Graphic card report

A Design Project Report

Presented to the Engineering Division of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Engineering (Electrical)

by

Noé Girand

Project Advisor: Bruce Land

Degree Date: May 2010

Abstract

Master of Electrical Engineering Program

Cornell University

Design Project Report

Project Title: Graphic card

Author: Girand Noé

Abstract:

Computer graphics has become more and more popular. Starting from representing 2D sprites by computing basic geometry (points, rectangles, triangles), it can now represent complex 3D scenes with different kind of lights, with explosions, fluids, mirrors… The writing of graphics programs for developers has become easy, thanks to the combination of:

- Easy-to-use API such as DirectX and OpenGL which facilitates the communication with the hardware
- Easy-to-use graphic cards, which make most of the computation (Back-buffering, lightening computation, Z-comparison…), which unloads this part from the program.

The objective of this project was to implement these both sides (software and hardware) in order to provide a final easy-to-use API in C for a NIOS processor to run on a DE2 board. This project was an extension of course ECE5760 of Bruce Land, in which I implemented some parts of the graphic card, in a static way.

These implementations faced some issues:

- Synchronization between the NIOS, the Graphic Card and the VGA controller;
- Performances, at different levels:
    - Rapidity
    - Low-hardware consumption
    - Flexibility
- Efficient management of memory, specifically the SRAM which is used by both the graphic card and the VGA controller;

The goal of these implementations was to implement common techniques of computer graphics:

- For the hardware (Graphic card), in Verilog:
    - Triangle rasterization
    - Linear extrapolation
    - Z-Buffering
    - Light computation
- For the software (API), in C:
    - Communication with the graphic card
    - Geometry structures to facilitate the manipulation of data
    - Matrix structures to compute transformations.

Report Approved by

Project Advisor:_____**Date:**_____

Executive summary

Computer graphics has become more and more popular. Starting from representing 2D sprites by computing basic geometry (points, rectangles, triangles), it can now represent complex 3D scenes with different kind of lights, with explosions, fluids, mirrors… The writing of graphics programs for developers has become easy, thanks to the combination of:

- Easy-to-use API such as DirectX and OpenGL which facilitates the communication with the hardware
- Easy-to-use graphic cards, which make most of the computation (Back-buffering, lightening computation, Z-comparison…), which unloads this part from the program.

The objective of this project was to design a solid global architecture for a graphic system with the common possibilities enumerated above. This needed to set up a logical hierarchy of components, as some robust communicating protocols for the communication of the high-level components (NIOS processor, VGA controller, Graphic card).

I have successfully implemented in hardware the main functionalities one expects from a graphic card: drawing triangle, z-buffering, basic (directional) lighting, double-buffering.

I have implemented a global architecture for managing the VGA controller, the NIOS and the graphic card, but some synchronization failures remain. There are detailed in the last part of this report.

The user downloads the FPGA design to the Cyclone II on a DE2 Altera board.

Then, he can easily write a complex graphic software application in C, using my easy-to-use library to draw to screen. He can then compiles it and executes it on the NIOS which is part of the FPGA project.

I have a demo running an application which builds a 293 points and 500 triangles rabbit, and renders it in 3D with light. The user is able to rotate the rabbit and modify rendering options by interacting with the buttons on the FPGA board.

Design problem and system requirements

The objective of this project was to design a solid global architecture for a graphic system with common 3D functionalities. The pipeline would encapsulate from the high-level API functions in software to the VGA display in hardware.

. This notably implied to have a logical hierarchy of components to make the project easy to understand, to facilitate debugging, and to understand the dependencies, the timing limits. I also needed to implement some robust communicating protocols for the communication of different components (NIOS processor, VGA controller, Graphic card).

There were important issues when elaborating the components. Rapidity was the most important consideration. Several points are to be considered in order to achieve this goal. First, the graphic card should be able to draw one new point in one (or two if including z-testing) clock cycle, if the memory responds in one clock cycle. Then, the synchronization between the different components should minimize the number of clock cycles. At last, the global architecture should optimize the access to the shared memory (SRAM) from the Graphic card and VGA controller, in the sense of minimizing the time in which neither uses it, but making sure that they do not overlap

Another goal was to have low hardware consumption. If the users want to add components to the project for other purposes, it is important to use fewer registers and multipliers as possible. Eventually, I also wanted to leave a lot of flexibility to the user. In order to ensure reusability, it is important to keep many options for the user, and use parameters at maximum. The components should be logical to be understandable.

Hardware implementation

At the hardware level, the objective was to implement a triangle pipeline including rasterization, lighting and z-buffering.

The first objective was triangle rasterization : how to fill a triangle so that any point inside it has a color and deepness in accordance with the distance from the extremities. This algorithm can be reduces to linear extrapolation (in 1D) : given two extremities of a segment, how to fill the points inside this segment in accordance to their distance to the two extremities.

In order to be realistic, results should appear as in 3D. Thus, I also had to implement the z-buffering technique, which ensures that geometries (and pixels) do not appear on the screen in the order in which they are drawn, but according to their z coordinate which are specified by the user.

There are two ways of filling a triangle. One can specify the colors at the vertices and use the rasterization technique we mentioned above. Or one can specify a single color for the triangle and a normal, as well as a light color and a light normal. The appearing color of the triangle is that of its material color affected by the light color and the directions in a more realistic way according to physics.

Software implementations

I had also some precise expectations from the software. The most important was to set up a library to communication with the graphic card. The user shoud be able to use high-level functions (like DrawTriangle, DrawTriangleWithMaterial, setLight…) which encapsulate the communication with the NIOS.

I also wanted to set up some geometry structures to facilitate the manipulation of data. Indeed, the data to describe geometry is very common to many applications: a scene is usually a light plus a collection of objects which are a collection of vertices, triangles with their normals and a material. Users want to use floating-points data to describe the geometry easily, instead of the format of the wires.

At last, I also wanted to implement some mathematical transformations using matrices, as this is commonly used in computer graphics, like projections, user view, rotations, translations…

Limitations

There were some critical limitations for my work. The main one was that the design of my architecture was completely based on the DE2 board. The main restriction this implied was the memory: the SRAM is only 512kBytes, which is 32 bits per pixel for a screen of 512*256 I used. This also means that all the SRAM was dedicated to the screen. I was also limited by the number of multipliers. This notably prevented me from doing 3D graphics computation (world, view, transformation) in hardware in parallel.

Previous work

Some other similar projects were realized, for ECE5760 for example. The objective was to find and solve the problems on my own and I did not inspire from their work. These projects can be found at website:

- http://instruct1.cit.cornell.edu/courses/ece576/FinalProjects/f2008/ap328_sjp45/website/introduction.htm
- http://instruct1.cit.cornell.edu/courses/ece576/FinalProjects/f2007/jas328_asl45/jas328_asl45/index.htm

Some of them were hardware only, and were not designed to host a user application on a NIOS.

System requirements

The components I used were:

- A standard VGA screen
- A DE2 board from Altera
- A computer with Eclipse (with NIOS IDE, notably the compiler) and Quartus II.

The graphic card is designed in Verilog (using the Quartus II software). Verilog permits to design a hardware circuit with elementary components (like wires, logical gates, registers and multipliers), but at a higher level of abstraction. The source code in Verilog describes modules organized hierarchically. In these modules, the values of variables and connections can be described (set or computed) “easily”. From this source code, Quartus II can generate a FPGA project file which only contains the elementary components. This project can then be exported to a FPGA that build the required circuit.

The graphic card module can be used on any kind of FPGA, but it has been designed to work with a 512*512 memory cells of 16 bits. The clock cycle can be any value tolerated by the memory used (the memory must answer a request in one cycle). In my case, I used the SRAM from the DE2.

The top cell of the project, which includes notably one graphic card, one NIOS and one VGA controller (designed by Bruce Land) is specific to the DE2 of Altera.

The DE2 Altera board contains the Cyclone II FPGA (which contains the graphic card, the VGA controller and the NIOS), a SRAM (contains the data to be displayed on screen, computed by the graphic card), a SDRAM (the data for the NIOS), some buttons input, and other kind of outputs (lights), and other components I did not use.

My software library has been written in C, and has been compiled through gcc to the NIOS instruction set. The user application should write its program in C to use it. He can then download his application to the NIOS to run it.

Description of solution

Global view

This is the global view about how data information about graphic geometry is translated into screen.

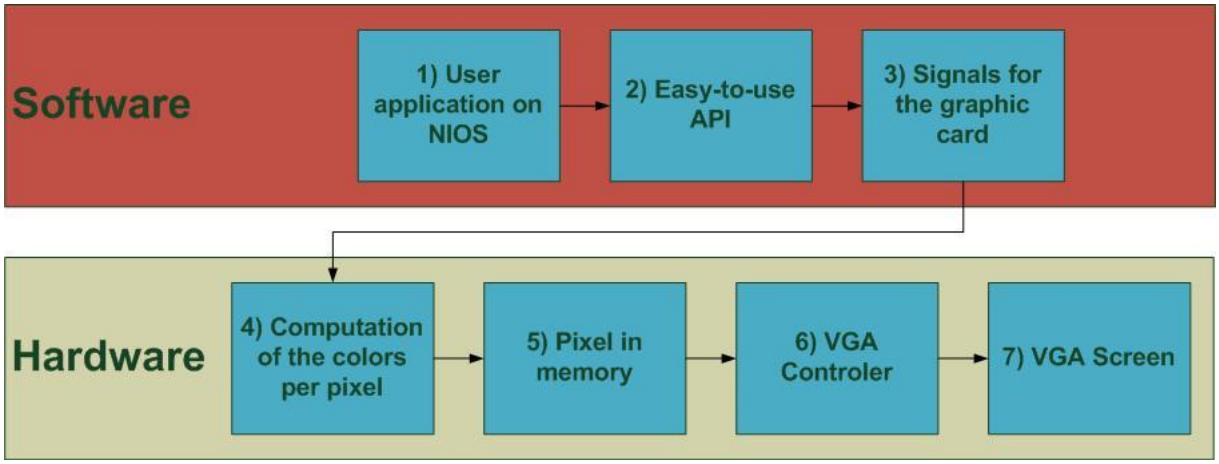


Fig 1: the chain of data transfer from user application to screen

1) Developers who write software applications customize their application on their own needs, by modeling their graphics objects as they want (cube, or rabbit).

