MLC- IE09

The Game Development Process: Hands-On

A Major Qualifying Report
Submitted to the Faculty
of the
WORCESTER POLYTECHNIC INSTITUTE
In partial fulfillment of the requirements for the
Degree of Bachelor of Science

Submitted by:

Andrew Cote

Felix Nwaobasi

Richard Pavis

Date: 1-12-2010

Professor Mark Claypool, Advisor

Abstract

Understanding the game development process is integral to achieving success in the game development industry. As the demand to know more about the game development process increases, so does the number of outlets that provide pertinent information to the public. There are numerous books, lectures, conferences and organizations that provide information on the game development process. Although these mediums are each informative in their own right, they are still restricted in the information that they convey and often may not reflect current industry practices. The information unarticulated by these mediums could be important for job placements or advancements in this industry.

In order to gain a better understanding of the game development process, we assimilated with the development studio GDX to help develop a game for the Wii videogame console. Our tasks involved designing and developing several of the mini-games within the game. As members of the team: we contributed ideas towards the direction of the game, heard feedback from the publisher on the game's progress, and most importantly, received hands-on experience with the pressure and excitement that is associated with meeting a deadline for a game's milestone. As a result of our work, we were able to gain invaluable insight on various aspects of the game development process as well as experience the delights and disappointments that are associated with this fast-growing industry.

Table of Contents

Table of Figures

1. Introduction

The video game industry is the fastest-growing entertainment medium today (1). In fact, the recently released *Modern Warfare 2* generated $550 million in its first five days. This number surpasses *Harry Potter and Half-Blood Prince*'s[1] five-day global box-office record of $394 million as the largest grossing five-day opening for any form of entertainment (2). With the current success of the industry, interest in the game development process has piqued to unprecedented levels.

The game development process is similar to the software engineering process, albeit with a few differences. To begin with, game development is comprised of many different components, some of which are game design, programming, graphics and audio. Due to the various components that go into developing a game, one of the core tenets in game development is that games should be developed iteratively. Iterative development advocates that developers ensure that a small portion of a game works before they move on to another portion as opposed to trying to code a large portion of the game in one attempt. This makes the game less error-prone and also gives the client a chance to view a working version of the game, therefore making it easier to assess the game's progress.

One of the major elements of the game development process is the relationship between the developer and the publisher. In game development, the publisher serves as the client. The publisher provides most of the funding that the game needs and also dictates certain aspects of the game's direction. It is the responsibility of the publisher to make sure that the game's progress corresponds to the path envisioned and to determine whether to invest further resources into the project. The publisher-developer relationship is important in order to ensure that a quality product is delivered, and a substantial amount of communication must occur between both parties.

There have been many attempts made by academia and industry-insiders alike to make those interested in game development more knowledgeable of the game development process. Numerous books have been written on the subject such as *Introduction to Game Development* by Steve Rabin (3) and *Object Oriented Game Development* by Julian Gold (4). Post-mortems, which highlight the successes and failures encountered while working on a completed game, are also available to the public. An abundance of lectures have also been given by people versed in the matter and annual conferences such as the Game Developer's Conference (GDC) present the public with the opportunity to network and learn more about the industry. Game development associations, particularly the International Game

[1] http://harrypotter.warnerbros.com/harrypotterandthehalf-bloodprince/dvd/index.html

Development Association (IGDA) have become increasingly popular as well. With chapters ranging from Boston to Berlin, the IGDA gives the public the opportunity to hear developers lecture about topics in game development and speak on emerging trends.

Although these various sources of knowledge have disambiguated certain aspects of the game development process, there are certain components that cannot be captured in mere text or speech. One cannot use words to accurately describe the euphoria or stress that is associated with working on a big budget title. Furthermore, there are a plethora of questions that are best answered through hands-on experience, some of which are:

1. Do the basic software engineering principles still apply in the game development process?
2. Good programming practices such as commenting, modularity, coding, are stressed in lectures and textbooks, but how often are they practiced in the industry?
3. How well do artists and programmers communicate with each other so that both parties are productive?
4. Exactly how much power do the publishers have over the direction of the game? How significant a role does the developer play in shaping the game's direction?

These were questions whose answers could prove integral in understanding the game development process. For instance, whether or not code is commented in the industry could give one insight into how code readability is maintained; or one could learn how a game developer works around a situation where unavailable art assets are required.

In order to get the answers to these questions, we assimilated with the Massachusetts-based company, Game Developer X[2] (GDX) to help develop a game for the Wii videogame console. The game we worked on was *Ultimate Swimming*[3]. It was centered around a series of water-based sports mini-games that the player could participate in, some of which were: 'Water Polo', 'Dodgeball', 'Swimming', and 'Rock the Boat'. Production on the game was in its preliminary stages when we arrived, and since we would only be there for four months, we were given the task of integrating with the team to get the product ready for its alpha build. The primary goal of our task was to further improve the artificial intelligence in four of the existing mini-games as well as design two new mini-games that would be

[2] Due to external circumstances, the developer wanted to remain anonymous for this report. We have given them the alias of Game Developer X, or GDX

[3] For legal reasons, we are not allowed to reveal the name of the game we were working on. To compensate, we have given the game the alias of "Ultimate Swimming"

available for alpha testing. Upon the completion of our tasks, we interviewed members of GDX to obtain answers to questions that were pertinent to game development. Our goal was to make correlations between the answers to these questions and the work that we had undertaken. This project provided us with the opportunity to work on a game for a major console, and more importantly provided information on aspects of the game development process that lectures and readings could not adequately encompass.

To better explain our Major Qualifying Project (MQP), the next five chapters will be presented in the following manner:

- a Background section that provides useful background information on topics that pertain to our project
- a Methodology section that provides an in-depth look at both our experience with the game development process and a few of the features we implemented
- an Interview section which contains a transcript of the interview we conducted on members of GDX.
- an Observation section that makes inferences on the game development process based on the work we accomplished and the answers we obtained from the interview
- a Conclusion section that summarizes our findings

2. Background

For this project, our group used Lua to help develop a casual game for the company GDX. By participating in this MQP, we were able to gain hands-on experience with the game development process as well as familiarize ourselves with industry standards. In order to properly address this project, some background information is needed on these topics.

2.1 The Game Development Process

In game development, there are two key entities. The first entity, the developer, generally comes up with ideas for games and creates all of the code and art for a game. The developer tells a second entity, called a publisher, about the game in a pitch. In the pitch, the developer tries to convince the publisher that a particular game is worth funding. If the publisher agrees, the publisher funds the game and assumes executive control of the project. At this point, the publisher makes all major decisions about game content. The developer may make suggestions, but because the publisher controls the funding, it ultimately has the authority to change anything about the game it deems necessary. Occasionally, this arrangement happens in the other direction, where the publisher has the idea and seeks out a developer to bring its idea to fruition. In this case, though, the publisher still has the funding and control.

Once a game has a developer and has secured funding from a publisher, development begins. Initially, a working component of the game is created. This smaller game component does not include final art or final code, but instead serves to show that a certain portion of the game functions properly. The working component is then shown to the publisher, so that the publisher can assess the game's progress. If a publisher does not like something about the working model, the publisher may tell the developer to make specific changes. These changes along with other improvements are then reflected in the next working model. After this process is repeated several times, the final game grows to resemble completion. This is known as the iterative process.

The game is generally shown to the publisher at certain milestones, which are deadlines where certain objectives about the game need to be met. For instance, perhaps basic pathfinding AI should functional by the first milestone while combat AI might need to be functional by a later milestone.

Milestones generally tend to be a few weeks or months apart. They serve as a means for the publisher to monitor whether the game is still worth funding. At particular milestones, a publisher may expect a vertical slice. A vertical slice is an in-depth prototype where all aspects and features of the game are viewable.

After the game has reached a certain level of development, it enters alpha testing. At this point, known as alpha build, the game has all of its main implementation complete with no major features scheduled to be added later on. The game is tested to make sure that it functions as the publisher and developer expect. Upon completion of alpha testing, the game is further polished and then enters beta testing. Beta testing, which is usually done by potential customers or a testing team, is meant to find any remaining bugs or aspects of the game that need to be modified before the release of the game. Once beta testing is complete, the game is released and sold in stores, for download, or for subscription. The game may also require support after release, such as expansions or patches, which are created by the developer and distributed by the publisher.

If at some point the publisher feels that the game is not headed in the direction desired, or that the market for a game is no longer strong enough to justify continued development, the developer can stop funding the game. In this case a game is cancelled, and the developer ceases work on that game.

2.2 Casual Games

Over the past two years, the Nintendo Wii has sold more units (5) than both the Xbox 360 (6) and the Playstation 3 (7). Many feel that the Wii's sales success is attributed to the innovation of its controller, the Wii Remote. The Wii Remote differs from traditional controllers due to its infrared sensor and accelerometers. With the Wii Remote, Wii games are able to track the movements of its player's arms. This motion control has garnered the interest of a number of people, many of whom do not often play videogames. Because the Wii draws in users who are not comfortable playing games of higher complexity, a large number of more casual games are purchased for the Wii. As more of these casual games are purchased, companies develop more of them in order to satiate the growing demand.

Typically, videogame players can tell the difference between hardcore and casual games. In hardcore games, the player typically takes on the role of an in-game character and engages in some kind of quest. Examples on the Wii might be *Metroid Prime 3: Corruption*[4] or *No More Heroes*[5]. The player usually makes progress in the game each time it is played and moves forward through a storyline. This storyline is one of the main differences between hardcore games and causal games. Most casual games have no consistent storyline. That is, a player can sit down to play a casual game, and at the end of the play session, have made little-to-no progress towards any long-term goal. The player is unlikely to have taken the role of a fictional character, or to have made decisions with long-term consequences in the game. An example of a casual game on the Wii would be *Wii Sports Resort*[6] or *Mario Party 8*[7]. Both of these games are comprised of a number of mini-games.

Mini-games are one of the main features of casual games. A mini-game is a small game that exists within a larger game. *Wii Sports Resort* is comprised entirely of these mini-games with no real larger game. This formula of making games which are comprised entirely of smaller mini-games has shown itself to be successful in a number of Wii titles, such as the *Raving Rabbids* (rabbids.us.ubi.com) series, or *Carnival Games* (www.2kgames.com/2kplay/carnivalgames/).

Because mini-games are small, they can be hosted on the Internet. Often small Internet games are referred to as Flash Games. They are given this name because they are often programmed in the ActionScript language. Games like *Alien Hominid* (www.alienhominid.com) and *N* (www.addictinggames.com/ngame.html) have been recognized as good Flash Games, and have since been made into full-scale and published videogames that are sold in stores. Because Flash Games are hosted entirely on the Internet, they can be updated regularly, and players will see those updates instantly. Massively Multiplayer games like *World of Warcraft* (www.worldofwarcraft.com) are also updated regularly and hosted online, but these games are large in scale. They are extremely expensive operations to develop and support, and they cater to hard core gamers instead of casual gamers. Webkins.com hosts a game where players can take care of a virtual pet. In order to access this game, though, the player must buy a stuffed animal with a code on it.

Our project, the *Ultimate Swimming* Project, is made up of a collection of casual mini-games.

