Design and Implementation of a Hardware Coprocessor for Ray Caster Volume Rendering*

Arnaldo Oliveira, Beatriz Sousa Santos

Resumo - **Este artigo descreve a implementação de um coprocessador de hardware destinado a volume rendering. Este tipo de aplicação requer bastantes recursos, quer de armazenamento, quer computacionais. Felizmente, os primeiros já não são tão problemáticos hoje em dia. No entanto, altos desempenhos de processamento não são facilmente atingidos sem a utlização de hardware dedicado. A abordagem proposta tira partido da utilização de hardware reconfigurável, permitindo que o mesmo seja partilhado por diferentes aplicações, contribuindo assim para uma diminuição dos custos de aquisição de coprocessadores especificos para cada aplicação.**

Abstract - **This paper discusses the implementation of an hardware coprocessor for volume rendering. This type of applications requires much storage and computational resources. However, high performance processing isn't easily achieved without dedicated hardware. The proposed approach takes full advantage of reconfigurable hardware, which allows sharing the same hardware among different applications, thus saving the investment cost of specific coprocessors for each application.**

I. INTRODUCTION

Volume rendering means rendering voxel-based data [1]. Rendering voxels currently finds two major applications: rendering CSG models and the visualization of scalar functions of three spatial variables. This work is concerned with the second case, which corresponds to one of the most important applications of Scientific Visualization. Such data, prior to the availability of hardware and software for volume rendering, was visualized using such "traditional" techniques as isocontours in cross-sectional planes.

A voxel volume is either produced by a mathematical model, such as in computational fluid dynamics, or the voxels are collected from the real world, as in the medical imaging. Visualization software generally treats both types in the same way. Medical imaging has turned out to be one of the most popular applications of volume rendering. It has enabled data collected from a tomographic system as a set of parallel planes to be viewed as a three-dimensional object [1, 2, 3].

The basic idea of volume rendering is that a viewer should be able to perceive the volume from a rendered projection on the view plane. In medical imaging we may want to view a surface, the volume, or just part of the volume. Thus, we view the extraction and display of "hard" surfaces that exist in the data as part of volume rendering problem. In many cases we may have a volume data set from which we may have to extract and display surfaces that exist anywhere within the volume. Rather than bounding surfaces of an object, we may be dealing with an object that possesses many "nested" surfaces – like the skin of an onion. If such surfaces are extractable by some unique property then we can render them visible by making them 100% opaque and all other data in volume 100% transparent. More generally, we may try to view either the whole volume or a subset of it by assigning a color and an opacity to each voxel and accumulating these values along a viewing direction.

This paper presents the results of a first approach to develop an hardware coprocessor for ray caster volume rendering. The work is divided in two main parts:

- Hardware coprocessor - based on Xilinx XC6200 development system [4];
- Software application - developed for Windows family of operating systems providing full access to the hardware and other useful functionality.

The paper is organized in eight sections. Section I is this introduction. Section II describes the ray caster algorithm. Section III provides a brief description of the development system used. In section IV some implementation guidelines are discussed. Section V presents the complete hardware architecture. In section VI the most important topics of the software application are described. Results are presented in section VII. Finally, some conclusions are drawn in section VIII.

II. RAY CASTER ALGORITHM

There are two major approaches to volume rendering: ray casting (backward mapping) [2] and plane compositing (forward mapping). We will consider only the first approach.

The ray caster algorithm has the following general structure:

```
for {each pixel}
{   fire a ray and find the voxels through
    which it passes }
```

The ray caster algorithm scheme is displayed in figure 1.

* Work developed, in the scope of the MSc course on Computer Visualization, during the academic year of 97/98.

This method accumulates information from all voxels that intersect the current ray casted through the current pixel. A single loop of the algorithm provides the final value for a pixel.