2) These applications use a standardized easy-to-use library which will provide basic functions to describe the graphic information, by setting triangles, materials, and lights. Transformations are also set up at this step (local transformations of an object in scene, transformation due to the point of view from user and transformation due to 3D - projection)

3) These functions will translate this information into signals which communicate with the graphic card.

4) The graphic card uses the information from the signal to compute the per-pixel data.

5) The per-pixel data is stored into a memory.

6) The VGA controller reads the information in the memory, and transforms this information into signals to the VGA display.

7) The data is displayed on the screen.

As we can see on the diagram above, the project was made of a hardware part and a software part. The two parts are detailed, beginning with hardware, because this is the most important.

Hardware

This section describes all the hardware of the project.

Overall design

There are some common protocols for all components which need to communicate. When the communication is direct (when the components are on the same clock signal, and they cannot be disturbed by other components), the low-level component has a ready output signal to indicate that it is on idle state, waiting for order. The higher level component has a start signal which can be enable only if host component is ready. If so, start is enable for one cycle. Because they are on the same clock signal, the component cannot miss the signal.

Many signals are common in the components. These signals are:

- input clock
- input n_reset: if 0, resets the component
- input n_pause: if 0, pauses the component
- input start
- output ready
- Many debug signals, which begin with a d_ followed by the name of the wire / register they are connected to. When debug signals are coming from a lower-level component, the current component adds the name of this lower-level component in the name of the signal. For example, from Triangle Renderer, the debug state signal of its top line triangle render is named d_tr_top_ltr_state

Furthermore, all components are ruled by a state machine, whose register is named “state”.

In conclusion: most of components share the following skeleton code:

```
Module module_name(
    input clock,
    input n_reset,
    input n_pause,
    input start,
    output ready,
    // Component specific signals
    // Debug signals
    ,Output d_state
);
    reg state;
    assign ready = (state == stIdle)
    assign d_state = state;
    always@(posedge clock)begin
```

```
if(n_reset)begin
        if(n_pause)begin
                switch(state)….
        end
end else begin
        state <= stIdle;
end
    end
endmodule
```

In order to understand their hierarchy, the components are described from high-level to low-level.

Top module (see source file in appendix : DE2_TOP.v)

My top cell instantiates one NIOS, one instance of a graphic card component and one VGA controller notably (but also many wires to connect to other components of the NIOS, like the key inputs, the memories…).

The graphic card and the VGA controller are both using the memory (SRAM), one to set the data and the other to read it. Thus, this is an overall view of the hardware of top-level components.

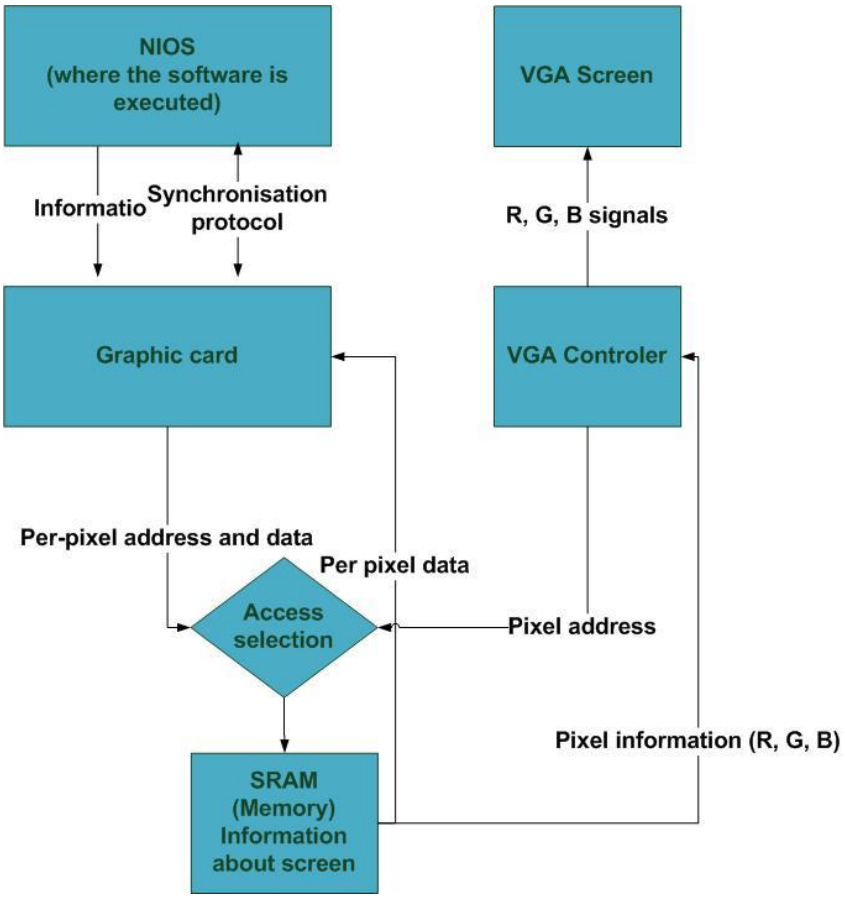


Figure 2 : the blocks detail of the top cell

The architecture of the graphic card is detailed later, in page 12.

The SRAM stores the information about the screen. Each pixel of the 512*256 available pixels occupies two consecutive cells of 16 bits each (one buffer for each cell). The repartition of the information is set up by parameters stored in the “headers” file. By default, they are set up so that the R, G, B, and Z channels occupy 4 bits each, in this order. One can change these parameters and rebuild the project.

Communication protocol

Because the VGA Controller and the Graphic card are both competing for the SRAM (the VGA Controller to read the data to display it to the screen and the graphic card to fill it), I have two signals to control which one is enabled. The finals wires which connect to the SRAM are connected to the adequate component according to these controls signals. The VGA controller has priority: it is enabled when the screen wants to draw a pixel fitting in the right range (the rectangle of 512 * 256 in the middle of the screen). I wanted the Graphic Card working on the highest frequency as possible. But as I could not have it working on the 27Mhz clock, I had set it to the same signal clock as the VGA Controller (see the conclusion).

The Graphic card and the NIOS must also communicate. I tried to implement a robust protocol to have them communicating, but this protocol can miss sometimes (again see conclusion).

The graphic card keeps a record of the parity of the last order received, and the NIOS that of the last order emitted. At “idle” regime, these numbers (0 or 1) match. When the NIOS has its entire inputs ready, and when the graphic card is ready, the NIOS switches its number. When the graphic card receives it, it becomes unavailable and change its number to announce that it received the order and is performing it. The NIOS can then compute the new outputs for future order, while waiting graphic card becomes ready. This is a two-level handshaking protocol.

This is a summary of the communication protocol, on the left the NIOS side (in software), and on the right the graphic card (in hardware).

