Frame 797 Skipped
.Frame 798 Skipped
.TraceDriverAGP::nextAGPTransaction() -> Disabling preload...
Dumping frame 799
ColorWrite => End of swap.  Cycle 910027
ColorWrite => End of swap.  Cycle 910027
ColorWrite => End of swap.  Cycle 910027
ColorWrite => End of swap.  Cycle 910027
DAC => Cycle 913168 Color Buffer Dumped.
.BBB.BBBBBBBBBBBBBBBBB.BBBBBBBBB.BBBB.BBBBBBBB.BBBBBBBBBBBBB.BBBBBBBBBBBB.BBBBBB
BBBBBBB.BBBBBBBBBBB.BBBBBBBBBB.BBBBBBBBBBBBB.BB.BBBBB.BBBBBBB.BBBBBBBB.B..BB.BBB
B.BBBBB.BBBBBBB.BBBBBBB.BBBBBB.BBBBBBBBBBBB.BBBBBBBBBBBB.BBBBBBBBBBBB.BBBBB.BBB
BBBBBB.BB.BBBBB.BBBBBBB.BBBBBB.BBBBBBBBBBBBB.B.BB.BB.B.BB.BB.B.BB.B.BB.BBBBBBB..
...B................B...................B......B...............B......BB.B...BBBB
B.BBBBBB.BBBBBBB..B.BBBBBB....BBBB.BB.BBBBBBB.BBBBBBBBB.BBBBBBBBBBBB.BBBB.BBBBBB
BBB.BBBBBB.BBB.BBBBB.BBBBBBBB.BBBBBBBBBBB.BBBBBBBB.BBB.BBBB.BBBBBB.BBBBBB.B..B.B
BB.BBB......B.B.BBB.B.BBB...BB.BBB.BBBB.BB.BB.B.B.B.BBBBBB.BBB.BB.BBBBBBB.BBBBBB
B.BBBBBBBBBBBB.BBB.BBBBBBB.BBB.BB.B.B.B.BBB.BBBBBB.BB..B.BBBBBBBBBBBBBB.BBBBBBBB
.BBBBBBBB.BBBBBBB.BBBBBBBB.BBBB.BBBBBBB..BBBBBBBBBBBBBBB.BBBBBBBBBB.BBBBBBBBBBB
BB.BBBBBBBBBB.BB.BB.BBB.BBBBBBBB.BBBBBBBB.BBBBBBBB.BBBBBBB.BBBBBBBB.BBBBBBBBB.BB
BBBBBB.BBBBBBBB.B.BBBBBB.BBBBBFile: ..\support\support.cpp  Line: 33
 Signal:writeGen => Error.  Max. BX exceeded (read conflict).  Signal "VS5::Shad
erCommand" cycle 2996339.
```

(c)



```
$ ./gl2attila.exe tracefile.txt 500
Simulator Parameters.
--------------------

Input File = tracefile.txt
Simulation Frames = 500
Simulation Start Frame = 0
OptimizedDynamicMemory => FAST_NEW_DELETE enabled.  Ignoring third bucket!
Converting 500 frames.