The voxels are stored in a three dimensional array and indexed by:

X = (x, y, z)

To apply this algorithm, each voxel must be mapped to a given color C(X) and opacity α(X) (where α=1 implies an opaque voxel and α=0 implies a transparent voxel). After that, we trace a set of parallel rays into the data. If a pixel coordinate is (i, j) we define a ray as the vector:

R = (i, j, k)

where:

(i, j) = r is the pixel index of the ray

k is the distance along the ray, k = 1,…, K

For each ray, we progress along it resampling the data at evenly spaced intervals, computing C(R) and α(R). As each color and opacity are computed, we accumulate those values progressing along the ray in a front-to-back order. This accumulation process uses the standard transparency formula and for a sample R we define an accumulated color and opacity after R has been processed as follows:

$$C_{out}^*(r,R) = C_{in}^*(r,R) + C^*(R)[1-\alpha_{in}(r,R)]$$

$$\alpha_{out}^*(r,R) = \alpha_{in}^*(r,R) + \alpha^*(R)[1-\alpha_{in}(r,R)]$$

where:

$$C_{in}^*(r,R) = C_{in}(r,R)\alpha_{in}(r,R)$$

$$C_{out}^*(r,R) = C_{out}(r,R)\alpha_{out}(r,R)$$

$$C^*(r,R) = C(r,R)\alpha(r,R)$$

Note that all colors (R, G, B) are premultiplied by their associated opacity and become (Rα, Gα, Bα) in the accumulation process. The reason for this is easily understood by considering simple examples. If α=0 the object is completely transparent and its color contribution to the accumulation must be (0, 0, 0). If, on the other hand, α=1 then the object is completely opaque and its color contribution is (R, G, B). Also when α=1, from thereafter in a (brute force) accumulation algorithm, α_{in}=1 and the second term of the equation is always zeroed. The final color is then the color accumulated when the opaque voxel is hit by the ray. At the end of the loop, for each ray, the final color is obtained by:

$$C(r) = \frac{C_{out}^*(r,R)}{\alpha_{out}(r,R)}$$

where R=(i, j, k).

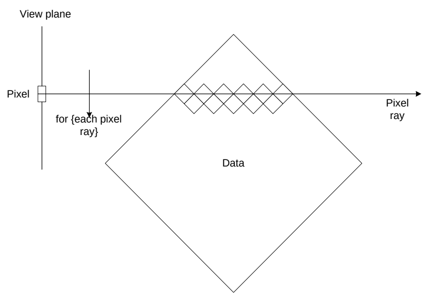


Figure 1 - Ray casting schematic representation (adapted from [1]).

III. DEVELOPMENT SYSTEM OVERVIEW

The platform used to implement the coprocessor was a Xilinx 6200 development system [4]. From its features, the more relevant for this application are:

- PCI based board;
- 16K User programmable gates within the XC6216;
- 512 Kbytes of fast SRAM;
- Plug and Play compliant.

It consists of a Xilinx 4013E FPGA and a compute element. The compute element is a Xilinx 6200 FPGA [5], four 8-bit wide SRAM's and six bus controller chips to control data flow. A Xilinx 4013E FPGA [6] is used as the PCI bus interface. The primary component of the compute element is a Xilinx 6216. The board architecture allows the XC6216 to be reconfigured through the PCI interface during run-time. The PCI interface provides direct access from the host PC to logic cells within the user's circuit. The compute element memory is organized into two banks, each bank consisting of two 128K × 8 bits SRAM's. A bank of RAM can be accessed from either the PCI interface or the XC6216.

Figure 2 shows the design flow that was adopted in order to implement the coprocessor in the development system. The starting point is the description of the circuit in a hardware description language, in this case VHDL [7]. This step is performed with a text editor and results in one or more VHDL source files (*.vhd) containing the description of all components needed. The next step is the compilation of these files using Velab [8] (available from Xilinx) which outputs a file containing an EDIF netlist. Next, based on the information contained in the netlist file, the XACT6000 [9] software (also available from Xilinx and specific of this FPGA family) performs the placement and routing of the circuit. When finished it can produce two types of output files:

- CAL - Contains the bit-stream used to configure the FPGA.
- SYM - Contains information about the placement of the components in the FPGA cells.

These files can be used as the input to a debugger such as IDELS [10] developed at the department or PCITest that comes with the board. Once the correctness of the circuit has been verified, they can also be used by the application in order to configure and access the hardware. However, in this application only the CAL file is used. Due to performance reasons, a low-level access mechanism is used.

IV. IMPLEMENTATION GUIDELINES

In this section some important implementation decisions will be presented.

