A Flexible Simulator for Exploring Hardware Rasterizers

Iosif Antochi, Ben Juurlink, and Stamatis Vassiliadis
Computer Engineering Laboratory, Delft University of Technology
P.O. Box 5031, 2600 GA Delft, The Netherlands
Phone: +31 (0)15 27 83644, Fax: +31 (0)15 27 84898
E-mail: {tkg|benj|stamatis}@ce.et.tudelft.nl

Abstract— **Owing to technological advances, applications that once run on PCs are now becoming available on battery-powered, mobile devices such as PDAs and even mobile phones. Due to specific limitations of these devices such as power consumption and gate count restrictions, until recently, their processing power was limited. To overcome the limited processing power, different multimedia accelerators have been designed and implemented. In order to design and evaluate the performance and power consumption of such multimedia accelerators, in particular hardware graphics accelerators, we started implementing a flexible OpenGL simulator. This paper describes this simulator suitable for design space exploration of hardware graphics accelerators at the architectural level.**

Keywords— **graphics architecture, rasterizer simulator.**

I. Introduction

The performance of interactive graphics architectures [1][2][3] has been improving at phenomenal rates over the past few decades. The demand was for faster and more powerful renderers, thus only solving the system design problem of how to achieve maximum rendering performance from the technology available to implement the system was an issue. On the other hand, with the advent of mobile platforms for computing and communications, system designers and integrators were confronted with a massive shortage of low-cost, moderate-to-high performance 3D graphics systems suitable for low-power operation. Only recently, a few proposed and commercially implemented 3D graphics architectures [4][5][6] have emerged amenable for the strict requirements of "green" mode of operation of embedded systems. For this shifted paradigm, viewed in the larger context of system-on-chip affordable today, the design methodology is still in its infancy and rather an unexplored field, nevertheless looking very promising to researchers and with a large room for innovation.

The main goal of our project is the design of a low-power 2D and 3D graphics accelerator for mobile terminals equipped with an ARM core. The purpose of this accelerator is to relieve the burden of graphical computations from the ARM CPU core. Initially, the accelerator concerns only the back-end stage of the graphics pipeline, more specifically, the rasterization stage.

In order to design and evaluate the performance and power consumption of such an accelerator, we needed an OpenGL architectural level hardware graphics simulator. We searched for a flexible OpenGL simulator and to the best of our knowledge, an architectural level hardware graphics simulator was not publicly available. There are, however, software OpenGL implementations that support profiling such as GLSim [7] from Stanford and GLDebug from SGI. Since we could not found an appropriate simulator four our purpose we started designing our own flexible simulator framework. Our simulator is based on the freely available Mesa library [8] (it is coupled with the Mesa library as a driver) and it simulates the back-end of the OpenGL graphics pipeline (the rasterization stage). The main advantage of the presented simulator is that it allows a high degree of flexibility for studying the different trade-offs (e.g. the format and the precision used to represent the coordinates of the graphics primitives) that can be made at the architectural level of a graphics hardware accelerator.

The simulator accurately measures the data traffic between different parts of the system. Since data traffic accounts for a large part of the overall energy consumption [9], reducing the power consumption can be achieved by optimizing the data traffic between different blocks of the accelerator and between the accelerator and the main memory/CPU. Early, high-level estimation of data traffic is needed in order to reduce the architectural design space, since complete exploration of the search space at, for example, the gate level is infeasible.

II. Background

The design of a hardware architecture for a computer graphics pipeline requires a thorough understanding of the algorithms involved at each stage, and the implications these algorithms have on the organization of the pipeline architecture. The choice of algorithm, the flow of geometry data through the pipeline, and bit width precision are crucial issues in the design of new hardware accelerators. Making these decisions correctly requires intensive inves-

tigation and experimentation.