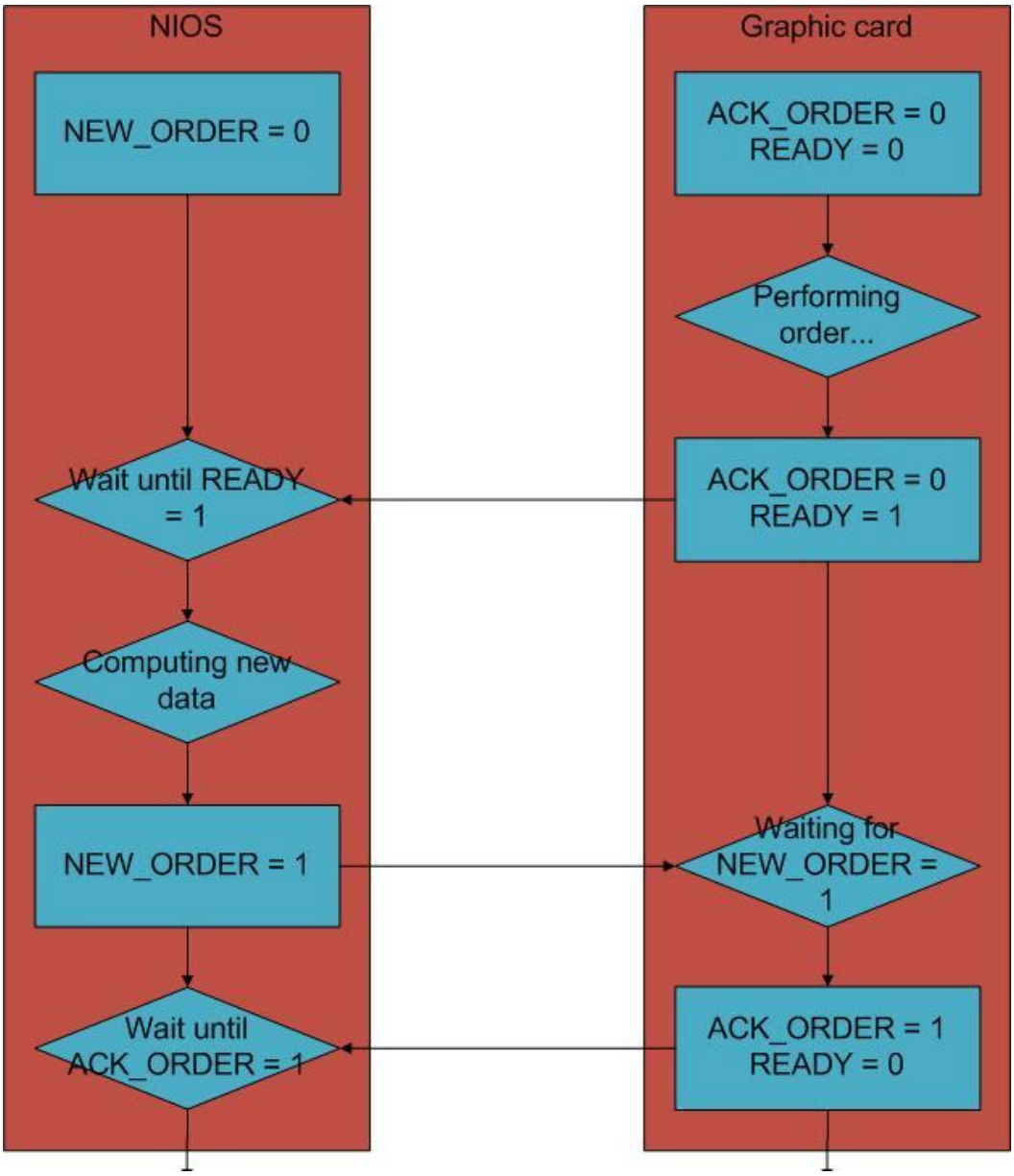


Figure 3 : The communication NIOS / Graphic card

Graphic card (see source file in appendix : GraphicCard.v)

This is the top-level component of my project. It may receive an order when it is not working, that means, when it is not already completing an order: switching buffer, clearing current buffer, drawing a triangle. Its clock signal must be set so that the memory should be able to answer in one clock cycle.

Specific inputs

- Communicating wires (order, new_order, ack_order)
- Data in memory at current position (in_data), to be used for the z-checking
- Graphic information, when ordering a new triangle draw (3 points, light, and material information)
- Rendering options (z_enable, light_enable, double_face_enable, double_buffer_enable)

Specific outputs

- Information to write in memory (write, address, data)

Description

The goal of the graphic card is to implement the communicating protocol with the extern world, to select the required component, and to manage the current buffer it is working on.

The graphic card encapsulates two components to respond to a specific order it may receive: one Clearer and one Memory Triangle Controller.

The graphic card has a state register, which is at Idle by default. When a new order is received, the graphic card switches to a state indicating that it is enabling a specific component (the clearer, or the Memory Triangle Controller), and connects its some of its wires (like the output wires of memory) to that of the component, until this component becomes ready (which means it completed its work).

Below is the representation of the hierarchy of the graphic card.

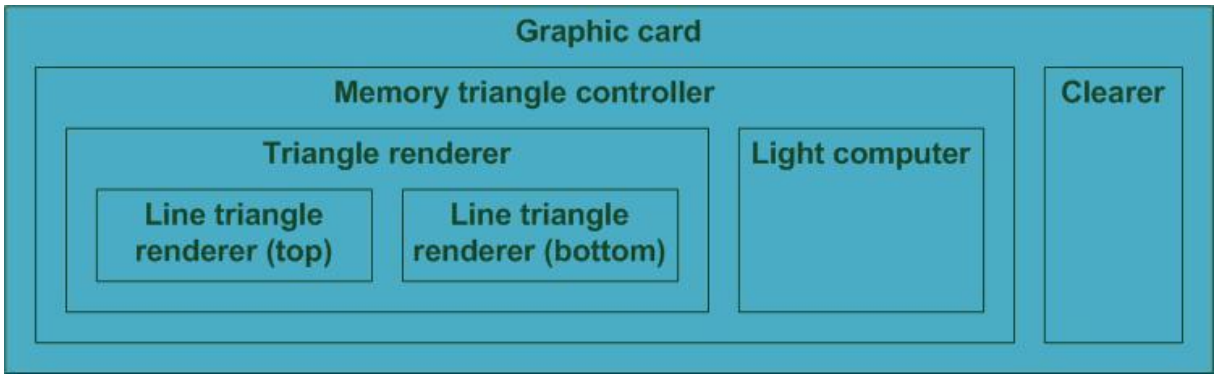


Figure 4 : Hierarchy of the graphic card

Clearer (see source file in appendix : Clearer.v)

This component clears one buffer (filling it with zero).

Specific inputs

- current_buffer: the buffer to clear

Specific outputs

- Three outputs to communicate with the memory (write, address, data)

Description

This component contains one register “current_addr”, to store the address it is currently clearing. When a new order is passed, on each following clock timing, clearer iterates on all the locations of the buffer until it completes.

Memory triangle controller

(see source file in appendix : MemoryTriangleController.v)

This is the component which is responsible to fill memory on a triangle.

Specific inputs

- Three coordinates and three color information for the three points which represent the triangle
- Rendering options : z_enable, light_enable, double_light_enable

- Light and material colors and normals
- current_buffer : which buffer to use if double buffering is used
- z_data : the z-value of the current point where the component should draw a new point

Specific outputs

- write, addr, data : to write some data at a specific address (in SRAM)

Description

Memory triangle controller instantiates one triangle renderer component (which is responsible to list the points inside the triangle) and one light-computer component. It has four different states:

- Idle
- Initialize drawing : to launch the triangle renderer
- Drawing: Waiting for a new point from Triangle renderer. When a new point is computer, is z-checking is enabled, then it set in the address that of the new point to receive its z and switches to GetZWriteData. Otherwise, if z-checking is not set, write data directly. If triangle renderer is done, it moves to Idle Mode.
- GetZWriteData: The component receives the z information about the pixel in the Z-buffer. If the new pixel to be drawn is in front, it overrides the previous one.

Light computer (see source file in appendix : LightComputer.v)

This is the component responsible to compute the illumination of a material. There are neither clock signals nor registers in this component, all the computation is done directly through wires.

Specific inputs

- Light and material colors and normal directions
- A signal to indicate that double-face is enabled. If this is the case, the normals are not oriented. Otherwise, if the material is illuminated from behind, it is rendered as black.

Description

The normals are represented as a 3 fixed point vector which is normalized. The first bit is a sign bit (0 positive, 1 negative), and then follows the absolute value which represents the decimal part (not 2 complementary). Thus, if one normal is aligned with x, y, or z axis, it is not normalized properly, because in this format, the value “1” cannot be represented. Instead, it is represented as (0, 0, 0.11111111) and a little difference occurs. But because the resolution of the computation of lighting is higher than that of colors in memory, the difference is not significant.

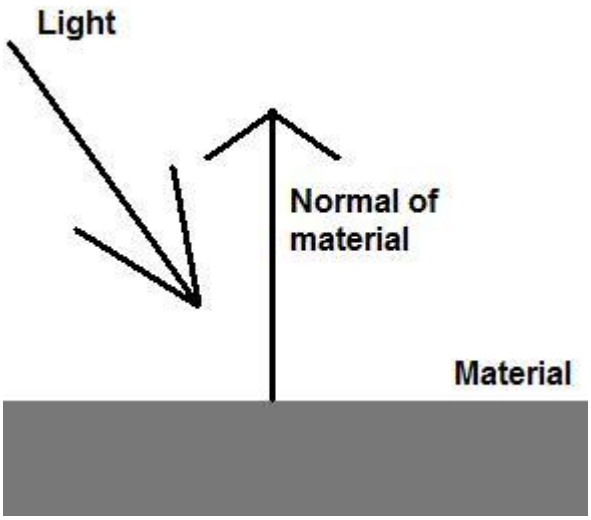


Figure 5 : Computation of light

The component computes the dot product of the normals. To do this, the component has two wires to compute the positive and negative terms of this dot product. If the result is positive, the normals were not facing (I use a different convention from the commonly used one), and the object is not illuminated. If double facing is enabled, then the light is computed with the absolute value of the dot product. Otherwise, the object is displayed as black.

Triangle renderer (see source file in appendix : TriangleRenderer.v)

Given three points inputs with three coordinates (x, y, z) and three color channels (r, g, b) triangle renderer provides the set of the extrapolated points between the two. This is where the triangle rasterization is implemented.

Specific inputs

- Three points with their coordinate (x, y)
- Four data channels (y, z, r, g, b) for each points

Specific outputs

- A point at current coordinate (x, y)
- Current data channels (z, r, g, b) for each points
- newPoint, an output to indicate higher-level component that a new current point has been computed

Description

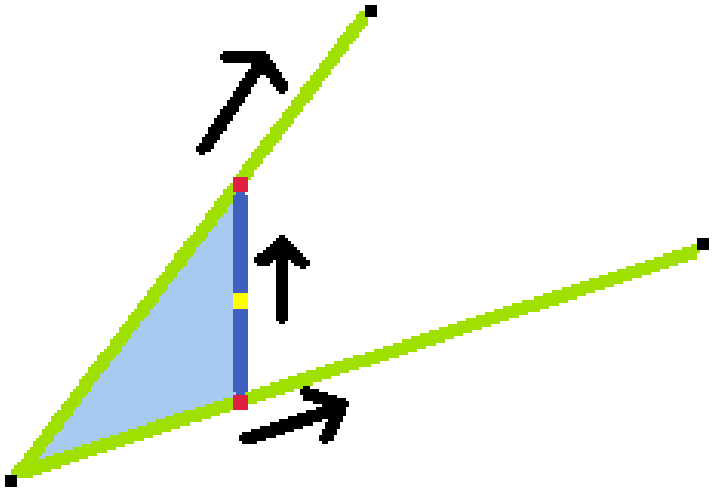


Figure 6 : Triangle rasterization

The component needs to draw a line from point to point and interpolate the data channels. This is the objective of the Line Triangle Renderer component detailed later. Triangle renderer uses two of these components: one for the top line and one for the bottom line. When they both reach a point with a new X, triangle renderer interpolates the four data channels (z, r, g, b) between the two points. This is done exactly as in Line Triangle Renderer. A pseudo-code for this component (when p1_x< p2_x < p3_x and p2 is above the line (p1; p3)) would look like this:

```
for(x = p1_x; x < p2_x; x++){
    getNewXPointFromTop(t_y, t_z, t_r, t_g, t_b);
    getNewXPointFromBottom(b_y, b_z, b_r, b_g, b_b);
    for(y = b_y; y < t_y: y++){
            z = b_z + (y – b_y) * (t_z – b_z) / (t_y – b_y)
            // Same for r, g, b…
            // We have our new point data…
    }
}
```

But an extrapolation in this way would be very expensive, because of the use of a divisor (and a multiplier). Thus, a special method is used, which is described in the last component : Line Triangle Renderer.

