Bouganis Hunt 3000

EIE1 Project by Dinmukhamed Yermembetov, Jonathan Zheng, Mark Zolotas

Imperial College London, 2013

INTRODUCTION

Our project is a first-person shooter arcade game, influenced by the NES game “Duck Hunt”, that functions on the Altera DE2 FPGA. The objective of this game is to shoot down the targets that appear at random, using the provided mini-laser toy. This report will outline the methods implemented into crosshair position tracking, the game engine, graphics synthesis, and sound production. In order to accomplish this, we used the parallel design enabled by the layout of the FPGA to increase efficiency.

TRACKING

To track the location of the crosshair we decided to detect colour; the algorithm we used calculates the average of all the coordinates that pass a threshold. It counts the number of pixels that pass the threshold, and at the same time adds their coordinates to a total, so that each time the camera cycles through all of the pixels, the total sum of coordinates is divided by the total pixels counted to get the average. To do this, we used global variables to preserve information across cycles, because program does not let us to use parameters as variables. Furthermore, as Catapult-C does not allow us to divide ‘ac_int’ variables, a special math library (‘math/mgc_ac_math.h’) had to be included to perform the division.

However, even with this algorithm in place we were not tracking colour properly because the amount of pixels sent from the camera in one cycle was unknown to us. Because of this, we decided to create our own 640x480 frame and coordinates by applying a buffer to both the left and right sides of the coordinates from the TV_to_VGA block. This was a major breakthrough in implementing our ideas. The whole tracking mechanism and graphics engine are based on this conversion as it allows us to know exactly where pixels are coming in and going out, as well as being able to split the display into 300 32 x 32 squares in a 15 x 20 grid (used to tile sprites to ‘draw’ a background). **[1]**

The algorithm for detecting whether a shot is attempted is quite simple. It takes in the colours seen by the camera and if a pixel crosses a threshold, a variable is set to 1, if no pixel crosses the threshold, a variable is set to 0. If the previous cycle of pixels is 1, and the current cycle of pixels is 0, it means that a colour disappears from the input, and a shot is fired. In real life, this was to be implemented by attaching a colour that passes the threshold to the player’s thumb, and having the player hide the colour by bending their thumb to hide it behind the other fingers.

During initial tests our crosshair was either not appearing at all or just shaking around, which meant our thresholds were not correct, and the background was severely affecting the pixel inputs. We tried using LEDs as a detectable colour source, but it turned out that the camera picks up too many components of other colours from the light source, creating a pretty much white input, not allowing us to reliably distinguish between colours. The camera also had trouble differentiating between normal colours, as it turns out that the camera produces a ‘washed out’ image, making very different colours in the real world blend into each other in the captured video.

Through a long process of experimentation, we found out that black felt has the property of being

very different to the camera than anything inside a normal background. In fact, it turned out to be the most reliable source of colour for our algorithm to track. The algorithm for detecting whether a shot is to be attempted was also unreliable despite any device we attached to the thumb, but luckily we also had possession of a toy keychain in the shape of a gun that produces red light when the trigger is fired. With the felt and the red light, we managed to set thresholds for colour that could be reliably picked up by the camera. However, we still sometimes encountered tracking problems due to an inconsistent background, so we added the ability to change the thresholds post compilation; this was done by making the values changeable through switches on the board, and implementing 7-segment decoders to see and write down the results.

GAME ENGINE

The game engine handles the positions and the states of all of the 'moving parts' of the game. This includes the player's health, score, fire rate, crosshair position, target positions etc. and as you can guess this means a long list of parameters and variables needed for the C code inside catapult. The problem is that all of these parameters depend on each other and so it's not possible to compute them in parallel. Because of this, we set the block to pipeline in the architecture constraints. This means that more than one instruction is performed every clock cycle inside the block, and so the throughput can be maintained.

We also created a block in quartus that produces a 30 Hz clock and used that for this block instead of using the 27 MHz clock used in the other blocks. This is because of the game engine making extensive use of counters to time the amount of time the 'moving parts' stay in different states. Using a 30 Hz clock allows us to reduce the size of the variables used in the counters to reduce memory usage, as well as locking the frame rate so that we can be sure of how many pixels a sprite will move per second, because their movement is calculated on a frame by frame basis.