The maximum voxel range must be 0 to 65535 (16 data bits). This range is large enough to accommodate both practical and simulated data. For most practical applications, data is sampled with less than 16 bit (normally between 8 and 16). In case of simulation, the

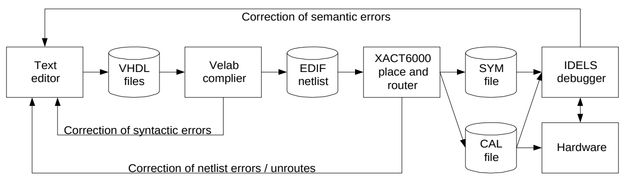

Correction of semantic errors
Text editor
VHDL files
Velab complier
EDIF netlist
XACT6000 place and router
SYM file
IDELS debugger
Correction of syntactic errors
CAL file
Hardware
Correction of netlist errors / unroutes

Figure 2 - XC6200 Design flow.

results can be represented in 16, 32, or 64 bit either in fixed or floating-point notation. We will assume that it's possible to convert from any of these representations to 16 bit fixed point without visible loss of information.

The implemented algorithm requires the mapping of each voxel value to color and opacity values. If this mapping is performed by one or several expressions, it can take some time (even if these expressions are calculated in hardware). To accelerate this step, a lookup table was used. This table is organized in the following way:

- The number of entries depends on the voxel range;
- Each entry is composed by a 32-bit word used to store a 24 bit RGB encoded color and an 8-bit opacity value.

The amount of memory occupied by the lookup table is negligible compared to the amount of volume data (even if the table has 64K entries corresponding to 256 Kbytes).

In the beginning of the project, we thought to implement the algorithm using floating-point arithmetic, however, after some research it was decided to use fixed-point arithmetic. The justification is the following: to perform an addition between two floating-point numbers in the form

$n_1 = m_1 \times b^{e1}$	m - mantissa
$n_2 = m_2 \times b^{e2}$	b - base (fixed)
	e - exponent

it is necessary to ensure that $e_1=e_2$. Each increment/decrement on the exponents corresponds to a shift in the mantissa, which unfortunately is a sequential process and can take several clock cycles, thus corresponding to a performance bottleneck. Moreover, the opacity range is limited to the interval [0, 1] (which helps to avoid floating-point representation). In order to perform any addition and multiplication in only one clock cycle, the opacity values are fixed-point encoded between $0(00000000_b)$ and $128(10000000_b)$, therefore corresponding to 129 levels. The number of opacity levels is large enough for most applications. Furthermore, we will see later that its adjustment is performed graphically in our application and the pointing device has a lower resolution.

A first design strategy was based on the transfer of voxel data already mapped to color and opacity values to the board local memory. The resulting rendered image was also stored in local memory and then transferred to the host memory. However, during implementation the following problems occurred:

- Due to the limited memory available on the board (512 Kbytes), and depending on the volume size, it could be necessary to perform several cycles of data transfer. This time, from the processing point of view is useless, which contributes to slow the overall system performance;
- The map table must be implemented in host memory, which is slower than the board static RAM, increasing the mapping time.
- Finally, to synchronize the datapath with the memory access, it is necessary a control unit. To ensure a correct operation of the circuit, the number of states must be greater than desirable for a high system performance.

Together, these problems lead to a change in the design strategy. The current design is asynchronous as will be shown in next section, however, some problems have not been solved yet. In the last section an approach that combines the best of the initial and implemented strategies, in order to achieve higher performances, is presented.

V. Hardware Architecture

The hardware architecture of the developed coprocessor is presented in figure 3. It can be divided in the following main components:

- A datapath composed by 4 multipliers (12 × 8 bits), 4 accumulators (12 bits) and 1 subtractor (12 bits). The circuit implements the ray caster equations presented in II and is located within the XC6216 FPGA cells;
- Output buffers, which align the outputs of all accumulators to the same FPGA column. This procedure allows the simultaneous reading of all accumulator outputs with a 32-bit bus width. These buffers are also

