CMU SMC 5873
5032 Forbes Avenue
Pittsburgh, PA 15289

December 11, 2006

Dear Professor Mai and Professor Marculescu:

Accompanying this letter is Fuggle's formal report, "18-545 Advanced Digital Design Project: Virtual Reality PacMan." This report examines the design and implementation of a first-person, eat-em-up, virtual reality game.

This report describes the process of designing the hardware, software, and interfacing of a new version of a retro game. It describes the process beginning with the system level architecture and proceeding to the hardware and software architectures. Included is the methodology for verification, techniques used to improve the design, and an analysis of the project as a whole.

If you have any questions or comments regarding this report, please contact us via e-mail at tzb@andrew.cmu.edu.

Sincerely,

Travis Brier, Astav Sacheti, and Julio Segundo

enclosure: paper entitled "18-545 Advanced Digital Design Project: Virtual Reality PacMan"
On-line Copy: http://www.andrew.cmu.edu/user/jsegundo/PacMan/

Abstract

This report describes a virtual reality Pacman game designed and implemented for 18-545, Advanced Digital Design Project. The report begins with an introduction that describes the game and the features implemented. Following that, there are more detailed explanations for each of the parts for the game. Then, the report examines the problems encountered and possible improvements in future iterations.

Introduction

At the beginning of the semester, we decided to implement to take the 1980's classic PacMan game and change it into a ramped up first-person virtual reality game.

The original concept of PacMan was a game of tag within a maze; the way to end the game was to reach every point within the maze without being tagged. This concept was implemented by four different colored ghosts and "PacMan," a large yellow ball that could eat small pellets. The game player would control PacMan to run around the game board and avoid getting eaten by the four ghosts. In addition to avoiding being eaten by the four ghosts, the game player had to maneuver PacMan to eat every pellet placed out on the board. Once every pellet had been eaten by PacMan, the level was completed and the user would progress into harder and harder scenarios, more complex mazes, faster ghosts, and fewer lives.

Fuggle took this original concept and introduced a 21st century twist: first-person playing. The original PacMan was an omniscient two-dimensioned view from overhead, Fuggle built an entire virtual world of mazes, ghosts, and of course PacMan. The player no longer looks at the entire maze from overhead; this new implementation simulates an environment that allows the player to search around the world for pellets or ghosts as if he or she were actually running around the maze grabbing up pellets.

High-Level Architecture

Below is a drawing of the system level architecture:

QuickTime™ and a
TIFF (LZW) decompressor
are needed to see this picture.

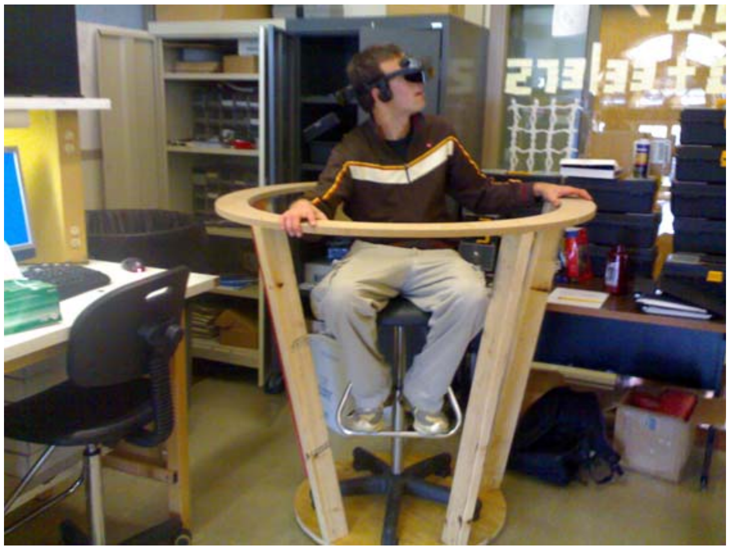
The guts of the system are at the hardware level. Contained within the Verilog code is logic that keeps track of the entire state of the game. The Verilog uses a direction input from the user, which tells the absolute direction that PacMan's facing, to keep track of where PacMan is in the maze and which spot is the next he's moving to. The Verilog keeps track of PacMan's position, each of the four ghosts (and the Artificial Intelligence which they use to chase PacMan down), and also ensures that the pellets and power pellets disappear from the game state when PacMan runs them down.

All this information is written into the BRAM, the interface between the generated Verilog logic and the software on the PowerPC Core (PPC). Both the PPC and the hardware have write access to the BRAM and consequently it is the primary method of communication between the two different devices on the board.

