E pldan Beck (ib54)
Rohit Gupta (rg242)
Wednesday 4:30

Lab 4: Video game with analog control
March 8th and 15th, 2006

Introduction

The purpose of this lab is to design a game with the aim to deflect balls using a paddle to prevent them from hitting the side of the screen. The collisions of the balls against the walls, paddle, and other balls follow hard-ball dynamics. The paddle is controlled using an analog input, specifically a trimpot. If the balls are deflected into the bin, the score is incremented and the balls are removed from the screen. New balls enter the screen at an increasingly faster rate during game play. Balls that are left on the screen for a certain period of time are removed, and the score is incremented. If the balls hit the left side of the screen, the score is decremented, and the ball is removed from the screen. Drag is taken in to consideration when the ball moves from one side of the screen to the other. The user is given sixty seconds to finish the game.

Homework

1. *Why is the assembler* `sleep` *command used in the main loop?*

   The sleep command is used to make entry into sync ISR uniform time; it forces the ISR to start at the same time. If commented out, ISR can start different times causing glitches, such as lines not being straight on the TV. So the sleep command forces the ISR to start every twelve cycles.

2. *The NTSC video signal we will generate is not interlaced. Explain what interlacing is and how our video signal differs from NTSC video. You may want to refer the Stanford page or other links on the Video Generation page.*

   Interlacing is a method to improve the quality of images displayed on an image device without increasing analog bandwidth. This is achieved by having lines drawn at a slope, and having it be partitioned into two sections: the odd field containing the odd numbered lines, and the even field which consists of the even

numbered lines. Each frame is completed by scanning the even fields and then the odd fields. It is used when available bandwidth is not large enough to transmit full frames fast enough to prevent flicker. Because of persistence of vision, even though only one set of frames are scanned at any given time, we see the full frame.

It is different than our code, since our goal is to generate a non-interlaced black and white video.

3. *Write a macro to perform fixed-point absolute value.*

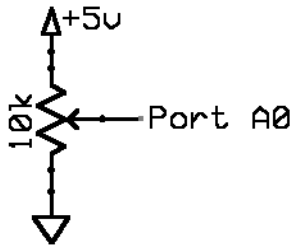
```
#define abs(a) ((a < 0)? –a : a )
```

Design and Test

Initial Setup

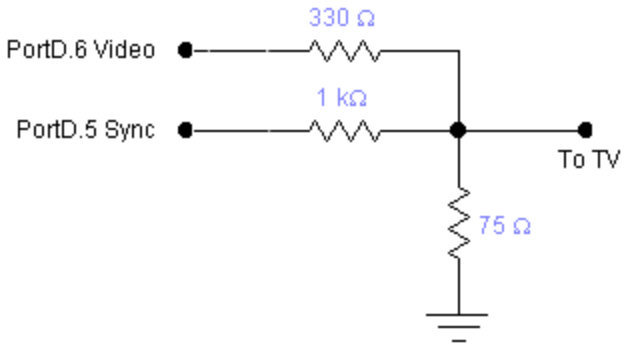
Hardware

The hardware setup consisted of the MCU in the STK500, a potentiometer wired up into the ADC of the MCU for analog control of the paddle's position in the game, and the video output wired up. The setup of the MCU was as usual. No LEDs or push-buttons were used for this project. Port A0 was used for the Analog to Digital conversion as shown in the flowing schematic:



Port D was used for the video output, specifically Port D6 and D5. These ports were wired up for the video output as shown below.

The voltage levels of .3 V and 1.0 V were used for the display of a pixel of black and white respectively. The sync level used was 0 V. The resistors used were to get the appropriate voltage levels.



Software

The software setup was the same as all previous labs except the compiler options were set to optimize for speed, the data stack size was downsized to 100 bytes, and the char was set to unsigned. The code was written mainly in C in the CodeVisionAVR IDE. Code was mainly written before the lab period and the following header files were included for the indicated purposes:

```
#include <Mega32.h>
```

This was used for the MCU defines and register names. This is the standard header that is included for projects using the Mega32 MCU

```
#include <stdio.h>
```

This was included for the purpose of a few important functions such as sprintf( ) for the purpose of copying a specific string to a buffer for displaying it on the screen.

```
#include <stdlib.h>
```

This was included for the purpose of our random velocity generator for the new balls being shot at the paddle. This was a special feature since the user could not predict the trajectory of the balls and is fun for the whole family.

```
#include <math.h>
```

This was included for the purpose of a few math functions such as `lsqrt` and others.

```
#include <delay.h>
```

This was included for the video generation ISR code. This provided the correct intervals such that the screen was centered correctly and that the new line and new frame inputs were correctly recognized.

