

## **Penrose World**

CSEE4840 Embedded System Design

03/26/2014

Project Design

### **Group member**

Yuanhui Luo(y13026)

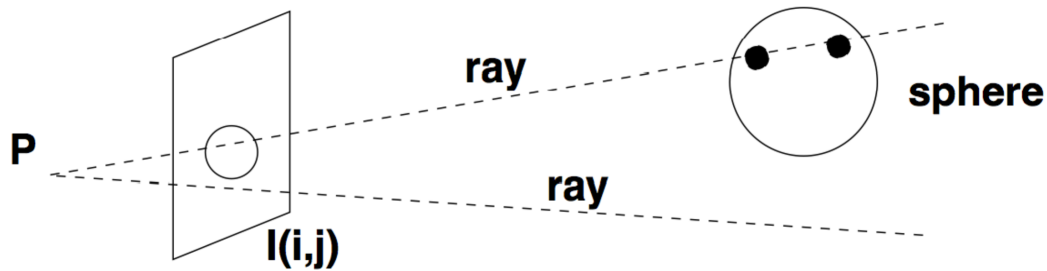
Yao Luo(y13028)

Cong Zhu(cz2311)

## Implement details: Algorithm

### Induction

We consider use Ray-Casting algorithm to implement our game. Ray-Casting is a kind of basic graphic projection technology which is frequently implied in 3D area. In our game, we need to project a visual 3-D object into our image plane (screen) by Ray-Casting. The sketch map is shown as below.



It just like see the projection of a cubic object on a plane from a eye. We connect every point on the surface of our visual 3-D structure and our eye point P, and then calculate projection of that point on the image plane (screen) (or we can call it image plane). Also we will divide our image plane (screen) into many pixel, each pixel has a set of value which represent its range, the color value (just like lab3) of pixels are depended on the projection results of points in visual cubic structure. Finally, we can finish the display on image plane (screen) by the value of set of pixels on image plane (screen).

### Below is the flow path of our algorithm

1. Build a computer-held visual 3-D composed cubic structure. We store the boundary constraint of all planes to represent a composed cubic structure in visual space.
2. Initialize and record the value of a three-dimension vector  $R(\theta, \gamma, \varphi)$  which represents the rotation angles according to x-axis, y-axis and z-axis. The method of rotation will be described later.
3. Generate enough points in our visual 3-D space in each plane by the initial boundary constraint, which is the initial points in our visual 3-D space. Location of every point is represented by a three-dimension vector. And then by multiplexing the initial location matrix of each point to rotation matrix which is generated from rotation vector R, we get the location of all points after rotation.
4. Renew our image plane (screen), which is constructed by  $480 \times 680$  pixels, besides its location, each pixel records the plane number and the distance of a point which is projected on that pixel. It is possible that more than one point projected on the pixel, we will also record the related value of the nearest quailed point to our pixel. If no point is projected to that pixel, we just record NULL.

5. For each plane, we project every point on that plane to image plane (screen). If the point is projected in the area of a pixel, we record the distance between that pixel and point and the plane number of the point in the pixel. We always renew the distance and plane number when the distance between new point and that pixel is smaller than the prior one.
6. After projecting all planes, the four-dimension pixel matrix (x-coordinate, y-coordinate, plane number, distance) is passed to image plane (screen) controller block by data bus. And the image plane (screen) controller can generate the projected graph on image plane (screen).
7. Repeat step 2-6 when new command signal from gamepad is sent to CPU.

### The method of Rotation

In our game, we will rotate our visual space cubic object accord to x-axis, y-axis and z-axis. We implement the operation by rotate every point which compose the whole cubic object. We consider the rotation in 2-D space. The original point P(x,y) is shown in the figure below. The vector (x,y) is rotated by angle  $\theta$ . Assume the original vector is  $(A\cos\varphi, A\sin\varphi)$ . After rotation, the new vector is  $(A\cos(\varphi + \theta), A\sin(\varphi + \theta))$ .