The PPC core, while handling communication with the Verilog Logic, also handles communication with the workstation running the OpenGL, the direction input from the chair. Essentially, it's the hub of communication between every device and it routes the signals to the appropriate devices.

The initial loading phase of the game is done by the PPC. It reads the map parses it and sends it off to the openGL workstation. Then it waits for the hardware to set up its data, and finally waits for the user to press the start key.

The game player input, coming from the chair in the diagram below, is also handled by the PPC. The VR helmet motion sensor is hooked up directly to the OpenGL workstation, which is the only part of the design that needs it input. The VR helmet has a direction tracker that allows the game player to change the viewing direction simply by changing the direction that their head is facing. The chair input is a PS/2 mouse input that is used to represent the direction which PacMan is facing. This is fed into the PPC and written to the BRAM for hardware access to the data, and then sent to the workstation as well.



The workstation is a large OpenGL engine for the entire game. The PPC constantly transmits the game state to the workstation and the workstation runs OpenGL code that translates the game state into the virtual world being viewed by the game player. The workstation is given the initial board state so the updates it is sent are only the objects that change from the initial state or previous transmission, this makes our UDP transmissions small but numerous to maintain a constant flow of data from the hardware to the OpenGL. The workstation's output goes back to the VR helmet and the virtual world is displayed for the game player.

Lower Level Architectures

PPC Architecture-

The PPC cores are responsible for many of the major functions of the game: initial map load up, interface to input controllers, and network connection to the workstation. The PPC handles the very beginning of the game initialization: the initial memory load-up.

When the game starts up, the Power PC processor runs a routine that loads a map configuration file into the BRAM. The PPC also handles one of the two inputs to the game system. There are two input controllers in our design of the game, the first is the Chair interface. The shaft of the chair is attached to a PS/2 mouse, which is polled by the board to get information about the direction of the chair. This direction is stored in a 2-bit register as an absolute direction [N=00, E=01, S=10, W=11].

