Xil_Roids

18-545 Fall 2008

Anthony Cartolano

Jun Lee

Brad Miller

Junqing Wei

Contents:

What we built

Game Description:

Xil_Roids was designed to be a two player version of Asteroids. In Xil_Roids, players have the ability to shoot both asteroids and their opponent, and are rewarded with points accordingly. A player is killed when he collides with either an asteroid or the bullet of another player. Each player begins the game with 6 lives, and the game terminates when either player has run out of lives. The player with the most points when the game terminates wins.

System Overview:

Xil_Roids runs primarily on the Virtex-II Pro FPGA board and uses Nintendo Wii remotes to obtain user input. The code which interacts with the Wii remotes is the only code which does not run on the FPGA board. Wii remotes communicate using Bluetooth, hence the code which interacts with the Wii remotes runs on a Linux workstation with Bluetooth. The Linux workstation communicates user input to the FPGA board using an Ethernet connection.

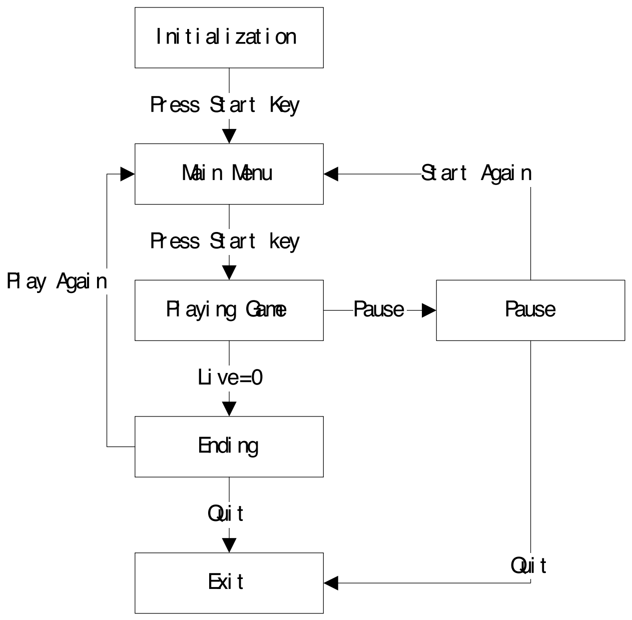
Xil_Roids maintaines a software representation of the game state, and updates the state, screen and audio output buffer at a regular rate according to the interaction of objects within the game state and input from the user. The representation of the game state which Xil_Roids maintains was designed to be both elegant and efficient, so that all timing requirements can be met and the complexity of the game can be reasonably increased. In particular, there was motivation to write the software for Xil_Roids in a sufficiently efficient manner to allow the game to run entirely in software, preventing added complexity required by the incorporation of hardware.

In addition to the core Xil_Roids logic which handled user input, maintains and updates the state of the game, and generates audio and video output, additional software creates a finite state machine surrounding the game to handle transitions from setup, to game-play, to game-paused, etc. The Xil_Roids implementation is also supported by a custom file format which facilitated both collision detection and increases the efficiency with which Sprites can be painted on the screen. A file converter was created which runs on Linux and convertes BMP files to this custom format.

System Componet Description:

Game Finite State Machine

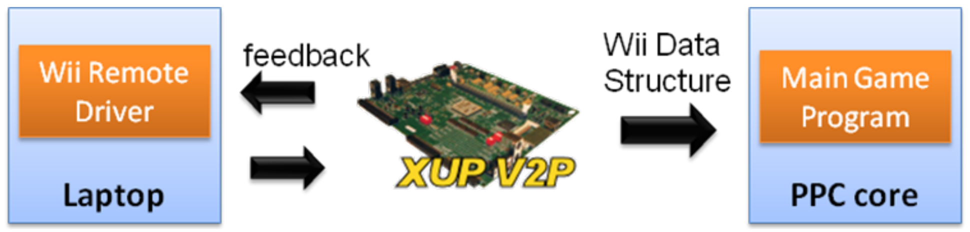
The game state is an important of the playability of the game. The game state of our system is like this.



To implement this, we used a game state structure to control the game state and also store some important configure information of the game, such as whether a player is still alive, the game is in multi or single player mode, etc.