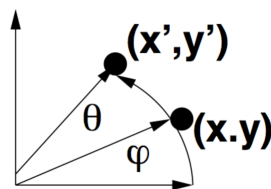
$$x' = A\cos(\varphi + \theta) = A(\cos\varphi\cos\theta - \sin\varphi\sin\theta) = x\cos\theta - y\sin\theta$$

$$y' = A\sin(\varphi + \theta) = A(\sin\varphi\cos\theta + \cos\varphi\sin\theta) = y\cos\theta + x\sin\theta$$

so we can take a rotation matrix R to finish operation above.

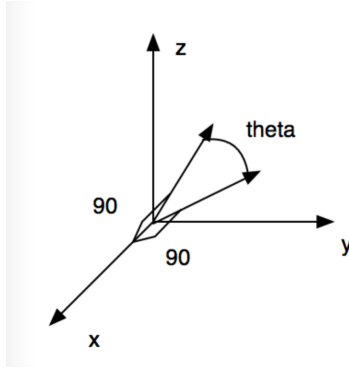
Which means

$$\begin{bmatrix} x' \\ y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} x \\ y \end{bmatrix}$$



In the same way, we know when we rotate a point according coordinate axis. For example, x-axis. The x coordinate of that point will not change, and the y coordinate and z coordinate will change with the rotation angle.

So  $R(\theta, \gamma, \varphi)$  which  $\theta, \gamma, \varphi$  represent the rotation angle separately.



So we can define three Rotation Matrixes  $RotX(\theta), RotY(\gamma), RotZ(\varphi)$ , and calculate the rotated coordinate of each point  $P$ .

$$RotX(\theta) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$

$$RotY(\gamma) = \begin{bmatrix} \cos\gamma & 0 & \sin\gamma \\ 0 & 1 & 0 \\ -\sin\gamma & 0 & \cos\gamma \end{bmatrix}$$

$$RotZ(\varphi) = \begin{bmatrix} \cos\varphi & -\sin\varphi & 0 \\ \sin\varphi & \cos\varphi & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

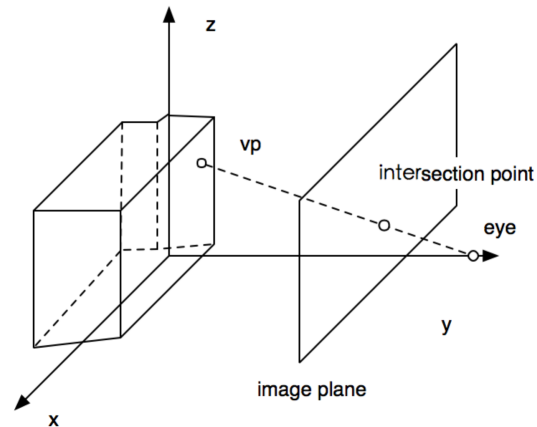
$$P = \begin{bmatrix} x \\ y \\ z \end{bmatrix}, P' = \begin{bmatrix} x' \\ y' \\ z' \end{bmatrix}$$