TraceDriver: Setting resolution to 1024x768
File: ..\support\support.cpp  Line: 33
 TextureObject::Mipmap:set2D => Unexpected internal format
```

(d)

Figure 4.4. Output of ATTILA non-unified version (bGPU) for (a) Volumetric Lighting II, (b) Doom3 and (c) Prey. (d) Shows the error returned by gl2attila with a trace from QuakeIII.

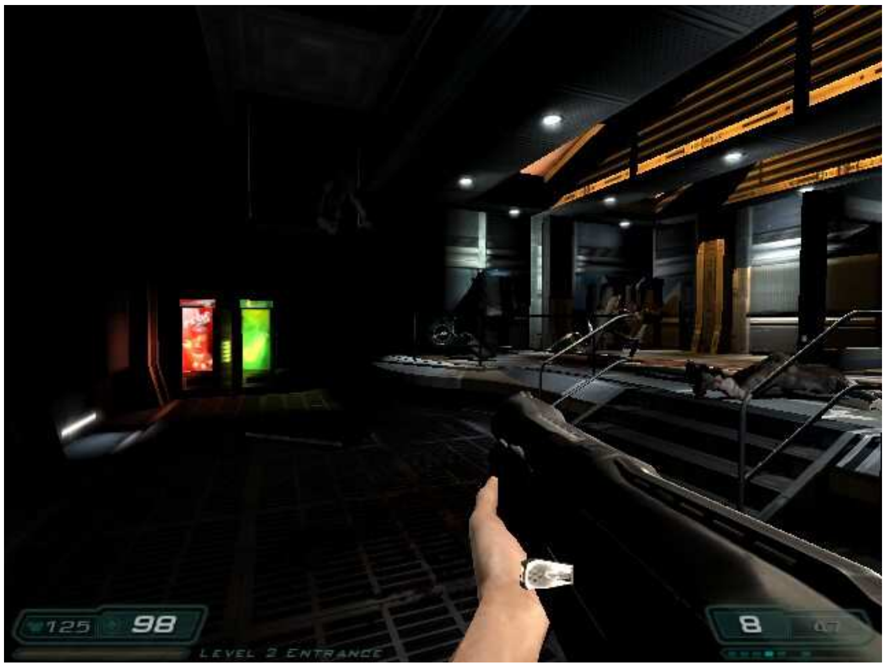
4.2.2.4 Analysis of ATTILA's outputs

When the execution of the simulator finishes (correctly or incorrectly), the outputs files are placed in the same folder than the simulator. These files were explained on section 3.3 and involve two images, one showing the final frame simulated and other a latency map (a per fragment quad map storing the execution latency of the last quad written in a framebuffer position, see [5]), and a statistic file. There is not so much information about what is the purpose or the complete meaning of the latency map.

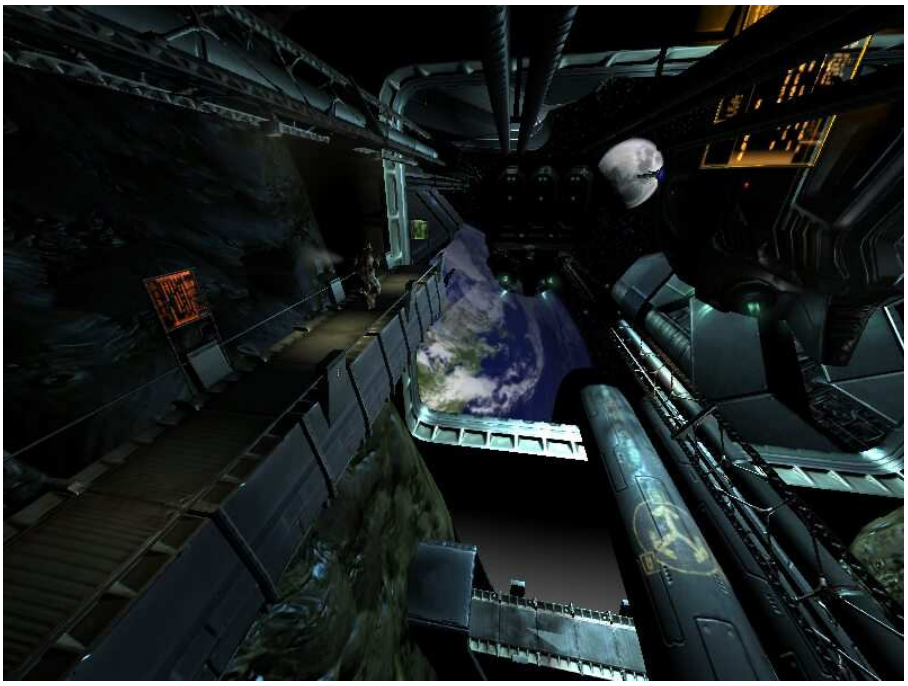
An example of the statistic file is figure 3.5, and examples of outputs frames are shown on figure 4.5. As seen in this figure, the outputs frames are exactly as shown in a real GPU. But not always it is like this, for example, on [5] there is a comparison between two frames from Unreal Tournament 2004 and there are some differences (problems when simulating). Of course, a frame that was not perfectly simulated means that the statistics will show data that not corresponds to the reality. In the example from [5], the fragments that correspond to lines or letches in the forest have not a correct value, so the statistics about these fragments can be incorrect. However, these statistics are very near to the reality, and the most part of these frames are correct. In short, if a simulation finishes correctly, the data, frames and statistics can be enough for making conclusions about the architecture simulated.

The way to check simulated frames is more or less by intuition if GLPlayer did not work with the trace. For example, we can see that the frame from Doom3 (4.5b) is ok if it is compared with the frame from GLPlayer (figure 4.3). But the frames from figures 4.5a (VLII) and 4.5c (Prey), that come from traces that GLPlayer could not play, have to be checked with intuition: just see if there is some typical problems with the rendering (shadows, objects, lights… missing and mistakes) or try to play again to the game, capture the frame manually and see that the simulated frame is ok. In any case, every capture from figure 4.5 seems to be correct.

(a)



(b)

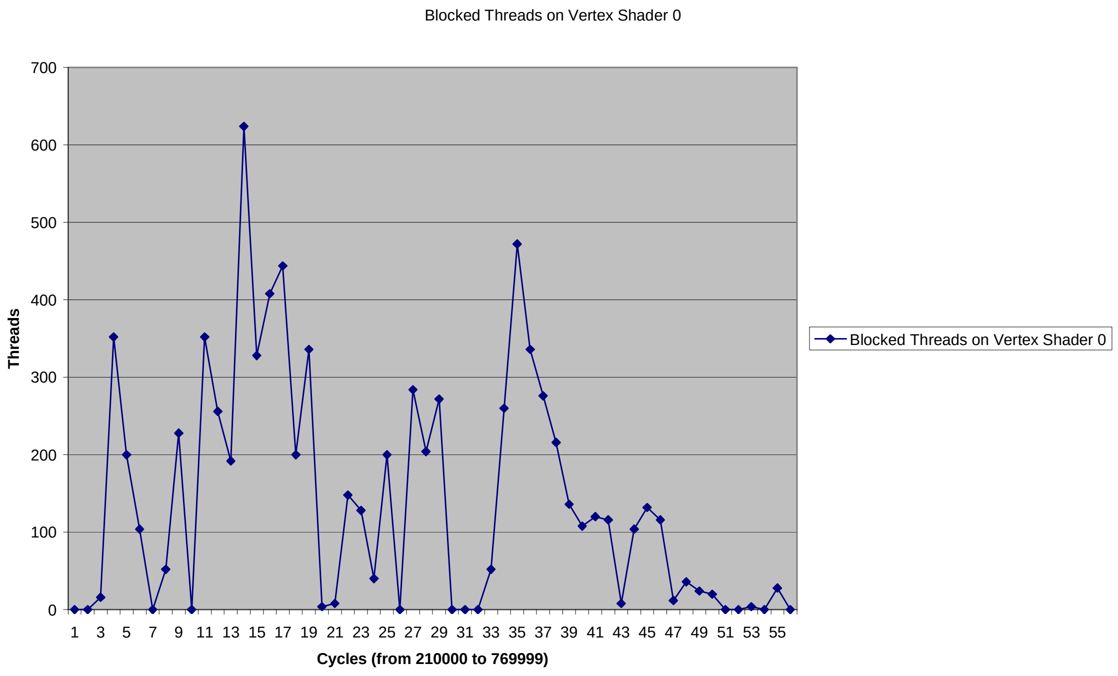


(c)

Figure 4.5. Frames simulated from (a) Volumetric Lighting II (frame 60), (b) Second level of Doom3 (frame 200), and (c) Second level of Prey Demo (frame 800)

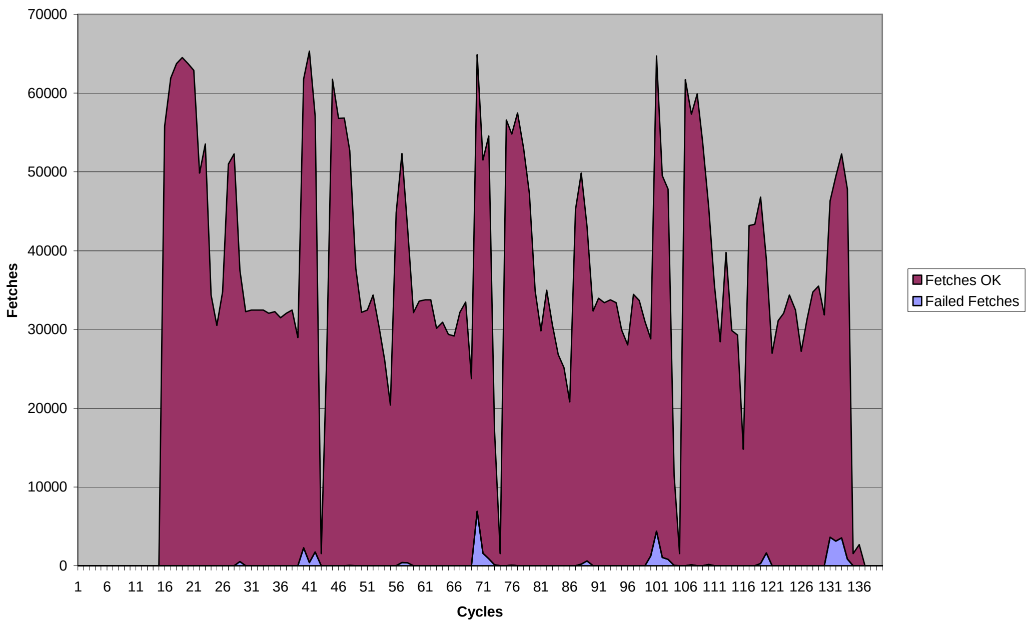
Once the frame is checked, the statistics can be opened and analyzed. The problem with the file stats.csv is that it is too big and has too many columns, so some programs like Microsoft Office Excel will open only a limit number of columns (commonly 255). The problem of opening the .csv file with a text editor is that it is no longer understandable because the use of the separator character ";" make difficult to follow what a column means (while the size of a number is short, the name of the column is too long, so they are not synchronized). Fortunately, there are some cheats about how to open a file .csv with Excel that has more than 255 columns: making some conversions to .txt and importations in Excel (a bit difficult) or writing a macro (see http://support.microsoft.com/kb/272729).

The variables measured are shown on Appendix B, and from this data it is possible to make some graphics like shown on figure 4.6. In this example it can be possible to take some conclusions, for example, figure 4.6b shows that the fetch stage of texture units works well because there are not so much fetches failed in comparison with the fetches OK.



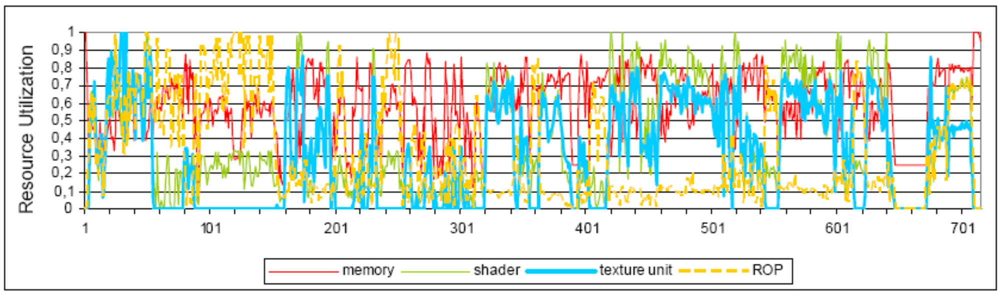
Blocked Threads on Vertex Shader 0
Threads
Cycles (from 210000 to 769999)
Blocked Threads on Vertex Shader 0

(a)



Fetches
Cycles
Fetches OK
Failed Fetches

(b)



(c)

Figure 4.6. Examples of graphics with statistics from (a) Prey (Blocked threads on vertex shader between cycles 210000 and 769999). (b) VLII (Fetches OK vs Fetches Failed on the Texture Unit of Fragment Shader 0 during all the execution 1048574). (c) Doom3 (from [6], resource utilization at frame 377 using a thread window with 3 Texture Units).

4.3 Discussion about ATTILA

Now that the internal architecture and the way of working were explained, it is time to make a discussion about ATTILA. Weak and strong sides will be analyzed and finally a conclusion will be described at the end of this section. The discussion will be made from a point of view of users, that in this case, a user is a GPU developer.

As commented on section 3.1.2, a depth change involves going more far away than the parameters and modifying the source code. The simulator's source code is easily to change because it is object oriented and box organized. But on the most cases, these depth changes imply a transformation on the ATTILA's OpenGL library (now gl2attila or gllib.a before). For example, when a developer would like to support GLSlang in his new simulated GPU (and maybe a new design for a future graphic card) he is not able to implement it because GLInterceptor and gl2attila have to be able to capture and translate GLSlang code. Other example is when a novel architecture like several GPUs on chip is tested, and then the library has to be able to share and distribute the workload among the new resources. The problem is that the source code of the ATTILA's OpenGL library is not open, so for these cases the solution is to build an own library that can support these new features, or if it can be important, ask the authors for a new version of the library.

Nowadays the lack of documentation for ATTILA is a problem for users. It is necessary to contact the author by the ATTILA mail list distribution when having doubts. The only documentation available is the source code, some files that the author posted in the last versions, papers and reports from the authors and the mail list. Sometimes they are helpful (for example, when installing ATTILA or executing and capturing traces), but in many cases they are not enough (for example, the code has to be read when trying to understand the statistics variables, which is a hard task). On the definition of good software, as well as having the binaries, libraries and source code, it is included documentation and manuals too. But it is lightening with a mail list [9], where the author and even the community of users answer questions as soon as possible. But this problem will be finally solved with time by the authors; they are planning to public some documentation on the web next months.

The simulator is quite inefficient in terms of time spent for simulations. For only one frame from Doom3 it can take from 13 to 20 minutes in a single processor, and more or less the same in a core duo. The code explained on section 3.1.2 can show the reason (figure 3.4): In each cycle, each stage has to wait to the previous for being executed (and just one cycle), when in a real GPU it does not happen (each stage executes one cycle at the same time). That the motivation of using buffers is having stand alone stages (that is, stages that do not depend for timing). One solution for this problem can be seen on Appendix D. This Appendix is based on a previous work [18] from the same author of this current report, and shows how to improve the simulation time of ATTILA using parallelization methods. Moreover, there is a lot of intercommunication between the different components (boxes, the simulated stages of the GPU) even if it is just an "I am ready" or "I am busy" signal.

Also the current OpenGL framework does not offer a good compatibility with many applications. Firstly, GLInterceptor and the simulator can not work with applications out of OpenGL (not with Direct3D or OpenGL ES), and inside OpenGL programs the wide of choice is too small. Just some programs are known that work with ATTILA (see section 4.1), and it does not provide to the developer a perfect framework where work and test his GPU architecture with the programs that he wants to use. Moreover, the use of the framework is limited only for Windows platforms, and do not permit to the developer works only in UNIX systems (as it is known that most developers and researchers work with non-Windows systems). In these cases, the solution is that the developer has to make his own framework, but not the simulator because the code is portable (not highly, but can be used in Windows, Linux, Cygwin and Mingw32).

GLInterceptor can have potentially another bad aspect. The capture of the OpenGL API calls can show the internal OpenGL code behaviour, and then, it is possible to have the OpenGL code with some inverse engineering of, for example, a game engine. GLInterceptor is very helpful and needed if ATTILA wants to work with real applications, but it has to be manipulated carefully because it is possible to infringe some intellectual laws. But this problem is potential and not completely real because one can always monitor what is happening in his system, including API calls, and the real code is still coded inside the binary and nobody can read it.

But not all are bad aspects, now talking about strong sides we can see that ATTILA can support real data for testing the architecture. The part of OpenGL supported is small, but maybe enough for most developers. If ATTILA could simulate frames from complex games like Doom3, Prey or Unreal Tournament 2004, it means that it is relatively powerful and with time it can simulate a lot of OpenGL applications.