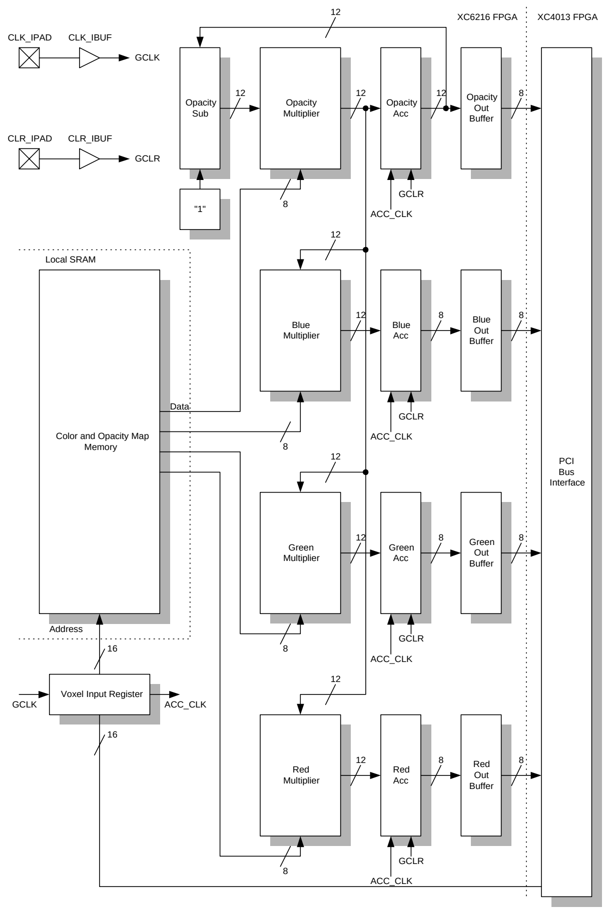


Figure 3 - Hardware coprocessor architecture.

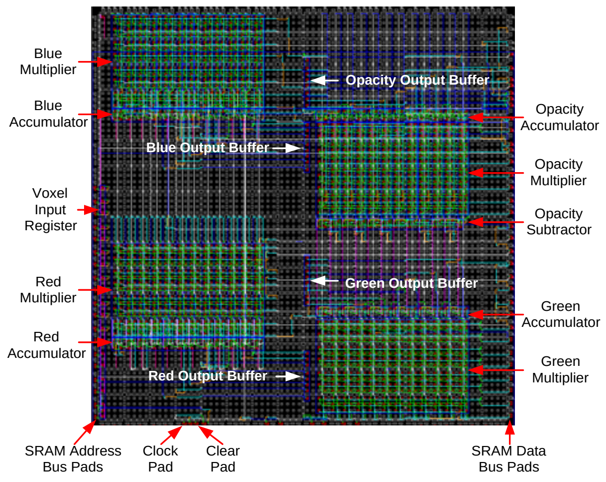


Figure 4 - FPGA global layout.

implemented in the XC6216 FPGA;

- A color and opacity map memory that converts the voxel data into color and opacity values, acting as a lookup table. This block is implemented on the board memory built with static RAM. The great advantage of this approach is the fast conversion rate that can be achieved (approx. 50 MHz). For 16 bit voxel data, 24 bit color and 8 bit opacity the required memory is 256 Kbytes, corresponding to half of the total memory available on the board (512 Kbytes);
- A voxel input register that latches and routes the voxel value from the FPGA data bus to the memory address bus. Each time a new value is written in this register, a clock pulse is applied to the accumulators, thus updating their outputs;
- A PCI bus interface circuit. Its function was already introduced in section III.

Figure 3 also represents the FPGA input pads used for clock and clear purposes. The clock signal is used only to synchronize the Voxel Input Register write operations. All the remaining circuit is completely asynchronous. The clear signal when asserted resets the contents of the accumulators.

One feature available in reconfigurable hardware systems is the possibility of hardwire values frequently used in calculations, saving chip area and improving performance, this approach is used in the opacity subtractor. The additive is fixed and equal to 1 (in fact is 128, see section IV for details). The complete layout of the circuit is shown in figure 4.

VI. SOFTWARE APPLICATION

The software application was developed using Microsoft Visual C++ 5.0 and is intended to run on Windows 95/98/NT operating systems. It provides full access to the hardware and software renderers. Additionally it offers the following functionality:

- Create, open and save volume files
- Open and save colormaps
- Select render type
- Generate a test volume (sphere with custom radius and voxel values)
- Data analysis (data histogram)
- Render error analysis (error versus precision bits)
- Edit colormap
- Render direction, quality, bounding box and scale.