$$P = RotX(\theta)RotY(\gamma)RotZ(\varphi)P'$$

### Ray casting

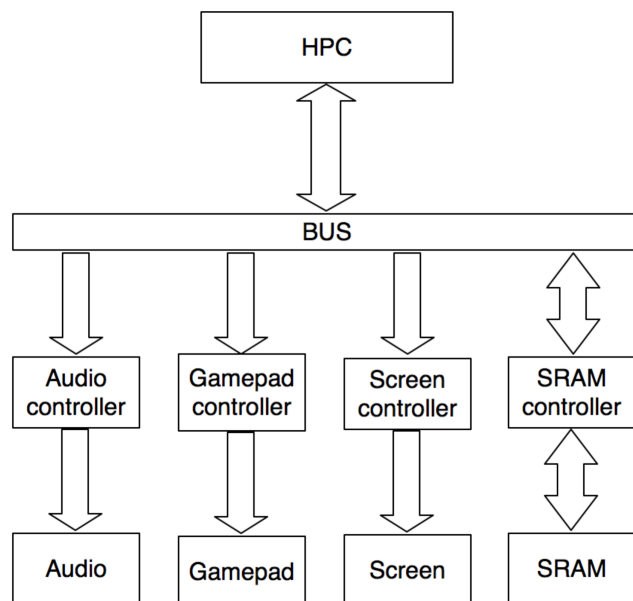
We need to calculate if points on the visual cubic structure can project on the image plane as well as the location in where it projects. We implement it by calculate the intersection points of the image plane and the line from eye point to points on cubic structure(vp). we set the eye and the image plane into a fixed position. The eye will locate on y-axis,  $E(0, y_e, 0)$  represent its space location. Obviously , the image plane is perpendicular to y axis. We can represent the location of pixels in a  $480 \times 640$  image plane by a set of coordinate  $P(x, 0, z)$ , x vary from -319 to 320, z vary from -239 to 240.

To calculate the projection coordinate from points on cubic structure(we name it vp) on image plane, we need at first represent the direction vector from eye point to vp, and then give the value of y coordinate  $y_e$  into the connection lines above to calculate x and z, which are the coordinate of intersection point. Finally set the value of those points and calculate the distance between vp and that intersection points. The sketch map is shown as below.



## Hardware Peripheral

The components of our system include the HPS and FPGA core on the Sockit Board as well as the SRAM block. One task for the FPGA is mainly used to build the controller (driver) of SRAM, VGA, and the gamepad. The FPGA is also responsible for the parallel computation of point projection on screen, generate input data stream into the VGA screen. We use gamepad to control direction of rotation, store the location information of 3-D object and its 2-D projection in the SRAM. Finally, all the communications are happened on the Avlon bus. Further details about each part is described as bellow:



## FPGA

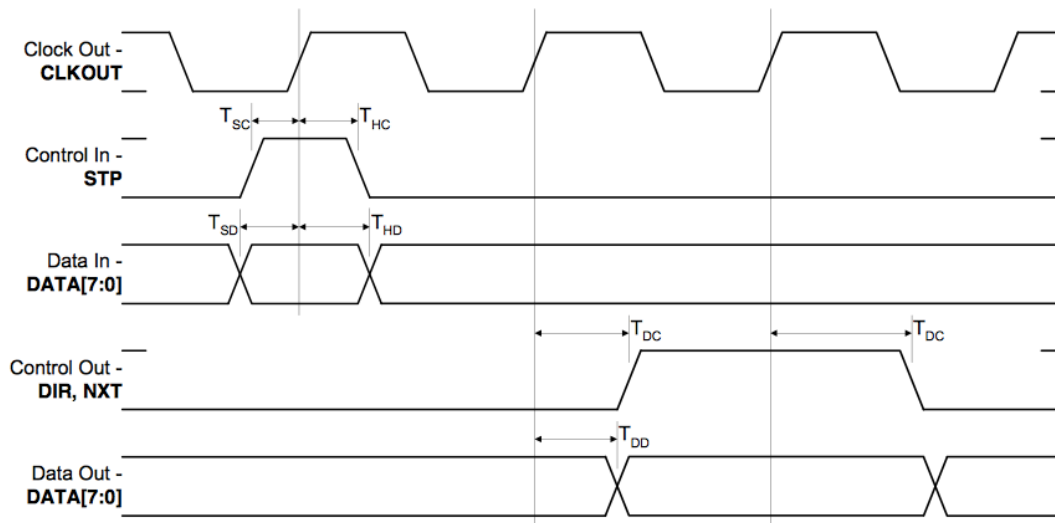
Besides the controller of all the peripherals, FPGA is also responsible for the parallel computation of each point on the object onto the screen. In each clock cycle, we generate several rays which pass through the eye and each point on the surface of target object, record the two dimensional location on the screen and calculate the distance from the screen to the object's surface. We can do the calculation of each line in parallel for the reason that every single computation has no data dependency with others. The detail of every calculation is shown in the algorithm implementation.