One of the most important terms encountered in computer graphics is **Color**. Due to the fact that computers have finite resources, all the values used for computation in computer generated graphics have a certain degree of approximation. In the case of the color values, the RGB representation it is widely used. In this representation each color that we want to represent will be obtain by specifying its Red ,Green and Blue components (channels), so each color (or approximation) can be represented as a combination of the three primary components mentioned above. As a practical example, most present graphics cards represent colors either as a 16-, 24-, or 32-bit values, each representation having a number of bits allocated per RGB component (e.g. A 16-bit color value has a 5-bit Red component, a 6-bit Green component, and a 5-bit Blue component). In some cases, for instance when it is necessary to simulate transparency, another channel (named Alpha) can be added, and the new tuple is called an RGBA color. There are also other systems to represent colors based on luminance or intensity but we do not describe them here. For the same reason (limited resources) the output device (display) of a computer has a discrete nature being organized as an array of elements, each element being able to store a color component. This organization of the output system has an impact on the visual quality of the represented scenes since, if the number of points per output device is to small, unpleasant visual effects can occur. One of this effects is aliasing. This effect is observed even in the simple case of drawing a line that is neither vertical nor horizontal. Instead of displaying a continuous smooth line, the display can represent a line that has a "stair-case" shape.

Another term often used in graphics is **texture**. Usually a texture represents an image that will be mapped on a 3D object in order to make its appearance more realistic. The advantage of using this approach to draw a 3D object, instead of defining each point of the object, is that the computations required to draw the object are drastically reduced due to the fact that some computations are executed only on the vertices of the respective object. For some applications (e.g. games) a texture might represent more than a pattern that will be mapped onto a object. For instance, if we choose to represent a wooden box, besides the look of wood material also the sound that is produced when we hit the box is important so it becomes part of the texture, but from the graphics point of view this is irrelevant since only the color components are used to represent objects.

3D graphics refers to systems used to create and manipulate a modeled "world" or scene on a computer and to display the world on the 2D computer screen in a realistic way. The world is typically constructed from objects made up from meshes of adjoining triangles or polygons each defined by its vertices. Each vertex has a number of properties including its position in 3D space (x, y, z) and color. Each polygon additionally has some global properties such as texture. To allow users to interact with the 3D world, either to view it from a new position or to change objects within it, the entire world must be processed to produce a new image to be displayed on a screen "viewport". This processing consists of three main steps: transform and lighting, hidden surface removal, and texturing and shading. Transformation alters the world as objects move within it or as a user's point of view changes. Each change can affect the position of any or all of the vertices within the world. Lighting then performs the calculations necessary to simulate the effect of different lights on objects in the scene, affecting the color of each vertex as a result. Finally texturing and shading determine the color of each pixel in the scene by taking into account both the color of the polygons and their texture. Textures are stored in memory and the relevant pixel from the texture map (called texels) are retrieved and used to texture each pixel before it is written into the display memory. Depending on the texturing technique, each output pixel requires a different number of texels. The simplest technique, called point sampling texturing, uses a single texel from the texture map. More advanced texturing schemes interpolate among more texels to accurately estimate the texture for the output pixel.

Even though we focus on 3D graphics, it is worth mentioning that 2D graphics is also an important part of a computer graphics accelerator. A 2D graphics accelerator should include circuitry able to speed up the executions of functions like line drawing, block transfers, area fill, logical and arithmetic mixing, map masking, scissoring, and hardware cursor. On the other hand, traditionally, the 3D graphics generation process has a pipeline structure. The main function of a 3D graphics pipeline is to generate three dimensional objects on a two dimensional device while taking into consideration factors such as light sources, lighting models, textures and other information that would have an impact on rendering quality.

III. Proposed Framework

In order to implement the framework for the simulator, besides the Mesa library, we have searched for a portable and flexible library to be used for the simulator graphical interface (windows, menus) but also for the underlying components of the simulator such as communication and synchronization. One important criterion in choosing the library was retargetability, so that the simulator to be eas-

ily ported and run on different systems. Also an object oriented approach was preferred. The best available solution was found to be the Qt library [10], which is easy to learn and, hence, suitable for the implementors and the extenders of the simulator.

A. Global Structure of the Simulator

The structure of the simulator framework is depicted in Fig. 1. Input for the simulator is generated by running applications (benchmarks) through the Mesa library, which is augmented so that it sends the specific rasterization primitives to the simulator. Based on the performance evaluation of the obtained results the rasterizer architecture can be modified, and the whole process is restarted.