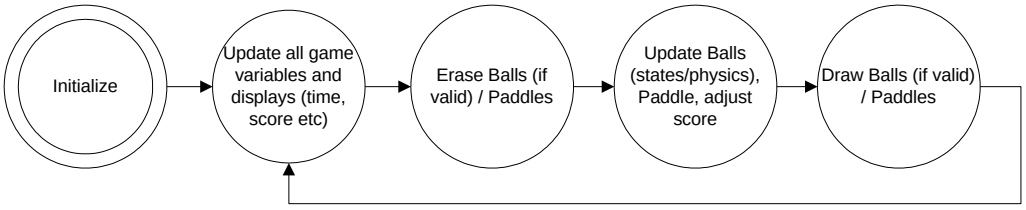
Design

Design of the Hardware

The design of the hardware was pretty much outlined in the lab assignment. We used no new or strange designs in hardware and all of the work that we did was based on software algorithms. The hardware design was as described in the hardware set up. The MCU's ADC is wired up into the potentiometer and the MCU's portD drives the video input of a NTSC television with the circuit shown above.

Design of the Software

The design of the software for this lab assignment was rather intensive and long. On a top level the following diagram demonstrates the top level view of how the program works. This includes what the main program loop had to do every time:



Our implementation was rather straightforward. We utilized a few neat tricks to make the program more manageable, potentially expandable, as well as memory efficient. These features are described in a more detailed manner below.

It is at this point where we need to split the description of the code into three sections. The first section is the one that involves the fixed point arithmetic. All of this code was standard as it was given to the class and needs little in-depth discussion but the basic functionality of the functions are explained below.

The second section is the video generation code including the ISR and frameBuffer manipulation code. This code was also standardized and given to the whole class. As we found that to achieve 16 balls on screen at one time we didn't need to hack into the assembly code the video generation code is unaltered in our project and thusly doesn't need to be explained in depth. The basic functionality of the functions, however, are briefly described for reference.

The third section is the code that is designated as game code. This includes all the rest of the code other than the code that was standardized and given to the class, or the

code that we wrote. This code is described in close scrutiny and detail and these are the functions that should be paid attention to by the reader.

Fixed Point Arithmetic Macros / Functions

```
#define int2fix(a)    (((int)(a))<<8)
```

This converts an integer into a fixed point integer

```
#define fix2intSlow(a)   ((signed char)((a)>>8))
```

This converts an fixed point integer into an integer but is not very efficient

```
#define float2fix(a) ((int)((a)*256.0))
```

This converts a float into a fixed point integer

```
#define fix2float(a) ((float)(a)/256.0)
```

This converts an a fixed point integer into a float

```
#define multfixSlow(a,b) ((int)((((long)(a))*((long)(b)))>>8))
```

This multiplies two fixed point numbers but is slow and not efficient

```
#define divfix(a,b)   ((int)((((long)(a))<<8)/((long)(b))))
```

This divides a fixed point integer a by a fixed point integer b

```
#define sqrtfixSlow(a)    (lsqrt(((long)(a))<<8))
```

This takes the square root of a fixed point number, however is slow and not too efficient.

```
char fix2int(int a)
```

This function converted a fixed point integer into an integer much faster and more efficiently than the macro.

```
int multfix(int a,int b)
```

This function multiplied two fixed point integers much faster and more efficiently than the macro version.

```
int sqrtfix(int a)
```

This function took the square root of a fixed point integer much faster and more efficiently than the macro version.

```
int sinefix(int TargetAngle)& int cosfix(int TargetAngle)
```

These functions returned the fixed point integer representation of the sine and cosines of the input target angle. These functions were never used in our code since we represented the paddle as a trapezoid since it was faster to compute

collisions for both the program and the user. This made the game much more playable and since it was acceptable it also allowed the game to be faster.