Ethernet

One of the highlight of our system is that we port the Wii Remote controller into our system as the user input device. To achieve this, we want to implement a framework as following:



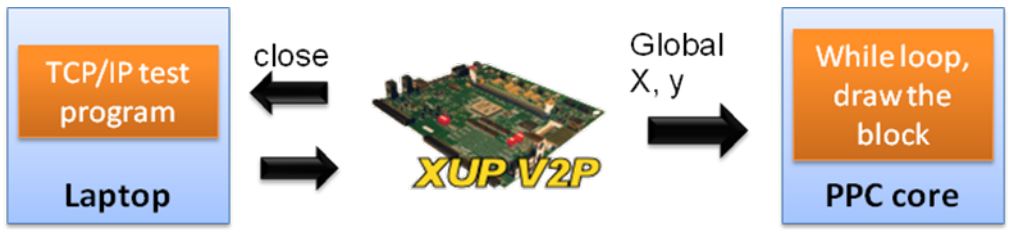
We separate this framework to two parts, one is on the PC and the other one is on the board. The Wii Remote Driver parts on laptop will be introduced in other sections. This section will introduce how the Ethernet works in the FPGA and its interface to our game.

To make the Ethernet work, there are mainly two libraries we can use in Xilinx developing system. One is light weight IP (LWIP) and the other one is Xilnet.

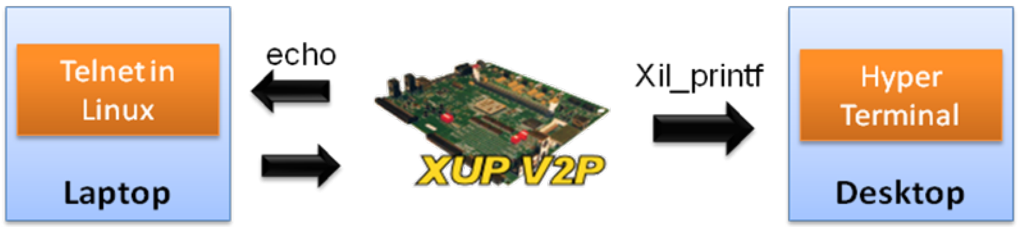
The main benefit of LWIP is that it is a higher level library and is more functional. It supports many mechanisms such as interrupt receiving mode. However, when we tried to use this library, the hardware configuration is a little bit confusing. After one week's efforts, it is still failed to work.

The Xilnet is a lower level, high efficiency and relative small library than LWIP. But it is not functional enough. The checking receiving data and interrupting mechanism is not included. And the latest version of Xilinx Developing Platform does not support this driver by default. So we modified the MSS and MHS file to port this driver into our system. Though there are so many difficulties in porting it, the example program is provided in the lab instruction, which accelerate our progress a lot.

Firstly, we implemented a while square drawing demo with a Linux GUI with TCP/IP connection. The framework is like this,



Since it is based on the lab provided http 1.0 protocol, each connection is closed automatically after echoed by the FPGA server. So it cannot be used directly in our game. Our next test demo is as following:



In this demo, the input is collected from the telnet terminal, and the FPGA can receive and echo the data it received. Also the connection is a standard continuous TCP/IP connection. In this demo, we solved a CPU blocking program by the TCP_accept function of the Xilnet library. When there are no TCP data input, this function just keeps on waiting. To solve this, we modified the EMAC driver, IP driver and the TCP driver, so that when there are no data input, the program will just skip and jump to the following part.

Based on this demo, we implemented the game controlling with the keyboard in another laptop using the Ethernet. And then the Wii data is also received successfully using this module.

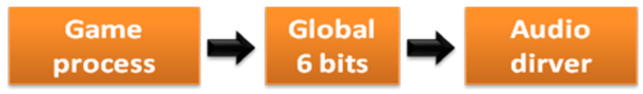
Wii Remote Interface

After the data of Wii is received, we implemented a data structure to store it. The Ethernet connection is checked at the same time as the repaint screen, so it runs at about 30Hz. However, the Wii driver may send data over 100 times per second. So we may receive more than one datagram each run. To keep the system input up-to-date, we only keep the latest X-Y coordinate input from the Wii driver. But for the button input, this will miss many valid data. So we implement a mechanism that iterate each data frame to check whether there is a button input and keep it in a buffer. And after all the date frames are checked, we send the buffered button input to the game.