With these settings in place, the computation inside the game engine can be written. To keep things simple, we used only one target, although the code could be easily duplicated, used in parallel, and set with different coordinates to produce multiple targets on the screen at a time. Whether a shot can be fired because of a fire rate cap is first calculated, and the position of the crosshair and position of the target are compared to check for intersection. Other algorithms and counters are used to calculate the new position of the target, what sprite to display, health remaining etc. and they are all assigned to output parameters and saved in global variables to use in the next cycle of the block. For further details, see catapult C file that contains comments and explanation of the operation of the different algorithms used ('game_engine.cpp').

GRAPHICS

In terms of graphics design, we first have the current coordinates of the pixel being sent to the VGA converted into it's position in a 15 x 20 grid (named 'block_coord'), along with it's position within the block (named 'inblock_coord') sent into a schematic named 'background_display'. The purpose of this block is to retrieve the colour of the pixel in the background at those coordinates; within this block a number of other functions take place.

The images displayed on the screen (sprites) are created through the usage of ROMs and memory initialisation files. To construct an image, we made use of these files by creating ROMs with 1 bit words and 1024 addresses (used as a 32 x 32 square); inside each word a 1 means foreground and a 0 means background. By manually placing a number of 0s and 1s in the form of a visual pattern, we can create an image that contains two colours. **[4]**

The reason for only using 1 bit per word and not more (which would allow more colours and therefore more detail) is the fact that there isn't much memory space in the DE2 (a total of a 400 kb, not including SRAM that requires an extra controller) and using more than 1 bit words very quickly breaks this limit. This gave us the option to 'paint' a 32 x 32 sprite with 2 colours by attaching a 12 bit number to each sprite (6 bits each for foreground and background) giving us a 64 colour palette. With a little creativity, a nice looking graphics interface can be created despite these constraints.

To draw the background, each 32 x 32 sprite is given a 6 bit address, allowing for 64 total sprites to picked. The 15 x 20 'block_coord' coordinates of the pixel being sent to the VGA pass through a ROM called 'background_sprite_layout', which contains the addresses of the sprites that exist within each block in the 15 x 20 grid. The output is then used as the select line of a 64 line multiplexer to pick the correct sprite.

At the same time, 'inblock_coord' (the address inside the current 32 x 32) is sent into every sprite ROM in parallel and the outputs (1 or 0 for foreground or background) are sent into their corresponding address as multiplexer inputs. The output of 'background_sprite_layout' also passes through to another ROM called 'sprite_colours' where the foreground and background colour of the current 32 x 32 square is read. The 1 bit foreground/background output acts as the select line between the colour selected through 'sprite colours' to finally pick the colour of the current pixel being sent to the vga in the background.

Displaying the 'moving parts' of the game uses a similar method, except that their coordinates aren't predetermined, and so they cannot be read from a ROM. Because of this, an algorithm is used to compare the coordinates of a 'moving part' (the top left pixel of the sprite) with the current pixel being sent to the VGA to see if it lies within its area. If it is, the coordinates of the pixel within its area are calculated and then sent to another ROM. The output of this ROM is then used as a select line between the foreground colour and background colour of the sprite, or used as a select line between the background colour and another colour to make the sprite transparent. Because there are many 'moving parts', they may collide with each other, so an order of preference has to be determined. This is done by placing many multiplexers in front of each other to act in a way similar to multiple 'if...else' statements in a row.

SOUND

The sound engine is based on the design behind the music box that we made previously in a lab experiment with the FPGA, so we had already written most of the code beforehand. The basis

behind it is that a musical note is just a periodic signal of a certain frequency, and if we can generate waveforms with predetermined frequencies, then we can sequence the different frequencies in a predetermined order to produce a tune.

To do this we made use of an onboard frequency generator on the FPGA that provides a 50 MHz clock signal. This is divided using a counter, producing a square wave of controllable frequency **[2]**. To create a 'keyboard', all we had to do was reference a chart of the frequencies of different notes, and string 25 of the divider blocks together, connecting them all to a multiplexer allowing the select line to act as the 'keys' that select different notes.

Creating the sequence of notes that get sent to the 'keyboard' used a similar principle to the note generator, except this time the divider was set as a much bigger number to produce a much slower output that is used as the tempo at which the tune is played. This slower output is used to increment another counter that is used as the input for a switch statement, with the different cases inside set to values that are sent to the select lines of the 'keyboard' multiplexer. All that was left was to compose some tunes and change the cases inside the switch statement accordingly, and the signal that needs to be sent to the speaker is finished **[3]**.