Video Generation Code

```
interrupt [TIM1_COMPA] void t1_cmpA(void)
```

This function basically makes stuff show up on the screen that should look like a pixel representation of our screen buffer, screen[1600]. To do this it generates the correct sync pulses that indicates to the TV that a new line is starting and then dumps all of the information of the appropriate screen buffer to the D.6 port. At the end of the line the ISR waits for the TIM1_COMPA flag to be thrown and the next line is drawn until the whole frame is drawn on screen. This happens very fast. After 251 lines the ISR simply outputs nothing for a specific amount of time to indicate to the TV that a new frame is about to start.

```
void video_pt(char x, char y, char c)
```

This function basically updates the screen buffer such that a new point is going to be placed at position (x,y) where the char c determines it's color: 0 for black, 1 for white, 2 inverts whatever color is already there.

```
void video_putchar(char x, char y, char c)
```

This function alters the screen buffer such that a new "big" character is placed at position (x,y) with the char c acting the same as in the function above.

```
void video_puts(char x, char y, char *str)
```

This function alters the screen buffer such that a string of "big" characters is placed originating from (x,y) where char *str is the string to be put onto the screen.

```
void video_smallchar(char x, char y, char c)
```

This function is the same as `video_putchar()` but puts a "small" character instead of big one onto the screen buffer and thusly on screen.

```
void video_putsmalls(char x, char y, char *str)
```

This function is the same as `video_puts()` but puts a "small" string of characters instead of big ones onto the screen buffer and thusly on screen.

```
void video_line(char x1, char y1, char x2, char y2, char c)
```

This function computes the proper computations to plot a straight line of arbitrary slope between the two points (x1, y1) and (x2, y2). The char c is the color of the line and behaves like before.

```
char video_set(char x, char y)
```

This function returns the color of a point in the screen buffer. It returns 0 for black and 1 for white.

Game Code

This section is organized in the most straight forward way possible. Generally in the order that main( ) calls each of the functions. It is important to note that we used an array of structures to hold all of the ball information. To minimize on the amount of stack memory needed to deal with the structures we had to implement a global pointer so that we could access one specific structure from the array at any given moment through the pointer. The same was done for the paddle as it was a more convenient way to write the code it was also more memory efficient.

```
void initBalls(void);
```

This function initialized the structure array which held all of the information about each of the balls. It initialized the hitcounter to 0. This basically allowed us to increment the hitcounter and not apply any collisions to this specific ball when the value is greater than 0. The deadtime was set to the global deathtime, this was so that during the game play as soon as the ball stopped moving we could decrement the deadtime and use it to decide when to delete a specific ball from the screen when we were trying to create a new one. All balls were set to invalid (or non existent) and their (x,y) coordinates chosen as the origin of the shooting balls (which we chose to be (115, 32) making it about centered on the y axis and near the right wall). The velocities were initialized to 0 as we didn't want the balls to move around while invalid and pop up anywhere incorrect due to some bug or other difficulty.

```
void updateADC(void);
```

This is a very simple function that basically takes the converted ADC value at ADCH and assigns it to the global Ain which refers to the input at port A0. This value is between 0 and 255 and refers to an analog control for the paddle which is done in the updatePaddle(void) function.

```
void initPaddle(void);
```

This function initializes all of the paddle structure member variables, updates the ADC for the first time and draws the paddle for the first time. It initializes the initial (x, y) position of the paddle. The radius (height) of the paddle and it's width radius. These are used for both collision testing as well as drawing the paddle. The paddle is then drawn.

```
void drawPaddle(void);
```

Drawing the paddle was done using a bitmap. This bitmap is defined at the top of the code, padbmp, and we experimented with a number of different bitmaps until we were happy with the one we were using. At first the paddle was 8 pixels tall but this seemed too much so we approximated a 45 degree trapezoidal arrangement so that the paddle collision math could be simpler and filled it in with white. The code itself is very similar to the code used to display characters on screen. It's simply scanning code that plots a pixel at the specific location based on some conditional statements.