Triangle renderer must always order the points in a convenient way, to have p1_x <= p2_x <= p3_x. It must assign bottom / top lines accordingly whether p2 is then above line (p1; p3) or under. This is done by watching the sign of det(vect(p1 to p2); vect(p1 to p3)).

There is a register called “second” to store the information that we are in the second part of the draw, which means that we reached p2, and we switch the top or the bottom line as the diagram bellow explains:

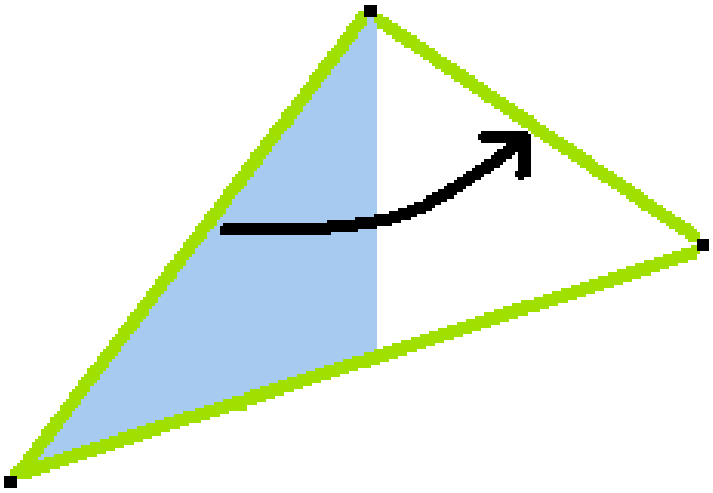


Figure 7 : Triangle renderer switching

The triangle renderer component has also to deal with situations where abscises of two or three points are aligned.

Line Triangle renderer

Given two points inputs with three coordinates (x, y, z) and three color channels (r, g, b) such as p1_x= < p2_x, line triangle renderer provides the set of the extrapolated points between the two.

Specific inputs

- Two points with their coordinate (x)
- Five data channels (y, z, r, g, b) for each points

Specific outputs

- A point at current coordinate (x)
- Current data channels (y, z, r, g, b) for each points
- newX, an output to indicate higher-level component that a new current point has been computed

Objective

- Current points should have their x differed of exactly one
- The current y data channel is right, whereas r, g, b data channels can be approximated for performance (see later for details).

Description

Should this be programmed in computer science, one would probably write some code like this one (omitting type's problems):

```
for(int x = p1_x; x <= p2_x, x++){
        int y = p1_y + (x – p1_x) * (p2_y – p1_y) / (p2_x – p1_x) ;
        int z = p1_z + (x – p1_x) * (p2_z – p1_z) / (p2_x – p1_x) ;
        …
```

```
        int b = p1_b + (x – p1_x) * (p2_b – p1_b) / (p2_x – p1_x) ;
        // We have our new point data…
}
```

But this requires the computation of 1/ (p2_x – p1_x), which is very expensive in hardware (a division), as some multiplication. Instead, we use the Breslenham algorithm (see detail at http://en.wikipedia.org/wiki/Bresenham%27s_line_algorithm) described as fellow.

Let's suppose p1_y <= p2_y, …, p1_b <= p2_b for now, just to understand the algorithm (it extends naturally to other cases).

Let's consider that the two points in red are that we want to connect (through a line in green). The points in blue are the points that are to be computed. In the algorithm, we mention the y extrapolation, but the same stands for r, g, b data channels.

The idea of the algorithm is:

Start with x = p1_x, y = p1_y.

We loop on the x, and when needed, we increase the y of 1. This works when circumstances when p2_x – p1_x >= p2_y – p1_y, otherwise we would need an average increase of y of > 1. We'll discuss how we deal with this problem later.

We have to increase the y when by increasing it, we get a point which is closer to the line. The check is computed by checking if the point at position (x, y+1/2) is above or under the line.

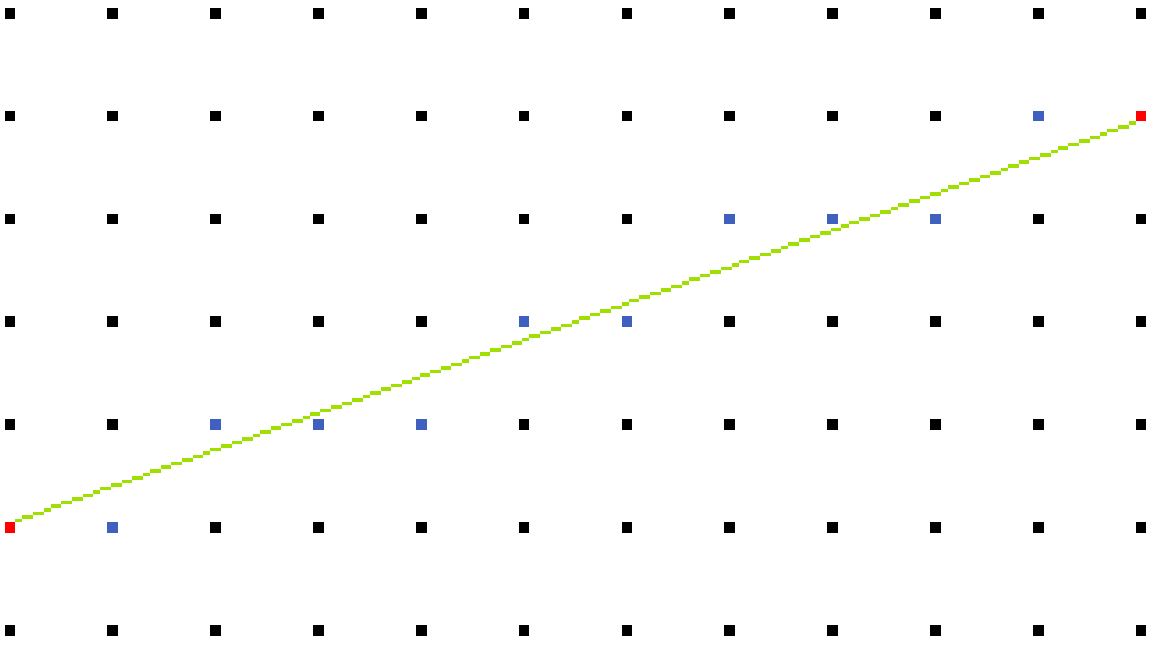


Figure 8 : Breslenham algorith

```
Int y = p1_y;
// Same for r, g, b
for(int x = p1_x; x <= p2_x, x++){
        if(vect_product(vect(x-p1_x, (y + ½) -p1_y), vect(p2_x-p1_x, p2_y-p1_y))>0)
```

```
            y++ ;
        // Same for r, g, b
        // We have our new point data
    }
```

In the implementation, the vect product is stored between each loop to have the computation of the next vect product faster, and avoids the multiplication in it (ey… registers).

In addition, I used a sign signal (sy…) to consider the case p2_y < p1_y.

This algorithm is fast (it provides one new point on each cycle) and low-consuming (no division, and multiplication by 2, which is only a shift and does not need multipliers). But it can increase the data channel of a maximum of 1 when x increases of 1.

This is acceptable for data channels which have very low resolution and because the user is little likely to render geometry where the red channel moves from 0% to 100% in a few pixels at the screen.

But this is not acceptable for the y channel, because the user is very likely to render geometries with lines which incline more than unit slope. The component would draw a line like this one:

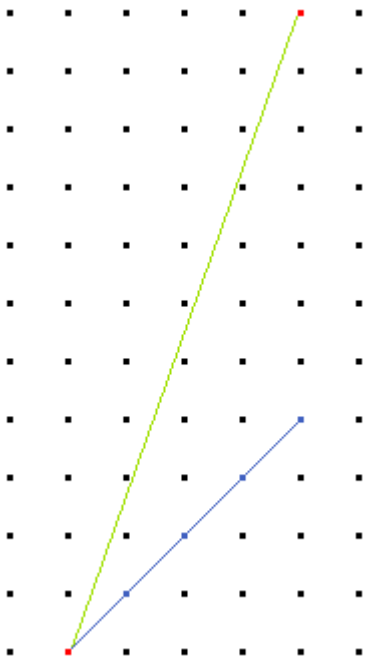


Figure 9 : Problem if dy > dx

To solve this problem, we can notice that if

p2_x – p1_x < p2_y – p1_y

p2_y – p1_y < p2_x – p1_x

We solve the problem by switching x and y.

There is one signal (coordMax) to check which coordinate to use for the loop.

HexadecimalToSevenSegments

This is an auxiliary debugging component which permits to convert a hexadecimal value (a 4-bits wire) to the 7-bits wire that are to be connected to a seven segments display to represent this number.