Audio Driver

Basically, the audio module is a separate part of the game. We use a global structure to store the data need to be transferred between the game state and the audio driver. It is consists of the pointer to each sound effect wave file, the length of those sounds, the current state of each sound and a 8-bit triggering flag for those sound effects.

If the main game program wants to play a sound, it only needs to just set the corresponding bit to 1. And the audio driver will check the triggering bits 500 times per second. If a triggering signal is found, the sound will be mixed automatically with the current playing sound.



Another feature of this mechanism is that the triggering bits only need to be set once, and the sound will be played till end. So that the game codes do not need to worry about an out of date triggering block the new playing command. Our 6 bits flags are set as following:

1st bit: Background Music

2nd bit: Shooting Sound

3rd bit: Moving Sound

4th bit: Sound when the ship is hit by other objects

5th bit: Sound for game start

6th bit: Sound for game end

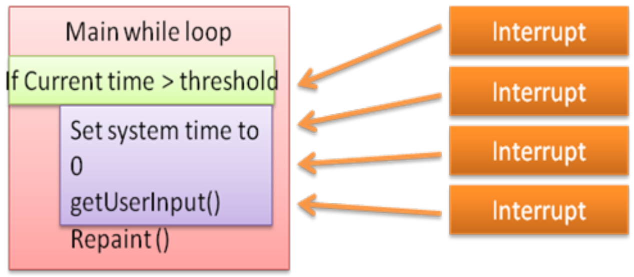
We use the programmable interrupt timer to control refresh rate of the audio module. To achieve the best audio quality we set the sample rate to 44.1k, and the bit rate is 1411kbps. In Xilinx audio driver, the FIFO size is only 512byts, which means we at least need to refill the FIFO about 400 times to keep the audio output continuously. Based on this, we build a FIFO feeder runs at 500Hz.

When we test this module separately without the game state, it works quite well. However, when we begin to porting it into our main program, we found that the FIFO writing is much slower than we expected. It even takes longer time than the repaint functions. By analyzing the codes in the Xilinx audio library, we found that this is caused by the default FIFO writing mechanism. When a new byte need to be put into FIFO, the driver will check the FIFO status, if it is full, then the driver will block the CPU to wait until the earlier data is pulled by the decoding module. To make the system runs smoothly, we tried to modify the module and change it from the waiting mechanism to a checking mechanism. When the FIFO is full, the driver will just return a full signal to notify the upper layer codes, and then the game codes will no longer try to write data to FIFO in this run of interrupt handler.

This modification is successful. The audio driver only takes a neglectable time to feed the FIFO which does not influence the main game program at all. And our calculation of the bit rate is also correct. The average bytes number written to the FIFO is 470, just less than 512, which keeps our audio driver runs at the highest efficiency.

System Time and Refresh Rate Control

There is only one programmable interrupt timer in the PowerPC core. As mentioned in the Audio Driver section, we use it for writing the FIFO. So to keep the screen repaint in a constant rate, we build another timing system based on the systemTime functions of PowerPC core.



We mainly utilize two functions in the Xilinx PowerPC library, setSystemTime() and GetSystemTime(). As shown in the diagram, in each while loop, the current system time is checked first. If it exceeds a threshold, which represents the expected interval between two repainting, then the repaint codes will be run. Otherwise, it will just skip the repaint codes. Through this mechanism, though there might be many interrupts happened between two intervals, the screen repaint rate will keeps constant.

Representation of Sprites and Game State

The representation of the game state was designed to allow for efficient collision detection and rendering, as these were determined to be the two most computationally intensive tasks associated with the game. Collision detection for every pair of objects on the screen was determined to be an unacceptable solution, given that each player had the ability to add an arbitrary number of bullets to the screen. Likewise, blindly re-calculating each pixel every time a new frame was rendered required too many memory instructions. Hence, our representation of the game was designed to minimize the amount of calculation required for collision detection, and to minimize the number of pixels which were updated when a sprite changed position.