[4] www.metroid .com
[5] nomoreheroesgame.us.ubi.com
[6] www.wiisportsresort.com
[7] www.marioparty.com

2.3 Programming in Lua

As a developer, GDX does a large amount of programming. When working on the Wii, a large part of this programming is done in a scripting language called Lua. Lua is similar to C++ in most ways, but it is different in a few key areas. First of all, the syntax in Lua is very loose. For instance, in Lua, variables are not typed. That is, one can declare a variable without specifying whether it is a string, integer, double, or function; Lua determines this at run-time. The second large difference in Lua is the table. A Lua table can be thought of as a hashtable whose keys and values can be of any type. These tables, which can be read by other Lua or C++ files, are useful for keeping data organized. The third and major difference is that Lua is interpreted, rather than compiled. That is, after editing a Lua file, the programmer simply has to save the file and run it as opposed to having the computer build a separate executable.

Aside from these distinctions, Lua is very similar to C++. It has the ability to implement objects and classes, as well as the ability to import and inherit from other classes. Lua's advantages lie in its simple, flexible syntax, and the way that it does not need to be rebuilt at every modification. This makes it faster for development and easier to use in many cases than C++.

Although Lua provides many benefits, certain limitations such as its lack of pointers, make it difficult to code an entire game in the language. To compensate for Lua's shortcomings many game developers code specific portions of a game in a language like C++ and use a library called Luabind to expose certain C++ code to Lua and vice versa. With Luabind, certain C++ functions and classes can be made usable by Lua and certain Lua tables and functions can be used by C++. This library makes it possible to combine the benefits of Lua and C++, thus providing programmers with the tools to write code in an effective and time-efficient manner.

3. The Development Process

This MQP was an excellent opportunity to put into practice all we had learned in our classes, as well as learn things we had not. GDX's Studio Head made it clear we would integrate with the team as new engineers. While we were not working on the payroll, our work would still be counted as official code if it reached a standard of quality. Our supervisor, Mr. X[8], performed the task allocation. He took time to talk with us, asking what we would be interested in working on, and where our talents would be most useful to the team and the project as a whole. We soon discovered that our work on *Ultimate Swimming* would consist of developing a series of independent mini-games concurrently. This allowed each of us to focus almost exclusively on two mini-games, except when we would need to help a partner clarify or solve a problem.

Concurrent development was possible because each mini-game was entirely separate from the others, except for a planned character persistency throughout the game. Each of us worked on separate tasks within our mini-games. We discussed what we were interested in, and each of us took the mini-games that were most interesting to us, and matched our skills. Within each mini-game were a series of tasks to get the mini-game completed. Each game came in various stages of completion; some were partially complete while others were still in the design phase.

As development progressed, the game would evolve or change based on the publisher's feedback and the ideas of the developer. The most notable case was with the mini-game "Obstacle Course." It was made clear at the beginning of development that this particular game was still in the design phase, and nothing dealing with its form or function was decided. We would often have to wait for decisions to be passed up the chain, and then wait as the publisher was unable to be contacted or gave vague answers.

Meetings were somewhat sporadic during our development on *Ultimate Swimming*. GDX has a concept of a "Daily 15" which is a daily meeting, 15 minutes long, to discuss all the pertinent issues affecting the game. While the meeting was called the "Daily 15" there was no guarantee it would be held every day. Often the producer would be out of the studio, or a meeting simply wouldn't be necessary. Our schedule did not allow us to be at GDX every day, so we simply were not able to attend every "Daily 15" that took place. As our time progressed, it became clear a more effective method was

[8] Our supervisor asked to remain anonymous in our report. For the remainder of this report, he will be referred to as Mr. X

to work through Mr. X. He would meet with us at the beginning of each day, and we would keep him apprised of our progress throughout the day. Then, at project meetings Mr. X would be able to speak as to our progress.

Our first issue was clearly the Lua programming language itself and the development environment. We were using tools we had never seen before, and even though we grasped the fundamental concepts quickly, there was still an irreducible learning curve. For example, Lua is a powerful high-level scripting language, with strange syntax conventions. To familiarize ourselves with Lua, we read the tutorials that were present at http://lua-users.org/wiki/TutorialDirectory. Due to straightforward examples, we were able to begin programming quickly, but even after a few weeks, there would be an occasional error due to our unfamiliarity with a Lua. As for our software tools, we received them as we needed them, so sometimes we would be faced with a problem we couldn't solve because we didn't have the proper program, or files.

Two of the tools that we became familiar with were SumoTool and Perforce. SumoTool is a program used to work with 2D graphics and heads-up displays (HUDs). The other tool, Perforce, is a revision control system which was used for content management across all of the projects we worked on, as well as across our own mini-games.

One major problem was our inability to implement Wii gesture controls. A significant feature of the swimming mini-game was going to be the integration of the Wii Remote gestures. However, we found ourselves waiting on hardware and software, as we did not have the engine libraries or the Wii Remotes and Bluetooth modules necessary to begin development. These issues however did not stand in the way of our development. Because of concurrent modular programming, at any given time each of us had two mini-games to work on, thus minimizing time wasted waiting for a crucial item.

When a milestone was reached, submitting our work was not a formal affair for our team. As we would contribute to the codebase we would keep Mr. X apprised of our progress. As milestones approached, we were warned to not submit any new work that may have errors. The team would perform test builds throughout the day to make sure features were properly implemented and no bugs were present.

The best way to answer questions we had about the game development process was through integrating ourselves with the GDX team and writing code with them. This gave us the opportunity to observe coding practices, learn about tools used in the industry, and identify important aspects of working in the game development process. We worked on six mini-games, each of which provided us with insight into certain facets of the industry.

3.1 Polo

The basic backbone for the Artificial Intelligence (AI) in the Polo mini-game was in place when the code was given to us. The AI was implemented using finite state machines (FSM). When we started there were 5 basic states that the polo players could be in:

- *Offense*. While in this state, the player moves towards the opponent's goal, in an attempt to score. The player who has possession of the ball is the only player who can be in this state; all other players would be in the 'support' state.
- *Defense*. While in this state, the player stays by the goal in order to prevent the opponent from scoring. If an opposing player has possession of the ball, and is within a certain distance of the player, then the player in the 'defense' state will attempt to steal the ball.
- *Support*. While in this state, the player moves towards the opponent's goal in an attempt to position themselves in an opportune spot to receive a pass or score a goal.
- *Cover*. While in this state, the player attempts to cover any player that is in the 'offense' or 'support' state. If an opposing player has possession of the ball and is within a certain distance of the player, then the player in the 'cover' state will attempt to steal the ball.
- *Seek*. The player only enters this state when no one on either team is in possession of the ball (i.e. loose ball). When in this state, players move towards the position of the ball.

Figure 1 demonstrates how the different AI states in "Polo" transition from each other.

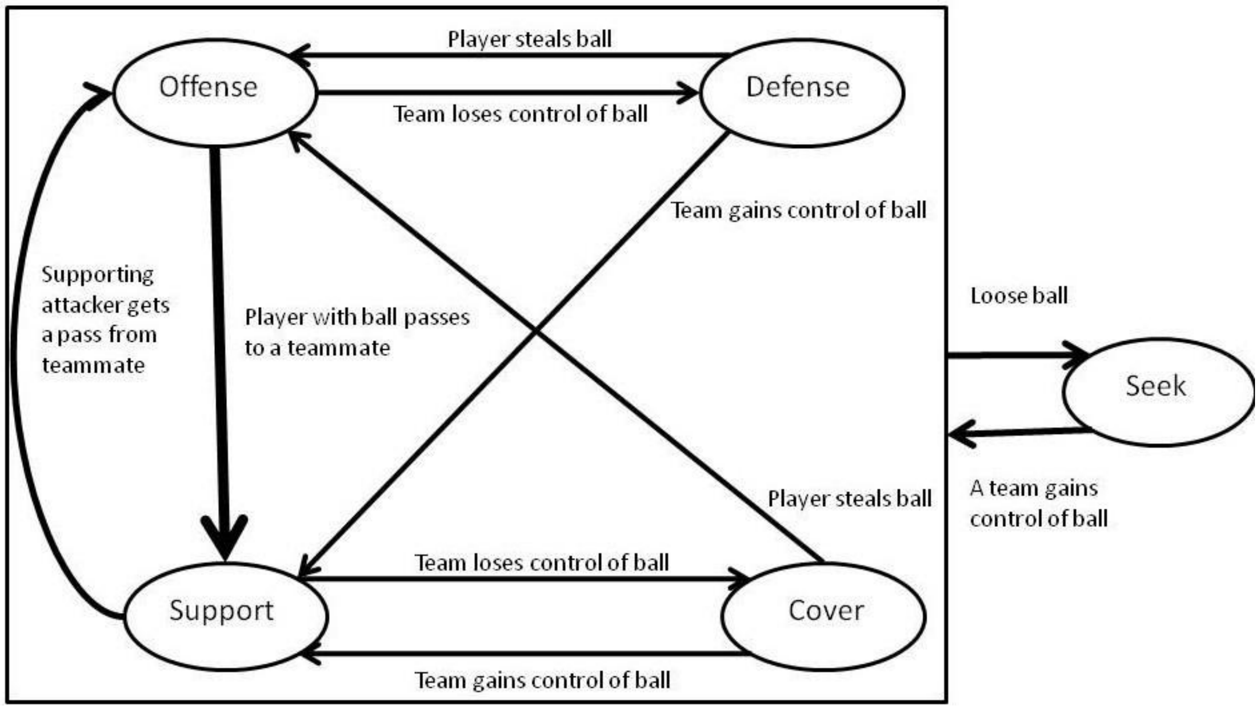

Player steals ball
Offense
Defense
Team loses control of ball
Team gains control of ball
Supporting attacker gets a pass from teammate
Player with ball passes to a teammate
Loose ball
Seek
A team gains control of ball
Player steals ball
Team loses control of ball
Support
Cover
Team gains control of ball

Figure 1: Diagram that represents the FSM used in 'Polo'

The 'state' of the player is analogous to the 'mindstate' of human beings. For instance, if a player is in the 'offense' state, then their primary focus would be on scoring a goal, if they were in the 'defense' state then their primary concern would be preventing a goal from scoring, etc. A state machine works by alternating the state of each player based on a change in their environment. For example if a player is on offense and its team loses control of the ball, then its state would switch to defense.

Before we arrived at GDX, the states were not at the level of complexity described above. For example, the 'defense' state and the 'cover' state were practically identical; this lead to awkward defensive positioning on the field and minimum defense by the goal. Furthermore, the fact that players were unable to steal the ball from another opponent made gameplay monotonous and uneventful. These deficiencies in character behavior, along with others, hampered the illusion of intelligence in the "Polo" mini-game. In order to rectify this situation and get this portion of the game ready for the alpha build, significant changes needed to be made. To accomplish this task, we created a goalie player whose job was to prevent the other team from scoring. To add proper functionality to the goalie, we added a 'GoalTend' state which caused the goalie to interpose himself between the ball and the goal when a player from an opposing team got within a certain distance of the goal. We also implemented a feature that consistently made the goalie track on the ball. This further added to the illusion of intelligence from the goalie. The goalie was then programmed to intercept the ball from an opponent when that opposing player attempts a shot on the goal. These features made the game more interesting and made scoring more difficult than it originally was. It should be noted that the ‘GoalTend’ state was the only state that the goalie could be in and that no other players besides the goalie could enter this state.