Software

The goal of this project was also to prove an easy-to-use API for a user application. I have written the basic functions for the NIOS to communicate with the graphic card, but also higher-level functions to facilitate the job for the user. Most of these functions were inspired from DirectX.

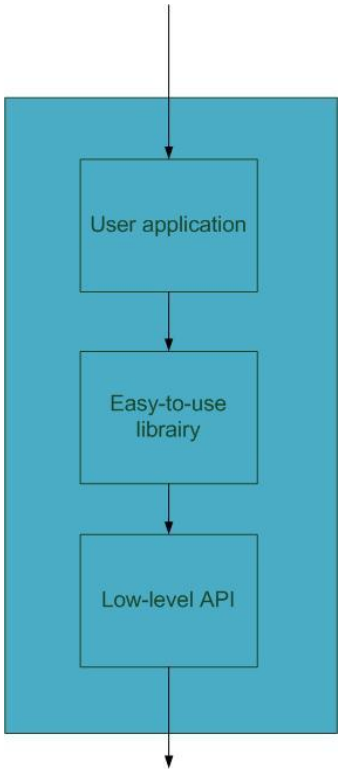


Figure 10 : Software in NIOS

Graphics.h

This file contains all the low-lever functions to communicate with the graphic card. It is mandatory.

void ResetSynch(void);

This is the first function to call before using the graphic card. It synchronizes the new_order signal to the ack_order signal from the graphic card.

void ClearScreen();

To clear the current screen buffer.

void Flip();

Displays the current buffer to screen. Moves current buffer to next buffer.

void getSW(char * SW);

To get the SW input values from the DE2 into a char array.

void getKEY(char * KEY);

To get the KEY input values from the DE2 into a char array.

void setZENABLE(char enable);

void setLIGHTENABLE(char enable);

void setDOUBLEFACEENABLE(char enable);

void setDOUBLEBUFFERENABLE(char enable);

These four functions permit to define the rending options of their name.

Geometry.h

To deal with the geometry. Should be used otherwise the user does not render triangles.

typedef struct {…} Vertex;

This is the vertex format to communicate data with the NIOS. Because the data is formatted accordingly (complicated), user should use the following functions to manipulate them.

void SetVertexColor(Vertex * v, float r, float g, float b);

Specify the color of a vertex. Color channels range from 0 to 1. In case of overflow, the values are truncated, and the results are unpredictable.

void SetVertexPos(Vertex * v, float x, float y, float z);

Specify the position of a vertex. Coordinates range from 0 to 1. In case of overflow, the values are truncated, and the results are unpredictable.

typedef struct {…} Triangle;

void InitTriangle(Triangle * t, Vertex * a, Vertex * b, Vertex * c);

Build a triangle according to the vertices.

void DrawTriangle(Triangle * t);

Draw a triangle.

Lighting.h

To deal with the lighting. Not mandatory if the user does not use lights.

typedef struct {…} Light;

This is the light format to communicate data with the NIOS. Because the data is formatted accordingly (complicated), user should use the following functions to manipulate them.

void InitLight(Light * l, float r, float g, float b, float x, float y, float z);

Buid a light with the parameters. r, g, b must range 0 to 1. x, y, z design the normalized direction of the light, must range from -1 to 1 and verify $x^2 + y^2 + z^2 = 1$.

void SetLight(Light * l);

Specify the light to be used to compute the drawing of the next objects.

typedef struct {…} Material;

void InitMaterial(Material * m, float r, float g, float b);

void SetMaterial(Material * l);

Same schema, for the material of an object.

typedef struct {…} Normal;

void InitNormal(Normal * m, float x, float y, float z);

void SetNormal(Normal * n);

Same schema for the normal of the triangle.

Because most of graphic libraries do so, I also implemented a library to deal with very common 3D transformations. Functions use is obvious according to their name.

Vector4.h

This API uses homogenous coordinates, to represent 3D vectors and 3D points. Vectors have a 4^{th} coordinate of 0, points of 3. Thus, intuitive operations (difference between two points, addition of a vector to a point…) can be operated easily. This is widely used in computer graphics, mainly because it permits to represent translations as linear operations with matrices. Using matrices for transformations permit to take benefit of hardware (massive parallel multipliers and adders in GPU of modern graphic card) which I did not implement.

typedef float Vector4[4];

void InitVector4(Vector4 v, float x, float y, float z, float w);

void Normalize(Vector4 v);

void Cross(Vector4 r, Vector4 v1, Vector4 v2);

float Dot(Vector4 v1, Vector4 v2);

void MakePoint(Vector4 v);

Normalize the vector according to the fourth coordinate (so that this coordinates becomes one).

Matrix4.h

typedef float Matrix4[4][4];

void MatIdentity(Matrix4 r);

Build an identity matrix.

void MatMultVect(Vector4 r, Vector4 p, Matrix4 m);

Multiplies a vector p by a matrix m, stores result in r.

void MatMult(Matrix4 r, Matrix4 m1, Matrix4 m2);

Same with matrices.

void MatTranslat(Matrix4 m, float x, float y, float z);

Build a translation matrix.

void MatRotX(Matrix4 m, float angle);

void MatRotY(Matrix4 m, float angle);

void MatRotZ(Matrix4 m, float angle);

Build rotations matrices around the according axis.

void MatView(Matrix4 m, Vector4 eye, Vector4 at, Vector4 up);

Build a transform matrix so that view is positioned at point “eye”, looks at points “at”, with up direction as “up”.

void MatPersp(Matrix4 m, float fov, float aspect, float znear, float zfar);

Build a perspective-projection matrix. Aspect should be 2.0f as we use 512*256 screen. Field of view should be typically pi/3. Z-near and Z far should be set as the most appropriate bounding box of the scene to optimize the z-resolution in the z-buffer.

Demo

I implemented notably a demo to have a cube rotating. The goal of this demo is to test the graphic card and measure the performances, but also to convince that the use of the API is very easy.

my_program.c

```
#include "system.h"
#include "sys/alt_irq.h"
#include "altera_avalon_timer_regs.h"
#include "altera_avalon_pio_regs.h"
#include <stdio.h>
#include <math.h>
#include "cube.h"
#include "graphics.h"
#include "matrix4.h"

int main(void)
{
   printf("Starting\n");
   // Init the graphics
   ResetSynch();

   // Init light
   Light light;
   InitLight(&light, 1.0f, 1.0f, 1.0f, 0.0f, 0.0f, 1.0f);
   SetLight(&light);

   // Init cube
   Cube cube;
   InitCube(&cube);
```

```
// 3D computation
Matrix4 rotX, rotY, trans, view, proj, viewProj, transViewProj;
Vector4 eye, at, up;
InitVector4(eye, 0.0f, 0.0f, -2.75f, 1.0f);
InitVector4(at, 0.0f, 0.0f, 0.0f, 1.0f);
InitVector4(up, 0.0f, 1.0f, 0.0f, 0.0f);
// Build matrix view
MatView(view, eye, at, up);

// Build matrix projection
MatPersp(proj, 3.14159f / 3, 2.0f, 1.0f, 5.0f);

MatMult(viewProj, view, proj);

// Rotation angle around Y axis
float alpha = 0.0f;
// Rotation angle around X axis
float beta = 0.0f;

char KEY[1]; // 4 KEY fill in one char
char SW[2];  // 18 SW fill in 3 chars

while(1){ // Infinite loop

   // Get inputs from USER
   getSW(SW);
   getKEY(KEY);

   // Set Z enable to SW0, read first bit of SW
   setZENABLE((SW[0] >> 0) & 1);

   // Set light enable to SW1, read second bit of SW
   setLIGHTENABLE((SW[0] >> 1) & 1);

   // Set double buffer enable to SW2, read third bit of SW
   setDOUBLEBUFFERENABLE((SW[0] >> 2) & 1);

   // Set double face enable to SW3, read fourth bit of SW
   setDOUBLEFACEENABLE((SW[0] >> 3) & 1);

   // Turn right if users press KEY3
   if((KEY[0] >> 3) & 1)
     alpha += 0.00314159f;

   // Turn left if users press KEY0
   if((KEY[0] >> 0) & 1)
      alpha -= 0.00314159f;

   // Turn down if users press KEY2
   if((KEY[0] >> 2) & 1)
```

```
            beta -= 0.00314159f;

        // Turn up if users press KEY1
        if((KEY[0] >> 1) & 1)
            beta += 0.00314159f;

        // Build the two rotating matrices
        MatRotX(rotX, beta);
        MatRotY(rotY, alpha);

        // Build the total world matrix
        MatMult(trans, rotX, rotY);

        // Build the total transorm matrix = trans * view * proj
        MatMult(transViewProj, trans, viewProj);

        ClearScreen();
        ComputeCubeVertices(&cube, transViewProj);
        ComputeCubeNormals(&cube, trans);
        DrawCube(&cube);
        // Switch buffers to screen
        Flip();
    }
}
```