Another good aspect of ATTILA is that it has a very structural source code. As seen on section 3.1.2, this source code is object-oriented, implementing boxes that cover the details of each part of the pipeline. These boxes, signals and statistic managers can help to the construction of a new architecture, with the bounds of a structural code with abstractions and other good aspects from the object-orientation.

Of course, it can not be possible without an open source code. The code available is just from the simulator, the rest of the framework (GLInterceptor and GLPlayer) are private code, and if it is needed to change, then a new framework must be implemented as commented so far.

Finally it is good to comment that the lots of statistics and the wide set of parameters that configure highly the pipeline help a lot to analyze with detail and work with a lot of different architectures with ATTILA, respectively. Without the statistics it is impossible to understand and analyze the architecture simulated, so the high number of statistics (about 204 variables) is enough, and if it is not, Statistic Manager helps to add new statistics to the architecture.

In conclusion, ATTILA is a completely framework for researching new GPU architectures. It is still young: slow on simulations, not compatible with the most graphic applications, and so on. Now it provides a good environment for working with graphics, and helps to research and develop new GPU applications with the tools that it provides. Among time of development, it will be possible to achieve a compatible and a comfortable (maybe with a GUI) simulator.

5. Conclusions and future work

This project made an evaluation of the new GPU simulator called ATTILA. Firstly I made an introduction to GPUs, some aspects of design and current general architecture of the majority graphic cards. After that, I made a thorough explanation of the simulator: simulated architecture, architectures that ATTILA can simulate, introduction to the source code, statistics and the framework that ATTILA provides. When this theory was explained, I exposed some experiences with ATTILA when installing, choosing testprograms and running them on the simulator. Afterwards, I was able to make an evaluation and some conclusions on section 4.3 with the information recompiled on sections 3 and 4.

Some conclusions that can be taken out from this project are:

- The evolution and technology of GPUs go separately from the development of CPUs. That is because GPUs work with other kind of data and they can have a specialized architecture. It is a good reason for developing new simulators for only GPUs.
- ATTILA provides a good environment because it supports real applications and gives a collection of classes and objects that help to the design of a new architecture.
- The simulators for CPUs are more developed than GPU simulators, it is because the discipline about GPUs is younger than CPUs, but with time it will be as important as CPUs, with a lot of software and tools that helps to research in this area.

This simulator is still young and with time it can be powerful. Some future works that can be interesting to develop are:

- Make a good documentation, with examples and explanation of every detail like the statistics.

- Make ATTILA more compatible with graphic applications: support for Direct3D and OpenGL ES, support of a more wide set of OpenGL (that is, support of more OpenGL applications).
- More help to the user, not only the code, maybe a GUI that interactively show how the architecture is working and change it in an easy way. Or even more, a support for modules, making possible to add easily a new device like memory or another GPU.
- Make ATTILA more efficient. For example, parallelize it allowing the developers to execute the simulator in a multiprocessor faster than in a single processor (see Appendix D).

Finally I hope that this work can help to understand a bit better the simulator ATTILA, and encourage to GPU researchers use this simulator because it is interesting for test new architectures and features.

References

[1] Thomas Scott Crow. "Evolution of the graphical processing unit". Dec. 2004. www.cse.unr.edu/~fredh/papers/thesis/023-crow/GPUFinal.pdf

[2] Pedro Trancoso, Maria Charalambous. "Exploring Graphics Processor Performance for General Purpose Applications". 2005. http://www2.cs.ucy.ac.cy/~pedro/publications/dsd2005-gpu.pdf

[3] Emmett Kilgariff, Randima Fernando. "The GeForce 6 Series GPU Architecture". Chapter 30. 2005. Mc Graw Hill. http://download.nvidia.com/developer/GPU_Gems_2/GPU_Gems2_ch30.pdf

[4] Pushkar Joshi, Leslie Ikemoto. "Harnessing the GPU for General Purpose Computation: A case study". 17th dec. 2004. www.cs.berkeley.edu/~ppj/publications/gpu_paper.pdf

[5] Victor Moya del Barrio. "An end to end, highly detailed simulator for the ATILA GPU microarchitecture". 2005. http://personals.ac.upc.edu/vmoya/docs/ATILASim.pdf

[6] Victor Moya, Carlos González, Jordi Roca, Agustín Fernández and Roger Espasa. "ATTILA: A cycle-level execution-driven Simulator for modern GPU architectures". 2006. http://personals.ac.upc.edu/vmoya/docs/ISPASS%20-%20ATTILASim.pdf

[7] Ziyad S. Hakura, Anoop Gupta. "The design and analysis of a cache architecture for texture mapping". ISCA 1997.

[8] Victor Moya del Barrio. "ATTILA log". http://personals.ac.upc.edu/vmoya/log.html

[9] Victor moya del Barrio. "ATTILA GPU Simulator Mail List". http://tech.groups.yahoo.com/group/attilasim

[10] I. Back, T. Foley, D. horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream Computing on Graphics Hardware. ACM Transactions on Graphics, 23(3):777-786. 2004.

[11] Jordi Roca, Victor Moya, Carlos González, Chema Solís, Agustín Fernández and Roger Espasa. "Workload Characterization of 3D games". 2006 http://personals.ac.upc.edu/vmoya/docs/IISWC-Workload.pdf

[12] Victor Moya, Carlos González, Jordi Roca, Agustín Fernández and Roger Espasa. "Shader Performance Análisis on a Modern GPU Architecture". 2005. http://personals.ac.upc.edu/vmoya/docs/vmoya-ShaderPerformance.pdf

[13] Victor Moya, Carlos González, Jordi Roca, Agustín Fernández and Roger Espasa. "A Single (Unified) Shader GPU Microarchitecture for Embedded Systems". 2005. http://personals.ac.upc.edu/vmoya/docs/EmbeddedGPU.pdf

[14] J. W. Sheaffer, D. Luebke, and K. Skadron. "A Flexible Simulation Framework for Graphics Architectures". 2004. http://qsilver.cs.virginia.edu/gh2004.pdf

[15] Tiberiu S. Popa. “Compiling Data Dependent Control Flow on SIMD GPUs”. 2004. http://qsilver.cs.virginia.edu/

[16] Miguel Ángel Martínez del Amor. “GPU ARCHITECTURE”. 2006. www.idi.ntnu.no/~thorvan/tdt24/GPU%20ARCHITECTURE.ppt

[17] Miguel Ángel Martínez del Amor. “ATTILA SIMULATOR”. 2006. www.idi.ntnu.no/~thorvan/tdt24/ATTILA%20SIMULATOR.ppt

[18] Miguel Ángel Martínez del Amor. “PARALLELIZATION OF ATTILA SIMULATOR WITH OPENMP”. 2006. www.idi.ntnu.no/~thorvan/tdt24/PARALLELIZATION%20OF%20ATTILA%20SIMULATOR%20WITH%20OPENMP.ppt

APPENDIX A: DEFINITIONS AND ABBREVIATIONS

3

3D Objects: Objects that represent the objects of the modelled word. The coordinates are expressed on 3D coordinates.

3DNOW!: is the name of a multimedia extension created by AMD for its processors, starting with the K6-2 in 1998. It is an addition of SIMD instructions to the traditional x86 instruction set, designed to improve a CPU's ability to perform the vector processing requirements of many graphic-intensive applications. (Source: www.wikipedia.org).

A

AGP: The (**A**ccelerated **G**raphics **P**ort (also called **A**dvanced **G**raphics **P**ort)) is a high-speed point-to-point channel for attaching a graphics card to a computer's motherboard, primarily to assist in the acceleration of 3D computer graphics. Some motherboards have been built with multiple independent AGP slots. AGP is currently being phased out in favor of PCI Express. (Source: www.wikipedia.org).

AI: (**A**rtificial **I**ntelligence) can be defined as the study of methods by which a computer can simulate aspects of human intelligence. (Source: www.wikipedia.org).

ALU: (**A**rithmetic **L**ogic **U**nit) is a digital circuit that calculates an arithmetic operation (like an addition, subtraction, etc.) and logic operations (like an Exclusive Or) between two numbers. The ALU is a fundamental building block of the central processing unit of a computer. (Source: www.wikipedia.org).

Alpha: In computer graphics, alpha compositing is the process of combining an image with a background to create the appearance of partial transparency. (Source: www.wikipedia.org).

API: (**A**pplication **P**rogramming **I**nterface**)** is a source code interface that a computer system or program library provides in order to support requests for services to be made of it by a computer program. (Source: www.wikipedia.org).

ARB: (The OpenGL **A**rchitecture **R**eview **B**oard) is an industry consortium that currently governs the OpenGL specification. It was formed in 1992, and defines the conformance tests, approves the OpenGL specification and advances the standard. On July 31, 2006, it was announced that the ARB voted to transfer control of the OpenGL specification to Khronos Group. As of November 2004, the voting members are 3Dlabs, Apple Computer, ATI, Dell, IBM, Intel, Nvidia, SGI and Sun Microsystems, plus other contributing members. Microsoft was an original voting member, but left in March 2003. (Source: www.wikipedia.org).