The following portion of code updates the goalie’s behavior every frame:

```
function tAI:UpdateGoalTend(fElapsed)
      -- goalie position.
      local PosX, PosY, PosZ = self.Player.Object3DData:GetPosition()
      local BallX, BallY, BallZ = PoloGame.ball:GetPosition() --ball position
      -- net position
      local NetX, NetY, NetZ = self.Player.TeamInfo.Goal:GetPosition()
      .
      .
      .
      -- if an opponent gets within this distance of the goal, move towards
that player in an attempt to steal the ball.
      local NoScoreZone = 150.0

      -- if the goalie is directly in front of the ball, and the ball is
      -- being carried by an opposing teammate, then grab the ball
            if    (DistanceBetween(PosX, PosY, PosZ, BallX, BallY, BallZ) <
            60) and (PoloGame.ball.State == BallStates.CARRIED) and
            (PoloGame.ball.AttachObject.TeamInfo != self.Player.TeamInfo)
            then

            PoloGame.ball:Stop() -- stop ball’s velocity
            PoloGame.ball:Attach(self.Player) -- attach ball to goalie

            -- put the goalie’s team on offense, opposing team on defense
            PoloGame:BallSwitchedHands(self.Player.TeamInfo)



            -- if the ball is being carried by the goalie then find the
            closest teammate and pass him the ball
            elseif(PoloGame.ball.AttachObject == self.Player) then

            local myTeammate = self:GetOpenTeammateClosestToOwnNet()

      -- if there is a teammate available, then face them and pass it to them
            if (myTeammate > 0) then
                  MatePosX, MatePosY, MatePosZ =
            self.Player.TeamInfo.Players[myTeammate]:GetPosition()
            self.Player.Object3DData:FacePoint(MatePosX, MatePosY, MatePosZ)
            g_TempVecOne:SetVector(MatePosX - PosX, MatePosY - PosY, MatePosZ
            - PosZ)
            g_TempVecOne:Normalize()
            g_TempVecOne:SetLength(500.0)
            PoloGame.ball:Throw(g_TempVecOne, BallStates.GOALTHROW)
      -- else throw the ball towards the center of the field
            else

                  g_TempVecOne:SetVector(0 - PosX, 0, 0 - PosZ);
                  g_TempVecOne:Normalize()
                  g_TempVecOne:SetLength(300.0)
                  PoloGame.ball:Throw(g_TempVecOne, BallStates.GOALTHROW)
            end
            .
            .
            .
end
```

With the work done modifying the present states in “Polo” as well as the work done to implement a new AI state for the goalie, it became apparent why the state machine structure was ideal for this type of casual game.

Major changes were also made to the other team players. The first of which was adding functionality for stealing the ball from an opponent. The main problem that arose when trying to implement stealing was determining how often players should be able to steal from each other. It would be bad game design to make the players steal the ball every time an opposing player got close to them, because it would have made the game too difficult and in turn made the game less fun. In order to work around this, we devised a rudimentary but efficient implementation that used a random number generator to determine whether or not the player would be able to steal the ball.

Below is the code for stealing:

```
      -- if the ball is being carried and a player gets within a certain
distance of the ball, then attempt to steal
      if (DistanceBetween(PosX, PosY, PosZ, BallX, BallY, BallZ) < 55) and
(PoloGame.ball.State == BallStates.CARRIED) then
-- the string canSteal is what determines if a player can steal the ball.
-- if a four character string is equal to “3121”, then a steal will be
-- successful. If not, then the string is set back to “” and four characters
-- that range in values from 1-3 are chosen at random.
            if (string.len(self.Player.canSteal) == 4) then
            self.Player.canSteal = ""
             end
            -- choose random number from 1-3
             self.Player.canSteal = self.Player.canSteal..(math.random(3))
      if (self.Player.canSteal == "3121") then
                  if(PoloGame.ball.AttachObject:GetState() ~=
                  AIStates_Polo.GOALTEND) then
                  PoloGame.ball:Stop() -- stop ball velocity
                  PoloGame.ball:Attach(self.Player) -- attach to this player
                  self.Player.TeamInfo:SetToOffense()
                  self.Player.canSteal = ""; -- set can steal to empty
```

```
    end
```

The rest of the changes made to the AI in Polo were minor changes to make the players more realistic. Code was implemented to make the players pass the ball to a teammate if they were being overmatched by the defense. To feign this type of intelligence, we had the player who was in control of the ball check if two or more opponents were within a certain proximity. If they were, then the ball would be passed to another teammate.

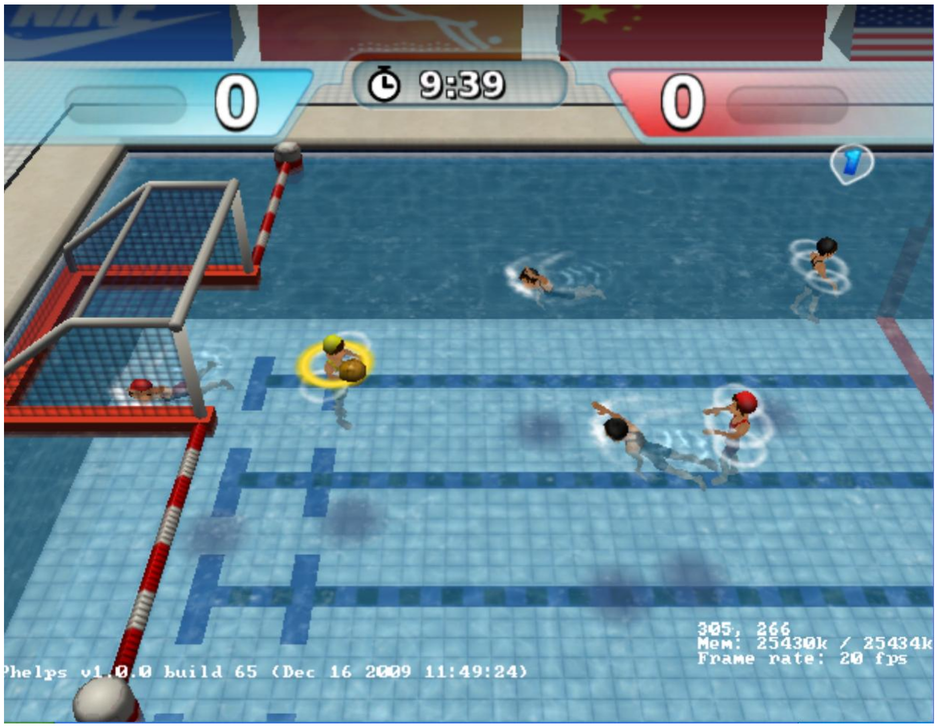
Formations were another behavior that was added to the AI present in “Polo”. There were to be five different formations that the teams could use to move up-field. The players would pick which formation they would travel in based on certain environmental factors such as the score of the game, the amount of time left in the match, etc. Each formation would provide the team with strategic advantages. Some formations would be defensively beneficial while others would be more offensively beneficial.

In conjunction with the AI, there were also small non-related AI components of the Polo mini-game that needed to be improved. Most importantly, an interface needed to be designed. When we first started working on the Polo mini-game, there was nothing that kept track of the score or how much time was left in the game. To rectify this, an interface was designed using Sumo Tool where the score of each team would be monitored, as well as the amount of time that was left in each match. A buzzer sound was also added to provide an audio cue to the player whenever a goal was scored.

Although there were no major issues that arose while coding the AI for the polo portion of the game, there were minor problems that presented themselves. While coding the “Polo” portion of the game we discovered that Lua ran slowly when it had to deal with local tables (tables that were created inside a function). This meant that whenever a local table was created in a function, a significant decline in the framerate would become visible. This not only made the game seem glitchy, but it also hampered the overall experience. In order to work around this issue, local tables were only used in situations where it was imperative. If it was not absolutely needed, then global tables were used or another data type that served as a worthy substitute.

The modifications we made to the “Polo” mini-game were all geared towards making the game more enjoyable. Whether it was adding the functionality to steal or implementing code that allowed the polo players to move in certain formations, each change made was implemented to make the AI behave more realistically and thus increase the game's fun factor.

Figure 2, shown below, is a screenshot taken from the "Polo" mini-game.



```
Phelps v1.0.0 build 65 (Dec 16 2009 11:49:24)
```

```
305, 266
Mem: 25430k / 25434k
Frame rate: 20 fps
```

Figure 2: Screenshot from the "Polo" mini-game

3.2 Dodgeball

The "Dodgeball" mini-game was a lot further down the line in development when we started modifying the code and many of the standard dodgeball mechanics were already present in the game. Similarly to the "Polo" mini-game, the AI in "Dodgeball" was implemented using state machines. There are three states that the dodgeball players can be in, and they are the following:

- *Offense*. While in this state, the player will throw the ball at an opposing player. A player can only be in the 'offense' state if it has possession of the ball.
- *Dodge*. While in this state, players will either attempt to avoid a ball being thrown at them or attempt to make a catch. If the ball is caught, then a teammate who was no longer in the game, would be able to return.

- *Seek*. In this state, the player moves towards the position of any ball at rest in hopes of obtaining it.

A diagram representing the interaction between states is shown below in Figure 3.

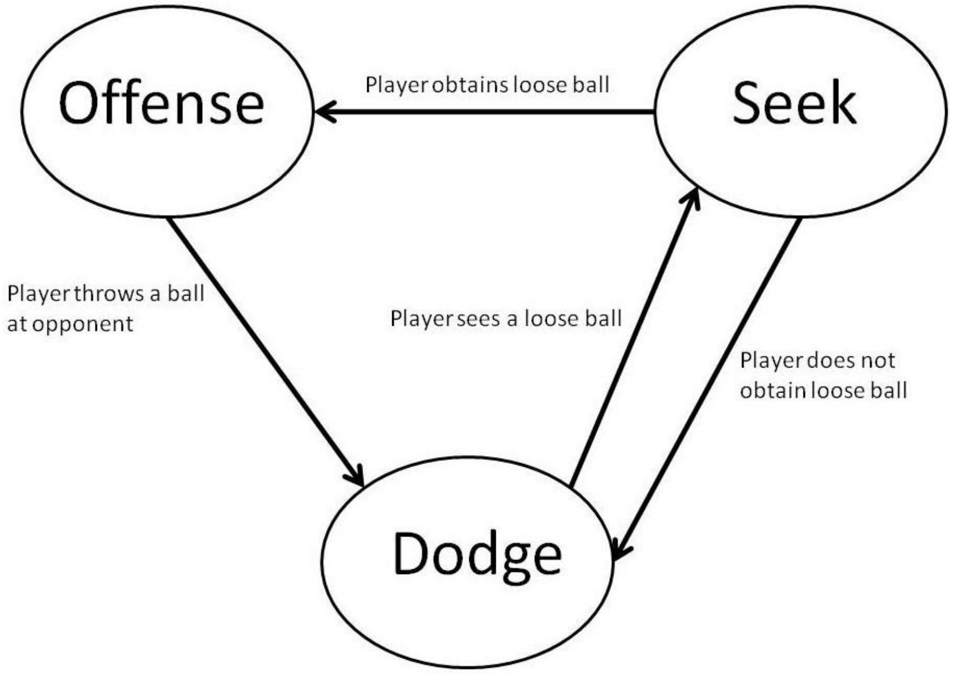

Offense
Player obtains loose ball
Seek
Player throws a ball at opponent
Player sees a loose ball
Player does not obtain loose ball
Dodge

Figure 3: Diagram that represents the FSM used in 'Polo'

Being that the basic functionality of Dodgeball was in place, it was our duty to improve upon the existing model and make the AI behave more realistically. There were a few tasks that were paramount to improving the AI of the dodgeball players. In order to do these things we decided to add a few extra features to the iteration of Dodgeball that we were working on. We added functionality so that if a player had a ball in hand, it would be able to deflect any incoming balls that were thrown in their direction. This not only added another strategic element to the game but also made the game more similar to real-life dodgeball. We then drafted team strategies that would add another layer of dimension to the game. One of these strategies was code that would have opponents throw two balls at one player at the same time if it was determined that they had a good chance of getting that opponent out of the game.