Cube.c

```
#include "cube.h"
#include "graphics.h"
#include "geometry.h"
#include <math.h>

typedef struct {
    Vector4 points[8];
    Vertex outputPoints[8];
    Triangle triangles[8];
    Material material;
    Vector4 normals[4];
    Normal outputNormals[4];

} Cube;

void InitCube(Cube * c){
    // Best size without overflowing in the bounds
    float f = 0.8f;
    InitVector4(c->points[0], -f, -f, -f, 1.0f);
    InitVector4(c->points[1], -f, +f, -f, 1.0f);
    InitVector4(c->points[2], +f, -f, -f, 1.0f);
    InitVector4(c->points[3], +f, +f, -f, 1.0f);
    InitVector4(c->points[4], +f, -f, +f, 1.0f);
    InitVector4(c->points[5], +f, +f, +f, 1.0f);
```

```
        InitVector4(c->points[6], -f, -f, +f, 1.0f);
        InitVector4(c->points[7], -f, +f, +f, 1.0f);

        SetVertexColor(&(c->outputPoints[0]), 1, 0, 0);
        SetVertexColor(&(c->outputPoints[1]), 1, 0, 0);
        SetVertexColor(&(c->outputPoints[2]), 0, 1, 0);
        SetVertexColor(&(c->outputPoints[3]), 0, 1, 0);
        SetVertexColor(&(c->outputPoints[4]), 0, 0, 1);
        SetVertexColor(&(c->outputPoints[5]), 0, 0, 1);
        SetVertexColor(&(c->outputPoints[6]), 1, 1, 1);
        SetVertexColor(&(c->outputPoints[7]), 1, 1, 1);

        InitVector4(c->normals[0],  0.0f, 0.0f, -1.0f, 0.0f);
        InitVector4(c->normals[1],  1.0f, 0.0f,  0.0f, 0.0f);
        InitVector4(c->normals[2],  0.0f, 0.0f,  1.0f, 0.0f);
        InitVector4(c->normals[3], -1.0f, 0.0f,  0.0f, 0.0f);

        InitMaterial(&c->material, 0.0f, 1.0f, 1.0f);

        unsigned char i;
        for(i = 0; i < 8; i++){
            InitTriangle(&(c->triangles[i]), &(c->outputPoints[(i) % 8]), &(c-
>outputPoints[(i + 1) % 8]), &(c->outputPoints[(i + 2) % 8]));
        }
    }

    void ComputeCubeVertices(Cube * c, Matrix4 transViewProj){
        unsigned char i;
        Vector4 r;
        for(i = 0; i < 8; i++){
            MatMultVect(r, c->points[i], transViewProj);
            MakePoint(r);
            SetVertexPos(&(c->outputPoints[i]), r[0], r[1], r[2]);
        }
    }

    void ComputeCubeNormals(Cube * c, Matrix4 trans){
        unsigned char i;
        Vector4 r;
        for(i = 0; i < 4; i++){
            MatMultVect(r, c->normals[i], trans);
            InitNormal(&(c->outputNormals[i]), r[0], r[1], r[2]);
        }
    }

    void DrawCube(Cube * c){
        SetMaterial(&(c->material));
        unsigned char i;
        for(i = 0; i < 4; i++){
          SetNormal(&c->outputNormals[i]);
```

```
            DrawTriangle(&(c->triangles[2 * i]));
            DrawTriangle(&(c->triangles[2 * i])); // Cheating : drawing each triangle twice if one fails.

            DrawTriangle(&(c->triangles[2 * i + 1]));
            DrawTriangle(&(c->triangles[2 * i + 1]));
        }
    }
```

Results

I have a demo running an application which builds a 293 points and 500 triangles rabbit, and renders it in 3D with light. The user is able to rotate the rabbit and modify rendering options by interacting with the buttons on the FPGA board.

Some synchronization failures remain. There are detailed in the next section.

Performance

The main performance that users focus on is timing: how many triangles can be drawn per second with all the options enabled. But there is no precise number, because the time of rendering a triangle is almost proportional to its rendering size. Thus, the time to render a scene is very variable with the geometry.

Thus, I used the rabbit and an 8-triangles cube (without the top and bottom faces) as “ordinary” examples to give an idea about the performances.

Both shapes have been set to stretch over 256 * 256 pixels on screen. Performance does not depend whether the light is enabled or not. The shapes rotate to have an average about the size if the surface been displayed for each triangle, and to average about the ordering of triangle when z-buffering is enabled.

Below are the performances reached (in number of frames per minute). As one can see, it is very difficult to model it properly with the number of triangles because it mainly depends on the geometry.

Shape	# of triangles	With z-buffering enabled	Without z-buffering enabled
Sphere	760	68	71
SphereSquished (smaller triangles)	760	69	72
Rabbit	500	101	105
Cube	8	2000	2500

In order to provide an idea about how the speed evolves with triangles, I have set up a square with covers most of the screen. It is filled by n * n * 2 triangles, where n is any arbitrary given number.

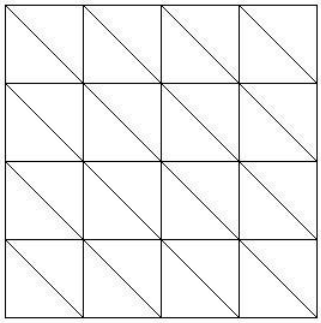
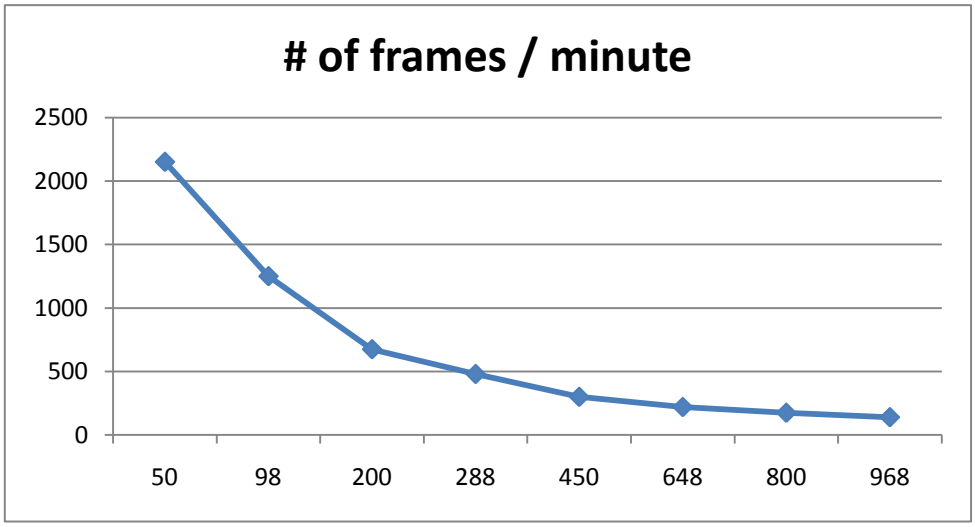


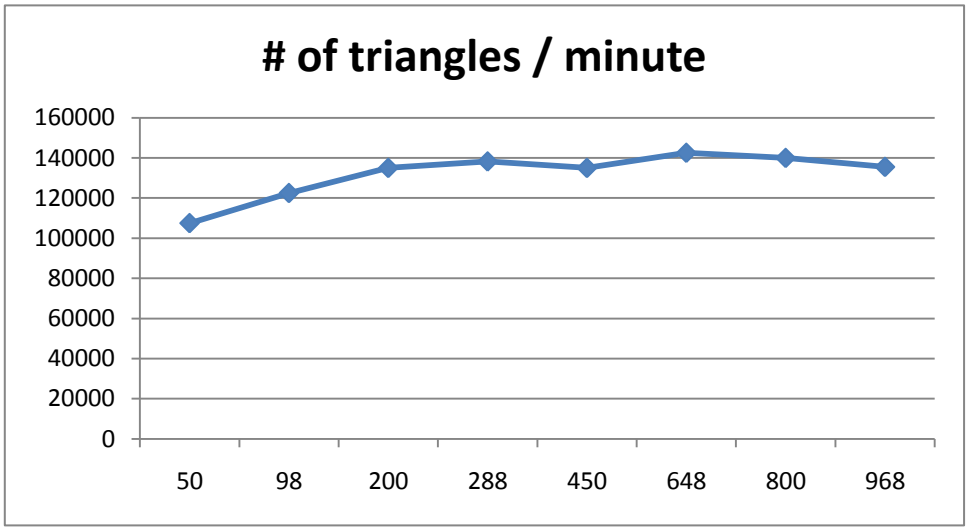
Figure 11 : Square with n = 4

By changing n, we can see how the performances evolve with the number of triangles, even if the covered area remains constant.

N	# of triangles = n * n * 2 per frame	# of frames per minute	# of triangles / frame * # of frames per minute
5	50	2150	107500
7	98	1250	122500
10	200	675	135000
12	288	480	138240
15	450	300	135000
18	648	220	142560
20	800	175	140000
22	968	140	135520

Note : the performances do not depend whether the z buffering is enabled because as this is a flat geometry, only one pixel in the geometry can be displayed in the pixel on screen.