As mentioned above, the simplest way to detect collisions would have been to check every pair of objects on the screen to see if they have collided. Unfortunately, the number of object pairs on the screen is proportional to the square of the number of objects. This means that as the number of objects on the screen grew, the number of comparisons became prohibitively large. Several methods to reduce the total number of collisions were considered, including making comparisons only between certain types of objects (ie. bullets and ships, not bullets and bullets) and various hardware accelerations. Ultimately, we decided that the most robust strategy would be restricting comparisons based on location.

In order to prevent comparisons of objects which were not remotely adjacent, it was necessary to maintain some sorting of the objects based on location. This lead to the Board data structure shown below. The Board consisted of a matrix of linked lists which store the various objects on the screen according to a hash of their current location. Objects in neighboring linked lists in the matrix are also neighboring on the screen. `CANARY_TOP_INIT` and `CANARY_BOTTOM_INIT` had no bearing on the functionality of the board, and existed for debugging and defensive programming purposes. Similarly, `obj_count` tracked the total number of objects on the board, but was not directly needed for the functionality of the board.

Found in board.h:

```
typedef struct board_struct {

        CANARY_TOP_INIT

        int obj_count;

        LinkedList board_data[BLOCKS_HIGH][BLOCKS_WIDE];

        CANARY_BOTTOM_INIT
} Board;
```

Each linked list node of the Board matrix included a pointer to the Sprite it represented. The Sprite struct was used to represent all objects on the screen, and is shown below. A Sprite struct was required to have pointers to the following functions:

- clear: remove all pixels associated with the Sprite from a frame buffer
- render: draw the Sprite on a new frame buffer
- update_state: update the state (generally position and direction) of the Sprite
- collision_detect: detect a collision between the Sprite and another Sprite
- collision_notify: notify Sprite of a collision with another Sprite
- get_rdrimg: get the RDR image currently representing the Sprite

Each Sprite was also expected to contain position and size data, a tag indicating which type of Sprite the Sprite is, and a union containing data specific to different types of Sprites. The complete Sprite struct is shown below, along with the structs which contain the data unique to each different type of Sprite.

Found in sprite.h

```
typedef struct sprite_struct {

      CANARY_TOP_INIT

      int type;

      short height;

      short width;

      short x_pos[NUM_FRAMES];

      short y_pos[NUM_FRAMES];

      short radius;

      short live;

      SpriteUnique unique;

      void (*clear)(struct sprite_struct *sprite, int buf);
      int (*update_state)(LinkedListNode *self);

      void (*render)(struct sprite_struct *sprite);

      int (*collision_detect)(struct sprite_struct *self, struct
sprite_struct *other);

      void (*collision_notify)(LinkedListNode *self, LinkedListNode
*other);

      int (*destroy)(struct sprite_struct *sprite);

      struct RDRimg_struct *(*get_rdrimg)(struct sprite_struct *sprite);

      CANARY_BOTTOM_INIT

} Sprite;


   enum {

         ASTEROID_TYPE = 1 << 0,

         BULLET_TYPE = 1 << 1,

         PLAYER_TYPE = 1 << 2,

         SCORE_TYPE = 1 << 3

   };


   typedef union sprite_unique {

         Asteroid asteroid;

         Bullet bullet;

         Player player;

         Score score;

   } SpriteUnique;


   typedef struct asteroid_struct {

         short x_vel;

         short y_vel;
```

```
        int angle;

        int destruct;

} Asteroid;


typedef struct player_struct {

        short x_vel;

        short y_vel;

        int destruct;

        int angle;

        int lives;

        int score;

} Player;


typedef struct bullet_struct {

        short x_origin;

        short y_origin;

        struct sprite_struct *owner;

        int distance;

        int angle;

        int speed;

} Bullet;


typedef struct score_struct {

        struct sprite_struct **player;

} Score;
```

By representing Sprites as a union with several properties unique to each Sprite, and several properties common to all Sprites, we were able to place distinct objects on the screen while treating them uniformly as necessary. For example, this representation allowed for refreshing the screen by simply iterating through all objects on the screen and asking them to clear and render themselves. This rendering technique also minimized the number of pixels which needed to be written to render a new frame buffer. The use of a unique rendering function for each Sprite also allowed for easy implementation of behavior such as spinning and exploding for each Sprite, and the Sprite was able to render itself according to its own, internal state.