ARB_Vertex_Program: Is a vertex shading language for OpenGL made by ARB.

ARB_Fragment_Program: Is a fragment shading language for OpenGL made by ARB.

ATI: **ATI Technologies U.L.C.**, founded in 1985, is a major designer of graphics processing units and video display cards and a wholly owned subsidiary of AMD, as of October 2006. (Source: www.wikipedia.org).

ATTILA: The name is based on '**Attila** The Hun'. It is a cycle accurate GPU simulator made by several authors (mainly Victor Moya) in the Polytechnic University of Catalonia.

B

BrookGPU: is the Stanford University Graphics group's compiler and runtime implementation of the Brook stream programming language for using modern graphics hardware for non-graphical, or general purpose computations (GPGPU). (Source: www.wikipedia.org).

Buffer: In computing, a **buffer** is a region of memory used to temporarily hold output or input data. Buffers can be implemented in either hardware or software, but the vast majority of buffers are implemented in software. Buffers are used when there is a difference between the rate at which data is received and the rate at which it can be processed, or in the case that these rates are variable, for example in a printer spooler. (Source: www.wikipedia.org).

C

C++: is a general-purpose, high-level programming language with low-level facilities. It is a statically typed free-form multi-paradigm language supporting procedural programming, data abstraction, object-oriented programming, generic programming and RTTI. Since the 1990s, C++ has been one of the most popular commercial programming languages. (Source: www.wikipedia.org).

Cg: (**C** for **G**raphics) is a high-level shading language created by NVIDIA for programming vertex and pixel shaders. Cg is based on the C programming language and although they share the same syntax, some features of C were modified and new data types were added to make Cg more suitable for programming graphics processing units. (Source: www.wikipedia.org).

Class: In object-oriented programming, **classes** are used to group related variables and functions. A class describes a collection of encapsulated instance variables and methods (functions), possibly with implementation of those types together with a constructor function that can be used to create objects of the class. (Source: www.wikipedia.org).

Class diagram: Is a diagram that shows how the classes from the source code of a program are related and interconnected (relationships between classes).

CPI: (**C**ycles **p**er **i**nstruction (**c**lock **c**ycles **p**er **i**nstruction ***or*** *clocks per instruction*)) is a term used to describe one aspect of a processor's performance: the number of clock cycles that happen when an instruction is being executed. It is the multiplicative inverse of Instructions Per Cycle (IPC). (Source: www.wikipedia.org).

CPU: (**C**entral **P**rocessing **U**nit or sometimes simply **processor)** is the component in a digital computer that interprets computer program instructions and processes data. (Source: www.wikipedia.org).

Cygwin: is a collection of free software tools originally developed by Cygnus Solutions to allow various versions of Microsoft Windows to act somewhat like a Unix system. (Source: www.wikipedia.org).

D

D3D: see Direct3D.

DDR: In computing, a computer bus operating with **D**ouble **D**ata **R**ate transfers data on both the rising and falling edges of the clock signal, effectively nearly doubling the data transmission rate without having to deal with the additional problems of timing skew that increasing the number of data lines would introduce. This technique has been used for the Front side bus, Ultra-3 SCSI, the AGP bus, DDR SDRAM (principal memory), and the HyperTransport bus on AMD's Athlon 64 X2 processors. (Source: www.wikipedia.org).

Direct3D: is part of Microsoft's DirectX API. Direct3D is only available for Microsoft's various Windows operating systems (Windows 95 and above) and is the base for the graphics API on the Xbox and Xbox 360 console systems. Direct3D is used to render three dimensional graphics in applications where performance is important, such as games. Direct3D also allows applications to run fullscreen instead of embedded in a window, though they can still run in a window if programmed for that feature. Direct3D uses hardware acceleration if it is available on the graphic board. (Source: www.wikipedia.org).

DRAM: (**D**ynamic **R**andom **A**ccess **M**emory) is a type of random access memory that stores each bit of data in a separate capacitor within an integrated circuit. (Source: www.wikipedia.org).

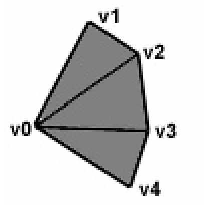
Driver: is a specific type of computer software, typically developed to allow interaction with hardware devices. This usually constitutes an interface for communicating with the device, through the specific computer bus or communications subsystem that the hardware is connected to, providing commands to and/or receiving data from the device, and on the other end, the requisite interfaces to the operating system and software applications. (Source: www.wikipedia.org).

E

Environment: is a type of computer software that assists computer programmers in developing software. In this report means the system and tools used for the development of these simulations.

F

FAN mode: In OpenGL, when defining the FAN mode for triangle means that the three first vertices defined correspond to the first triangle, and the next vertices defined make a triangle that has shared the first vertex defined and previous vertex.



(Source: http://web.cs.wpi.edu/~matt/courses/cs563/talks/OpenGL_Presentation/OpenGL_Presentation.html)

Fog: is a cloud in contact with the ground. In computer graphics means that the field of vision will be reduce.

Fragment: A fragment is a point in windows coordinates produced by rasterizer stage that has attributes as color, depth… It has the data necessary needed to generate a pixel in the frame buffer. One pixel (a dot of color) corresponds to multiple fragments.

Frame: is one of the many still images which compose the complete *moving picture*. The frame rate, the rate at which sequential frames are presented, varies according to the video standard in use. In North America and Japan, 30 frames per second is the broadcast standard, with 24 frame/s now common in production for high-definition video. In much of the rest of the world, 25 frame/s is standard. (Source: www.wikipedia.org).

Framework: In software development, a **framework** is a defined support structure in which another software project can be organized and developed. A framework may include support programs, code libraries, a scripting language, or other software to help develop and *glue together* the different components of a software project. In this report means to the set of tools and steps that ATTILA has for the development of GPU architectures for OpenGL. (Source: www.wikipedia.org).

Frustum volume: In 3D computer graphics, the **viewing frustum** or **view frustum** is the region of space in the modeled world that may appear on the screen; it is the field of view of the notional camera. The exact shape of this region varies depending on what kind of camera lens is being simulated, but typically it is a frustum of a rectangular pyramid. The planes that cut the frustum perpendicular to the viewing direction are called the *near plane* and the *far plane*. Objects closer to the camera than the near plane or beyond the far plane are not drawn. Often, the far plane is placed infinitely far away from the camera so all objects within the frustum are drawn regardless of their distance from the camera. (Source: www.wikipedia.org).

G

GCC: Originally named the **G**NU **C** **C**ompiler, because it only handled the C programming language, GCC 1.0 was released in 1987, and the compiler was extended to compile C++ in December of that year. Front ends were later developed for Fortran, Pascal, Objective-C, Java, and Ada, among others. (Source: www.wikipedia.org).

GDDR3: (**G**raphics **D**ouble **D**ata **R**ate **3**) is a graphics card-specific memory technology, designed by ATI Technologies. (Source: www.wikipedia.org).

GLES: See OpenGL ES.

GLSlang: (Open**GL** **S**hading **Lang**uage**)** is a high level shading language based on the C programming language. It was created by the OpenGL ARB to give developers more direct control of the graphics pipeline without having to use assembly language or hardware-specific languages. (Source: www.wikipedia.org).

GPGPU: (**G**eneral-**P**urpose Computing on **G**raphics **P**rocessing **U**nits**)** is a recent trend in computer science that uses the Graphics Processing Unit to perform the computations rather than the CPU. The addition of programmable stages and higher precision arithmetic to the GPU rendering pipeline have allowed software developers to use the GPU for non graphics related applications. Because of the extremely parallel nature of the graphics pipeline the GPU is especially useful for programs that can be cast as stream processing problems. (Source: www.wikipedia.org).

GPU: (**G**raphics **P**rocessing **U**nit (also occasionally called **V**isual **P**rocessing **U**nit or **VPU**)) is a dedicated graphics rendering device for a personal computer, workstation, or game console. Modern GPUs are very efficient at manipulating and displaying computer graphics, and their highly parallel structure makes them more effective than typical CPUs for a range of complex algorithms. (Source: www.wikipedia.org).

H

I