The code for this strategy is shown below:

```
function tTeam:TwoAtOne(Player, Player1)
      local OpposingTeam = Dodgeball.Team1
      if (self == OpposingTeam) then
            OpposingTeam = Dodgeball.Team2
      end
      local PosX, PosY, PosZ = Player:GetPosition()
      local closestEnemy = 1000000
      local test
      local thisPlayer = 0
      --finds the closest opponent, and sets that player as the target.
      for i = 1, NUM_DODGEBALL_PLAYERS do
            if (OpposingTeam.Players[i].bInGame == true) then
                  TargetX, TargetY, TargetZ =
                  OpposingTeam.Players[i]:GetPosition()
                  test = DistanceBetween(PosX, PosY, PosX, TargetX, TargetY,
                  TargetZ)
                  if test < closestEnemy then
                        closestEnemy = test
                        thisPlayer = i
                  end
            end
      end
-- set both players' state to offense
Player:SetState(AIStates_Dodgeball.OFFENSE)
Player2:SetState(AIStates_Dodgeball.OFFENSE)
-- have both players throw the ball the the same teammate
Player.AIController:ThrowBallAtEnemies(OpposingTeam.Players[thisPlayer])
Player2.AIController:ThrowBallAtEnemies(OpposingTeam.Players[thisPlayer])
end
```

Other strategies included having the AI strictly attempt to catch the ball if they were outnumbered by the opponents and having a player who had a ball in their hands stand in front of a defenseless player – and behave as a shield – as the defenseless player attempted to retrieve a loose ball. Unfortunately, we did not have enough time to implement these strategies.

Working on "Dodgeball" further reinforced the notion that iterative development is the ideal way to develop any type of software. For instance, in order to make the AI enough of a challenge to the player, we focused on adding one new strategy at a time, until it was determined that the player and the AI were evenly matched up. This provided for code that was easier to follow as well as less error-prone.

Figure 4, shown below, is a screenshot taken from the “Dodgeball” mini-game.

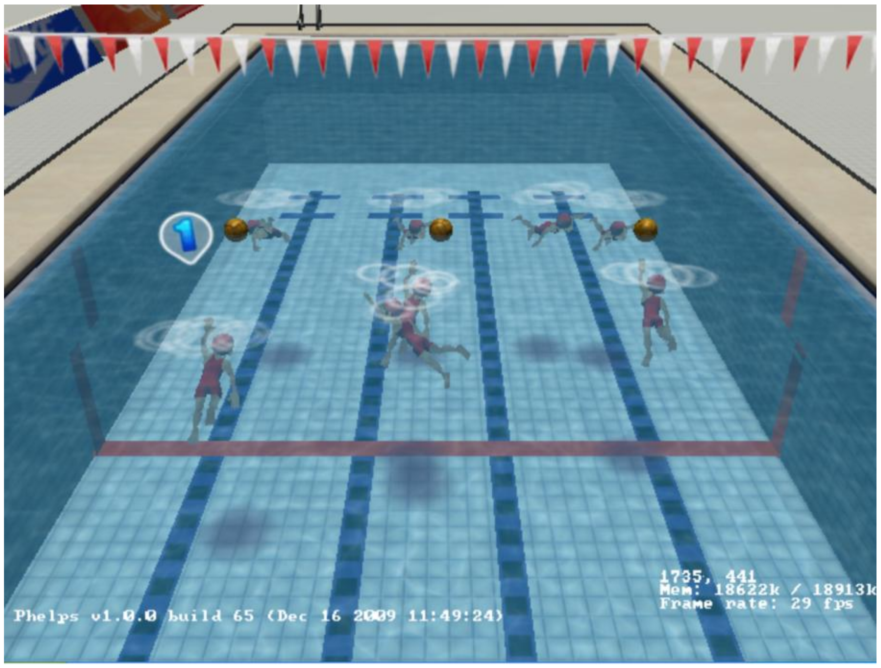


Figure 4: Screenshot from the “Dodgeball” mini-game

3.3 Swimming

It was decided by GDX that the “Swimming” mini-game would be a race in which four players could move their characters in one direction across the screen. Once the players reached one side of the pool, they had to turn back around and swim towards the other side. In order to move their character, the player would rotate the Wii Remote and Nunchuk attachment in a clockwise motion. To come up for air, players would have to press the B button after a certain period of time had elapsed. If a character does not come up for air, its movement speed will greatly decrease.

When implementing the game, we needed to create target ball objects for the player at the edges of the pool. First, one target would appear at the far side of the pool. Once that target was reached, a second target would appear at the other side of the pool. The targets were represented as `tEdge` objects. Each `tEdge` had a player property, which determines the player that is responsible for that target, as well as a lap property that determines where the target should be placed.

```
tEdge.player = 1   --which player we are
tEdge.lap = 200    --based on the lap, the z value of the target
```

The `SetPosition`() function sets the position of the `tEdge` objects. Depending on which player is responsible for the target, a different value is used for the x value, to distinguish lanes in the pool. The player's current lap number determines the position of the target, which is a coordinate corresponding to the side of the pool the target appears on.

```
function tEdge:SetPosition()

      if(self.player==1) then
            self.Object3DData:SetPosition(-100, 0, self.lap)
      end
      if(self.player==2) then
            self.Object3DData:SetPosition(-33, 0, self.lap)
      end
      if(self.player==3) then
            self.Object3DData:SetPosition(33, 0, self.lap)
      end
      if(self.player==4) then
            self.Object3DData:SetPosition(100, 0, self.lap)
      end
end
```

The `Reach`() function resets the position of the target, once a player has reached it.

```
function tEdge:Reach()
      self.bReached = true
      self.lap = self.lap * -1
      self:SetPosition()

end
```

We also needed to implement behavior for breathing. This was handled by placing two variables in the `SwimmingPlayer` file, and by changing the movement methods in the file. The `fBreathCooldown` variable keeps track of how much time must pass before the player needs to push the breath button. This time is calculated by counting the number of times that the game is updated. The `Rising` variable keeps track of whether or not the player's character is in the process of rising.

The `fBreathCooldown` variable is decremented at every update of the game, until it equals 0. The `MoveForward`() function accepts the variable `rise`, which is just the current value of the `Rising` variable. While rising, the y value is incremented by 0.3 and the character's forward position is updated. If the y value is already greater than 0, the character has reached the proper altitude, and the `Rising` variable is set to false. The `fBreathCooldown` variable is what determines whether or not a player has run out of breath. If this value is greater than zero, the player has not run out of breath and can still move forward. If not, then the player's speed is multiplied by .02, slowing it down, until it goes up for air.

The code for `MoveForward` is shown below

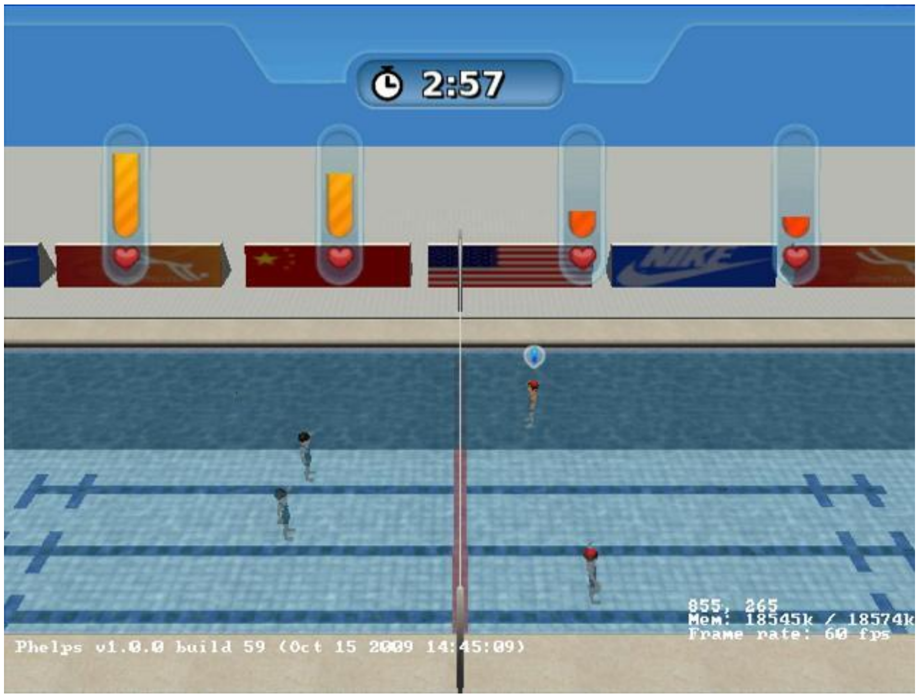
```
function tPlayer:MoveForward(rise)
      .
      .
      .
      local fSpeed = g_fElapsed * self.MovementSpeed
      local x, y, z = self:GetPosition()

      self.Object3DData:SetPosition(g_TempVecOne.x + g_TempVecTwo.x *
      fSpeed, (g_TempVecOne.y + g_TempVecTwo.y * fSpeed) +.3,
      g_TempVecOne.z + g_TempVecTwo.z * fSpeed);

      if(rise) then

            if(y>=0) then
                  self.Rising = false
            end

      elseif (self.fBreathCooldown <= 0) then


            fSpeed =  fSpeed * 0.2

      end

end
```

The AI behaves as a player would in that it pursues its current target, and goes up for air at regular intervals. Different AI-controlled characters would move at different speeds and be programmed to use different tactics while swimming towards their targets. The `Update`() function below handles this functionality.

```
function tAI:Update(fElapsed)
      if bAIToggle then
            --speeds chosen by testing

            if(self.Player.Number==2) then
                  self.Player.MovementSpeed = 140.0   --fastest
            end
            if(self.Player.Number==3) then
                  self.Player.MovementSpeed = 100.0   --slowest
            end
            if(self.Player.Number==4) then
                  self.Player.MovementSpeed = 110.0   --medium
            end


            if (self.Target == nil) then
                  self:ChooseTarget()
            end

            self.InitialCooldown = self.InitialCooldown - fElapsed

            if (self.InitialCooldown > 0) then
                  self.Player.fBreathCooldown = self.InitialCooldown

            else

                  if    (math.random(1,10)-1==0) then

                        self.InitialCooldown = 1
                        self.Player.Rising = true

                  end

                  self.Player.fBreathCooldown = self.InitialCooldown


            end

            self:MoveTowardsTarget()

            self:CheckForTargetReach()
      end
end
```

One of the most notable aspects of working on the “Swimming” mini-game was its similarity to the other mini-games. When we began work on the game, nothing was implemented. We created the players, AI, and other game aspects entirely by ourselves. Because of the way that the mini-games are so modular, though, this task was simple. We were able to take player classes from the other games and simply place them into the Swimming environment. It was simple to take the already implemented AI from other games and remove all of the game-specific strategies. This way, we were given basic swimming AI, which was all that was needed for this mini-game. This re-usability of code seemed to be present in each of the games, and was a huge help.