The origin of the paddle, however, had to be kept in the middle of the paddle for physics sake so the scanning code is actually (x – p->ry) where ry was a member of the paddle structure that was set by the initPaddle(void) function to be the appropriate y "width radius" (since the paddle is oriented sideways).

```
void erasePaddle(void);
```

This function simply erased the paddle pixel by pixel. The code is the same as the drawPaddle( ) code but simply draws black where the white is.

```
void drawbins(void);
```

This was a very basic function that draws the bin bitmap twice at the proper location where the bins need to be. The code is just like that of the code that displays a character on the screen. This function only needed to be called once since our code was written such that no moving objects on the screen will ever overwrite any of the non-refreshing pixels.

```
void drawBall(char a);
```

This function simply drew the ball that was currently being referred to by the global ball array pointer *bp. The char a parameter was used to both draw and erase balls. This function was useful as it allowed us to make tiny changes to the look of the ball as well as both erase and draw using the same function.

```
void initialize(void);
```

The first thing that this function does is call the initBalls(void) function. As described above this function basically initializes the structure array of balls to the proper values. It then initializes the paddle structure as done for the balls and goes on to set the seed for the random number generator. No specific number was chosen and since it is a constant the set of random numbers generated will be the same every time the game is played and thusly the same "random" trajectories generated for the new balls shot at the paddle.

The function then initializes the ADC to the proper conversion prescaler. Timer1 is then initialized and the TIM1_COMPA interrupt enabled so that the ISR will be called and the MCU generate video. The D port is initialized to an output port for ports 5 an 6 for the generation of video. The sync constants and lineCount is initialized so that we can properly begin generating video.

Next we draw a few lines that represent our playing field and the score board. We only need to do this once since our code was tweaked such that the sidelines are never overwritten. Then the software timer is initialized and it's value placed on screen with a string. To finalize our superficial aspect of the project we call drawbins(void) which simply draws the bins on our playing field. These also only need to be drawn once since the code does not allow that part of the screen buffer to ever be overwritten. Finally the score is initialized to 0, sleep mode enabled, and ISRs cranked up.

```
void updatePaddle(void);
```

This function was rather intuitive. It's basic design required erasing the paddle in it's old position, updating the ADC, converting this to a proper paddle height range, and then drawing the new paddle. The first two are done by calling the proper functions updateADC( ) and erasePaddle( ). To convert Ain to the proper range we used a temporary display of the Ain variable on screen to

determine where the paddle should stop moving for both ceiling and floor. When the A in value was greater/lower than these we set the value of the paddle's y to a specific value that was hugging the wall but not overlapping as to not overwrite our sidelines. Much of this code is empirical since every time we made the paddle change in size this code had to be altered in a very tinkered way. By using the setting of the y value after a specific threshold system we eliminated the paddle jumping around and acting glitchy.

```
void newball(void);
```

This function was very much like the initBalls( ) function except that it searched through the ball array for any balls that were invalid. When it found one it initialized it to valid, and set it's velocity. During the next call of updateBalls( ) this ball will then be shot out at the paddle on the screen.

```
void updateBalls(void);
```

This was the meat of our game code. Although the main( ) function did most of the scripting this function took care of all of the ball physics and game scoring. This function was intensive since it had to check every ball in the ball array with every other ball in the ball array. To allow our code to run a greater number of balls at any specific time we spread the computations over two frames. To do this half the balls were computed on one frame while the other half were computed on the next. To do this the even ones were computed individually and non-even computed individually. Notice, however, that during the collision checking they are tested against all of the balls and not just even or odd ones.

The first thing that needs to be done for every ball is to erase the ball in it's old position. This was done with a call to drawBall(0). The next thing was that the position of the ball had to be updated by adding the velocity times the Δt, which was 2 in our situation since we were splitting up the computations over 2 frames.

A very common bug with collision detection is that often times when an object approaches a boundary at a great enough velocity the object will never "touch" the boundary but be present before and after it. This discreet nature of computer simulation needs to be dealt with and is done so with checking if an

object jumps over any side wall and if it does to immediately place it at the boundary instead of beyond it. This is done with every ball for all of the walls except the left wall.

Next the drag is applied to the balls by negating from the velocity a proper fraction of the velocity. Again this fraction is multiplied by Δt to make up for the computations being spread over two frames.

First we check for collisions with the walls. The previous method of velocity "jumping" ensures that if a collision with the wall and a ball occurred the position of the ball can only be at a specific place (this is why = = is used instead of >=/<=). In this section of code the ball is also checked for collision with the left wall. If this occurs than the ball's member functions are reset, it is set to invalid, and the player's score is decremented by one. Originally this was a function called resetBall( ) but because memory and pointer issues it was hard coded into updateBalls( ) when it was needed.