However, simply sending the signal to an output pin runs into problems. The audio line out port on the board does not take in a one bit signal, it requires 4 different pins inside quartus to be connected to produce an output that a speaker can understand. After trying all kinds of combinations to try and get a working output without any luck, we turned to the internet. Luckily for us, on the Altera forums, someone had the same problem as us, and a helpful user posted a block that contained the codec necessary to convert a 32 bit signal into the correct form for the line out port. We connected our one bit signal to all 32 bits of the input to the codec, connected up the pins, and sound was up and running.

The changes made for this project were small details. To produce the 'thump' sound for a shot being fired, a randomly changing frequency needs to be sent to the speaker, so we simply incremented the divider inside a note block by a large prime number to produce a close to random sequence of frequencies. Another little detail was the ingame music gradually increasing in speed as the player's score increases to create a sense of urgency. Again, this is done by changing the value of divider, except this time decrementing it by a multiple of the score being sent into the block. Interestingly, these are both technically methods of frequency modulation, except in a digital form.

CONCLUSION

We have learnt a lot over the course of this project. Our main discovery was that no amount of planning can account for unforeseeable circumstances, such as our problem with the camera not being able to distinguish colours accurately. In addition, all of our Catapult files are pipelined with an initiation interval as small as possible so as to increase the throughput. This is due to the fact that the signal from the camera and to the VGA contains one pixel per clock cycle and thus there is no time for lengthy computation in series.

If we had more time and resources for the project, we would have invested in a more sensitive camera and added more features to the game engine, such as different targets, levels, more detailed graphics etc. The mini laser toy that we use for tracking is clearly not ideal, and a device designed purely for the intention of tracking by the camera would increase the quality of gameplay.

Overall this was an enjoyable project, and acted as an opportunity for us to express our creativity. It has given us a good insight into high level synthesis and the differences between design in parallel as opposed to series, and a beginner's look into hardware on the whole e.g. how data is processed by the VGA and camera.

APPENDIX

```
always @(posedge clk)

   //if the current pixel passes the 185 pixel buffer, its coordinates inside the frame are calculated
   if(vga_x_in >= 185 && vga_x_in < 825 && vga_y_in >= 30 && vga_y_in < 510) begin
      frame_x <= vga_x_in - 185;
      frame_y <= vga_y_in - 30;
      frame_there <= 1;
   end
   else begin
      frame_there <= 0;
   end
```

1

```
reg[15:0] clkDivider = 23889;
reg[15:0] counter;

always @(posedge clk)
   if(counter == clkDivider)
      counter <= 1;
   else counter <= counter + 1;

reg out;
always @(posedge clk)
   if(counter == 1)
      out <= ~out;
```

2

```
   if(counter == clkDivider)
      counter <= 1;
   else counter <= counter + 1;

reg[5:0] state;

always @(posedge clk)
   if(counter == 1)
      state <= state + 1;

reg[4:0] out;

always @(state)
   case(state)
      0: out = 12;
      1: out = 11;
      2: out = 10;
      3: out = 9;
      4: out = 7;
      5: out = 25;
      6: out = 4;
      7: out = 25;
      8: out = 2;
      9: out = 25;
      10: out = 4;
      11: out = 25;
      12: out = 7;
      13: out = 25;
      14: out = 9;
      15: out = 25;
      16: out = 12;
      17: out = 11;
      18: out = 10;
      19: out = 9;
      20: out = 7;
      21: out = 25;
```

3

Addr	+0	+1	+2	+3	+4	+5	+6	+7	+8	+9	+10	+11	+12	+13	+14	+15	+16	+17	+18	+19	+20	+21	+22	+23	+24	+25	+26	+27	+28	+29	+30	+31
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
32	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
64	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
96	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
128	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
160	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
192	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
224	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
256	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
288	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
320	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
352	0	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
384	0	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
416	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0
448	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
480	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
512	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
544	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
576	0	0	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
608	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	1	0
640	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0
672	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0
704	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0
736	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0
768	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	1	1	1	1	0
800	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
832	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
864	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0
896	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0
928	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0
960	0	0	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0
992	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

4 This is an example of a sprite ROM. If you look closely you can see the outline of an 'S'.