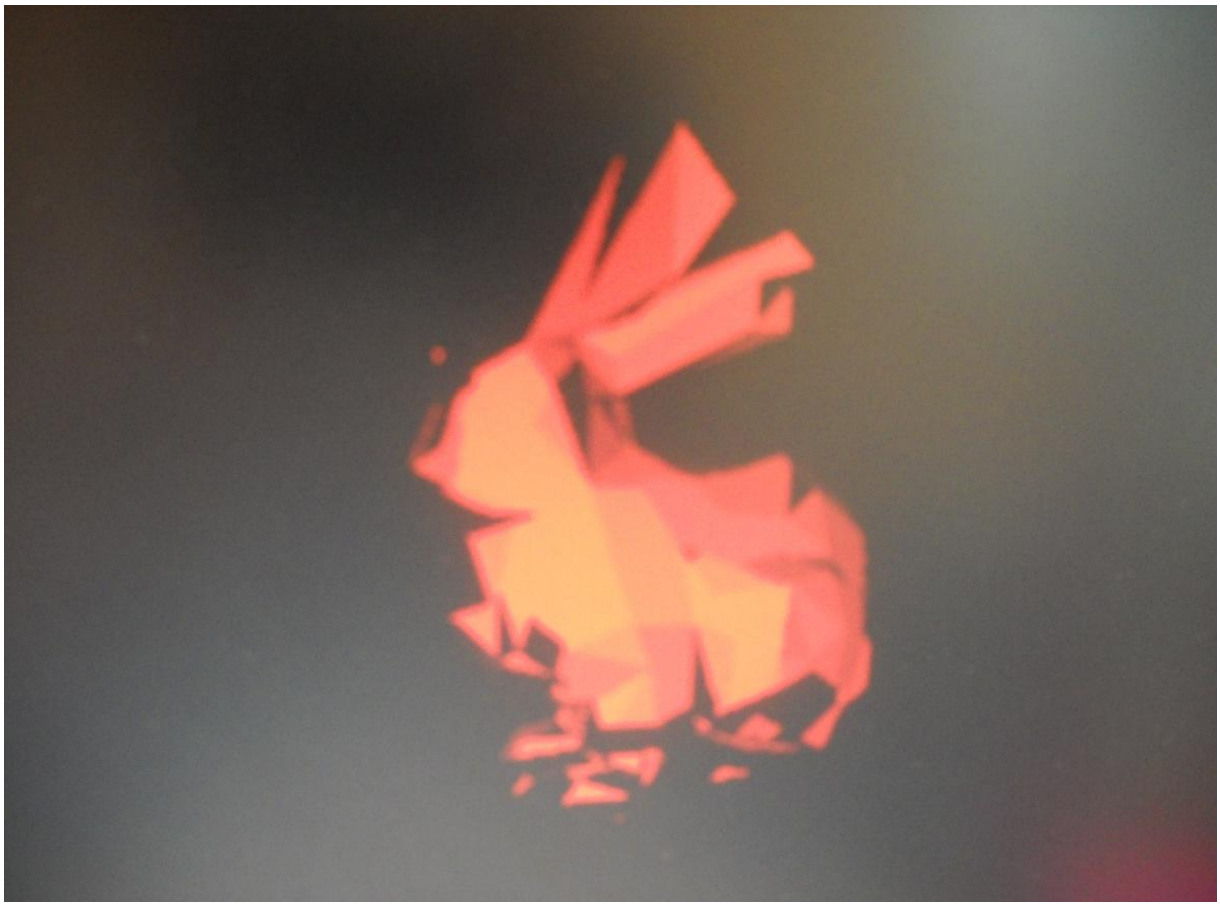


Figure 11 : Rabbit

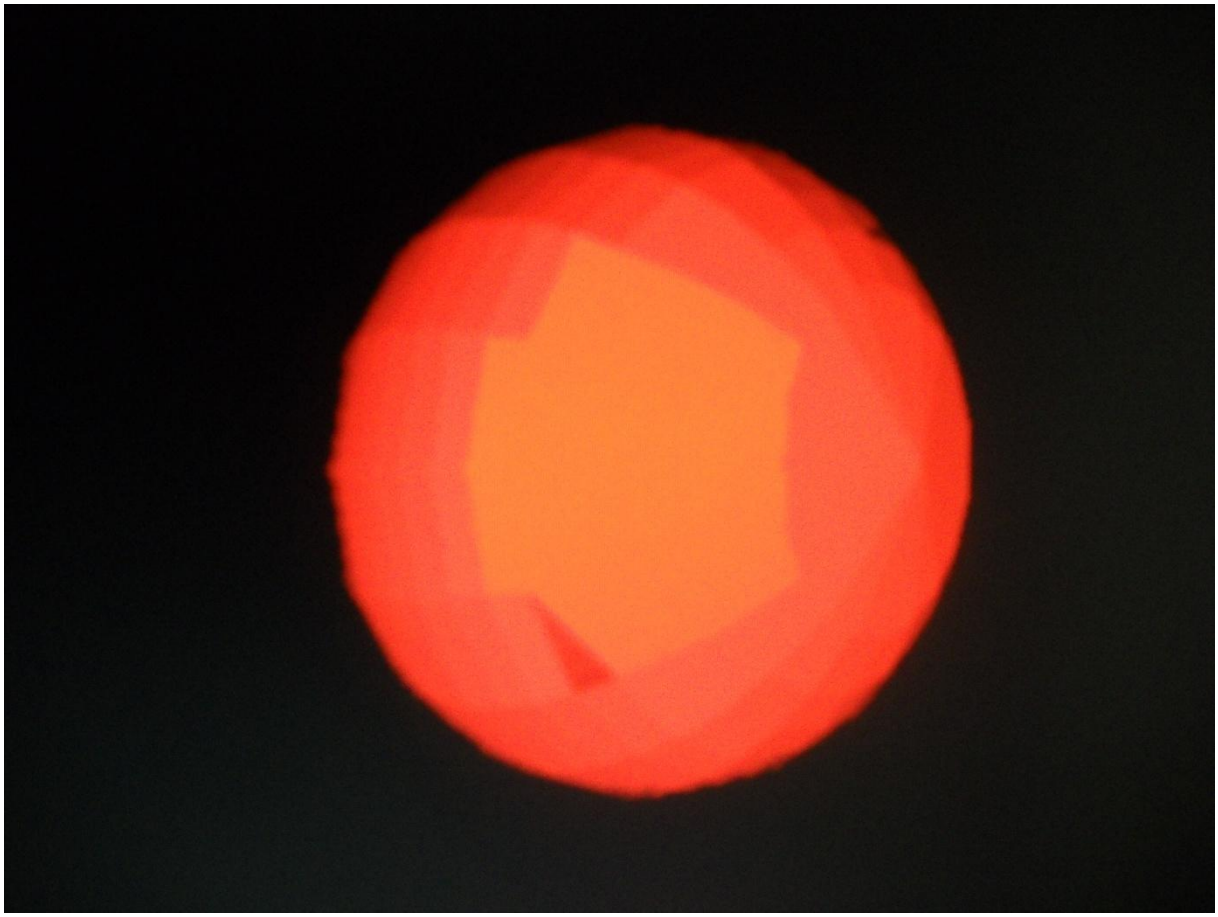


Figure 12 : Sphere

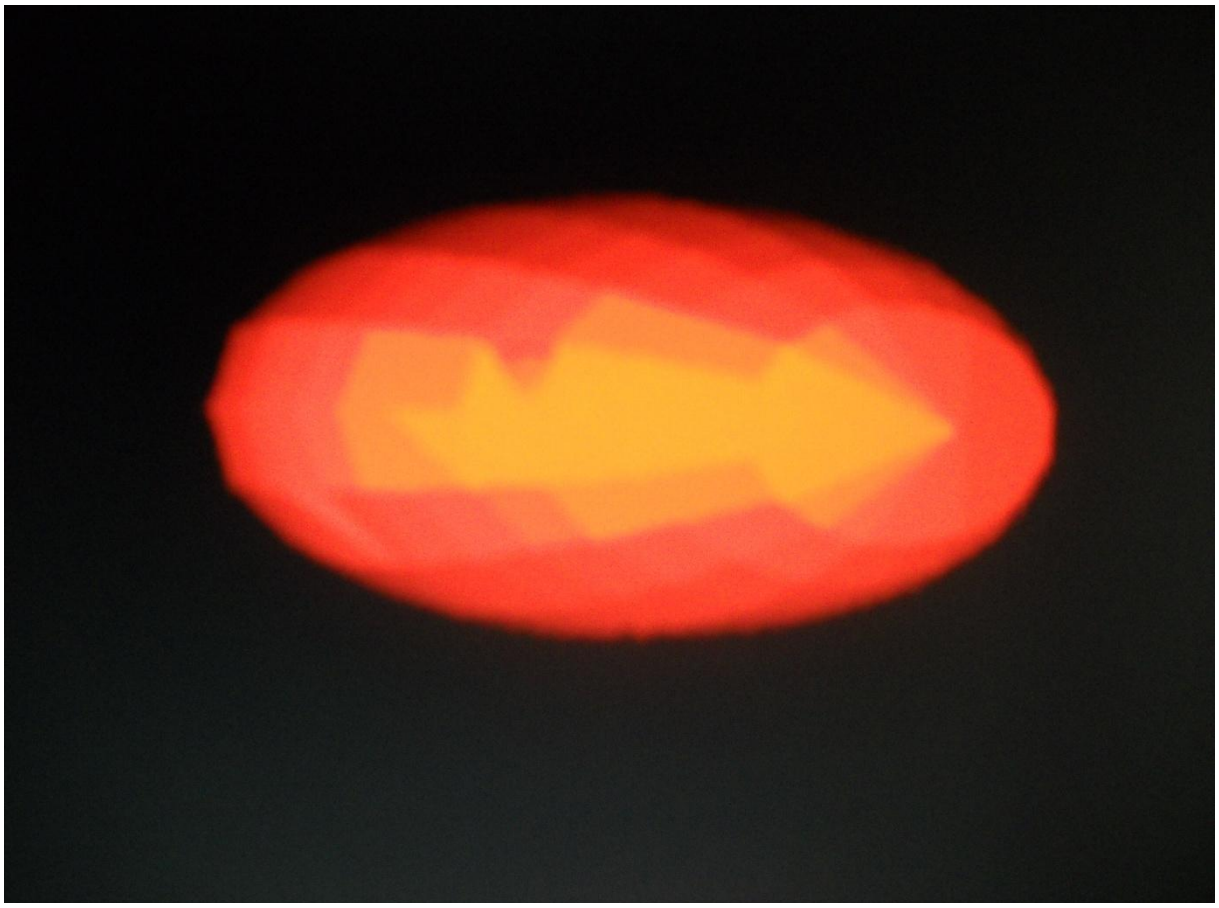


Figure 13 : Squired sphere

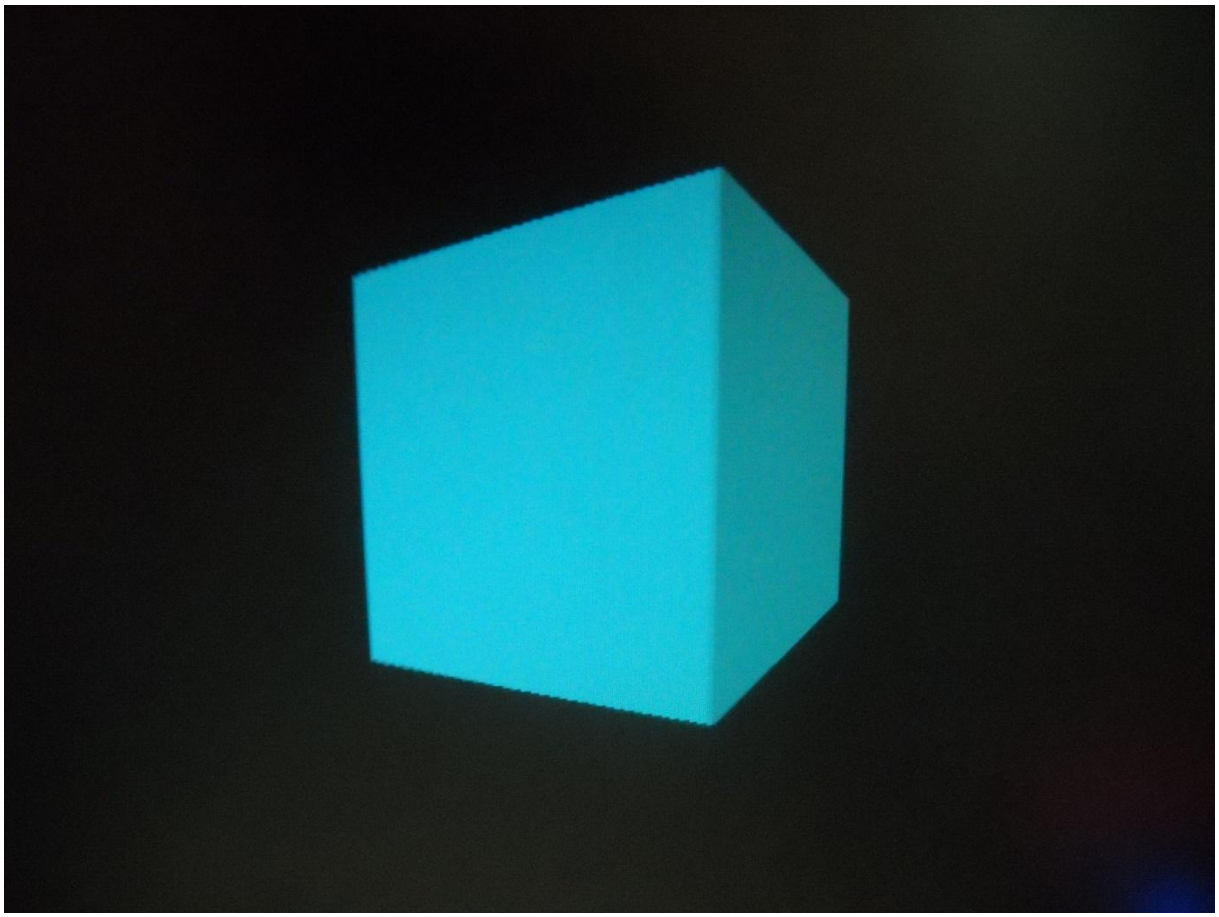


Figure 14 : Cube

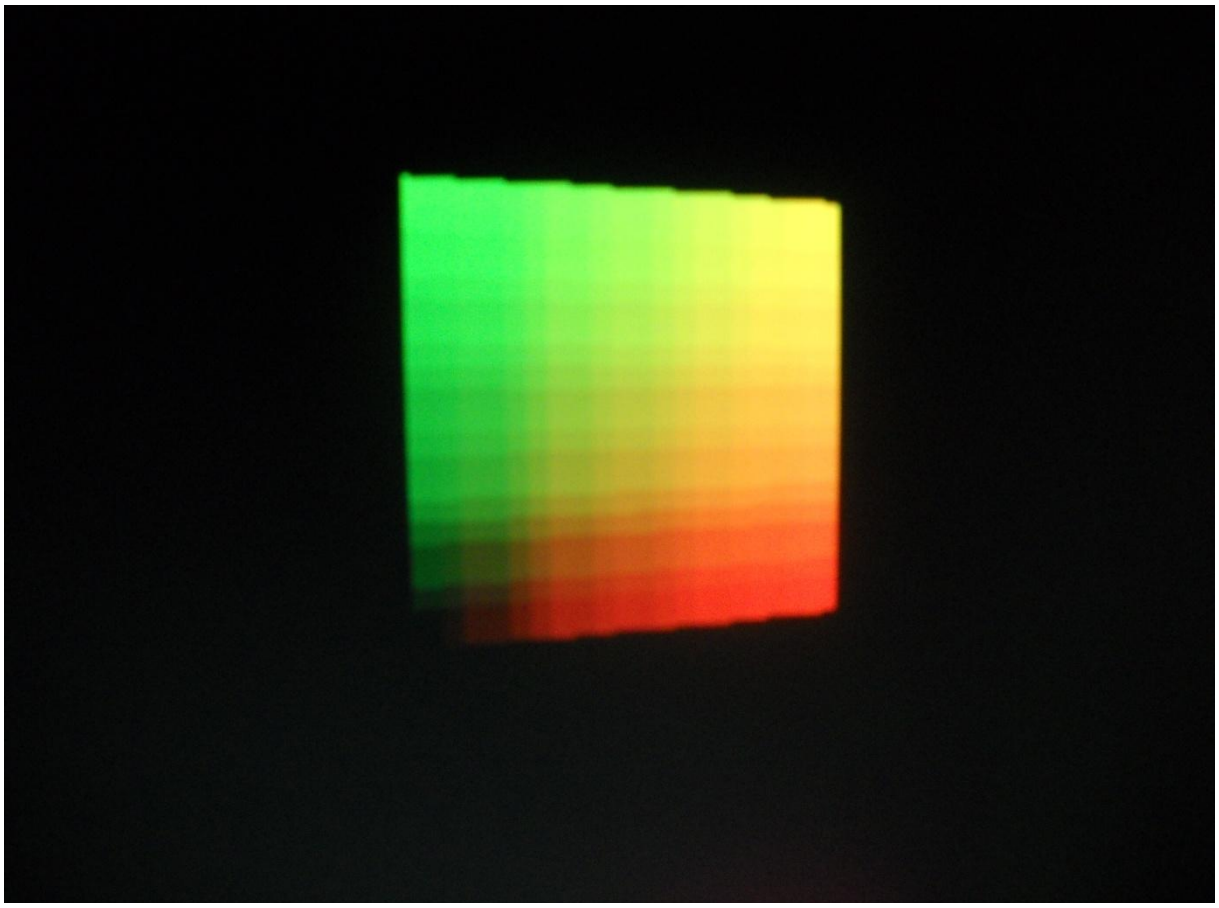


Figure 15 : The square

Conclusion - Comparison with my expectations

I have implemented an easy-to-use API which permits user to display their graphic software applications on a NIOS. I succeeded in implementing the chain from this high-level API to the rendering on the screen through the VGA controller. I But I missed several of my objectives.

Synchronization problem

The main failure is the remaining of a synchronization problem between the NIOS and the graphic card, which occasionally have the graphic card missing a drawing triangle instruction. Because the two components are not on the same clock, one output of one component can be “undetermined” (neither 0 nor 1) when the other reads it, if the front edge of the clock occurs to happen roughly at the same time.

Performance of Graphic card

The second main failure is that I wanted the graphic card to work on the highest clock frequency, possibly the 50Mhz clock from the board. But there were too much delay because of the high hierarchy, and I was not able to reach even the 27Mhz. Because the 50Mhz and 27 Mhz signals were already used in PLL, I set the input clock it to the same clock of the VGA controller (about 25 Mhz), which is below what the SRAM can stand.

Flexibility

I wanted to let the user customize the resolution of the rendering (the screen size, and given a certain amount of memory per pixel, be able to select how many bits are used for r, g, b, and z channels). One significant advantage would be to disable z-testing when rendering, because this operation is slow (an application can render the triangle in a z-sorted order), and re-use the bits which were used in z-buffering for the color channels.