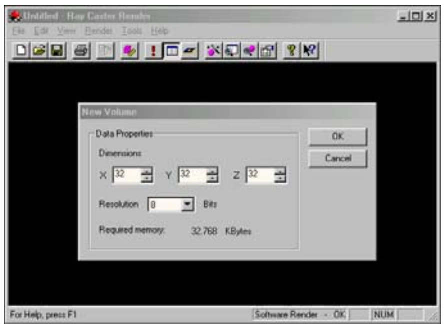
The application architecture is full object oriented. In addition to interface classes, the following classes were developed in order to simplify and increase access safety to the corresponding objects: CData, CRenderImage, CMapTable, CRender, CSoftRender and CHardRender. Together, they provide the application with important characteristics such as flexibility and extensibility. It is also possible to reuse them in other applications. The names are clear enough to justify their purposes. The CRender class serves as a base class for CSoftRender and CHardRender. It provides methods and encapsulates properties, which are common to software and hardware render objects. Additionally it allows the use of polymorphism [11, 12].

An interesting comment is related to the memory allocation inside class CData. Initially, the allocation was performed using arrays of pointers, where each pointer was used to allocate an array of voxels within the same voxel line. However, during program execution, we verified that for large data volumes, this operation takes too much time. Because Windows is a virtual memory operating system, we have decided to change the memory allocation strategy. Therefore, in the actual implementation all memory required to store the volume data is allocated in one step as a "large" array of voxels. This improves the allocation, deallocation and traverse times. To achieve the best program performance is recommended to fix the minimum swap file size to a large value (>50 Mbytes, depending on the volume size and running programs). Figure 5 displays the application main window. After startup the default volume size is 32×32×32. The maximum volume size allowed by the application is 256×256×256 voxels with a 16 bits resolution corresponding to a maximum of 33,5 Mbytes.

A. Functionality

The menu and toolbar placed on its top offers the following functionality:

- **File**
  - New – Creates a new volume (figure 5). The user can select the volume size and resolution.
  - Open – Opens a previously saved volume data file (*.vdt). The accepted file format is binary and specific of this application.
  - Save – Saves the current volume data into a "vdt" file.
  - Save As – Saves the current volume data into a "vdt" file under a new name.
  - Open Map Table – Opens a previously saved map table, checking if the table to be read is compatible with the current resolution.
  - Save Map Table – Saves the current map table into a file.


Untitled - Ray Caster Render
File Edit View Render Tools Help
OK
Cancel

New Volume

Data Properties

Dimensions

X 32 Y 32 Z 32

Resolution 8 Bits

Required memory: 32.768 KBytes

For Help, press F1 | Software Render - OK | NUM

Figure 5 - Application main window with the new volume dialog box opened.

- Import Data – Opens a raw data file. The user should select the volume size and resolution. The file can have a variable length header and data can be stored in big endian or little endian formats.
- Print/Print Preview/Print Setup – Standard print commands.
- Exit – Exits the application.

- **View**
- Toolbar / Status Bar - Shows / hides the toolbar / status bar.
- Data Properties – Displays volume size and resolution information.

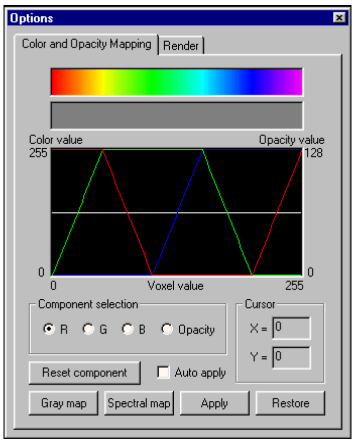
- **Render**
- Start - Starts the render process.
- Software - Switches to software render.
- Hardware - Switches to hardware render. If an error occurs during hardware initialization, the user is notified and the selection returns to software render.