Second we check the paddle and ball collisions. This was done by using a bounding box for the paddle. When the ball hits the top part of the box (which is top 45 degree inclined part) the ball's trajectory is reflected across the 45 degree angle with predetermined trigonometry to cut down on computation. When the ball hits the middle of the box it's x component of it's velocity is simply negated and when it hits the bottom the same happens as for the top 45 degree incline but the normal has a negative y component. Third we check for collisions with the bins. This is done as an expanded bounding box as well as to refrain from over writing the bin pixels. Whenever a ball is detected to collide with the bin the ball is reset as before but the score is incremented one.

Finally for the collisions we check for inter-particle collisions. This accounts for the major part of the processing requirements and the upper limit on how many balls we can put on the screen at any given time. Basically, instead of using Pythagorean theorem and many annoying multiplications and square roots we decided to approximate the distance between two ball centers as the difference between their x or y positions. When this value was below the value of twice the

ball radius (br) then we decided that a collision had occurred. We used the following equation to determine the new velocities of the two collided balls,

$$\vec{r}_{ij} = \vec{r}_i - \vec{r}_j$$

$$\vec{v}_{ij} = \vec{v}_i - \vec{v}_j$$

$$\Delta\vec{v}_i = \frac{-\vec{r}_{ij}}{\left\|\vec{r}_{ij}\right\|} \cdot \frac{\left(\vec{r}_{ij} \bullet \vec{v}_{ij}\right)}{\left\|\vec{r}_{ij}\right\|}$$

The code is basically these equations with some hacks to try and get as much performance out of every line of code. For instance we use a constant for magnitude of the distance between the balls as to not get strange "explosive velocity" artifacts from it becoming smaller than 4 (since br = 2, and the distance between the centers will always be 4). Also it is important to note that we use the hitcounter system. If both of the ball's hitcounter's are 0 then we allow the collision to happen (and we set the hitcounter to hitcount which we define globally) and if either is greater than 0 we decrement the ones that are greater than 0. This allows us to turn the hitcounter system on or off by adjusting hitcounter from zero or non-zero.

Finally we draw the ball. This is only done, however, if the ball is still alive. To check this we evaluate whether it's moving or not. We then check whether it's deadtime is zero or not. If it is we reset the ball as before and increment the score and we don't draw the ball thusly making it disappear off the screen. If it's deadtime is non-zero and the ball is not moving then we decrement the deadtime by one and draw the ball. In all other cases we draw the ball.

```
void main(void)
```

This is where all of it comes together. It starts off by setting the speed to 60 and assigning t2 to this variable. This is a counter and initial counter state that is used later in main for the purpose of shooting balls faster and faster adaptively at the paddle instead of every 2 seconds. We chose to implement this as we thought it would provide more interesting game play.

The main function then calls initialize() which initializes all of the important variables and registers of the MCU as described above. Then the program enters the main program loop. Unlike previous projects we decided that the game should be over after 60 seconds which in those 60 seconds the game play increases in skill level through a more rapid rate of balls shooting at the paddle with random trajectories. The MCU is then sent to sleep mode so that the sync ISR is in uniform time. The next conditional basically allows the main program code to happen only while we are in vertical blanking. This is when we reach the LineCount of 231.

First we increment the seconds clock and update it's display in the screen buffer. Also, every two seconds we decrement the speed so that the balls are shooting more rapidly every two seconds. We then update the display of the score on the screen. This is a little tricky for negative numbers and a special symbol needs to be made out of individual pixels for the negative sign. We then decrement the t2 counter. If it reaches zero than it is reset to the value of speed and a new ball is initialized through the use of newball( ). Finally we call updatePaddle( ) which includes updating it's position through the ADC and on screen and then we call updateBalls( ) which updates all of the collisions and ball physics of the balls, draws them onscreen or resets them appropriately as explained above.