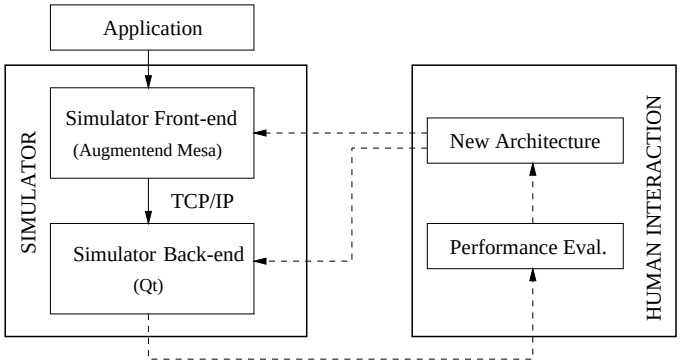

Fig. 1. The simulator framework.

Actually, the simulator is designed as two distinct entities:

- The Front-end. This part corresponds to the augmented version of the Mesa library. It receives data to be rendered from the benchmarks and implements the Transform and Lighting stages of the OpenGL pipeline. The rasterization instructions are sent to the back-end of the simulator.
- The Back-end. This unit is the core of the simulator. This unit receives data from the augmented Mesa unit and simulates the rasterizer.

The communication between the Front-end and the Back-end is implemented using the TCP/IP protocol. This allows the benchmark and the Front-end to be run on one machine and the simulator on another. This approach can be useful for running applications that are grabbing the input devices, so that the user cannot interact with other applications while the current application is running, or run full-screen since the simulator can be controlled separately on another machine. Furthermore, the Front-end can use the Mesa software rasterizer to display the result of the rendering process. By running the rasterizer simulator on another machine, a visual comparison of the image generated by the Mesa software rasterizer (the reference) and image generated by the simulator can be performed. Since the simulator might implement different algorithms in order to improve performance or reduce power (e.g. using different precision to represent colors or coordinates), the image quality can be affected so in order to evaluate the effect of the trade-offs to the image quality, a comparison to a reference image can be performed.

The simulator can also read data from files written by the Augmented Mesa unit for off-line processing.

B. Front-end (Mesa)

Since we are mainly concerned about the rasterization part of an OpenGL hardware accelerator, we searched for a software OpenGL implementation that can be used to perform the required operations needed before the rasterization stage of the graphics pipeline. After examining current software OpenGL implementations, we determined that Mesa is one of the most complete software implementations of OpenGL and that it can be considered a reference implementation. Recently, SGI released its free reference OpenGL implementation [11], but it seems that this implementation requires an X Server in order to be functional, since it is an extension to an X server.

The structure of the Mesa Library is depicted in Fig. 2. Mesa provides a kernel “Mesa Core” that handles the interface with the applications, the 3D pipeline generation, and context management. The Software Rasterizer (Off-Screen Mesa) is a device independent renderer which can be compiled without major modifications on any architecture, since it provides the simplest structure of a graphics device by directly using the main memory for output of the rendering process.

For each hardware device that needs to be controlled by Mesa, a device driver table must be filled in which describes its capabilities regarding the rendering of triangles, lines, and/or points. If the device does not implement in hardware one of the above mentioned primitives, then the default software rasterizer is used to render the respective primitive. This ensures that whenever a primitive is not directly supported in hardware, it will always be supported in software. This device table is actually where we hooked up into Mesa and provided our own rasterization routines. The real rasterization process is implemented as a separate application so that the Mesa part of our simulator (the front-end) is completely separated from the actual simulator (the back-end).

C. Back-end (Qt)

In this section we describe the function of each unit and discuss the issues that need to be investigated.

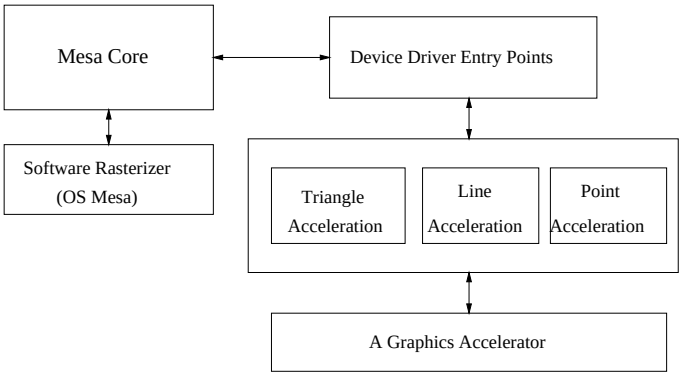

Fig. 2. The Structure of Mesa Library v3.4.