ISA: (**I**nstruction **S**et **A**rchitecture**)** is (a list of) all instructions, and all their variations, that a processor can execute. (Source: www.wikipedia.org).

J

K

L

Library: is a collection of subprograms used to develop software. Libraries contain "helper" code and data, which provide services to independent programs. This allows code and data to be shared and changed in a modular fashion. Some executables are both standalone programs and libraries, but most libraries are not executables. Executables and libraries make references known as *links* to each other through the process known as *linking*, which is typically done by a linker. (Source: www.wikipedia.org).

Linux: is a free open-source operating system based on Unix. *Linux* was originally created by Linus Torvalds with the assistance of developers from around the globe. *Linux* was developed under the GNU General Public License and the source code is freely available to everyone. (Source: www.orafaq.com).

M

Mingw32: (**Min**imalist **GNU** for **W**indows) is a software port of the GNU toolchain to the Win32 platform. MinGW includes a set of Windows header files (W32API) for native Win32 development. It was originally a fork of Cygwin (version 1.3.3). (Source: www.wikipedia.org).

MMX: (**M**ulti**M**edia e**X**tensions) is a SIMD instruction set designed by Intel, introduced in 1997 in their Pentium MMX microprocessors. It developed out of a similar unit first introduced on the Intel i860. It has been supported on most subsequent IA-32 processors by Intel and other vendors. (Source: www.wikipedia.org).

N

North Bridge: also known as the **M**emory **C**ontrolle**r** **H**ub (**MCH**), is traditionally one of the two chips in the core logic chipset on a PC motherboard, the other being the Southbridge. Separating the chipset into Northbridge and Southbridge is common, although there are rare instances where these two chips have been combined onto one die when design complexity and fabrication processes permit it. (Source: www.wikipedia.org).

NUMA: (**N**on-**U**niform **M**emory **A**ccess or **N**on-**U**niform **M**emory **A**rchitecture) is a computer memory design used in multiprocessors, where the memory access time depends on the memory location relative to a processor. Under NUMA, a processor can access its own local memory faster than non-local memory, that is, memory local to another processor or memory shared between processors. (Source: www.wikipedia.org).

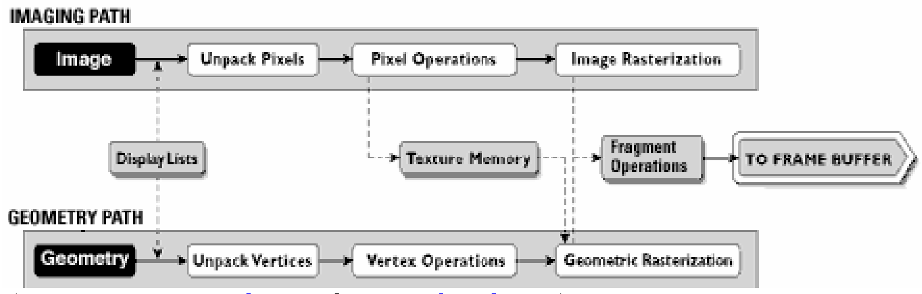
NVIDIA: **NVIDIA Corporation** (NASDAQ: NVDA) is a major supplier of graphics processors (graphics processing units, GPUs), graphics cards, and media and communications devices for PCs and game consoles such as the original Xbox and the PlayStation 3. NVIDIA's most popular product lines are the GeForce series for gaming and the Quadro series for Professional Workstation Graphics processing as well as the nForce series of computer motherboard chipsets. (Source: www.wikipedia.org).

O

Object: In the programming paradigm, object-oriented programming, an **object** is an individual unit of run-time data storage that is used as the basic building block of programs. These objects act on each other, as opposed to a traditional view in which a program may be seen as a collection of functions, or simply as a list of instructions to the computer. Each object is capable of receiving messages, processing data, and sending messages to other objects. Each object can be viewed as an independent little machine or actor with a distinct role or responsibility. Also, an object can be seen as a instance of a Class. (Source: www.wikipedia.org).

Object-Oriented programming: (OOP) is a programming paradigm that uses "objects" to design applications and computer programs. It utilizes several techniques from previously established paradigms, including inheritance, modularity, polymorphism, and encapsulation. Even though it originated in the 1960s, OOP was not commonly used in mainstream software application development until the 1990s. Today, many popular programming languages (such as Java, JavaScript, C#, C++, Python, PHP, Ruby and Objective-C) support OOP. (Source: www.wikipedia.org).

OpenGL: (**Open** **G**raphics **L**ibrary) is the premier environment for developing portable, interactive 2D and 3D graphics applications. It is a standard specification defining a cross-language cross-platform API for writing applications that produce 3D computer graphics (and 2D computer graphics as well). The interface consists of over 250 different function calls which can be used to draw complex three-dimensional scenes from simple primitives. OpenGL was developed by Silicon Graphics. OpenGL operates on image data as well as geometric primitives (see the similarity with the architecture explained on section 3.1.1):



(Sources: www.opengl.org and www.wikipedia.org).

OpenGL ES: (**OpenGL** for **E**mbedded **S**ystems) is a subset of the OpenGL 3D graphics API designed for embedded devices such as mobile phones, PDAs, and video game consoles. It is defined and promoted by the Khronos Group, a graphics hardware and software industry consortium interested in open APIs for graphics and multimedia. (Source: www.wikipedia.org).

P

PCI: (**P**eripheral **C**omponent **I**nterconnect or **PCI Standard)** specifies a computer bus for attaching peripheral devices to a computer motherboard. (Source: www.wikipedia.org).

PCI Express: is an implementation of the PCI connection standard that uses existing PCI programming concepts, but bases it on a completely different and much faster full duplex, multi-lane, point to point serial physical-layer communications protocol. (Source: www.wikipedia.org).

Pixel: (short for **Pic**ture **El**ement, using the common abbreviation "pix" for "picture") is a single point in a graphic image. Each such information element is not really a dot, nor a square, but an abstract sample. With care, pixels in an image can be reproduced at any size without the appearance of visible dots or squares; but in many contexts, they are reproduced as dots or squares and can be visibly distinct when not fine enough. The intensity of each pixel is variable; in color systems, each pixel has typically three or four dimensions of variability such as red, green and blue, or cyan, magenta, yellow and black. (Source: www.wikipedia.org).

Pixmap: is a three-dimensional array of bits. Also, a pixmap is normally thought of as a two-dimensional array (matrix) of pixels.

Q

R

Render: is the process of generating an image from a model, by means of computer programs. The model is a description of three dimensional objects in a strictly defined language or data structure. It would contain geometry, viewpoint, texture and lighting information. The image is a digital image or raster graphics image. The term may be by analogy with an "artist's rendering" of a scene. 'Rendering' is also used to describe the process of calculating effects in a video editing file to produce final video output. (Source: www.wikipedia.org).

S

Shader: is a piece of code that programs certain parts of the graphic pipeline. Specifically, it is a set of instructions, a computer program used in 3D computer graphics to determine the final surface properties of an object or image, executed by the GPU. This often includes arbitrarily complex descriptions of texture mapping, light absorption, diffusion, reflection, refraction, shadowing, surface displacement and post-processing effects. There are two types: vertex shader and fragment shader. (Source: www.wikipedia.org).

SIMD: (**S**ingle **I**nstruction, **M**ultiple **D**ata) is a technique employed to achieve data level parallelism, as in a vector or array processor. (Source: www.wikipedia.org). An operation (performed by a unit) over two arrays using this technique will make the calculation over each element, issuing the resulting array element by element to the next unit. By this way it is not needed to wait for the realization of the operation (A) over the whole array for continuing with the next operation (B) that needs the resulting array from the previous operation (A).

SMP: (**S**ymmetric **M**ulti **P**rocessor) is a multiprocessor computer architecture where two or more identical processors are connected to a single shared main memory. Most common multiprocessor systems today use SMP architecture. (Source: www.wikipedia.org). SMP is also called UMA.

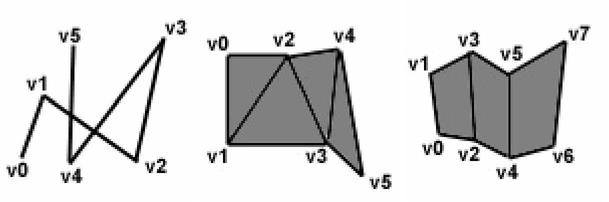
SMT: (**S**imultaneous **M**ulti**t**hreading) is a technique for improving the overall efficiency of superscalar CPUs. SMT permits multiple independent threads of execution in the same superscalar processor to better utilize the resources provided by modern processor architectures. (Source: www.wikipedia.org).

SSE: (**S**treaming SIMD **E**xtensions, originally called **ISSE**, **I**nternet **S**treaming **S**IMD **E**xtensions) is a SIMD (Single Instruction, Multiple Data) instruction set designed by Intel and introduced in 1999 in their Pentium III series processors as a reply to AMD's 3DNow! (which had debuted a year earlier). (Source: www.wikipedia.org).

Stencil: Stenciling, like z-buffering, enables and disables drawing on a per-pixel basis. You draw into the stencil planes using GL drawing primitives, then render geometry and images, using the stencil planes to mask out portions of the screen. Stenciling is typically used in multipass rendering algorithms to achieve special effects, such as decals, outlining, and constructive solid geometry rendering. (Source: OpenGL bluebook).