## Gamepad

We use the gamepad to control the rotation of 3D object showing on the screen. There are four basic moving instructions in our design. The press of button will be detected by the gamepad and send to the HPS as an internal signal. Gamepad communicate with the main logic in HPS using USB port and Avlon bus. These moving instructions then define the moving operations and the angle of moving to the FPGA to generate the new rotated 3-D object. The gamepad is connected to the Sockit board using USB ports on the board. The USB is controlled by SMSC USB3300 controller. The interface of this controller is shown as bellow:

<b>PERIFERAL</b>	<b>DATA_TYPE</b>	<b>FUNCTION</b>
HPS_USB_CLKOUT	output std_logic	60MHz reference clock for signals synchronous.
HPS_USB_DATA	inout std_logic [7:0]	Transfer control signals from gamepad to HPS.
HPS_USB_DIR	output std_logic	Controls the direction of the data bus, pull it high when gamepad buttons are pressed.
HPS_USB_NXT	output std_logic	The PHY asserts NXT to throttle the data. When the Link is sending data to the PHY, NXT indicates when the current byte has been accepted by the PHY. The Link places the next byte on the data bus in the following clock cycle.
HPS_USB_STP	input std_logic	The Link asserts STP for one clock cycle to stop the data stream currently on the bus. If the Link is sending data to the PHY, STP indicates the last byte of data was on the bus in the previous cycle.
clk	input std_logic	clock signal

The timing diagram for this mode is shown as below:

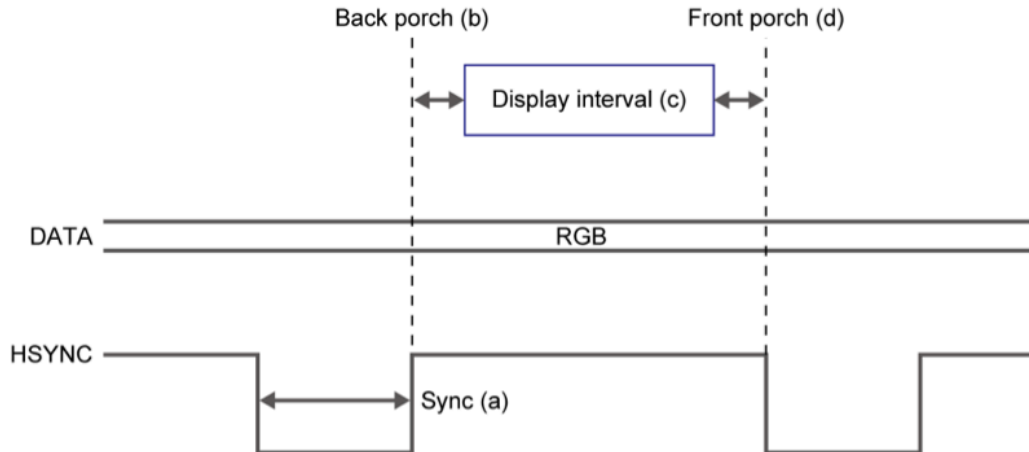


### VGA Display

The data processed by main logic in FPGA and HPS has the form of a 3-D array in which the first two columns are the X-Y coordinates of points that will display on the VGA screen. The third column indicates which surface the current shape is projected from. The VGA controller will first analyze the coordinates of input data to decide that which pixel on the screen is chosen to be light up. Then, the third column of input data will be used to decide the color of this projected surface in order to distinguish with different surfaces. The whole protocol is much like the lab3 where VGA controller response only for lighting up specific area of the screen based on the input data.