RDR File Format and Collision Detection

Lab 1 provided support code for displaying BMP images, and for displaying only a portion of a rectangular image through the use of a black and white mask. Unfortunately, the support code applied the mask to the image each time the image was to be displayed. This resulted in considerably reduced performance since the rendering of a single pixel required two memory references and logic to decide if the pixel should be displayed or not.

In order to reduce the amount of computation necessary to render a sprite, a separate file format was developed with properties better suited to displaying sprites. The modified format has been given the file extension "rdr", in honor of the Revolving DooRs. The modified file format improved the BMP display process by storing the location of the first pixel of each row and the length of each row of pixels in the header of the file, and then storing the actual pixel data in the main data portion of the file. This eliminated the need for a double memory reference necessary to display a single pixel. In order to allow for transparency within a line, an extra byte was padded to the beginning of each 24 bit pixel value to indicate transparency.

In addition to improved memory utilization, the RDR file format greatly facilitated precise collision detection. By inherently storing the outline of the image, a pixel perfect comparison of two RDR images to determine if they were overlapping was computationally feasible. This operation required careful comparison of the beginning of a given line of one image, and the end of the corresponding line (adjusted for screen position) of the other image.

As a note regarding BMP files: There appears to be a wide range of padding schemes used in BMP files. Even BMP files authored by the same program, having the same size, did not always have the same number of bytes. Padding schemes were observed whereby each row of pixel information would be terminated by 0, 1 or 2 bytes of zero, and the entire file would be terminated by 0 or 2 bytes of zero. Despite having dissected several BMP files in a hex editor, it is not evident to this author that the padding information is directly included in the header of the file. For any future concerned parties, it seems to me the best approach is to read the file size and header offset from the header, and work backwards to determine the padding scheme which is in use.

How it was built

Data Driven Design

Although the goal of creating a version of Asteroids playable with Wii Remotes was relatively constant throughout the life of the project, the implementation changed drastically.

Initially, our team considered a design which involved porting a current implementation of Asteroids to the Xilinx board and modifying the implementation to be compatible with the Wii remote. As an initial test of the feasibility, we first modified the Asteroids code to run on Linux and receive input from the Wii remote. The initial Linux implementation worked well, but appeared to become less feasible as we moved the design to the board.

Had we successfully completed the port, there would have been many advantages to this approach. The largest advantage is that we would not have been hindered by our limited knowledge of graphics. Instead, we would have been able to take full advantage of the knowledge of the original creators of our game. Also, we would have been able to take advantage of other the other full features of the game, such as multi-player play over a network.

As we worked to complete the port, it became increasingly clear that our team was poorly matched to the task. Given that a near-miss on a port would likely produce nothing playable, we decided to re-focus our efforts. As a simple demonstration, a group member had written some code which caused bubbles to float around the screen, bouncing off the edges. The code was able to render about 20 frames per second with about 10 bubbles of size 48x48 pixels on the screen. The code was also able to preform collision detection between the bubbles, although this lead to a noticeable decrease in performance.

Noting that in principle this simple demonstration contained much of the same calculation required to implement our variation of Asteroids, our team decided to depart from our original course of action and work towards a more basic, from-scratch implementation. From this point, we began to develop a new game design which utilized concepts and skills we had developed through the attempted port to develop a new game created from scratch.

Design Partitioning

In an effort to separate the design into separate pieces which could be tackled in pairs or individually, rather than as an entire group, the design was partitioned into several sub-systems. The key sub-systems which we identified included Audio, Input (Wii remote), Ethernet, Game FSM, Graphics and Game Logic. Ultimately, these sub-systems proved to be isolated enough, that we were able to produce progress in some systems though there was delay in others.

One difficulty of our game design was that the game was not playable with a Wii remote unless there was a working Ethernet connection. This unfortunately meant that many aspects of the game could not be fully tested until a working Ethernet connection was established, which proved to be more difficult than would initially seem.