The Back-end of the simulator was implemented using an Object Oriented approach, each component is represented by an object. By using a separate object for each unit, each unit can be easily modified as long as the interface remains the same, which makes the simulator very flexible. furthermore, by using a highly portable library as a back-end of our simulator, the Qt library, the machines that can be used as simulation platforms can be easily extended.

Depending on the implementation solutions, there are some specific blocks to each accelerator but there are also some common blocks that each hardware rasterizer should contain. The datapath with the control logic of a simple 3D graphics accelerator is depicted in Fig. 3. The implementation is organized as a pipeline which is composed of the following stages: *Bus Interface*, *Triangle Setup*, *Span Interpolation*, *Texture Processing*, *Pixel Processing*, and *Video Memory*.

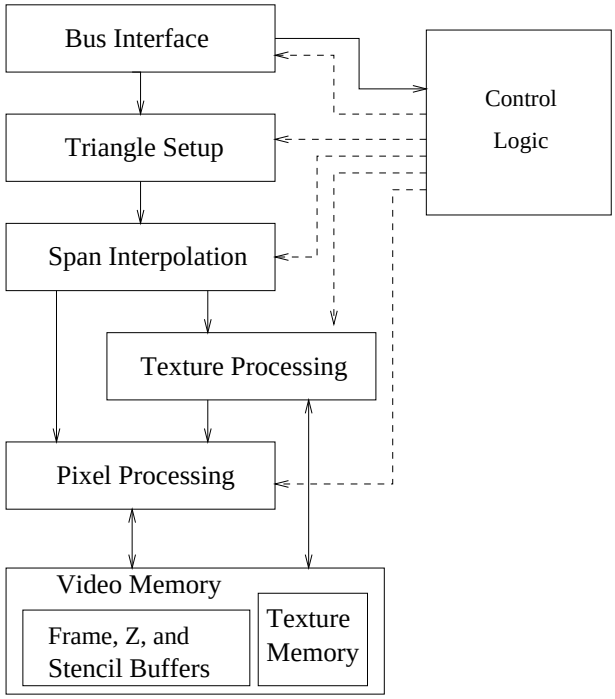

Fig. 3. The datapath with the control logic of a simple 3D graphics accelerator.

C.1 Bus Interface

This unit is responsible for the data transfers among the rasterizer and the processor or main memory. Instructions sent by the processor, are received here and parts of them (the opcodes) dispatched to the *Control Logic* unit. The remaining parts of the instructions (the operands) are buffered and later sent to the units that require them such as Triangle Setup or Texture Processing.

C.2 Triangle Setup Unit

Considering a complete software implementation of a graphics library as a reference, we can distinguish increasing degrees of acceleration of the graphics primitives by moving their execution from the host processor to a dedicated graphics processing device. Most 3D graphics architectures include a rasterizer to which the 3D vertex coordinates in image space, their associated color values, and for some architectures also texture coordinates, are sent. For rasterization it is common to use triangles or triangle strips as basic drawing primitives. The rasterizer interpolates the depth and color values for all the pixels bounded by the edges which define the triangles. Since triangles are planar shapes, a first idea to draw a triangle would be to linearly interpolate the vertex parameters along the edges and then linearly interpolate the edge values along scan lines. The problem that might appear is that linear interpolation is correct only when all three vertices have the same depth (z distance) from the observer. Otherwise, a “perspective correction” is needed since the human perception of depth is not linear, but hyperbolic.

Moving the slope and setup calculations for triangles to the rasterizer off-loads the host processor from intensive calculations and can significantly increase 3D system performance. Besides freeing the main processing unit, computing triangle setup data on the hardware rasterizer has the additional advantage of freeing the data bus, or in other words, to allow a higher triangle transfer rate to the hardware accelerator. The usual amount of geometry information required to draw a triangle is presented in Table I. *If the triangle setup stage is implemented in hardware, then the group called “Setup parameters” in Table I will be computed by this unit, so the required geometry data traffic between the CPU and the hardware accelerator is reduced considerably*.