PERIFERAL	DATA_TYPE	FUNCTION
VGA_CLK	output std_logic	clock output to drive VGA screen
VGA_HS	output std_logic	horizontal synchronize
VGA_VS	output std_logic	vertical synchronize
VGA_BLANK_n	output std_logic	control signal for blanking
VGA_SYNC_n	output std_logic	control signal for synchronize
VGA_R	output std_logic [7:0]	chroma of red
VGA_G	output std_logic [7:0]	chroma of green
VGA_B	output std_logic [7:0]	chroma of blue
writedata	input std_logic [335:0]	data of object and angle of view. 288 bits for vector information, 48 bit for object's surface.
clk	input std_logic	clock signal

The timing diagram for this mode is shown as below:

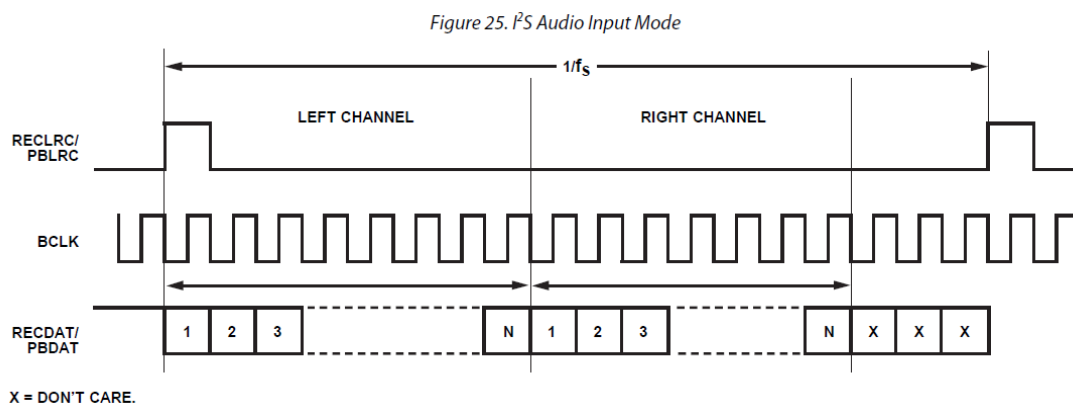


### Audio

We use the SSM2603 Audio CODEC block on the Sockit Board to display the back ground music. The music and some kind of sound effects are stored at the SRAM on board and the block is controlled by the HPS core. All the music are played automatically when the system boots and the sound effects are played whenever the HPS required. The interface of Audio block and FPGA is shown as bellow:

PERIFERAL	DATA_TYPE	FUNCTION
AUD_XCK	input std_logic	Audio CODEC Chip Clock
AUD_BCLK	input std_logic	Audio CODEC Bit-Stream Clock
AUD_DACDAT	input std_logic_vector [23:0]	Audio CODEC DAC Data
AUD_DACLK	inout std_logic	DAC Sampling Rate Clock
AUD_MUTE	input std_logic	DAC Output Mute, Active Low

The default work mode of audio output is I2S mode, the timing diagram for this mode is shown as below:



### SRAM and data storage

The object we show and rotate on the screen is actually stored as an array consists of many 3-D coordinates indicating a specific point on the object's surface. To generate this object, we only stored all the vertexes of this object. The points on every surface are actually



generated in every execution and these generated points are then used to calculate projections of each surface. After all the surfaces are correctly generated, a 2-D array is stored into the SRAM where each line of this array means the location on the screen. On top of that, each pixel is attached with a flag indicating which surface it belongs to so that VGA can display different color in different area.

### **Software Peripheral (HPS)**

The software peripheral in our design is relatively small because most of the operation happens in hardware. Every time when the gamepad sends a control signal to the system, we need to pass it to the software peripheral. It is the software that generate output to other block and hardware to react to the control instruction. Once the software detect a moving instruction is given by the gamepad, it generates a rotate instruction to send to the FPGA so that the main logic block in hardware could do the computation of rotation. The registers set to communicate between software and hardware are defined as bellow:

<b>REGISTER NAME</b>	<b>FUNCTION</b>
rotate_left	rotate to the left
rotate_right	rotate to the right
rotate_upward	rotate up
rotate_downward	rotate down
au_reset	reset the audio
au_on	begin playing music
direction_rotate	direction of rotation
angle_rotate	angle of rotation