Figure 5, displayed below, shows a screenshot of the “Swimming” mini-game.



```
855, 265
Mem: 18545k / 18574k
Frame rate: 60 fps
Phelps v1.0.0 build 59 (Oct 15 2009 14:45:09)
```

Figure 5: Screenshot from the “Swimming” mini-game

3.4 Obstacle Course

The “Obstacle Course” mini-game is about gathering and avoiding objects. The player controls their character by moving it around the pool, while gold medals float across the screen. The more medals the player collects, the higher the player’s score. Logs and inner tubes also float across the screen, though. If the player collides with a log, the character becomes dizzy, and its controls are switched, so that buttons that previously moved the character in one direction now move it in a different one. If the player collides with an inner tube, the player’s character is slowed down, and it cannot move as fast as it used to be able to. Both of these effects are temporary, and wear off after a certain period of time.

When our work on the “Obstacle Course” game started, it did not exist at all. We created all the files needed to manage the game. Most of the files were modeled very heavily after the “Dodgeball” game. The majority of our work on the game was done in the `ObstacleCourse` file, `AI_OC`, `Player_OC`, `Medal`, `Log`, and `Tube`.

In the `Medal` file, there is one important method, `Pickup`(), which is called when a player collides with a medal. At that point, the medal object is repositioned above the screen, so that it looks to the player as if the medal has been picked up. Once the medal is above the screen, it can begin drifting back down the screen, as if it was a new medal.

The same is done in the `Log` and `Tube` files. The actual drifting of the objects is handled in the Obstacle Course file, where the game iterates through all of the floating objects and updates their positions, so that they appear a little bit further down the screen. If the objects float off the screen, the game updates them by placing them back above the screen at a different horizontal position, so that they can begin drifting again as if they were a new object.

```
ObstacleCourse.Drift = function(self)
for i = 1, NUM_OC_MEDALS do
      local x, y, z = ObstacleCourse.Medals[i]:GetPosition()

      if(z>400) then
      x= math.random(1,300)+x
      y= y
      z= -300
      end
      if(x>150 or x<-150) then
         x = ((x+150)%300)-150
      end
      ObstacleCourse.Medals[i]:SetPosition(x, y, z+2)

end
-- we do the same for logs and tubes (omitted)
.
.
.
end
```

We check whether an object has been collided with or not in the `ObstacleCourse` file as well. If a player has collided with a medal, it is picked up. If a player has collided with a tube, the character is "slowed down". If the player has collided with a log, the character becomes "dizzy". We calculated collisions by iterating through all floating objects and checking their positions against the positions of all players.

```
ObstacleCourse.CheckCollisions = function(self)

 .
 .
 .
 --so we check out collisions between a player and a medal

      for m = 1, NUM_OC_PLAYERS do

            for n = 1, NUM_OC_MEDALS do

                  PosX, PosY, PosZ = self.Players[m]:GetPosition()

                  TargetX, TargetY, TargetZ =
                  self.Medals[n]:GetPosition()

                  g_TempVecOne:SetVector(TargetX - PosX, TargetY -
                  PosY, TargetZ - PosZ)
                  if (g_TempVecOne:GetSquaredLength() <
                  fPlayerRadiusX2) then

                  self.Players[m]:SetScore(self.Players[m]:GetScore()+s
                  elf.Medals[n]:GetValue())
                  self.Medals[n]:Pickup()

                  break
            end


      end

end

-- we do the same for logs and tubes (omitted)
.
.
.

end
```

The characters are actually programmed to become slower or dizzy in the `Player_OC` file. In the `SetDizzy`() and `SetSlow`() methods, we set the appropriate variables in the player, set the cool down time, and make sure that the two conditions are mutually exclusive.

```
function tPlayer:SetDizzy()
      self.Slow = false
      self.Dizzy = true
      self.MovementSpeed = 120
      self.fEffectCooldown = 10
end

function tPlayer:SetSlow()
      self.Dizzy = false
```

```
        self.Slow = true
        self.MovementSpeed = 50
        self.fEffectCooldown = 10
    end
```

In the `Update`() method, we check to see if the effect has cooled down long enough and if the character can be set back to normal. If not, we scramble the player's movement when the character tries to move.

```
    function tPlayer:Update(fElapsed)
        self.Object3DData:Update()
        if (self.fEffectCooldown > 0) then
            self.fEffectCooldown = self.fEffectCooldown - fElapsed
        else
            self.Dizzy = false
            self.Slow = false
            self.MovementSpeed = 120
        end
        if (self.OwningController == 0) then
                self.AIController:Update(fElapsed)
        else
            if(self.Dizzy) then
                self:ScramblePlayerMoves()
            else
                self:HandleInput()
            end
        end

        self:CapVelocity()
    end
```

Once the aspects of the game that affect the player are set up, we were to create an AI that played like an actual person. To accomplish this, the AI component of the game had to be able to differentiate between medals and obstacles; this implemented in the `MoveTowardsTarget`() function. If an AI player is close to a log or a tube, it senses it and moves in the other direction. If neither a tube nor a log is nearby, the AI continues to pursue the target as normal.

```
    function tAI:MoveTowardsTarget()
        local myX,myY,myZ = self.Player:GetPosition()
        local itsX, itsY, itsZ = 0, 0, 0

        for i=1, NUM_OC_LOGS do
            itsX, itsY, itsZ = ObstacleCourse.Logs[i]:GetPosition()
                if ((((myX - itsX) < 25) and
                    ((myX - itsX) > -25))) and
                      ((((myZ - itsZ) < 25))) then

    self.Player.Object3DData:FacePoint(-itsX, myY, myZ)
                                    self.Player:MoveForward()
                        self.Logs = 1
                else
```

```
                        self.Logs = 0
                  end
      end
      for i=1, NUM_OC_TUBES do
            itsX, itsY, itsZ = ObstacleCourse.Tubes[i]:GetPosition()
                  if ((((myX - itsX) < 25) and
                      ((myX - itsX) > -25))) and
                        ((((myZ - itsZ) < 25))) then

self.Player.Object3DData:FacePoint(-itsX, myY, myZ)
                                    self.Player:MoveForward()

                        self.Tubes = 1
                  else
                        self.Tubes = 0
                  end
      end
      if(self.Tubes == 0 and self.Logs ==0) then

self.Player.Object3DData:FacePoint(self.Target:GetPosition())
            if(self.Dizzy) then
                  self:ScrambleAIMoves()
            else
                  self.Player:MoveForward()
            end
            else
                  self:ChooseTarget()
            end
end
```

Figure 6, below, shows the basic AI of the “Obstacle Course” mini-game, as a state machine.

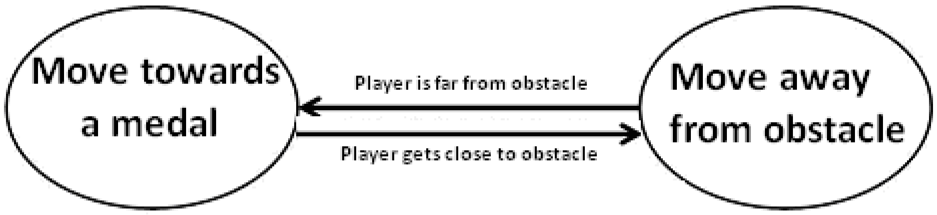


Figure 6: AI in the “Obstacle Course” mini-game

Working on the “Obstacle Course” mini-game provided good insight into the publisher-developer relationship. Throughout the development of this mini-game, most of the specifics of the gameplay had not been decided. At first, the team had talked about having medals that sank through the pool, instead of drifting across the screen. At another point the game was proposed as a modification of the “Swimming” mini-game, but with obstacles. The publisher did not seem to have any clear idea about what this mini-game was going to be. This issue temporarily halted development for this mini-game, as our group was not able to add any specific implementation to the game until late into the project.

3.5 Chicken

The game of “Chicken” when referring water games, refers to a 4 player game with one player on the shoulders of another player. These two person teams use either their arms or foam swords of javelins to unseat the other team. The game was implemented to allow for 4 person play, each person selecting a team and position.

“Chicken” was in a workable but incomplete state when we joined. The mini-game could receive, and process input, but the game had no concept of state, such as health or rounds. Placeholder art was in place until the artists finished the final products.

The AI for “Chicken” was designed and partially implemented when we joined the development team. It was our task to expand the functionality of the AI beyond simply repeating the same attacks. The AI was designed using a finite state machine which had several states:

- *Offense*. Actively attack the opponent, with a small amount of defense.
- *Defense*. Absorb attacks until a good attack or counter opportunity presents itself.
- *Counter*. Execute special attacks to remove a large amount of the opponent’s balance, at the risk of being caught off guard.

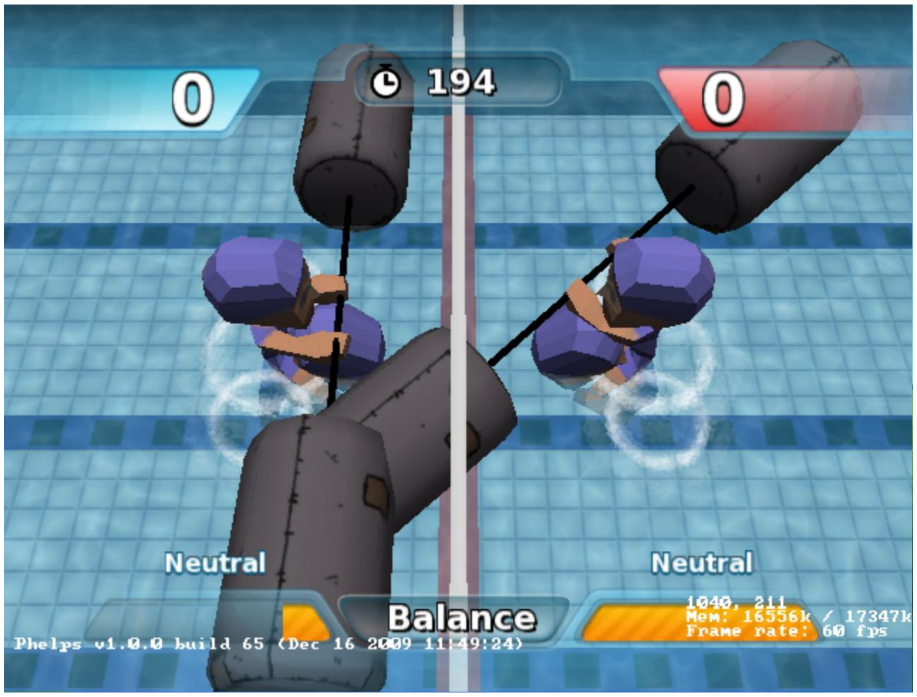
Our initial task was to give the mini-game a concept of state. The first was to track balance, which would represent each player’s overall stamina. Originally this was implemented as an integer, but later refactored as a double when tied to the interface. The player’s balance decrements a certain amount each time they were struck by the other player’s weapon. When their balance is completely diminished, the losing team falls over and the round ends. The winning team’s score is then incremented, and a new round begins.