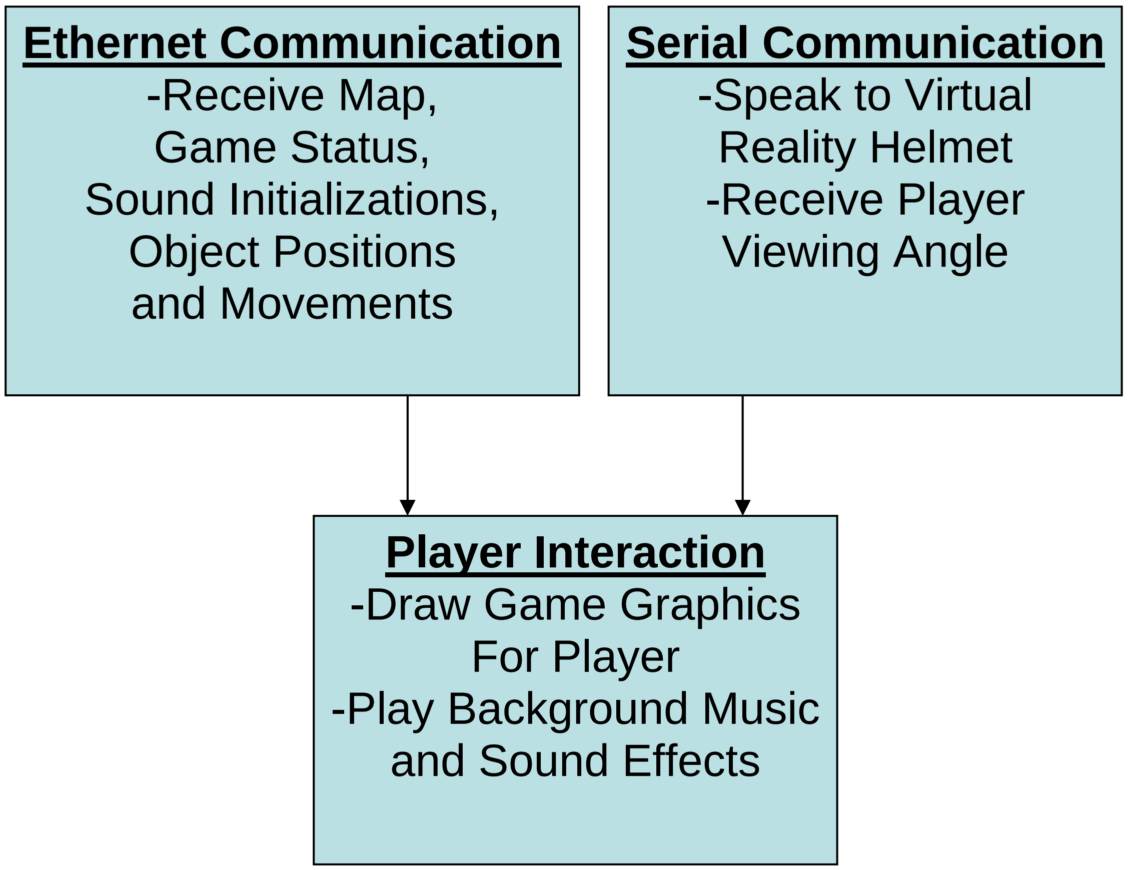
Network Connection to the Workstation-

The system uses a pseudo-UDP protocol for communication between the board and the workstation. The header used for a data-packet will be as shown below:

+	Bits 0 - 7	15-Aug	16 - 23	24 – 31
0	Source address			
32	Destination address			
64	Source Port		Destination Port	
96	Length		Type	
128	Data			

The network connection sends out the map configuration at the start of the game, and after that it sends only the updates to the five entities (pac-man, and four ghosts) to the board.

Workstation Architecture-



The role of the workstation is to act like a giant multimedia card. For this reason, the workstation handles Ethernet, OpenGL commands, audio commands and the motion sensor on the VR helmet. The Ethernet works as a form of communication between the workstation and the FPGA board controlling the game. The OpenGL is then used to draw the graphics of the game according to the received data.

Our team has decided to use a pseudo UDP stack over the Ethernet port for communication between the FPGA board and workstation. The reason for it being pseudo is due to our removal of the CRC check, and addition of an extra parameter. Since we only use a crossover cable between the board and workstation, we are assuming that no errors will occur between packets sent, which explains our removal of the CRC check. However, in order to better match the architecture of our game, we decided to add a *type* parameter, which will be used to define what the packet data should be used for. The information sent over our UDP will be a game status, initial map, map changes, pacman and ghost positions and rates of movement, and score.

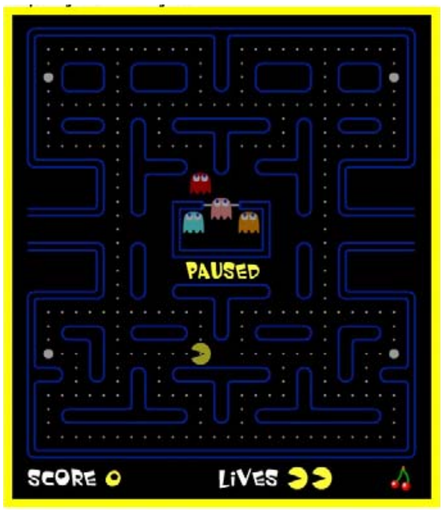
After the Ethernet forms a connection with the FPGA board and begins reading data, the workstation then knows where, what, and when to display splash screens, ghosts, pellets, power pellets, edges, walls, and floor tiles, and the score. This is all done by displaying the data received over Ethernet on the screen with OpenGL.

The map is sent initially before the game starts to the workstation from the FPGA board and stored in memory, as well as other game state variables. The map consists of currently, a 28 by 31 large grid, where each block is represented by one byte. The workstation then identifies each byte, and draws a 3 by 3 block containing that pellet, power pellet, edge, solid wall and/or floor, defined by that byte. When the game begins, and Pacman begins to eat pellets, the map then changes as the eaten pellets disappear. However, the entire map is not resent for every pellet that disappears, instead, we only receive changes from the initial map, every instance that a pellet disappears. This way the map scanner reads the most updated map available from the board, while saving bandwidth over the UDP.