	Parameters
Triangle data	
Vertices	$x_0,\ y_0,\ z_0,\ x_1,\ y_1,\ z_1,\ x_2,\ y_2,\ z_2$
Colors	$r_0,\ g_0,\ b_0,\ \alpha_0,\ r_1,\ g_1,\ b_1,\ \alpha_1,\ r_2,\ g_2,\ b_2,\ \alpha_2$
Texture Coordinates	$u_0,\ v_0,\ u_1,\ v_1,\ u_2,\ v_2$
Setup parameters	
Edge interpolation increments	$\Delta x/\Delta y$ (for 3 edges)
color increments	$\delta r/\delta x,\ \delta r/\delta y,\ \delta g/\delta x, \delta g/\delta y,$ $\delta b/\delta x,\ \delta b/\delta y,\ \delta\alpha/\delta x, \delta\alpha\ \delta y$
depth increments	$\delta z/\delta x,\ \delta z/\delta y$
texture increments	$\delta u/\delta x,\ \delta u/\delta y,\ \delta v/\delta x,\ \delta v/\delta y$

TABLE I
DATA NEEDED TO SETUP (DRAW) A TRIANGLE.

C.3 Span Interpolation Unit

The function of this unit is to linearly interpolate a set of parameters passed from the Triangle Setup unit. To understand what the problems are at this unit we should recall that on each vertex of an object in the initial 3D scene the following transformations are applied:

1. Transform coordinates form local to world coordinates.
2. Transform coordinates from world to eye coordinates.
3. Transform coordinates (perspective projection) from eye coordinates to device (window) homogeneous coordinates.
4. Perform primitive clipping.
5. Transform to normalized window coordinates.

The point in enumerating the above coordinate system transformations is that the last coordinate system used before device coordinates are the eye coordinates. Linearly interpolating all the parameters relative to window coordinates does not always correspond to a linear interpolation of the same parameters in eye space. The only case when linear interpolation in window coordinates corresponds to linear interpolation in eye space is when all the vertices of the primitive have the same depth (z) value in the eye coordinates. To obtain a correctly rendered primitive, theoretically, a division per parameter (e.g. color components and texture coordinates) that has to be correctly interpolated, should be performed, at each interpolation point. If a vertex in the eye coordinates has the following homogeneous representation: $P_e = [x_e, y_e, z_e, w_e]$, to obtain the window coordinates, a perspective projection should be performed and the respective homogeneous coordinates are $P_w = [x_w, y_w, z_w, w_w]$. To obtain the normalized window coordinates each component has to be divided by the w_w component. Thus the window coordinates of the vertex P are $P'_w = P_w/w_w = [x_w/w_w, y_w/w_w, z_w/w_w, w_w/w_w] = [x'_w, y'_w, z'_w, 1]$. A similar process should be applied to the other parameters of each vertex (colors, texture coordinates) (more details are presented in [12]). So to correctly render using a linear interpolation method the following steps have to be performed

1. Construct a vector of values for each vertex of the triangle

$$V = [x_w, y_w, z_w, w_w, p_1, p_2, p_3, ..., p_n, 1],$$

where n is the number of parameters that have to be interpolated for each vertex, and the parameters $p_1, ..., p_n$ represent colors or texture coordinates.

2. Divide V by w_w. The new vector is

$$V' = [x'_w, y'_w, z'_w, 1, p'_1, p'_2, p'_3, ..., p'_n, 1'],$$

where $1' = 1/w_w$.

3. Linearly interpolate all the elements of V' along polygon edges and across scan lines inside the triangle.

4. At each pixel divide the parameters p'_i by the corresponding $1'$ value to get the proper perspectively projected p values.

It is guaranteed that $1' \neq 0$, because after clipping all w_w are positive.

Also, since division is slower than multiplication, if there are many parameters p to interpolate then instead of computing n divisions per pixel, each having the same divisor, one division and n multiplications can be computed instead.

All parameters computed at this unit are sent to the following units on a "per fragment" base. Thus from now on fragments are processed instead of primitives. Mostly a fragment corresponds to a pixel, but this is not always true since a fragment can be discarded so it might never be part of the final image or it can be altered before reaching the frame buffer.