Information such as score and balance were represented in Chicken’s interface. Adding the interface to *Chicken* was our first experience integrating programming variables with art assets. The Lua bindings made the experience straightforward. We used SumoTool to locate which variables controlled the interface. A few function calls to the SceneManager within the OnUpdate() function allowed us to update these variables every frame.

An example of the working interface code is shown below:

```
for i = 1, 2 do

            SceneManager_ChangeTextString("CHICKEN\\SCORE_TEAM"..i,
Chicken.Teams[i].Score)

            SceneManager_SetKeyFrame("CHICKEN\\BALANCE_TEAM"..i,
"KEYFRAME_"..(100 - (math.floor(Chicken.Teams[i].fBalance))) ,
"KEYFRAME_"..(100 - (math.floor(Chicken.Teams[i].fBalance)))    )

      end

      SceneManager_ChangeTextString("CHICKEN\\TIME",
math.floor(Chicken.Time))
```

Figure 7, shown below, is a screenshot taken from the “Chicken” mini-game. It demonstrates the interface that resulted from our work with SumoTool.



```
1040, 211
Mem: 16556k / 17347k
Frame rate: 60 fps
Phelps v1.0.0 build 65 (Dec 16 2009 11:49:24)
```

Figure 7: Screenshot taken from the “Chicken” mini-game

3.6 Rock The Boat

“Rock The Boat” is a water game involving a number of players on a slippery raised surface. This surface is above water, such as a pool, and each player attempts to be the last one standing by pushing and shoving opponents into the water. This game is also known as “King of the Hill”, or “Last One Standing”.

Like many of the other mini-games, the basic structure for “Rock The Boat” was already coded and our task involved improving the code so that it was ready for its alpha build. Since the players in “Rock the Boat” already had a simple AI component guiding their actions, our work consisted mainly of tweaking certain variables, and refining certain aspects of gameplay. Other than refinements, we integrated art assets into the mini-game and refactored the code to reflect the boundaries defined by the actual art.

3.7 Ultimate Swimming Cancellation

During our break we received an email from GDX’s Studio Head informing everyone that all work would be suspended indefinitely on the *Ultimate Swimming* project. When we returned to GDX on Monday, we were told that the client was unsatisfied with the game's progress and pulled funding. We were given the option of continuing to work on *Ultimate Swimming* until the end of our project, as was the agreement when we began working, or to begin working on something new. We talked amongst ourselves, and we decided that we would rather contribute to the team, and work on a project that could use our help. That project would end up being the *Explorer Girl Game*[9] *(EGG)* expansion.

3.8 Explorer Girl Game

Our work on *EGG* was more limited than our work on *Ultimate Swimming*. *EGG* is a computer game that works with an Explorer doll. The Explorer doll plugs into the computer via a USB cord. A software program is run on the computer that checks to see if the doll is plugged in. If a doll is present, a number of small games as well as a larger mystery game are available to the player. Our task was to create a way that Flash games could be played within the *EGG* software. We were provided with examples from past projects where Flash had been used, and asked to find some way to use that code to assist in implementing this feature in *EGG*. We succeeded in running a Flash game by adding `CFlashAppDialogue` and `FlashWindow` files as well as by altering the `BaseWindow` and `MainMenu` files.

[9] For reasons similar to “Ultimate Swimming”, we are not allowed to divulge the name of this game either. We have given it the alias of “Explore Girl Game” or “EGG”.

The `CFlashAppDialogue` file was based largely on a class that already existed in another one of GDX's projects. Its primary helpful function is `DoFlashApp`(), which accepts the name of a Flash file to be played, and calls a function called `DoMovie`() to play the file.

```
VOID
CFlashAppDialog::
DoFlashApp(const CHAR *pszFileName)
{

    U32 nZOrder = Z_HINDMOST;

    CxCursor::Hide();

    FoxShowOSCursor();

    EGG::GetApp()->GetCurRoom()->Erase();
    EGG::GetApp()->GetCurRoom()->Pause();

    // Get our path name

    CHAR szSaved[FOX_MAX_PATHNAME];
    CHAR *p, szPath[FOX_MAX_PATHNAME];

    OSFileGetCurrentDir(szSaved, FOX_MAX_PATHNAME);

    FoxFileGetFullPath(szPath, pszFileName, FOX_MAX_PATHNAME);

    if ((p = strrchr(szPath, '/')) != NULL) {
        *(p + 1) = '\0';
    }
    OSFileSetCurrentDir(szPath);

    CFlashAppDialog cDialog;

    cDialog.SetZOrder(nZOrder);

    // Play the flash file
    cDialog.DoMovie(pszFileName);

    // Restore the CWD
    OSFileSetCurrentDir(szSaved);

    FoxHideOSCursor();

    EGG::GetApp()->GetCurRoom()->Resume();
    EGG::GetApp()->GetCurRoom()->Paint();
    CxCursor::Show();

}
```

We wanted to be able to use this function to play our Flash files inside of a Lua file that would run in *EGG*. In order to do this, we needed to register this function with the Lua state, so that Lua would know that it existed and be able to use it. We did this in the `BaseWindow` file. We created an instance of our `CFlashAppDialog` class, and created a new function that performs the same action as `DoFlashApp` but instead, accepts input from Lua. Once the Lua state was set up in the *EGG* code, we registered our new function as "`DoFlashApp`" in the Lua state.

```
CFlashAppDialog *fad = NULL;

int Lua_Bind_DoFlashApp(lua_State *L)
{ fad->DoFlashApp(luaL_checkstring(L, 1)); return 0; }


//...some code ommitted

CxLuaEngine *pScriptEngine = NULL;
if ((pScriptEngine = CxApp::GetApp()->GetScriptEngine()) != NULL)
{
      lua_State *pLua = (lua_State *)pScriptEngine->GetVM();
      BIND_GLOBAL_FUNCTION_FROM_MEMBER(pLua, "CreateScreenshot",
      *(CBaseWindow *)this, &CBaseWindow::CreateScreenshot);
}

fad = new CFlashAppDialog();


if ((pScriptEngine = CxApp::GetApp()->GetScriptEngine()) != NULL)
{
      pScriptEngine->RegisterFunction("DoFlashApp",
      Lua_Bind_DoFlashApp);
}
```

In our new `FlashApp` Lua file, we created a blank window and called our `DoFlashApp` function from it. Here, `DoFlashApp` is running a game called "CopyThat_AS2."

```
return CxDialog {

DoFlashApp('CopyThat_AS2.swf'),

minX = 350 - 155,
maxX = 520 - 155,

OnInit = function(self)
---------------------------------------------------------------
-- Background Overlay
--
self:LoadSurface("Textures/FlashPlayer.jpg")
self:SetZOrder(ZORDER.TOPMOST)
self:LinkRoot()
self:Paint(0, 0, EPOS.CENTER, EPOS.CENTER)
self:Hide()


end,

}
```

In *EGG*'s `MainMenu` file, we created a button that would load our new window. When that button is pressed, the `roomname` variable is set to "`OpenFlash`," which runs our `FlashWindow` Lua file.

```
Room.names=
{

      {button="BUT_MainMenuEnterWorld_00.fmv",  roomname="World",
            x=316, y=486},
      {button="BUT_MainMenuEnterCode_00.fmv", text="MyFlashButton",
            roomname="OpenFlash", x=10, y=30},
      {button="BUT_MainMenuEnterCode_00.fmv",
            roomname="GameSelectMenu", x=59, y=531},
      {button="BUT_MainMenuOptions_00.fmv",
            roomname="OptionsMenu", x=527, y=487},
      {button="BUT_MainMenuExit_00.fmv", roomname="Exit", x=527,
            y=531},
}

--some code omitted

function Room.OnStartRoom(self)
      local room = self

      self:LoadSurface("BKG_MainMenu_00.jpg")
      self:SetZOrder(ZORDER.HINDMOST)
      self:LinkRoot()


      for k,v in pairs(Room.names) do
      local b = CxButton
      {
            parent = self,
            roomname = v.roomname,
            OnClicked = function(self)
                  PushRoom(self.roomname)
            end,
      }
      b:LoadSurface(v.button)
      b:Link(self)
      b:Paint(v.x, v.y)
      table.insert(self.buttons,b)


      --some code omitted
```

```
        if ( v.roomname == "OpenFlash" ) then
              b.OnClicked = function(self)
              local opt = dofile("FlashWindow.lua")
              opt:Show()
        end
end
```

Figure 8 and Figure 9, below, show the *EGG* menu and the loaded Flash game.

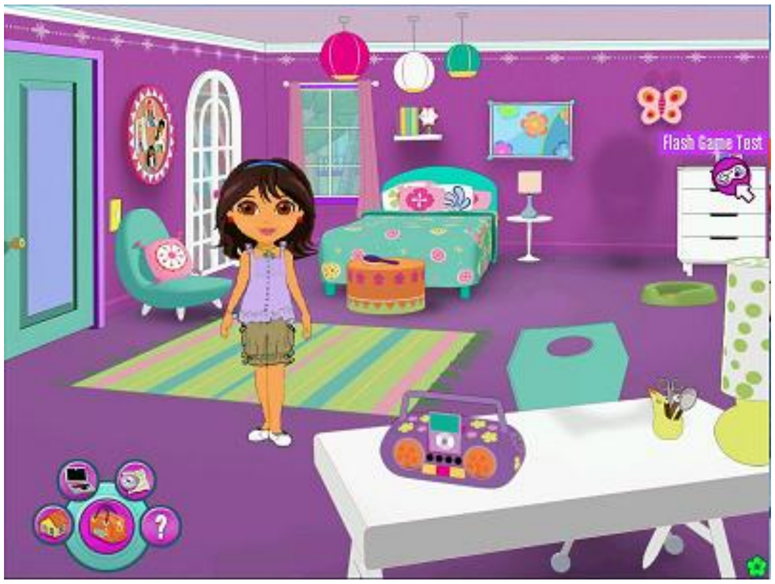


Figure 8: *EGG* Menu, With Flash game button at right of the screen

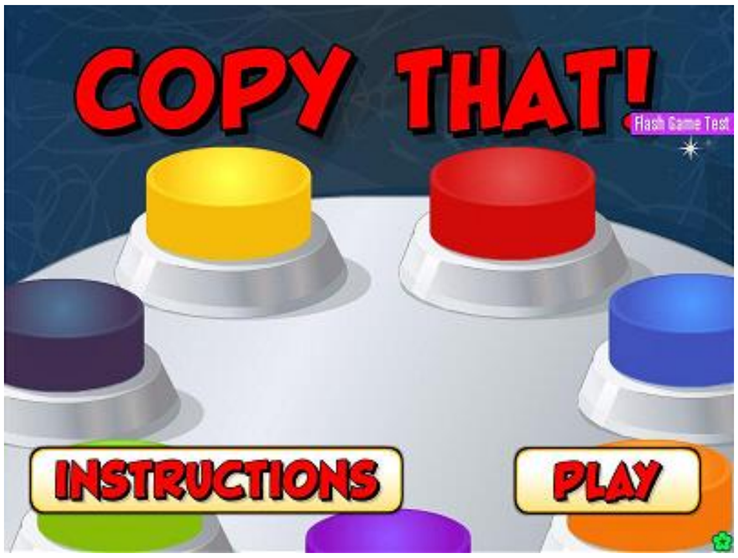


Figure 9: Flash game, successfully loaded in *EGG*

With these additions and modifications to the *EGG* Project, a Flash Game was able to be loaded and played inside of the *EGG* software.