Tools and Design Methodology

Our source code was largely written using the development environment provided by the Xilinx tool chain. Although we made some attempt at using version control, this met with varying degrees of success. This was partially due to lack of familiarity with the Windows Subversion interface and with version control in general. However, the single most significant factor was that several critical aspects of the design were not easily kept under Subversion (ie. hardware configuration files), and so ultimately a messier system of "zip" files had to be used anyway.

In general, design decisions were made by consensus of at least 2 team members. There were very few decisions which were made by the entire team, but this was ultimately not necessary. Given the size of the team, involving the entire team in a design decision often involved greater complication than reward

Personal Report

Junqing Wei

Though it took us quite a long time to finish the project, this course is interesting and useful for me. I learned lots of skills of using FPGA, implementing hardware, software projects and integrating them together. I also learned how to balance and tradeoff the objective with our time and ability restriction from the game developing progress.

There are many parts I tried but failed which are not included in the main report:

1) Dual-core

I spend two weeks on this, and I almost succeed. Basically, we need two Bram controller and some additional Bram blocks to implement this.

2) IWIP

I think that the 9.1 XPS will support IWIP well, but I don't know where I made some mistake and I have to give up using IWIP since our time is limited.

3) C++

The Xilinx SDK support C++, but in our program, we give up using it just because no one in our group is familiar with the eclipse developing platform.

4) Laser Tracking

In my opinion, this is still a great idea for user input, but our team decide to use Wii Remote, so I just give up trying to implement it.

And here are the works I have done in this project:

1)	Testing Laser Tracking Algorithm in MATLAB	5h
2)	Finishing lab1, lab2, lab3	20h
3)	Implementing the first animation and audio demo and integrate them together	5h
4)	Implementing the Ethernet connection using http1.0 protocol	15h

5)	Modifying the Ethernet codes and building the demo for DR3	15h
6)	Build the first version infrastructure of our game	10h
7)	Testing on Dual-core	10h
8)	Porting audio module into the game and build the software interface	10h
9)	Porting Ethernet module into the game	5h
10)	Game State	5h
11)	Testing the Game and debug	10h

Personal Report

Brad Miller

Work Summary

Subversion and Doxygen 5hrs

- set up repository
- arranged commit emails
- compiled and created Doxygen configuration file

Labs 0, 1, 4 15hrs

Selection of initial project 10hrs

- development of initial possible game concepts
- selection of concept to implement in group meetings
- research as to the feasibility of different ideas
- reading old project reports

Wii Remote driver 10hrs

- finding suitable open-source driver
- compiling driver

Simple DirectMedia Layer 25hrs

- familiarization with code base, compiling for Linux
- modifying to include Wii Remote input over Ethernet
- moving into Xilinx build environment

Maelstrom modification 15hrs

- handle Wii Remote events, in addition to existing input forms

Planning of Xil_Roids implementation 10hrs

General Xil_Roids game state and logic 15hrs

Development of individual Sprites for Xil_Roids	15hrs
Implementation of image sequences for spinning, explosion, etc.	5hrs
BMP decoding: picking apart hex dump to determine format	10hrs
Collision detection	5hrs
RDR images	20hrs
• Linux file converter	
• batch converting BMP images	
• software to display RDR image on game screen, including on screen edge	
Integration of entire Xil_Roids game	15hrs
Design of Wii Remote user interface (calibration)	5hrs
Report writing and presentation creation and preparation	25hrs
• Final report/presentation	
• DR1 report/presentation	
• Personal weekly reports	

Feedback regarding the course:

I definitely enjoyed taking 18-545. I think the opportunity to write software and design hardware to better support the software is fantastic, and we had opportunity to do that in this course.

I think one of the most challenging aspects of the course is the wide skill range of the class, and the group nature of the project. Some people in the class had more of a hardware skill set, while other had a stronger software skill set. Successfully engaging this diverse skill set in a single project can be difficult.

I think one way to improve the class would be to have a heavier focus on the group aspect, possibly through group dynamics discussions in the lectures and a heavier focus on project management techniques.

I would like to thank Dr. Ken Mai and Dr. William Nace for teaching this class, supervising the projects, planning a public presentation and supplying us with all the hardware we needed.