Testing, Hurdles and Solutions

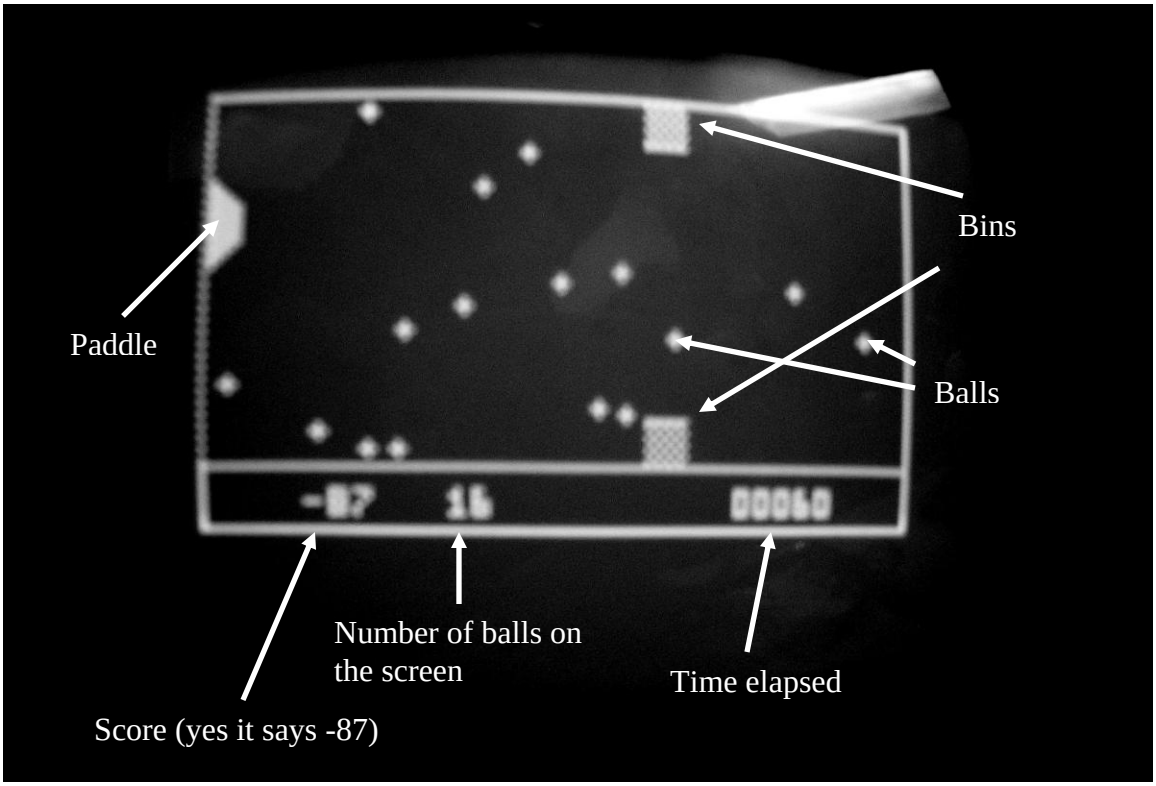
Testing our code was one of the most important aspects to the design of our project. From making sure the paddle looked symmetric to making sure splitting the computations did not have an effect on actual game, we used testing extensively to weed out every mistake we had.

At first we made sure that the ball dynamics were correct by having a couple balls bump into each other in a blank screen. Using this method we were able to see what mistakes we had in implementation of ball physics. Once this was completed we added the paddle, and repeated testing. Through testing, we were able to determine how to scale

the analog conversion to our playing screen, to make sure the paddle never went outside the boundaries that we defined.

One of the biggest hurdles we experienced was not being able to obtain the at least sixteen ball limit. The first implementation worked for about eleven balls, and after some simplification was able to get it up to twelve. In the end, we realized that we could split up the computations into two different screens without many adverse effects to game play.

Below is a screen capture of our game in the middle of play. Notice that there are 16 balls currently on screen and that the ball counter is working. The ball counter was not necessarily part of the project and not described in the code description but was a useful tool in determining how many balls were on screen instead of halting the game. It was maintained mostly in the main( ) and updateBalls( ) function.



Conclusions and observations:

This lab was successful in that all requirements of the security system were met. At first we were unable to achieve over eleven balls on the screen. However, after a suggestion from Richard, we were able to split up the computations even further and obtain at least sixteen balls. We feel that we could have made the interface more pleasing, time permitting. This includes making the balls circular rather than being crosses, and the paddle spherical. Also, we could have expanded the lab by including extras such as a top 10 score at the end of the game, and possibly make it two player game by competing for the highest score. Furthermore, some background music could be implemented with unique sounds for when a score is made and reduces. Any noises for ball-ball interaction, or ball-wall interaction would just be annoying. In general we were happy with our results, and if given more time could have made it an even futher enjoyable game to play.