A possible initial map file is shown here (this is the classic PacMan board):

```
EEEEEEEEEEEEEEEEEEEEEEEEEEEE
EPPPPPPPPPPPPSSPPPPPPPPPPPPE
EPSSSSPSSSSSPSSPSSSSSPSSSSPE
EpSSSSPSSSSSPSSPSSSSSPSSSSpE
EPSSSSPSSSSSPSSPSSSSSPSSSSPE
EPPPPPPPPPPPPPPPPPPPPPPPPPPE
EEEEEEEEEEEEEEEEEEEEEEEEEEEE
.
.
.
```

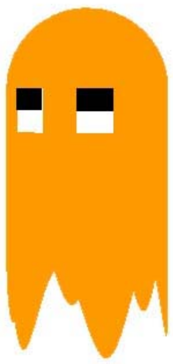
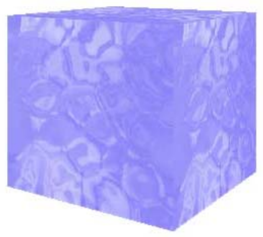
In the standard two-dimensional omniscient view, this board would be mapped to this:



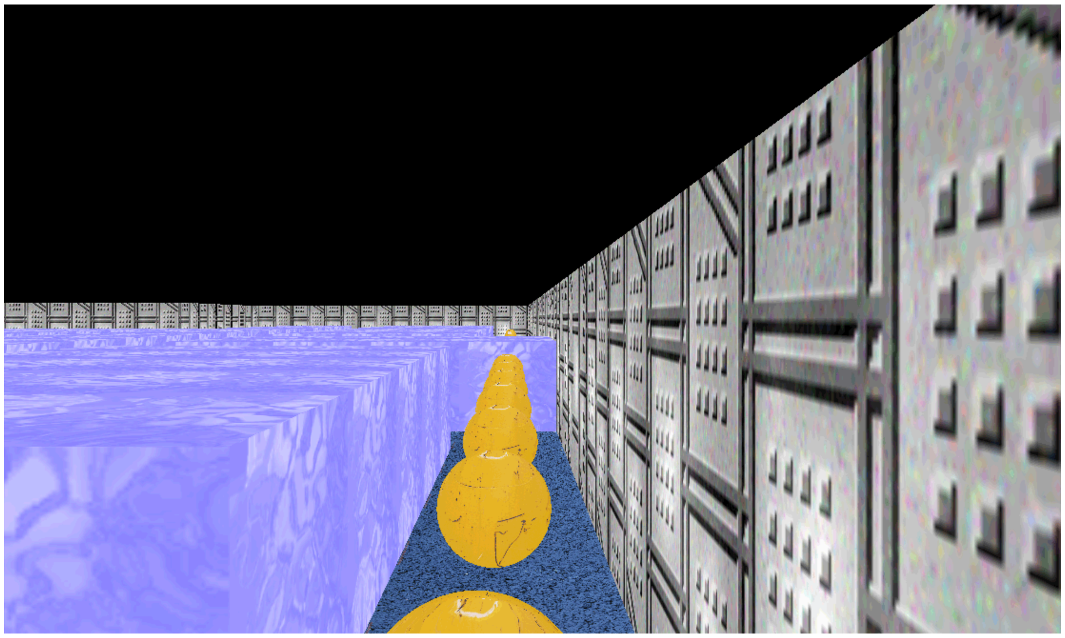
Pacman and ghost positions are sent separately from the map. Initial points are sent when first beginning the software for initial home and variable declarations, while current and destination points are then submitted while playing the game to keep Pacman and ghosts moving around the map. The OpenGL considers both Pacman and the ghost to be separate instances from the map, and will move independent of the map during regular game play.

$$\begin{tikzcd}
\text{TCP/IP} \arrow[r] \arrow[drr] & \text{loadMap()} \arrow[r] & \text{Init()} \\
 & & \text{Render()}
\end{tikzcd}$$

While time limits the amount of optimization we can do for the game, one method used is building lists. This means that instead of recreating each object (ghost, solid wall, edge, etc.), a list allows us to create each object once at the initialization of the game, and we simply re-display the same object at different locations. Doing this reduces the amount of processing necessary for each frame of the game. Shown below are several of the objects that we have included for redisplaying rather than redrawing:

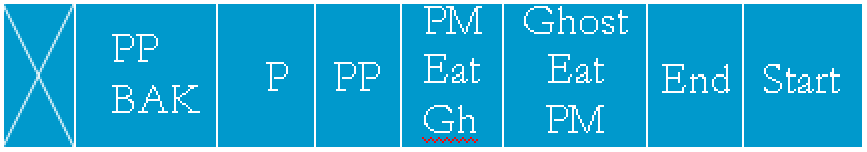
			
Ghost	Pellet and Floor	Solid Wall	Edge

Shown below is a screenshot of the virtual world that is generated by the bit map written above:



The second controller is the motion sensor on the VR Helmet. This motion sensor is also polled by the workstation and returns a 16-bit number with the direction the person wearing the helmet's head is facing.