The most striking part of working with *EGG* was the inscrutability of GDX’s code. The *EGG* project was extremely large and poorly documented. As a group, we were asked to look through GDX’s old code and see if we could use any of it in the *EGG* project. The code we were given was buggy and uncommented. Lines which would appear to load a new screen into the *EGG* game in fact loaded completely separate mini-games. The only way to determine that this was what was happening was by following calls in the code through several large files which contained literally no comments. This was an extremely difficult issue to work through, and only made the importance of clean and commented code more evident.

4. The Interview

From our work at GDX, we were able to learn industry practices and standards in a practical environment. Unfortunately, only a limited amount of knowledge can be obtained by immersing one's self in a game development environment for a short period of time. For instance, the programming that we did on *Ultimate Swimming* gave us an idea of how often comments are used in code or an idea of the way AI is traditionally implemented in casual games, but it could not provide an answer to the question about what game developers look for in a software engineer.

In order to obtain answers to questions such as these, we decided that it would be in our best interest to interview members of the GDX development team to get their perspective on general topics that were pertinent to game development. The purpose of the interview was to get further insight into specific aspects of game development that we could not gain by simply coding. The answers obtained from the interviews would complement the knowledge that we acquired from our hands-on experience and further aid us in learning more about the game development process.

We decided to conduct the interview in one joint session so that members at GDX could expound upon another's point if need be, as well as not answer if they felt a sufficient answer had already been given. The three members of GDX's team that we chose to interview were the Studio Head, the Technical Director, and the Senior Producer[10]. They were chosen because their respective positions provided them with the ability to give high-level answers to certain questions as well as the ability to speak of practices in the game development process that a programmer or low-level developer may not have information on. Our questions touched upon topics such as what do development companies look for in a team member, what high-level trends are the game development industry moving towards and what specific elements were essential to the game development process

The answers that we received were not only extremely helpful in gaining a firmer grasp of the game development process, but also provided for some surprising answers. Below is a transcript of our interview.

[10] The members of GDX we interviewed chose to remain anonymous. In the interview they will be referred to by their respective titles.

4.1 Questions

Interviewer: How important is Iterative Development?

Studio Head: I think that it is really important. I guess there are multiple levels to that question. I mean, it is important to make good gameplay. You can put in a document that a game needs to be fun. For [*Alvin and the Chipmunks]* they removed a lot of game mechanisms to achieve fun. But you also have to remember that we make games as a business. When you have a fixed budget and you have a fixed timeline – also people cost money – so you're limited in what you can do. You kind of have to make a tradeoff between scope and how fun to make the game. It's important to make the best game possible, but business limits the amount of iterations we can do. So because everything is limited by money, I'd say what's most important is being smart about iteration and determining how many iterations can be done with a given budget.

Technical Director: No one will make a game that is fun at first. You must iterate to achieve fun. We had to do a bunch of iterations on *Alvin and the Chipmunks* to achieve a certain level of fun. The key to maximizing the profit is to minimize the iteration time. The longer iteration time is, the more time we're wasting. You want to make iteration time as short as possible.

Interviewer: How important are things like comments in code?

Technical Director: For maintenance and reuse, clean code and commenting is superior to hacking. Inevitably somebody who may not be you may have to extend, maintain or fix bugs in the code later on. This is a lot harder to do when the material is crappy. I mean, if it's a throwaway prototype it's not really important that it be clean. Still, that's not how I go in. I write every system like it's going to be used a million times, so I make sure it's as clean as possible even though it may not be reused again. We don't have a time machine to see if it'll be reused again – better safe than sorry.

Studio Head: Well, like everything, there are two sides to every story. If it's going to be reused or if it is part of a library, then it's infinitely important to write clean code. If it's just a prototype, then it doesn't really matter.

Interviewer: What do you look for in a Software Engineer as a team member?

Technical Director: Independence. Their ability to own a subsystem without the need for a lot of oversight [is very important]. So like, they can take a system and implement it and not need to ask for help every 10 minutes. Also, the initiative to voluntarily enhance the system beyond what they're asked to do is important. The ownership to make it as good as possible given certain requirements is key, but taking the initiative to make it better is more important.

Senior Producer: [I'd agree with the Technical Director and say independence is] really important. Also important is communication from them and knowing when and how to ask for help. It's not really efficient if you spend time creating this function that solves a problem, only to find out that the issue had already been resolved – don't recreate the wheel. Good time management is always important too. But like [the Technical Director] said, basically someone who can own [their assignment], can create their own tasks, and cares about the quality.

Studio Head: Notice how no one said anything about technical proficiency in one field or another. It's so easy to find a resource that's technically-efficient, but to find someone who can be reliable and make sure that the task that they are assigned is going to be done is hard to find. You can take what you learn in school and be technically-efficient, but someone who is reliable is more important.

Interviewer: What do you see as the most important trends in Game Development?

Senior Producer: I think there's a huge trend towards social and networked multiplayer games. Also there's a trend towards reusable middleware components and engines, such as Unity and Havok. At this point, lower level coding is less important and there is now a reliance on outsourced resources. Because of this our end-coders have to interface with a lot more external people and be able to work with them. Another trend that I've noticed is that people are making less games for introverts and more games for social and casual gamers. You've kind of seen this rise of casual games over the last few years. I mean, there are tons of tiny games to play – Flash games, iPhone games, etc.

Studio Head: See, I wouldn't even call the social aspect of gaming a trend anymore. Ever since the first game I worked on 20 years ago, multiplayer was huge. I feel social aspects are now the ever-increasing need to find more people with the same interest. It's like natural evolution. You start with *Pong*, then you have modem play and now network play. I think the next step is to make games more accessible to casual gamer.

Technical Director: I think that the idea behind being able to accommodate any new trend is the flexibility of the studio to hit the ground running when that trend emerges. Writing our own systems to target those trends is not how we do it quickly. The best way to hop on a trend is to use middleware.

Interviewer: Are there any elements you feel that are essential to the Game Development Process?

Studio Head: Good people.

Senior Producer: I think the planning production of the process is one of the most important things that are overlooked. Also people who can manage their schedules and time.

Technical Director: I'd agree and say planning is really essential as well as spending more time prototyping.

Interviewer: Can you give us any insight into why *Ultimate Swimming* was cancelled?

Studio Head: Sure I have no problem talking about it. I'd say it wasn't one thing in particular. There were failures here from project management side as well as problems from a technical standpoint. And as far as I'm concerned, we also had problems from the client side. Some of their expectations for the game were not realistic especially with the time and the budget we were given.

Senior Producer: We went into this project knowing it would be risky. We were already working on projects here, prior to taking on *Ultimate Swimming*. We didn't choose the right middleware. That, combined with the staffing issues and our lead programmers changing, people being shifted around and not properly adjusting the project schedule and timeframe. You can't plug people in the same hole and have the same expectations done in same time frame. That's just not the way it works. Also, it goes back to communication. Client communication was just off in what we were delivering and what we were developing. They wanted a playable prototype that accurately demonstrated the gameplay almost immediately, we promised that later on. It didn't help that it wasn't clear who held the cards on the client side. There were just lots of communication issues.

Interviewer: Based on our experience here at GDX, we've kind of gotten a glimpse at the balance of power between the publisher (client) and the developer. Can you further explain how power is balanced between the two?

Technical Director: Publisher has all the power. They decide whether they accept the milestone or not. We can make it as perfect as anyone can expect, but if the publisher doesn't like it and tells us to make changes, we have to suck it up and do it.

Senior Producer: Just to add to that, I'd say that the publisher has the most power at beginning but as the game comes to an end, the developer gains more power. At that point, the publisher has already invested so many resources into the game that they'd do anything to make sure that the game gets completed.

Studio Head: It's like any contract you enter, you sign up to deliver something, and you're expected to do so. In the case of games though, fun is an integral part of what you're delivering, and it's hard to define what fun is. We can deliver crap and no one is happy. In all fairness, publishers take a lot of risk [that's why they have so much power over developer]. They put out a lot of money into development cost, marketing, etc. And now, it's tough for publishers because the stores now have the power. Stores will tell publishers you're only getting this much shelf space, or, as an example, we are no longer stocking games that have monkeys in them.

Technical Director: Exactly. For example, there was this game we had completed that we had to go back and work on since Wal-Mart wouldn't accept games with lethal weapons. Every weapon that was lethal had to be changed to a non-lethal weapon. The time it took us to make those changes hindered our ability to ship by Christmas. Yeah, Wal-Mart has all sorts of power. As a matter of fact, when *EGG* was released, Wal-Mart told Fisher Price that the Explorer dolls that the game used were too expensive so Fisher Price had to make a new doll to accommodate Wal-Mart.

Studio Head: And Wal-Mart will say that the end-user has all power. It's a risk-rewards thing. The more rewards, the more risks that are probably involved. Different people will tell you who has the power. The stores will say the user, the publisher will say the store, etc.

4.2 Interview Afterthoughts

The interview turned out to be more beneficial than we had expected. There were unanticipated particulars that were mentioned such as the use of middleware software to simplify the coding process that we did not expect to hear. As will be discussed in the Observation section, these points helped provide a more complete, and in some cases, surprising view of the game development industry.

Before this section comes to close, we feel that is important to make the distinction between the interview that we conducted with GDX's development team and the game development lectures that we spoke about at the onset of this report. Many of the questions that we asked were specifically related to situations that we had experienced while at GDX. For instance, we asked the question about the publisher's power to further understand why GDX's developers stressed satiating the needs of the publishers. We asked the question about iterative development to get a broader understanding of how they determined which objectives need to be met for a particular milestone, etc. This differs from the lectures in the sense that the information derived from lectures tends to be generalized and not as specific in scope. Also, the reason why the answers from the interview turned out to be more beneficial than expected was because when they were coupled with the knowledge we gained by working on *Ultimate Swimming*, we were able to get a more thorough understanding of the game development process. As two separate entities, both integrating with a development studio and interviewing industry insiders can serve as great learning tools, but when combined, it provides for one of the most comprehensive ways of learning about game development.

5. Observations

When we set out to complete this project, we had a core group of questions that we wanted answered. To our delight, we left with more information than expected. Based to the work that we did at GDX and the answers we were able to obtain from our interviews, we gained surprising and invaluable insight about the field of game development. Below is a list of the most pertinent information that we were able to gather.