- **Tools**
- Generate Data - Generates voxel data for testing purposes. On the current release, only a sphere can be generated. The user can specify its radius, inside voxel value and outside voxel value.
- Data Analysis - Calculates and displays the data histogram. It can be useful to build a first color map draft.
- Render Analysis - Performs the analysis - render quality versus number of bits used in accumulators.
- Options - Displays the options window containing two pages: Color and opacity mapping and render

B. Selection of application and render parameters

The options window is supposed to provide a usable interface to select application and render parameters. It is divided in two tabs:

- Color and opacity map (figure 6) - This tab provides the controls necessary to build an arbitrary map table. The color and opacity map layouts are displayed on the two upper rectangles. The third rectangle (components plot) shows the relation between each color component (R, G, B, Op) and the voxel value. To modify the map, the user should first select the desired component and then specify its configuration with the mouse on the components plot. The button "Reset" allows to perform the reset of the selected component. Two pre-defined maps are available: gray and spectral. They are automatically built by pressing the respective buttons. The "Apply" button makes the displayed map available to the render module (either



Component selection: [✓ R] G B Opacity

Cursor X = 0 Y = 0

Auto apply: unchecked

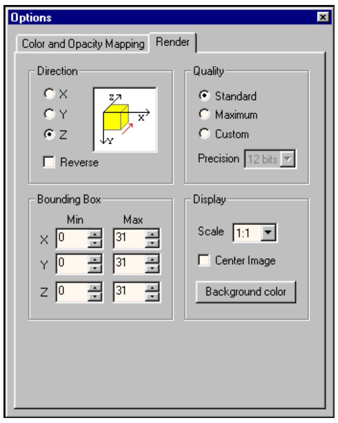
Figure 6 - Color and opacity map options page.

software or hardware). The "Restore" button replaces the displayed map by the render map, thus discarding any changes.

- Render (figure 7) - The render tab gives access to render options. The user can select one from the six render directions available: X, -X, Y, -Y, Z, -Z. Three render qualities are available: standard, maximum and custom. Only standard quality is implemented in hardware. Its precision is enough for most applications (12 bit accumulators). The maximum quality uses double precision floating point arithmetic. In this case, we assume that the error introduced by the render is negligible. The selection of custom quality activates the precision combo box, where the user can specify the number of bits to use in the accumulation process (8 – 16 bits). When the area of interest is limited to a volume section and/or the response time must be improved, it's possible to apply a bounding box. The display group allows the user to change some view-related parameters (scale, center and background color).

C. Error analysis

For performing a render analysis it is assumed that the maximum quality is achieved using double precision floating point arithmetic. After rendering, the resulting image is stored in memory for comparison purposes. Next, a render is performed for each value of



Direction: X Y [✓ Z]

Reverse: unchecked

Quality: [✓ Standard] Maximum Custom

Precision 12 bits

Bounding Box

	Min	Max
X	0	31
Y	0	31
Z	0	31

Display

Scale 1:1

Center Image: unchecked

Figure 7 - Render options page.

accumulation bits. All the resulting images are compared with the maximum quality image. At the end of the process a dialog box displays the results. No intermediate images appear on the screen. The expressions used to calculate the render error are the following:

$$I = \sum_{i=0}^{M-1}\sum_{j=0}^{N-1}\sqrt{\left(R_{ij}^{M}\right)^2 + \left(G_{ij}^{M}\right)^2 + \left(B_{ij}^{M}\right)^2}$$

$$E = \sum_{i=0}^{N-1}\sum_{j=0}^{M-1}\sqrt{\left(R_{ij}^{M} - R_{ij}\right)^2 + \left(G_{ij}^{M} - G_{ij}\right)^2 + \left(B_{ij}^{M} - B_{ij}\right)^2}$$

$$Er = \frac{E}{M * N * I}$$

where:

M * N – spatial resolution of the image;

$R_{ij}^{M}, G_{ij}^{M}, B_{ij}^{M}$ - values of the components of each pixel in the maximum quality image;

R_{ij}, G_{ij}, B_{ij} - values of the components of each pixel for a parameterizable quality image;

I - measure of the total image intensity;

E - value of the absolute error of a given image compared with the maximum quality image;

Er - value of the relative error, per pixel and intensity independent (corresponds to the absolute error normalized by the total number of pixels and image intensity.

The values obtained depend on the current color and opacity map configuration and render settings (direction and bounding box). For a given application it is necessary to vary these parameters and perform several render analyses in order to determine the optimum accumulator size.