C.4 Texture Processing Unit

Texture mapping is one of the methods employed to obtain realistic picture generation. To obtain real-time performance, usually a hardware implementation is required. Texture mapping introduces two major problems: texture coordinates computation and texture filtering. Computing texture coordinates is an operation which can introduce rendering latency compared with the computation of pixel coordinates. Usually, pixel coordinates are computed by means of linear interpolation, but for "perspective corrected" texture coordinates, linear interpolation is insufficient [13] and a hyperbolic interpolation is used or approximated. Texture filtering is also an important operation since it can introduce more rendering latency in the graphics pipeline. A texture unit is a part of the graphics pipeline which computes the physical texel coordinates and applies a texture filtering method on each pixel presented at the input. Textures can have different dimensions (usually powers of 2). Therefore, to obtain independence of the texture size, all units before the texture unit use relative values (e.g. real numbers in the range [0,1]), for texture coordinates, and at the texture unit these coordinates are transformed to physical texture coordinates. According to the filtering method (point sampling, linear, bilinear, or trilinear), the texture unit computes the RGB components for the current pixel using 1, 2, 4, or 8 texels. It is important to note that for each pixel as many as 8 accesses to the texture memory might be needed, so the bandwidth between the texture unit and the texture memory should be about 8 times larger than the bandwidth between the span generator and the texture unit. In order to reduce the data traffic from the texture memory to the texture unit and also the power consumption which is important in low power devices, and due to the high spatial locality, a texture cache can be used between the texture unit and the texture memory. In a previous work [14], we proposed employing a very small (128-512 bytes) texture cache between the texture unit and the texture memory. Some issues that need to be investigated at this unit are:

1. Mip-map level selection when using mip-maps
2. What happens if the texture coordinates are larger than the texture size?
3. How is the color from a texture map combined with the incoming (RGBA) color?

C.4.a Mip-map Level Selection. One reason for using mip-mapping is to increase the texture filtering quality by using prefiltered (smaller) copies (texture planes) of the original textures. The problem is that it needs to be known how to select the "best" texture plane(s). The "best" texture plane is the plane for which the ratio texels per pixel is the closest to 1. There can be two situations:

1. Magnification - one texel is mapped to multiple pixels.
2. Minification - multiple texels are mapped to the same pixel.

In order to find the best texture plane(s) the following parameters can be used. Let $\rho(x, y)$ be a scale factor and let

$$\lambda'(x, y) = \log_2(\rho(x, y)) \tag{1}$$

Then the "level of detail" parameter $\lambda(x, y)$ is defined as

$$\lambda(x, y) = \begin{cases} \textit{Max\_LD}, & \lambda'(x, y) > \textit{Max\_LD} \\ \lambda'(x, y), & \textit{Min\_LD} \leq \lambda'(x, y) \leq \textit{Max\_LD} \\ \textit{Min\_LD}, & \lambda'(x, y) < \textit{Min\_LD} \\ undefined & \textit{Min\_LD} > \textit{Max\_LD} \end{cases} \tag{2}$$

where *Max_LD* and *Min_LD* are constants corresponding to the minimum and the maximum level of detail. If $\lambda(x, y)$ is less than or equal to a constant c the texture is said to be magnified. If it is greater, the texture is minified. The value of c, the minification vs. magnification switch-over point, is computed with respect to the minification and magnification filters. If the magnification filter is given by LINEAR and the minification filter is given by NEAREST_MIPMAP_NEAREST or NEAREST_MIPMAP_LINEAR, then $c = 0.5$. This is done to ensure that a minified texture does not appear "sharper" than a magnified texture. Otherwise $c = 0$.

Considering that the u, v texture coordinates, for each interoplation point, are functions of x and y, that is $u = u(x, y)$,$v = v(x, y)$ then the ρ function used in Eq. (1) is defined as:

$$\rho(x, y) = \max\left\{\sqrt{\left(\frac{\delta u}{\delta x}\right)^2 + \left(\frac{\delta v}{\delta x}\right)^2}, \sqrt{\left(\frac{\delta u}{\delta y}\right)^2 + \left(\frac{\delta v}{\delta y}\right)^2}\right\} \tag{3}$$

While it is agreed that Equation (3) gives the best result when texturing, it is often impractical to implement. Therefore, an implementation may approximate the ideal ρ with a function f subject to the following conditions:

1. f is continuous and monotonically increasing in $|\delta u/\delta x|$, $|\delta u/\delta y|$, $|\delta v/\delta x|$, as well as $|\delta v/\delta y|$.
2. Let

$$m_u = \max\left\{\left|\frac{\delta u}{\delta x}\right|, \left|\frac{\delta u}{\delta y}\right|\right\}$$

$$m_v = \max\left\{\left|\frac{\delta v}{\delta x}\right|, \left|\frac{\delta v}{\delta y}\right|\right\}$$

Then $\max\{m_u, m_v\} \leq f(x, y) \leq m_u + m_v$.