Stream: is applied to hardware as well as software. There it defines the quasi-continuous flow of data which is processed in dataflow languages as soon as the program state meets the starting condition of the stream. (Source: www.wikipedia.org).

STRIP mode: In OpenGL, when defining the STRIP mode for lines, triangles or quads means that the first vertices defined will construct a first object (line, triangle or quad) and the next vertex defined will make a new object with the union of the last vertices defined. There are examples for Line Strip, for Triangle Strip and for Quad Strip, respectively:



(Source: http://web.cs.wpi.edu/~matt/courses/cs563/talks/OpenGL_Presentation/OpenGL_Presentation.html).

T

Texel: (**Tex**ture **El**ement (also **Tex**ture Pix**el**)) is the fundamental unit of texture space, used in computer graphics. Textures are represented by arrays of texels, just as pictures are represented by arrays of pixels.

Texture: an image used in computer rendering to give color and other apparent surface characteristics ("textures") to 3D objects.

Tile: A pixmap can be replicated in two dimensions to *tile* a region. The pixmap itself is also known as a *tile*. In this report it can be seen as a set of pixels, texels or fragments. (Source: http://barossa.ac3.edu.au/SGI_Developer/books/XLib_WinSys/sgi_html/go01.html).

Triangle: is one of the basic shapes of geometry. Because of this, every 3D object can be represented approximately as a set of multiple triangles.

U

UMA: (**U**niform **M**emory **A**ccess) is a computer memory architecture used in parallel computers. All the processors in the UMA model share the physical memory uniformly. Peripherals are also shared. Cache memory may be private for each processor. In an UMA architecture, accessing time to a memory location is independent from which processor makes the request or which memory chip contains the target memory data. (Source: www.wikipedia.org).

V

Vertex: is a corner of a polygon (where two sides meet) and, in OpenGL, a vertex can have associated several parameters as 3D coordinates, colour, depth, alpha, etc.

Visual Studio: is Microsoft's flagship software development product for computer programmers. It centers on an integrated development environment which lets programmers create standalone applications, web sites, web applications, and web services that run on any platforms supported by Microsoft's .NET Framework. (Source: www.wikipedia.org).

VLIW: (**V**ery **L**ong **I**nstruction **W**ord) refers to a CPU architectural approach to taking advantage of instruction level parallelism (ILP). A processor that executes every instruction one after the other (i.e. a non-pipelined scalar architecture) will have very poor performance. The performance can be improved by executing different sub-steps of sequential instructions simultaneously (this is *pipelining*), or even executing multiple instructions entirely simultaneously as in superscalar architectures. (Source: www.wikipedia.org). In this kind of architectures, the processor always issue and execute the same number of instructions at the same time (always 4 or 8, depending on the configuration of the processor). For that, the compiler has to dispose the instructions to the processor in a correct order avoiding interactions and bad results. The instruction nop (not an operation, do not do anything) is used to fill the places of instructions that can not be executed at the same time that the other (for example, two instructions where one need the result of the other operation).

W

Windows: is the name of several families of proprietary operating systems by Microsoft. They can run on several types of platforms such as servers, embedded devices and, most typically, on personal computers. (Source: www.wikipedia.org).

X

Y

Z

Z: see Z-buffering.

Z-buffering: is the management of image depth coordinates in three-dimensional (3-D) graphics, usually done in hardware, sometimes in software. It is one solution to the visibility problem, which is the problem of deciding which elements of a rendered scene are visible, and which are hidden. The painter's algorithm is another common solution which, though less efficient, can also handle non-opaque scene elements. Z-buffering is also known as **depth buffering.** (Source: www.wikipedia.org).

APPENDIX B: OUTPUT STATISTIC VARIABLES

NAME	UNIT CODE (to be explained by the author in future documentation, but the majority understandable)
AccessQueueOccupation	TU (Texture unit)
AddressALUBusyCycles	TU
AddressCalculationFinished	TU
AllocateFailed	CW-SU
AllocateOK	CW
AnisotropyRatio	TU
BackFacingTriangles	TS (Triangle Setup)
Batches	CP (Command Processor)
BilinearSamples	TU
BlendedFragments	CW
BlockCommands	ShDX FS (Fragment Shader)
BlockedInstructions	ShDx FS
BlockedThreads	ShF FS
Blocks	ShF FS
BytesRead	CP
BytesWritten	CP
Clear	CP
ClippedTriangles	CLP
ColorWriteReadBytes	MC (Memory Controller)
ColorWriteReadTransactions	MC
ColorWriteTransactions	MC
ColorWriteWriteBytes	MC
ColorWriteWriteTransactions	MC
CommandProcessorReadBytes	MC
CommandProcessorReadTransactions	MC
CommandProcessorTransactions	MC
CommandProcessorWriteBytes	MC
CommandProcessorWriteTransactions	MC
CulledFragments	CW SU HZ ZST
CulledHZFragments	HZ (Hierarchical Z)
CulledOutsideFragments	HZ
CulledTriangles	TS
DACReadBytes	MC
DACReadTransactions	MC
DACTransactions	MC
DACWriteBytes	MC
DACWriteTransactions	MC
DataCycles00 - 03	MC

Degenerated	PA
Draw	CP
EmptyCycles	ShF FS
EndCommands	ShDX FS
EndFragment	CP
EndGeometry	CP
ExecutedInstructions	ShDX FS
FailedFragments	ZST SU (Z-Stencil)
FakedInstructions	ShDX FS
FetchBankConflicts	TU
FetchCycles	FS
FetchFailed	CW SU ZST
FetchOK	CW SU ZST
FetchStallAddress	TU
FetchStallFetch	TU
FetchStallReadyRead	TU
FetchStallWaitRead	TU
FetchedInstr	ShF FS
FetchesFailed	TU
FetchesOK	TU
FetchesSkiped	TU
Fetches	StL
FilterALUBusyCycles	TU
FinishedThreads	ShF
Frames	CP
FreeThreads	ShF
FrontFacingTriangles	TS
GeneratedFragments	TT
HitsAlloc	CW InC TU ZST
HitsFetch	CW InC TU ZST
HitsHZCache	HZ
Hits	StOC
IndexesSent	StF
Indices	Stc StL StOC
InputActiveInputAttributes	ShF
InputActiveOutputAttributes	ShF
InputFragments	CW FFU HZ ZST
InputRegisters	ShF
InputTriangles	CLP FFU TS TT
InputVertices	FFU
Inputs	ShF StL
IntStampQueuesOccupation	FFU
InterpolatedFragments	FFU
LogicOpFragments	CW
MappedAttributes	StL

MemTransactions	StL
MemoryPreload	CP
MemoryReadBytes	MC
MemoryReadTransactions	MC
MemoryRead	CP
MemoryRequestLatency	Tu
MemoryRequests	TU
MemoryTransactions	MC StF
MemoryWriteBytes	MC
MemoryWriteTransactions	MC
MemoryWrite	CP
MissFailAlloc	CW InC TU ZST
MissFailFetch	CW InC TU ZST
MissFailMissAlloc	TU
MissFailMissFetch	TU
MissFailReqQueueAlloc	TU
MissFailReqQueueFetch	TU
MissFailReserveAlloc	TU
MissFailReserveFetch	TU
MissOKAlloc	CW Inc TU ZST
MissOKFetch	CW InC TU ZST
MissesAlloc	CW InC TU ZST
MissesFetch	CW InC TU ZST
MissesHZCache	HZ
Misses	StOC
NoFetches	StL
NoReads	StL
NoReadyCycles	ShF
OpenPagePenalty00 - 01	MC
OpenPages01 - 03	MC
OutputAttributes	Stc
OutputFragments	FFU HZ ZST
OutputTriangles	CLP FFU TS
OutputVertices	FFU
Outputs	ShF StC
OutsideFragments	CW ZST
OutsideTriangleFragments	HZ
OutsideViewPortFragments	HZ
PassedFragments	ZST
PreloadTransactions	MC
RAWDependence	CW ZST
RastStampQueuesOccupation	FFU
ReFetchedInstr	ShF
ReadBankConflicts	TU
ReadBytesMemoryBuss00 – 03	MC

ReadBytesMemory	TU
ReadBytesSystemBus	MC
ReadBytes	CW InC StF StL TU ZST
ReadFailed	CW ZST
ReadOK	CWZST
ReadToWritePenalty00 – 03	MC
ReadTransactions	CW TU ZST
ReadsFail	CW TU ZST
ReadsFailedTU	TU
ReadsHZBuffer	HZ
ReadsOKTU	TU
ReadsOK	CW InC TU ZST
Reads	StL
ReadyreadQuewueOccupation	TU
ReadyThreads	ShF
RegisterWrites	CP
RemovedInstructions	ShDX
ReplayCommands	ShDX
RequestQueueOccupation	TU
RequestedTriangles	TS TT
Requests	PA StC
ResultQueueOccupation	TU
ShadedFragments	FFU
ShadedStampQueuesOccupation	FFU
ShadedTriangles	FFU
ShadedVertices	FFU
ShaderOutputs	StC
SplittedAttributes	StL
StreamerFetchReadbytes	Mc
StreamerFetchReadTransactions	MC
StreamerLoaderTransactions	MC
StreamerLoaderWriteBytes	MC
StreamerLoaderWriteTransactions	MC
Swap	CP
SystemDataCycles 00 – 01	MC
Systemreadbytes	MC
SystemReadTransactions	MC
SystemTransactions	MC
SystemWriteBytes	Mc
SystemWriteTransactions	Mc
TestStampQueuesOccupation	FFU
TestedFragments	ZST
TextureRequests	ShDX TU
TextureResultLatency	TU
TextureResults	TU

TextureUnitReadBytes	MC
TextureUnitReadTransactions	MC
TextureUnitWriteTransactions	MC
TriangleInputQueueOccupation	FFU
TriangleOutputQueueOccuaption	FFU
Triangles	PA
UnblockCommands	ShDX
Unblocks	ShF
Unreserves	CW TU ZST
UnusedCycles	MC
UpdatesHZ	HZ
UsedResources	ShF
VertexInputQueueOccupation	FFU
VertexOutputQueueOccupation	FFU
Vertices	PA
WaitReadWindowOccupation	TU
WriteBytesMemoryBus 00 – 03	MC
WriteBytesSystemBus	MC
WriteBytes	CW InC TU ZST
WriteFailed	CW ZST
WriteOK	ZST
WriteToReadPenalty00 – 03	MC
WriteTransactions	CP CW ZST
WritesFail	CW TU ZST
WritesHZBuffer	HZ
WritesOK	CW InC TU ZST
ZStencilTestReadBytes	MC
ZStencilTestReadTransactions	MC
ZStencilTestTransactions	MC
ZStencilTestWriteBytes	MC
ZStencilTestWriteTransactions	MC

APPENDIX C: PARAMETERS

In this appendix the parameters that configure the ATTILA architecture will be explained. This appendix complements the section 3.1.1, so it is interesting to read them at the same time. Here only the most interesting parameters will be explained, because there are about 215 different parameters. To see all the parameters, just have a look to one configuration file (ATTILA-rei-5xx.ini) in the folder confs.

The parameters are defined in a configuration file that has a structure as shown in figure C.1. Label says the unit that will be configured, and parameter corresponds to an aspect of this unit to configure.