VII. RESULTS

This section presents the results of some tests used to measure the performance of the developed coprocessor and to compare it to some software solutions. Three different platforms were used:

- A Personal Computer equipped with a Pentium II processor, 128 Mbytes of SDRAM and operating at 233 MHz - PII233 - (software render);
- A Computer equipped with a Pentium processor, 32 Mbytes of DRAM and operating at 120 MHz - P120 - (software render);
- The developed hardware coprocessor connected to the previous computer - HWonP120 - (hardware render).

All experiments comprised the following steps:

1. Generate a new volume (File->New + Tools->Generate Data)
2. Build an appropriate color map (View->Options)
3. Select the desired render (Render->Software/Hardware)
4. Start rendering and measure the time.

The volume size used was 128*128*128 voxels. Table 1 displays the results obtained.

Only one volume size is presented. However, the render time varies linearly with the size.

Direction	Quality	Time(sec)		
		PII233	P120	HwonP120
X	Standard	1.1	2.9	1.7
	Maximum	1.4	3.0	Na
Y	Standard	1.1	2.5	1.5
	Maximum	1.4	2.7	Na
Z	Standard	0.7	2	1.4
	Maximum	0.9	2.3	Na

Table 1 - Rendering times for a volume size of 128*128*128 voxels.

Custom quality results are not displayed since this feature is provided for error analysis only.

During the experiments, we observed the following:

- As expected, the render speed is independent of the reverse parameter (render direction);
- The scale factor doesn't affect the overall performance. This parameter is used only at the end of the render process by the Windows GDI functions. The practical results demonstrate that the overhead generated by these functions is practically zero and the final image quality seems acceptable.

Analyzing Table 1 it is possible to draw the following conclusions:

- The ray caster algorithm runs faster (about 40%) on the developed hardware coprocessor than on the Pentium 120 computer. However, the same is not true if we compare to the Pentium II computer. Actually the last comparison is more reasonable today;
- On a Pentium II it is possible to run interactively the algorithm with a volume size of 128×128×128 or equivalent;
- The algorithm treats all render directions in the same way. Thus, the performance differences are caused by an external parameter. This parameter is the cache-hit rate. While in Z direction the voxels used to calculate a given pixel are always placed in adjacent memory positions, the same is not true for X and Y directions. In Y direction, the difference between adjacent voxels is "z dimension". In X direction this value is "y dimension × z dimension". This contributes for a bad data locality and consequently to a smaller cache-hit rate. The difference is not obvious on a Pentium II processor due to its large internal cache.

Figure 8 shows the concatenation of some rendered images. They correspond to a sphere generated with the following parameters:

- Volume size - 128×128×128;
- Resolution - 16 bits;
- Sphere radius - 64;
- Inside voxel value - 60000;
- Outside voxel value - 10000;

Only image a) presents a visible low quality. This is caused by the low precision of the accumulators (8 bits). The visual quality of all remaining pictures is very difficult to distinguish. The render analysis graphic confirms that beyond 12 bits precision (standard quality) the render error is practically zero (<0.5% with 12 bits precision).

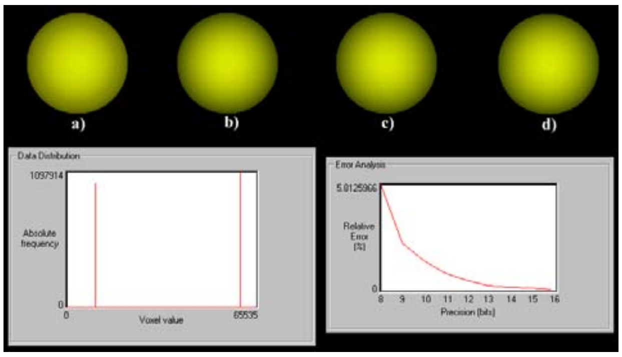


Figure 8 - Examples of rendered images - a) Custom 8 bit; b) Standard; c) Custom 16 bit; d) Maximum (floating point in s/w).

VIII. Conclusions

The results obtained during this work, and presented in the last section, prove the feasibility of implementing the ray caster algorithm in hardware. However, to be considered a real coprocessor, the hardware must process a given volume data set at higher performance than the available by software solutions. As referred in the previous section, it was not possible to achieve this goal when we compared the hardware performance to the performance of the last generation of Pentium II processors. The bottleneck is the mechanism used to transfer the voxel data from the host memory to the coprocessor board. Even using an optimized data traverse algorithm, which is independent on the render direction (but always orthogonal to two axes), the render time varies with the selected direction.