The audio interfacing between the user and the game is also handled by the workstation. All audio files are stored on the system in PCM Stereo 48kHz WAV files, standard format for our implementation. There is a software routine that reads an 8-bit register for audio output as shown below:



X	PP BAK	P	PP	PM Eat Gh	Ghost Eat PM	End	Start

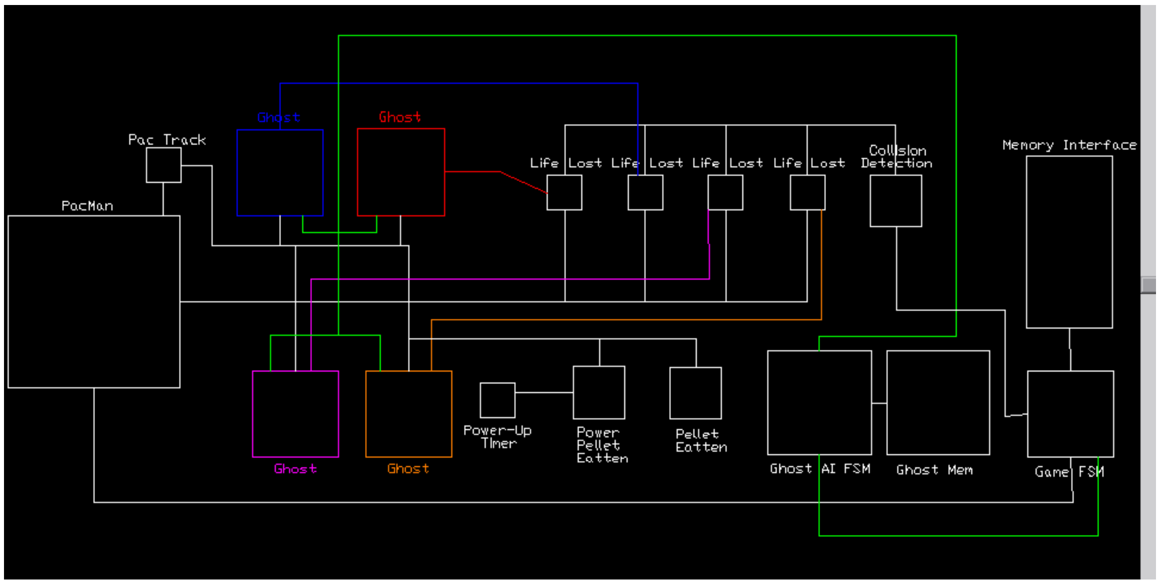
The above acronyms are as follows:

- PP BAK: Background Music when in "power" mode
- P: Just ate a pellet
- PP: Just ate a power pellet
- PM Eat Ghost: Pac-Man just ate a ghost
- Ghost Eat PM: A Ghost just ate a Pac-Man
- End: Game over
- Start: Game beginning

These bits are toggled by the hardware on the board, and sent to the workstation by the PPC core as an indication for the workstation to play the appropriate WAV file. The background music for the game plays continuously all the time (except when the PP BAK) is set to high.

Hardware Architecture-

Shown below a high level view of the hardware architecture:



Each of the modules above are instantiated within the Verilog code and written to the FPGA board to maintain the game state of the system. The forefront of the hardware would be the memory interface. The memory interface takes the communications from the individual modules who wish to know the state of a block of memory on the board and gets that information from the BRAM and transmits it back. It is the hub of all memory communications for the board.

The original plan for the BRAM was to have a large memory interfaced with the PPC and the user logic. However, due to the bus width, the problems associated with creating a large memory that could be used by both the PPC and the verilog would have been very difficult. As it ended up, the board was hardcoded into a BRAM accessed only by the verilog. There was a small, 32x4byte memory accessible by both the PPC and the user logic, this was the primary communication between the two.