1. **Fun takes precedence.** The purpose of a game is to provide entertainment to the player. In order to achieve this, developers try to maximize the amount of fun that the game provides. While working on *Ultimate Swimming* every change that was made to the game was made to increase the game's fun factor. Whether it was making the AI in the games smarter, or adding new features such as obstacles to challenge the player, fun was always the primary goal.

2. **Scripting is very important.** Many game development lectures and books stress that anyone interested in the field of game development should have a fundamental understanding of C/C++. Although it is imperative that one understand these languages, based on the work done on this project, we would say that it is equally important that one fundamentally understands scripting languages as well, primarily Lua. We emphasize Lua because it seems to be the emerging scripting language of choice in the game industry. The bulk of the work we did while working at GDX was done in Lua. In fact our knowledge of C++ was rarely put to use.

3. **Plan and prototype.** There are many people who like to start coding immediately, rather than plan out the tasks ahead of time. Although this may seem more time-efficient at the onset, it can actually create more hindrances later on, especially on games as large as *Ultimate Swimming* or *EGG*. For games with milestones that need to be met, it is much more efficient to do the planning, and put focus towards the implementation later on. This not only provides for a smoother development cycle, but it is also provides for better code modularity as well as improved game design.

4. **Choose the right AI type.** There are many ways to implement AI in videogames. There are state machines, goal trees, planning, etc. The different implementations work better for some games than they do for others. For instance, it would not make sense to have a more complicated

method like planning for a simple game like *Ultimate Swimming.* Making the right choice provides for more straight-forward coding as well as easier maintainability later on. This all ties back to the previous point of planning and prototyping. To plan out the actions for the AI in the design process, one first needs to decide which AI implementation is most suitable. In the case of *Ultimate Swimming* that implementation turned out to be the use of Finite State Machines since the AI in the game was not overly complicated.

5. **Choose the right software.** According to members of GDX, one of the reasons that the *Ultimate Swimming* project was cancelled was because they did not use the right middleware for the project. From this statement, it is evident that understanding when and how to use the right software is integral to the success of any game. One of the right software packages that GDX utilized while working on *Ultimate Swimming* was SumoTool. This software greatly simplified the process of designing the Heads-Up Display for the game and allotted us more time to work on more pressing issues.

6. **Reuse code.** Prior to working at GDX, we had an idea that code reuse was common in the game industry, but we did not know to what extent. Apparently the answer is: really common. In fact, at our first meeting with GDX, one of the topics of discussion revolved around using code from another game to accomplish a desired feature. Another instance of this occurred when Mr. X had us play *Mario & Sonic at the Olympic Games* so that we would have an idea of how the "Swimming" mini-game should be implemented. This all ties in to the point that the Senior Producer made in the interview of not trying to re-invent the wheel. If code has already been created that performs a desired functionality, then save development time, and reuse the code in the game. If direct code reuse is not possible, then spend time studying the game that contains the desired functionality and attempt to implement similar code. Although not as ideal, this still saves valuable design and prototyping time.

7. **Hacks.** As the Technical Director said, hacks are never ideal; they provide for messy and hard to follow code. There are times, though, when hacks should be used as a *temporary* remedy to save time. When no solution is readily available, hacks allow developers to allocate time to another part of development and subsequently help them reach fast-approaching milestones. For instance, while working on "Dodgeball", incorporating the arc that the ball had when it was being thrown was a lot harder than anticipated. One of the bugs that would occur was the game would classify the ball as moving even though it was stationary. This would cause players to be

taken out of the game if they tried to touch ball. In order to resolve this error, a hack was created where the ball was deemed stationary when its velocity reached a certain low value. With this hack in place, we were able to temporarily focus our attention elsewhere as well as present the publisher with a functioning model of the “Dodgeball” mini-game.

8. **Initiative counts.** In the interview, one of the characteristics that was stressed about being a good developer was the ability to take the initiative when necessary. This manifested itself to us early on due, in part, to the tasks that we were assigned. Though a few of these tasks were as straight-forward as “Add Steal Functionality”, many of the tasks were vague and left us in control to decide how to implement them. For instance, the task “Improve Mid-Field Presence” did not explicitly tell us how to go about doing so, rather we had to take the initiative and discuss amongst ourselves which tactics would increase the mid-field presence of the team. This type of initiative and independence is ideal when working in any type of business setting, especially in the game development industry.

9. **Understand the power structure.** One of the more surprising things we learned while at GDX is how the power structure is balanced in the game development industry. We came in knowing that publishers held the majority of the power with regards to a game’s development; what we did not know was how egregious the difference was. Even more astonishing was how much power the retailer has in the game development process (prior to the interviews, it had never crossed our minds). To our surprise, we found out that retailers have the power to tell a developer that they will not accept games with specific content(e.g. lethal weapons), which in some cases can cause a developer to re-enter development and make the necessary adjustments. We knew that the publisher wielded this type of power over the developer, but never imagined that stores did as well.

10. **Using comments.** Good commenting is always stressed in software engineering and was also stressed by members at GDX in our interview. However, although comments are beneficial to the understanding and reusability of code, commenting is not consistent in game development. There were certain files that had a decent amount of comments in them and thus were easy to use. But many of the files, especially the Lua files that were used for the *EGG* game were poorly commented. This made the code extremely hard to follow and caused us to waste a good portion of time trying to figure out what certain functions did. This was further exacerbated by the fact that there was no debugger in Lua. So not only did we not know what certain functions

did, we also did not have an efficient way of finding out; instead we had to resort to print statements and copious trial-and-error attempts. The verdict on commenting seems to be: use them if it’s absolutely necessary.

The observations mentioned above are the main deliverables of our project. As one can see, we amassed a good amount of useful knowledge while working at GDX. Some of the information was surprising, some was contradictory to what is traditionally taught in classrooms, but all should be helpful to anyone who is interested in the game development. We hope that others can take the observations gathered in this paper and use it to discern which facets of game development are the most fundamental. For any fledgling developer, this would be the first step to not only becoming the best game developer possible, but also to securing a job in the industry.

6. Conclusions

Game development is a large and growing field. There are many questions about the development process, though, to which answers are not readily available. Are basic coding principles like good commenting and modularity applicable to the game market? Do software engineering ideas actually have any bearing on real game development? How are artists able to speak with programmers about game development? How is the relationship between developer and producer balanced? In this project, our group answered these questions by working with a real game company on a title that was slated for release in stores.

As a team, our task in this project was to integrate with GDX and help the team develop a game. While doing this, we were able to experience and be a part of the game development process. We became familiar with the Studio Head, as well as the Senior Producer and worked with one of GDX's engineers. We were able to create working versions of every mini-game that belonged in *Ultimate Swimming*. We worked on Polo, Dodgeball, Swimming, Obstacle Course, Chicken, and Rock the Boat games. We learned how to use Lua to create and modify these games. We also worked on a second game, *EGG,* for GDX and were able to open a Flash Game inside of the *EGG* executable. We gave our work to the studio, so that they could extend and continue it.

A large portion of the game development studios operating today are funded by an external publisher and utilize iterative development cycles. During our project we had the opportunity to see a real publisher-developer relationship in action. This along with the iterative development practiced at GDX formed the majority of our experience. We saw how publisher feedback can radically alter the development schedule. We also saw how iterative development, combined with the skill of the developers can accommodate these changes mid-milestone.

We saw and contributed to real game code for a retail console game. We saw how enterprise tools are used by developers to write and modify their assets and code. Seeing how real developers comment and structure their code provided an invaluable angle into programming. We also saw how software engineering principles are respected, and realistically implemented. The concept of a "Daily Fifteen" is good in theory, but wise producers do not overload their staff with needless meetings. A "Daily Fifteen" will sometimes go long, but not exorbitantly so, and always in pursuit of greater communication.

Perfect code is not as prevalent in the game industry as it is in academia. Good commenting is not a common practice, and often code is misleading and difficult to understand. When working with *EGG* we encountered sections of undocumented code, which lead to lost productivity as we struggled to diagnose errors. This result shows that good coding is not present everywhere in the game industry; this only strengthens the need for it. It is worth noting that *EGG* is still in the early development stages, preparing for a spring release. *EGG* will undergo a significant amount of refactoring and improvement before it meets GDX's standards.

Despite commenting issues, GDX displays excellent software engineering in their products. In *Ultimate Swimming* each mini-game is modular, with well-defined interfaces between game functions. The game is built upon a combination of Lua and C++, with well-documented bindings. These good software principles are essential in the career of a game designer.

During our meetings we observed how artists and programmers interact. During these meetings, artists and programmers would speak in only general terms. It is unclear who is responsible for what, or when it would be available. When Mr. X asks for a ball model, he does not have a guarantee on when it will be ready. The meetings themselves would not always be smoothly run, or happen at consistent times. These meetings are essential to promote communication on a project, but the meetings themselves can be an interruption.

The relationship between publisher and developer is fundamentally unbalanced. The developer is in the employ of the publisher, and the publisher has the right to revoke funding if they are unsatisfied. This creates a clear separation between the developer and publisher, which can become a problem if the publisher is unable or unwilling to provide feedback. We experienced a lack of constructive feedback when working on *Ultimate Swimming*, which lead to an unsatisfied publisher and ultimately the cancellation of the *Ultimate Swimming* project.

After our work with GDX, our group has a clear view of how one game company works. We have experience working in the game industry, and have felt the pressures of a publisher, the tension of working with artists, and the difficulty of coding with a large team. We have answered the questions that existed when this project was started. Also, while our work on *Ultimate Swimming* may never be of use, *EGG* currently contains our code for launching a Flash Game. GDX intends to take that code and use it in future *EGG* releases.

Although we worked under the people and methods of GDX, it is possible that the entire industry does not follow the same protocol as GDX. In other companies, coding habits may be very different, or more control might be held by the developer. Other companies could be looked into, or other issues could be raised, such as how much money does it cost to develop a game? At GDX, more work could be done on the *EGG* game to help prepare for future additions and releases. Simply put: our work is complete, but there is still room for more to done.

Works Cited

1. **Hewitt, Dan.** The Entertainment Software Association - News Releases. [Online] September 29, 2009. [Cited: November 20, 2009.] http://www.theesa.com/newsroom/release_detail.asp?releaseID=76.

2. **Magrino, Tom.** Modern Warfare 2 five-day sales hit $550 million. [Online] November 18, 2009. [Cited: Novemeber 20, 2009.] http://www.gamespot.com/news/6240625.html.

3. **Rabin, Steve.** *Introduction to Game Development.* s.l. : Charles River Media, 2009. 978-1584506799.

4. **Gold, Julian.** *Object-Oriented Game Development.* s.l. : Addison Wesley, 2004. 978-0321176608.

5. **Niintendo Co., Ltd.** Consolidated Financial Highlights. [Online] October 29, 2009. [Cited: November 20, 2009.] http://www.nintendo.co.jp/ir/pdf/2009/091029e.pdf#page=9.

6. **Sinclair, Brendan.** MS: 17.7 million 360s sold. [Online] January 3rd, 2008. [Cited: Novemeber 20, 2009.] http://www.gamespot.com/news/6184291.html.

7. **Sony Computer Entertainment.** PlayStation®3 Worldwide Hardware Unit Sales (Unit: million). [Online] 2009. [Cited: November 20, 2009.] http://www.scei.co.jp/corporate/data/bizdataps3_sale_e.html.