An interesting point to be addressed in future work is the implementation of additional hardware components that allow the data to be transferred to the coprocessor independently on the render direction. This approach is more appropriate to implement in hardware since the voxels are always processed sequentially, making possible the use of efficient data transfer mechanisms. A requirement of this approach is the storage of the rendered image in local memory and its complete transfer to the host memory at the end of the render process. In the current implementation, each pixel is transferred individually after its final value has been calculated. The memory available on the development system allows the simultaneous storage of a map table with 64K entries (16 bit resolution) of 32 bits each (256 Kbytes) and an image of 256×256 pixels of 32 bits (R, G, B, Op) (256 Kbytes). A complexity that results from this approach is the need to multiplex the memory access. A real coprocessor must have two independent memory banks, one for color mapping and the other for image storing.

The improvement in the data transfer mechanism will also demand faster components in the datapath. The architecture of the used FPGA makes very difficult the implementation of carry look-ahead circuits due to the unavailability of gate primitives with 3, 4, … inputs. Consequently, to improve the performance of accumulators and subtractor the recommended scheme is based on carry-select circuits [13].

An improved design must also explore a better multiplier structure, instead of direct implementation of the algorithm. In [14] several multiplier schemes are proposed; the high performance requirements force a parallel implementation, however, is necessary to perform a study in order to adapt the scheme to the FPGA architecture.

Another important topic is the required precision in the accumulators and multipliers. For some applications, 12 bits can be enough to obtain a good quality image, however, depending on the color and opacity values used in the map table, it could be necessary to implement a 16 bits datapath to obtain full precision. The number of cells available in the used FPGA allows a maximum of a 14 bits datapath. To implement a 16 bit datapath is necessary to use a development system with a larger FPGA such as a XC6236 or XC6264 also from Xilinx.

Finally, the presented approach and the proposed modifications don't use the dynamic reconfiguration feature available in the used FPGA (introduced in section III). Thus it is possible to use another FPGA family such as the Xilinx XC4000 [6], which allows the implementation of more complex designs with a lower cost and better performances.

IX. References

[1] Watt, A., *3D Computer Graphics*, 2nd Ed. Addison Wesley, 1994.

[2] Elvis, T., "A survey of Algorithms for Volume Visualization", *Computer Graphics*, Vol. 26, N. 3, August, 1992, pp. 194-201

[3] Brodlie, K., L. Carpenter, R. Earnshaw, J. Gallop, R. Hubbold, A. Mumford, C. Osland, P. Quarendon, *Scientific Visualization, Techniques and Applications*, Springer Verlag , 1992.

[4] Xilinx, *XC6200 Development System*, Preliminary Datasheet, February 1997.

[5] Xilinx, *XC6200 Field Programmable Gate Arrays*, Product Description, (http://www.xilinx.com/partinfo/6200.pdf), April 1997.

[6] Xilinx, *The Programmable Logic Data Book*, 1996.

[7] Ashenden, P., *The designer's guide to VHDL*, Morgan Kaufmann, 1996.

[8] Xilinx, *Velab: VHDL Elaborator for XC6200 (V0.52)*, http://www.xilinx.com/apps/velabrel.htm.

[9] Xilinx, *Series 6000 User Guide*, 1997.

[10] Sklyarov, V., Monteiro, R. S., Lau, N., Melo, A., Oliveira, A., Kondratjuk, K., "Integrated Development for Logic Synthesis Based on Dynamically Reconfigurable FPGAs", *8th International Workshop, FPL'98*, September 1998.

[11] Stroustrup, B., *The C++ Programming Language*, Second Edition, Addison-Wesley Publishing Company, 1995.

[12] Prosise, J., *Programming Windows 95 with MFC*, Microsoft Press, 1996.

[13] Katz, R., *Contemporary Logic Design*, The Benjamin/Cummings Publishing Company, Inc., 1995.

[14] Cavanagh, J., *Digital Computer Arithmetic*, McGraw-Hill, Inc., 1984.