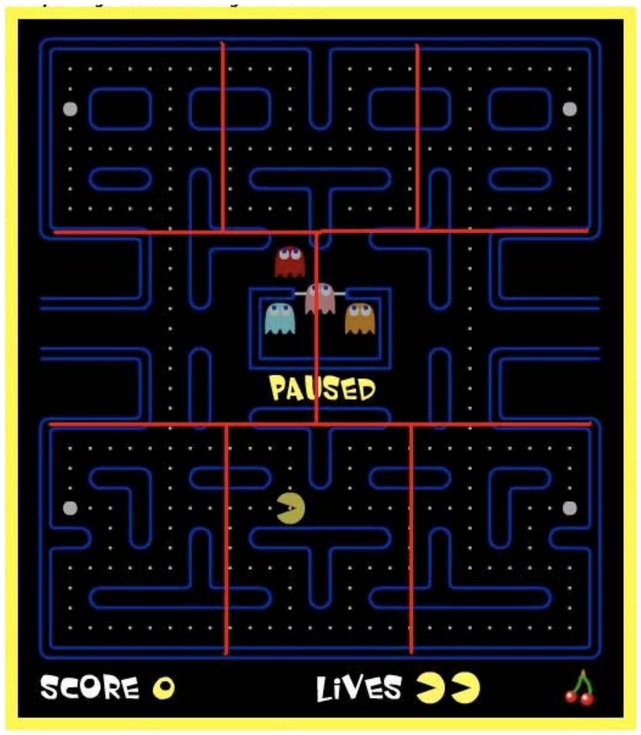
Ideally, if the memory block were able to be created, it would have been instantiated as shown above and functionally would have worked slightly different. When the Memory Interface gets a request to perform an operation on memory it receives the request in a multiple bit form. The bit encoding are as follows:

- 3:0-device ID, the device that wishes to access memory, memory interface has a map of what each device ID is associated with
- 13:4-block of board that the device wishes to access, the internal numbering scheme goes from 0-438 blocks, or if we have a larger board, it could potentially go up to 2^11-1, the internal modules only know this numbering scheme, the memory interface translates that number into the physical memory address associated within the BRAM
- 14-one bit indicating whether to read or write the data
- 36:15-Memory to be written (or 32'x if memory is meant to be read)

The memory interface then returns a success bit and if memory was read, it returns the data of the block. The memory interface consists of a synchronous FSM accessing the BRAM at a set clock rate and an asynchronous dual port FIFO. This allows for multiple devices to shoot commands at the memory interface and their data will be returned in the order in which it was received. This eliminates many complex timing issues that would have otherwise been associated with controlling multiple devices and allows for many devices to access memory at once without requests being forgotten.

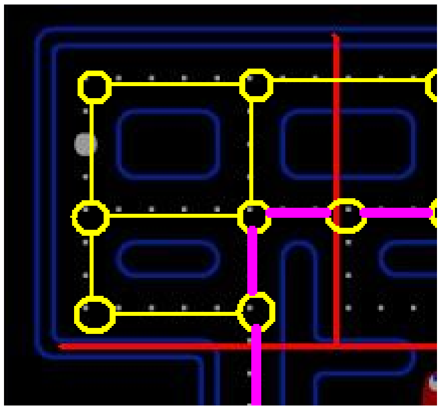
The Game FSM controls all the modules involved with the game. It sits still until the software has written all the initial game board data to the BRAM. At that point, it communicates to the Ghost AI to perform the Zoned Offense algorithm, described later in this paper. Once the Ghost AI has performed all the pre-emptive calculations, the Game FSM tells the software that everything at the hardware level is prepared to start. Upon receiving a start command, the FSM goes into the play state and keeps track of the number of lives of the user. When the number of lives reaches zero, the game FSM freezes the hardware logic and tells the software that the player has lost. The Game FSM then goes back to the awaiting start state. All communication between the Game FSM and the software is done through an eight bit BRAM logic block. The Memory Interface maintains memory of the address of this for the Game FSM communication.

The Ghost AI block provides each of the ghosts with the direction which they should go when they are chasing PacMan. Fuggle designed a simple algorithm which sets up a series of vectors that the ghosts travel along throughout the game play. The board is initially divided up into eight zones, as shown below:



Each of the ghosts has access to what zone pacman is in, this data is stored within the PacTrack Module. Depending upon what zone each of the ghosts themselves are in, they send a request to the Ghost AI to provide them with a direction to go. This is fairly straightforward except recognizing the paths to different zones can be calculation intensive and utilize a lot of hardware. To eliminate many of these time expensive calculations, a vector system was implemented. During the pre-loading phase, the Ghost

AI parses the board memory and determines where each “node” is within the board and which nodes connect to each other.