C.4.b Texture Warping Modes. Texture parameters received from the span interpolation unit are multiplied by the texture size for the selected plane (level) of the texture. After this procedure the obtained texture coordinates can be larger than the size of the texture. According to OpenGL specification in order to reduce the texture coordinates to the size of the texture space the coordinates can be clamped or repeated [15, pp. 124-125].

C.4.c Texture Blending Functions . After obtaining a color from a texture map, the next step is to define how this color is combined with the incoming fragment's color. According to [15], a separate constant RGBA color can also be used to compute the final color of a fragment besides the primary color of the fragment and the color obtained from texturing. Traditionally, there are four combining functions that can be used: *Replace*, *Modulate*, *Decal*, or *Blend*. Newer graphics accelerators (e.g. Geforce 3 and 4 from Nvidia, and Radeon 8500 and 9700 from Ati) have a more complex blending scheme since they implement a programmable texture unit called "Pixel shader". We only implement the four standard modes of blending, but the simulator can be extended to support more advanced blending functions. Furthermore, modern graphics accelerators are using multiple texturing units to reduce rendering latency, but this approach, in the case of low-power devices, for instance, might be less practical due to the gate count limitation and power consumption.

More detailed information about the texturing process can be found in [15].

C.5 Pixel Processing Unit

This unit is responsible for various tasks at the pixel ("fragment") level. As shown in Fig. 4, this unit can also be organized as a pipeline. For each block depicted in Fig. 4, a corresponding C++ class was implemented. Thus to simulate a different hardware implementation, it is possible to rewrite or inherit the provided implementation with an implementation that emulates the hardware unit to be simulated. Typical operations performed at this unit are various tests to eliminate pixels that were generated by the previous stages of the rasterizer but are not part of the final scene. Also at this level a pixel can have its properties modified (e.g. blending process).

C.5.a Tests. The most common tests (each of which can be disabled) are:

- Pixel ownership test - this test is used to determine if a pixel really belongs to the current graphic context or to another window that obscures the current context. If the pixel is not owned by the current context then it is discarded. Otherwise it is passed along the pipeline.
- Scissor test - this test determines if the pixel lies within a specified rectangle.
- Alpha test - this test is passed if the pixel's alpha value is in a certain (selectable) relation (e.g. less, less or equal, greater) with a fixed reference value.
- Stencil test - this test is passed if the value stored in the stencil buffer at the pixel's coordinate is in a certain relation (selectable) with a reference value.
- Depth buffer test - this test is passed if the pixel's z-value is in a certain relation with the z-value stored at the pixel's coordinate in the z buffer.

More details about the presented tests can be found in [15].

C.5.b Blending. At this unit a pixel's color components and alpha value are combined with the color components and alpha value of the pixel already in the frame buffer at the pixel's coordinates.

C.5.c Dithering. Dithering is an operation that assigns to a pixel a value that can be different than the initial color presented at the entry of this unit. The new assigned color can be a function of the pixel's initial color and the pixel's coordinates. The eventually new assigned color value should not exceed the maximum representable value in the frame buffer. The dithering process can be used when the number of bits of the colors stored in the frame buffer is smaller than the number of bits used to represent the colors in the rasterizer.

C.5.d Logical Operations. For the final phase a pixel can be combined with the pixel stored in the frame buffer by applying a logical operation such as AND, XOR, COPY, CLEAR and others.