```
[LABEL1]
ParameterName11 = value11
ParameterName12 = value12
…

[LABEL2]
ParameterName21 = value21
ParameterName22 = value22
…
```

Figure C.1. Scheme of the configuration file.

- [COMMANDPROCESSOR]: for the configuration of the Command Processor. One parameter is PipelinedBatchRendering (allow to process register writes (AGP_REG_WRITE) and non locked memory uploads (AGP_WRITE) while the rest of the pipeline is rendering).
- [STREAMER]: for the configuration of the Streamer of Vertex Fetch. It can be configured by up to 13 parameters as IndexBufferSize or VerticesCycle.
- [VERTEXSHADER]: for the configuration of the Vertex Shader (every vertex shader will have the same structure, but if unified shaders are used, then in the same shader unit can be mixed vertex and fragment shader) with up to 14 parameters, as ExecutableThreads, ScalarALU (it is a Boolean variable that only take the true value for pixel (or fragment) shader, the only one that supports scalar ALUs), FetchRate, InputsPerCycle,….
- [PRIMITIVEASSEMBLY]: for the configuration of the Primitive Assembly. The parameters in order to configure this stage are VerticesCycle, InputBusLatency, AssemblyQueueSize (the size of the memory that stores the last vertices) and TrianglesCycle.
- [CLIPPER]: some parameters that configure the Clipper stage can be ClipperUnits and ExecLatency, supporting up to 5 parameters.
- [RASTERIZER]: for the configuration of the whole Rasterizer stage.

  - The parameters for configuring the Triangle Setup are TrinagleInputLatency, TriangleSetupOnShader, StampsPerCycle, etc.
  - The parameters for configuring the Fragment Generator stage can be RecursiveMode, GenWidth, GenHeight, ScanWidth, etc.
  - The parameters for configure the HZ can build up the HZ cache or latencies, for example, HZCacheLines, HZCacheLineSize, HZAccessLatency, amongst others.
  - One important parameter for configuring the Interpolator stage is NumInterpolators.
- [ZSTENCILTEST]: Z and Stencil tests can be configured in a separately section from the Rasterizer. Some parameters can be used for configuring the ZCache (ZCacheWays, ZCachLines), and other about the compression unit (CompressionUnitLatency, DisableCompression). The number of the total of parameters is 19 parameters.
- [FRAGMENTSHADER]: In any case, the parameters for configuring the Fragment Shader are the same than for the Vertex Shader, plus the parameters for configuring the Texture Units, such as TextureBlockDimension, TextureCacheWays, TwoLevelTextureCache, TextureCacheWaysL1, against others.
- [COLORWRITE]: The parameters placed below this label can be StampsperCycle, ColorCacheWays, ColorCacheLines, ColorQueueSize, etc.
- [DAC]: It can be configured by the parameters BytesPerPixel, BlockSize, DecompressionUnitLatency (in case of use Compression of frames, but it can be always dismissed to only 1 cycle because the dump into a file is out of the GPU work), RefreshFrame … of the stage DAC.
- [MEMORYCONTROLLER]: This unit can be highly configured by parameters, such as MemorySize, MemoryFrequency, MemoryBusses, BankGranularity (size of banks), ReadLatency, WriteLatency, MemoryPageSize, etc.

The bus width of Command Processor, Streamer, ZStencil, DAC and TextureUnit can be configured with the parameters CommandProcessorBusWidth (on the baseline configuration is 8, and not 64 like in the others), StreamerFetchBusWidth, StreamerLoaderBusWidth, ZStencilBusWidth, DACBusWidth and TextureUnitBusWidht.

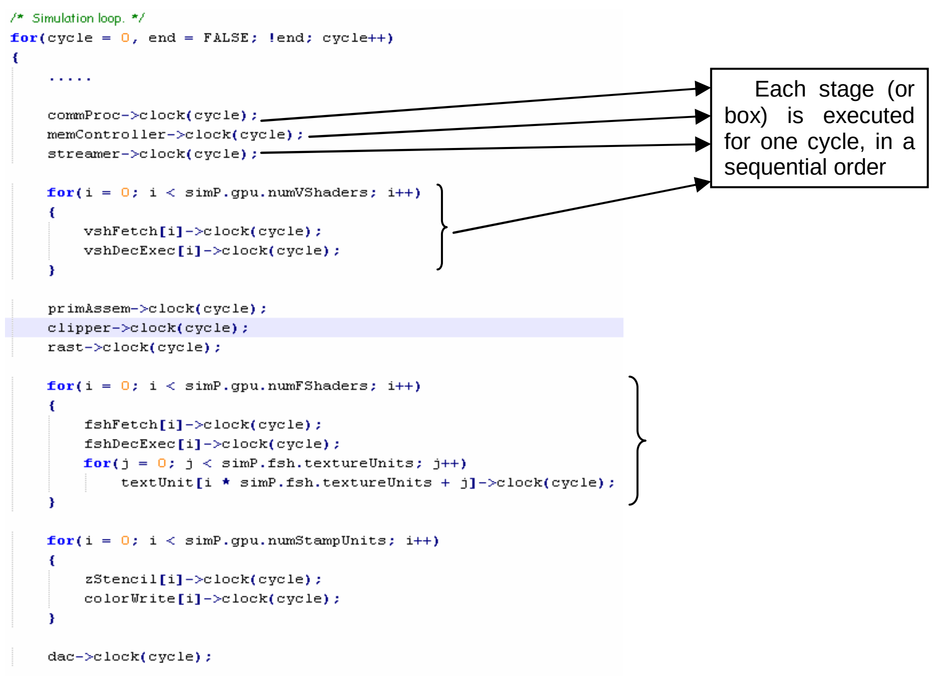
APPENDIX D: PARALLELIZATION OF ATTILA

This section is based on the miniproject [18] and further information found on the mail list [9]. The ideas exposed here are as theory; they were not tested before in a real machine yet but seem that can make the simulator to achieve a better performance.

As commented on the conclusion from section 4.3, ATTILA is a cycle-accurate simulator but inefficient when simulating. One simulation takes a lot of time when simulating, but it is logic when thinking that:

- The CPU has to do a work (simulation) that is very inefficient if it is not used specific hardware like GPU or other SIMD architectures.
- ATTILA is a cycle-accurate simulator, where everything is simulated on detail.
- The simulator code uses sequential model, with boxes and a lot of sometimes "useless" signals between them.

But it is possible to make the simulator more efficient trying some methods over the code like parallelization, avoiding change the whole design of the source code. We can see the way of working of the simulator in the main loop (called from the main function, file bGPU.c, see figure D.1) and it is possible to see two aspects:



```
/* Simulation loop. */
for(cycle = 0, end = FALSE; !end; cycle++)
{
    .....

    commProc->clock(cycle);
    memController->clock(cycle);
    streamer->clock(cycle);

    for(i = 0; i < simP.gpu.numVShaders; i++)
    {
        vshFetch[i]->clock(cycle);
        vshDecExec[i]->clock(cycle);
    }

    primAssem->clock(cycle);
    clipper->clock(cycle);
    rast->clock(cycle);

    for(i = 0; i < simP.gpu.numFShaders; i++)
    {
        fshFetch[i]->clock(cycle);
        fshDecExec[i]->clock(cycle);
        for(j = 0; j < simP.fsh.textureUnits; j++)
            textUnit[i * simP.fsh.textureUnits + j]->clock(cycle);
    }

    for(i = 0; i < simP.gpu.numStampUnits; i++)
    {
        zStencil[i]->clock(cycle);
        colorWrite[i]->clock(cycle);
    }

    dac->clock(cycle);
```

Each stage (or box) is executed for one cycle, in a sequential order

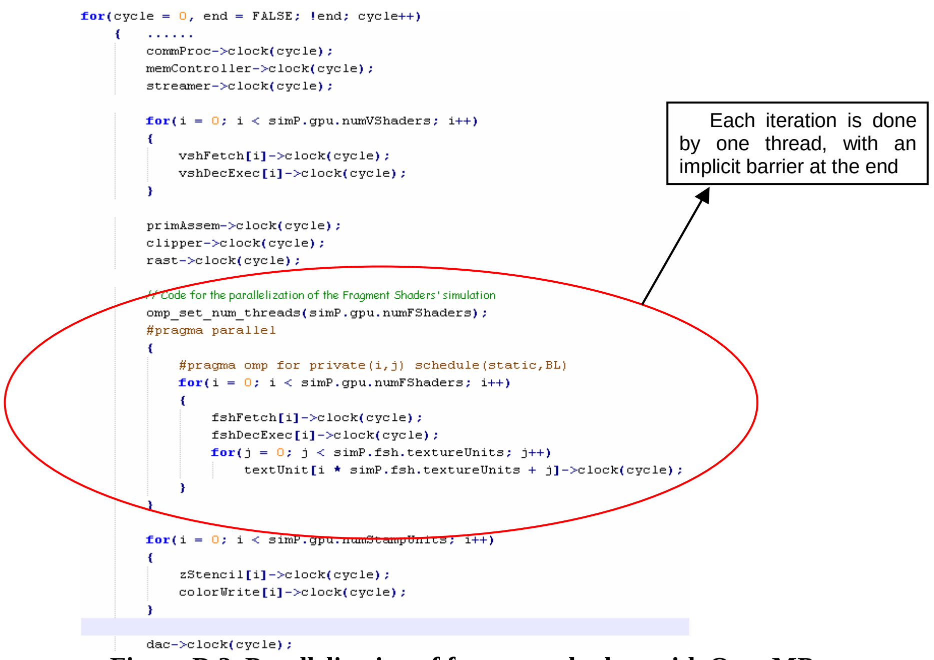
Figure D.1. Main loop of ATTILA simulator.

- It is sequential, first simulate one cycle in one box, and then simulate the next.
- Each box has input signals and output signals. One signal is a class that implements a kind of buffer with a cycle of delay. Then, each box simulates a single cycle stage (or stages) using local data and signal data produced in another box. But the data that comes from the signals was produced at least one cycle in the past so there are no data dependences between boxes simulating the same cycle.

Taking in account these aspects, it is possible to give solutions to many problems that statistics and experience can show us. Only two problems will be seen here:

- Problem 1: Statistics show that the most of simulation time is used on shaders (fragment shaders and their texture units). Also ZStencilTest takes so much time when doing depth or stencil passes (like stencil shadows engines as Doom3).

  The solution for this problem can be the implementation of parallel fragment shaders, but not in a sequential implementation as currently are, that is, with a parallel paradigm. For that, OpenMP will be applied over the existing code (see www.openmp.org). The parallelization of the Fragment Shaders can be easy if we put a barrier for waiting before going to the next stage. Each Fragment Shader has an input and an output private signal that only the boxes connected with them can read, that is, there is not interaction between the Fragment Shader Units (as commented before). See figure D.2 for an example of how implementation should be.