Shown above is an example of what the hardware determines. Each of the yellow lines represents a vector from node to node. When the ghost reaches a decision point, the module requests the AI to tell it which direction to turn. Based upon where PacMan is compared to the ghosts position, the AI has a pre-determined vector map that spits out a decision which will turn the ghost in the direction of the next closest zone to PacMan. If the ghost and Pacman are in the same zone, the ghost will make random movements until it either hits pacman or pacman moves out the zone and the ghost follows him to the next. If the intelligence of the ghosts need to be increased, the size of the zones can simply be made smaller. In the current implementation, the size of the zones are fixed, but in future iterations, the possibility of increases the ghosts intelligence using this system is a very plausible solution. The vector maps and use a dynamic parsing algorithm of the entire map to store it into the ghost AI memory block.

This AI algorithm did not get fully implemented into fuggle’s hardware due to time constraints. There were three backup algorithms implemented that were used during actual gameplay. The first was a “practice mode” algorithm where the ghosts were held steady within their initial starting locations. The second is “random mode” where each of the ghosts slightly favor a different direction and based on a variable input decides whether or not to turn at particular nodes. This was the algorithm used during the public demo session, it was efficient for allowing players to have an enjoyable experience without getting frustrated by constantly attacking ghosts. The final algorithm fully implemented was one simply for fun where ghosts had the ability to go through any walls and would constantly take the Manhattan distance towards pacman.

The PacMan and each of the four ghost modules all perform similar functions. The purpose of the modules is to maintain the exact position of PacMan and each of the ghosts, respectively. The standard the system uses for position calculation is fairly simple. Each module knows the initial position and the destination position of the object. To enhance the smoothness of the game, a timer is used to count in-between the block movements, thus providing a percentage of the way from initial to destination for each object. To speed up or slow down pacman or the ghosts, the only change that needs to be made is to increase or decrease the amount of time in-between each block.

The basic collision detection system used to determine if pacman is touching a ghost, a pellet, or a power pellet is mathematical and very straightforward. The system uses parameters and determines based upon the percentage through each of the blocks that the objects are, or if it's a pellet or power pellet they are fixed, and if the outside radii touch each other, a flag is thrown to either remove a pellet, kill pacman, kill a ghost, or remove a power pellet.

The power play timer determines how long PacMan can eat ghosts. During this time, if pacman and a ghost touch each other, the ghost PacMan collision detection alerts the memory to remove the ghost and rewrite it to its initial starting position. The PPC took care of the ghost position resets. This made it easier to reboot the game through the software reset.

Experimental Methods

Much of the experimental methodology was simply through trial and error. The Xilinx boards had a lot of quirks in them that weren't found until actual implementation of the different sections of our project. Much of the hardware was tested beforehand using ModelSim, however when actually loaded onto the board Xilinx seemed to use some sort of non-deterministic algorithm and odd optimization methods that caused multiple errors.

Problems

The biggest problem we had with designing the project was finishing all that we wanted to get accomplished in the amount of time available to us. By the time that we were able to efficiently use the capabilities of the board, much of the course was nearing completion.

Another problem we had was with the overall architecture of our system. We utilized the hardware to solve many problems that required a lot of logic to implement and debug, but when written in software would be just a few lines of code. Consequently, much of our hardware debugging time was wasted on small problems that we should have implemented on the PPC and we weren't able to fully implement some features that we were aspiring for (specifically really good ghost AI). Ideally, if we were to build this project again, the only features implemented in hardware would be pacman's position and each of the ghosts individual AI. The software would take care of the pellets, score, powerUp, and all the collisions.

All in all, better time management would have improved the project as a whole along with some more overhead in the initial planning phases.

Analysis and Conclusions

By the time for the public demo, the virtual reality pacman game took on the form that Fuggle had originally hoped for, a reasonable implementation that would suffice as a prototype for future iterations. The class as a whole was a good experience for having freedom and the resources to implement a really fun game. In the future it would be good to have one of the lab's to be implementing some sort of memory interface between software and hardware, that was one of the biggest bottlenecks when it came down to actual implementation of the project.

Sources Used

http://www.google.com
http://nehe.gamedev.net
Peter Nelson’s Brain
Xilinx documentation

Individual Reports – Astav Sacheti

We split up our design into three major components – Hardware on FPGA, Software on FPGA, and Software on the Workstation. Subsequently, we broke up our major tasks also according to the three components, but also helped each other with minor parts of the other components. The software on the FPGA (PPC Core) was the component that I took up and was responsible for. In addition to that I helped with the audio on the workstation, a few modules in verilog for the hardware on the board, interfacing with the motion sensor on the workstation, and few random things in the software on the workstation (mainly searching MSDN for windows versions of popular unix functions).