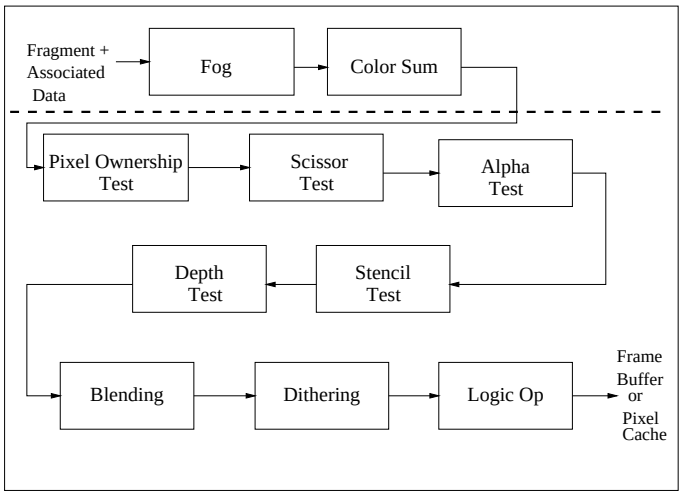

Fig. 4. Pixel Processing Unit.

C.6 Video Memory

The video memory is implemented as a flexible component that can be configured to simulate, at a high level, a memory module. Actually the memory component serves as an underlaying component for frame, stencil, and depth buffers, and also for textures.

D. The Control Logic

In order to control the units described in Section III-C we implemented a control unit. The operation of the control unit depends of the opcodes of the instructions received from the Bus interface unit. The control unit is responsible for the synchronization among the components that are part of the graphics pipeline and simulates the Control logic of a hardware rasterizer.

IV. CONCLUSIONS

In this paper we presented the structure of a framework for a flexible OpenGL rasterizer simulator. The framework allows the exploration of the architectural design space for an OpenGL compliant rasterizer. By using an object oriented approach we achieved a high degree of flexibility so that the evaluation process of different rasterizer architectures can be performed easily. Our simulator framework is ongoing work, and it has to be extended so that it can produce not only data traffic estimations but also power consumption estimations.

REFERENCES

[1] H. Fuchs, J. Poulton, J. Eyles, T. Greer, J. Goldfeather, D. Ellsworth, S. Molnar, G. Turk, B. Tebbs, and L. Israel. Pixel-planes 5: A heterogeneous multiprocessor graphics system using processor-enhanced memories. *Computer Graphics*, 23(3), 1989.

[2] M. Suzuoki et. al. A microprocessor with a 128-bit cpu, ten floating-point mac's, four floating-point dividers, and an mpeg-2 decoder. *IEEE Journal of Solid-State Circuits*, 34(11), 1999.

[3] M. Eldridge, H. Igehy, and P. Hanrahan. Pomegranate: A Fully Scalable Graphics Architecture. In *Proceedings of the Conference on Computer Graphics*, pages 443–454, New Orleans, 2000.

[4] B.-S. Liang, W.-C. Yeh, Y.-C. Lee, and C.-W. Jen. Deferred lighting: A computation-efficient approach for real-time 3-d graphics. In *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS)*, pages IV.657–IV.660, Geneva, May 2000.

[5] B.-S. Liang and C.-W. Jen. Computation-effective 3-d graphics rendering architecture for embedded multimedia system. *IEEE Transactions on Consumer Electronics*, 46(3):735–743, August 2000.

[6] PowerVR Technologies. 3D Graphical Processing, November 2000.

[7] GLSim, *http://graphics.stanford.edu/courses/cs448a-01-fall/glsim.html*.

[8] Mesa 3D Library, *http://www.mesa3d.org*.

[9] F. Catthoor, F. Franssen, S. Wuytack, L. Nachtergaele, and H. De Man. Global Communication and Memory Optimizing Transformations for Low-Power Signal Processing Systems. In *VLSI Signal Processing Workshop*, 1994.

[10] Qt Library, *http://www.trolltech.com*.

[11] Silicon Graphics Inc.'s OpenGL sample implementation, *http://oss.sgi.com/projects/ogl-sample/*.

[12] P. Heckbert. Survey of Texture Mapping. *IEEE Computer Graphics and Applications*, pp. 56-67, November, 1986.

[13] P. Heckbert. Fundamentals of Texture Mapping and Image Warping. Technical Report ucb/csd 89/516, University of California, Berkeley, 1989.

[14] Iosif Antochi, Ben Juurlink, Andrea Cilio, and Petri Liuha. Trading Efficiency for Energy in a Texture Cache Architecture. In *Proceedings of the 4th International Conference on Massively Parallel Computing Systems (to appear)*, 2002.

[15] Mark Segal and Kurt Akeley. *The $OpenGL^{TM}$ Graphics System: A Specification*. Silicon Graphics, April 1999.