```
for(cycle = 0, end = FALSE; !end; cycle++)
    {   ......
        commProc->clock(cycle);
        memController->clock(cycle);
        streamer->clock(cycle);

        for(i = 0; i < simP.gpu.numVShaders; i++)
        {
            vshFetch[i]->clock(cycle);
            vshDecExec[i]->clock(cycle);
        }

        primAssem->clock(cycle);
        clipper->clock(cycle);
        rast->clock(cycle);

        // Code for the parallelization of the Fragment Shaders' simulation
        omp_set_num_threads(simP.gpu.numFShaders);
        #pragma parallel
        {
            #pragma omp for private(i,j) schedule(static,BL)
            for(i = 0; i < simP.gpu.numFShaders; i++)
            {
                fshFetch[i]->clock(cycle);
                fshDecExec[i]->clock(cycle);
                for(j = 0; j < simP.fsh.textureUnits; j++)
                    textUnit[i * simP.fsh.textureUnits + j]->clock(cycle);
            }
        }

        for(i = 0; i < simP.gpu.numStampUnits; i++)
        {
            zStencil[i]->clock(cycle);
            colorWrite[i]->clock(cycle);
        }

        dac->clock(cycle);
```

Each iteration is done by one thread, with an implicit barrier at the end

Figure D.2. Parallelization of fragment shaders with OpenMP.

- Problem 2: In each cycle, each stage has to wait to the previous for being executed (and just one cycle), when in a real GPU it does not happen (each stage executes one cycle at the same time).

  The extensible model that ATTILA uses allows us to parallelize the execution of each stage in an “easy” way. The solution for that problem can be the execution in parallel of each stage, using the buffers with locks for the connections and barriers at the end of each stage for simulating only one cycle per stage at the same time (and no more). This solution is not easy. We have to implement a lock for the signals (buffers), statistics dumps, and not go to panic mode when there is no data in a signal. On the first cycles, all the buffers are empty, so each stage has to wait or perform an empty cycle (in exception of the first stage). With time, data will go through the buffers and stages. For example, when a fragment is created it has to go to the next stages, but for going from one stage to the next one, it has to wait at least one cycle. At the end, if it is possible to have an unlimited number of processors (for execute the thread of each stage in each processor) the slower box or 'stage' would be the one determining the simulator speed, like in a real GPU or processor.

  This scheme is shown on figure D.3. One thread will simulate the clock system, and is the one that synchronize the other threads (stages). In figure D.4 there are two example implementations with OpenMP and PThreads. The implementation in Pthreads was able to be compiled and executed, but not in a multiprocessor. These are some ideas that can work well with the simulator.

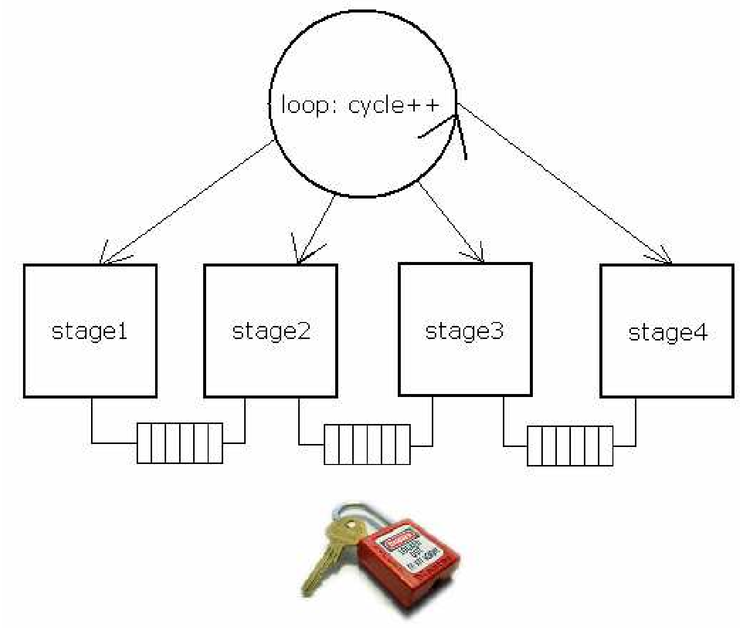
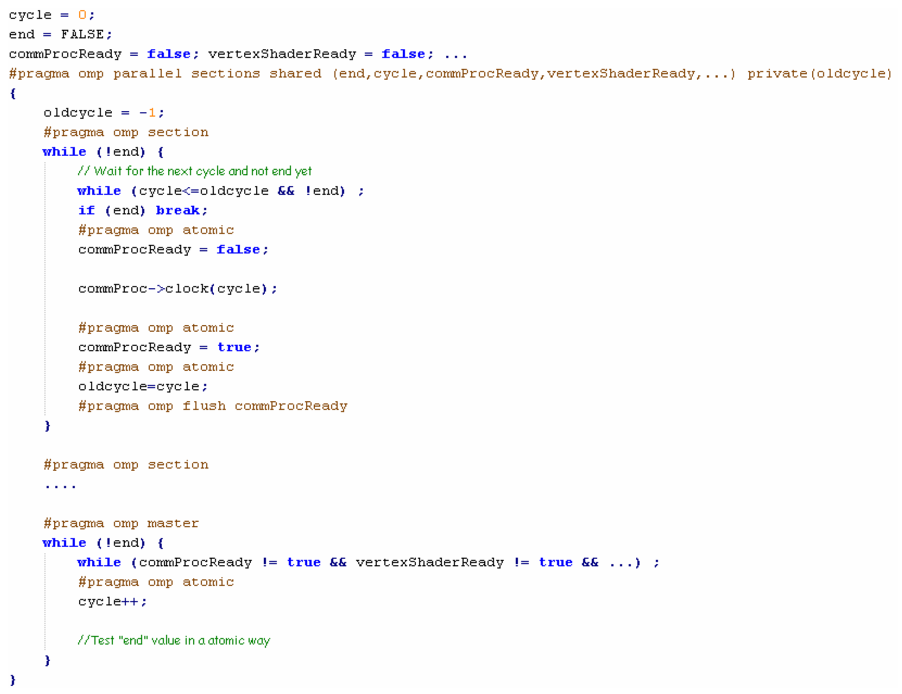
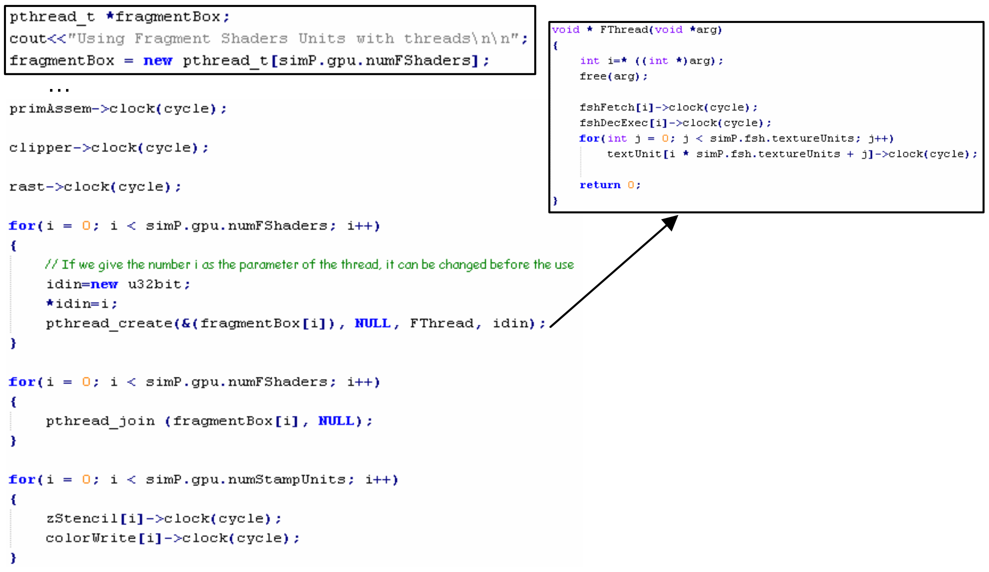


Figure D.3. Scheme of the execution in parallel of the stages.



```
cycle = 0;
end = FALSE;
commProcReady = false; vertexShaderReady = false; ...
#pragma omp parallel sections shared (end,cycle,commProcReady,vertexShaderReady,...) private(oldcycle)
{
    oldcycle = -1;
    #pragma omp section
    while (!end) {
        // Wait for the next cycle and not end yet
        while (cycle<=oldcycle && !end) ;
        if (end) break;
        #pragma omp atomic
        commProcReady = false;

        commProc->clock(cycle);

        #pragma omp atomic
        commProcReady = true;
        #pragma omp atomic
        oldcycle=cycle;
        #pragma omp flush commProcReady
    }

    #pragma omp section
    ....

    #pragma omp master
    while (!end) {
        while (commProcReady != true && vertexShaderReady != true && ...) ;
        #pragma omp atomic
        cycle++;

        //Test "end" value in a atomic way
    }
}
```

(a)



```
pthread_t *fragmentBox;
cout<<"Using Fragment Shaders Units with threads\n\n";
fragmentBox = new pthread_t[simP.gpu.numFShaders];
    ...
primAssem->clock(cycle);

clipper->clock(cycle);

rast->clock(cycle);

for(i = 0; i < simP.gpu.numFShaders; i++)
{
    // If we give the number i as the parameter of the thread, it can be changed before the use
    idin=new u32bit;
    *idin=i;
    pthread_create(&(fragmentBox[i]), NULL, FThread, idin);
}

for(i = 0; i < simP.gpu.numFShaders; i++)
{
    pthread_join (fragmentBox[i], NULL);
}

for(i = 0; i < simP.gpu.numStampUnits; i++)
{
    zStencil[i]->clock(cycle);
    colorWrite[i]->clock(cycle);
}
```

```
void * FThread(void *arg)
{
    int i=* ((int *)arg);
    free(arg);

    fshFetch[i]->clock(cycle);
    fshDecExec[i]->clock(cycle);
    for(int j = 0; j < simP.fsh.textureUnits; j++)
        textUnit[i * simP.fsh.textureUnits + j]->clock(cycle);

    return 0;
}
```

(b)

Figure D.4. Example implementation of the execution in parallel of each stage using (a) OpenMP and (b) PThreads

Content table

Figures table