The software on the Power PC Core had the following main components :-

- UDP communication with Workstation
  This required finding documentation on how UDP protocol works, a lot of trial and error to see if the correct packets are being transmitted, but relatively easy code in the end. Majority of the time was spent trying to get the code to behave nicely with EDK and Xilinx. The checksums for the UDP packets was hardcoded in the code for the various packet sizes that we were going to use.
- Polling the PS/2 mouse on the shaft of the chair
  Again, this required trying to find documentation on PS/2 protocols and how to initialize the mouse to talk to it. After finding the documentation and playing with it a little bit, the code was simple and short. PS/2 protocol allows a polled mode for the mouse which is exactly what we wanted.
- Syncing the hardware on the board with the software on the workstation
  This was probably our biggest hurdle after having a working implementation of the game. Even though everything worked nicely individually it didn’t work nicely together. There were little hacks that were written to glue the hardware and software to sync properly.

I also helped write the code for playing audio on the workstation. This involved searching for functions to create threads (for background music), Play wav files on windows PCs using Visual C++, and finally testing that it all works nicely together.

The third component, also had a few modules that I wrote. There was amux, a clock divider, and the main one was the ghost AI to move around the board (written with Travis).

Individual Reports – Julio Segundo

My primary role was to handle the OpenGL that resided on the workstation external to the board, as well as any other logic that handled that OpenGL. Some of those other items would be connecting to and reading serial data received by the head tracker worn by the player of the game and receiving game status data and transmitting, over Ethernet, acknowledgement packets back to the FPGA board.

To do these tasks I had to start by learning OpenGL using in-class books and the Gamedev website mentioned in the references. This took up a large amount of time, at least 12 hours a week for a few weeks after we finished the mandatory labs. I then used knowledge from 18-345, which I was taking at the time, to implement use of the Ethernet port on the external workstation. After instantiating that, the serial port was relatively new to me, I thus used the Windows API to learn how to receive access to the Serial port and talk to the Head Tracker to learn in which direction the player was viewing when playing the game.

The IDE used on the workstation was Visual Studio 2002.

One way to possibly improve the class is making more labs optional so that students can more appropriately use the labs that will benefit their particular project. Labs that would have helped our group in particular would have been hardware oriented and Ethernet oriented labs. Other than that, I loved the class!

Individual Reports – Travis Brier

My primary areas of focus were the hardware and the physical setup of the entire game. That being said, whenever anybody in our group ran into any problems within their particular areas, the other two group members would provide their eyes and talent to help solve the problem.

For the first portion of the project, I worked in conjunction with Astav to set up the interfacing between the hardware and software. Together, we figured out all the registers set up between the PPC and the hardware and the specific bootup sequence.

The first big part of the project was making the chair setup for pacman. While it wasn't really an ECE problem, it was an interesting engineering problem nontheless and a fair amount of time went into the math behind it. The biggest problem was making something that would allow us to be moved into lab, but also not be a cramped fit. I ended up designing the chair to expand as it did for the final demo so that everybody's legs would clear when turning around in the circle. The optical decoder used for determining the direction of pacman was plotted in AutoCAD and used in conjunction with a normal mouse being decoded on the PPC (yes, I did originally want to be an architect)

However, the vast majority of the project was spent writing the hardware code for the pacman board. Essentially, the hardware took care and kept track of everything.

Pacman - The pacman portion of the board kept track of pacman's position, direction, and time between blocks. One of the more difficult parts of this to implement was the timing for the turning. Due to the discrepencies between the openGL output and the hardware state, it was decided that pacman would only turn directly on top of every block, this just added in more timing issues.

Memory – The memory module consisted of building a BRAM out of verilog logic. The memory was used to provide the rest of the hardware with data on what was in which position at what time.

Interfacing memory – This was used between the PPC and the user logic verilog. It provided the communication for the resets and signals which instantiated the main FSM to allow the logic to reset and the ghosts and pacman to move about.

Ghosts AI – The ghosts AI is talked about in the above paper. While the vector logic was written, it never fully worked on the board and consequently we had to use the dumbed down random pattern generator implemented at the last minute in order to get a game that had ghosts moving without going through walls.

FSM's – Several large FSM's were written to control the state of the gameboard, during the initial software loading phase, the hardware loading phases and the signals required to sync the hardware and software the FSM's were utilized to maintain that everything on the hardware side was controlled synchroneously.

All in all, it was a wonderful class, I think future improvements to the class would be to do a lab involving connecting the PPC to a module of verilog.