I have some building parameters when compiling the project on Quartus in order to have variable widths for the r, g, and b data. But I was not able to simplify on an interface.

Memory limitations

Because the memory is small (512 kbits), to get a 512 x 256 screen with double buffering I had only 16 bits left per pixel. That was only 4 bits per color channel, and the same for the z channel. In first, I wanted to send the geometry data from the application to the graphic card, and store it there to be closer to usual graphic cards (like the user of Vertex Buffer for example in DirectX). This would have permit to have the concept of Index Buffer in the API to describe the geometry, with efficient use of memory. But the SRAM was already filled with the data for the screen, with already a low resolution.

Multiplier limitations

I wanted to move the transformation of the geometry (considering 3D transformation, user view, 3D projection) from the software to the hardware. But because this is implemented through matrices this would consume too many multipliers, this could not be performed in parallel to take benefit of the hardware. Thus, matrix multiplication is

done using the NIOS ALU. Nevertheless, it would have been interested to implement the multiplying in hardware even without the optimal parallelism to compare which one is faster.

Acknowledgements

I used the Altera website and the DirectX website (MSDN) to learn about linear algebra formulas.

I would like to thank Bruce Land for his availability and support.

This project reinforced significantly my training in microcontroller devices learnt in ECE5760 from Bruce Land in the first term.