Food Fight

Eric Fong

Dr. Zoe Wood

June 6, 2011

Contents

1 Introduction

Video gaming has become a major part of entertainment. In the past few years, its popularity has been increased by mainstream games such as Call of Duty, Halo, and World of Warcraft, along with smaller independent games like Super Meat Boy and Minecraft. Gaming is also one of the driving forces of computing technology, with complex games utilizing the power of new graphics and processor technology. Smaller and simpler games often expand the genre with an interesting artistic vision or simple gameplay. A subset of games, often labelled "casual" games, are simple easy-to-play games that appeal to a wide variety of gamers. Because complex games are difficult to develop and require a lot of resources, we decided that a casual game would be within the scope of our game development, and designed and developed a casual 3D video game for our senior project.

1.1 Course Structure

The course took place over two quarters, or roughly twenty three weeks. In the first quarter, we focused on game pitches, the basic game concept, setting up the programming environments, and programming the basic game system. We formed a team of three people in the first few weeks of the first quarter. Our team consisted of Chris Gilson, Annabel Hung, and myself. We programmed many aspects of our game without art assets, and used simple spheres and simple art in order to set up the game logic and basic gameplay. At the end of the first quarter, we had a basic game with simple gameplay and simple art.

For the second quarter, Chris Gilson left our team and took the Food Fight game in a different direction. Annabel and I continued work on our game, adding a few new features, fixing bugs and adding aesthetic changes into our game, and adding additional art. After playtests, where other people played our game, we altered the difficulty and asked them for their feedback, eventually incorporating it into the final project.

2 Project Overview

2.1 General Description

Food Fight is a casual arcade-style action game, where you control a kid and throw food at other kids. The setting is in a modern cafeteria, and the characters move around, throwing food at each other and eventually making the cafeteria messier as the gameplay progresses. The goal of the game is to hit the other kids with food, in order to score more points and make the cafeteria messier in a limited amount of time. The other kids also throw food at you, so avoiding them and avoiding the food they throw is key to getting a higher score. The game is played from a top-down perspective, with your character always in the middle of the screen. The amount of time, your current score, and your current inventory of food are all displayed on the screen. The game is controlled with the mouse and the keyboard; left-clicking on a location on the screen causes you to throw food at that location, right-clicking on a location on the screen causes your character to move to that location. Pressing the numbers 1, 2, or 3 causes your character to switch which food they are throwing.

The look and feel of the game is cartoony and unrealistic. The characters are exaggerated for aesthetic effect, and the proportions of food are unrealistic in order to emphasize certain aspects of the game. Food splats are bigger than they should be in reality, in order to give the player the sense of making the cafeteria messy. The other kids have exaggerated bodies so that it is easier to see where they are and hit them with food.

Because our game is a simple arcade-style action game, it does not have a long story like other games. The game is at its core about a kid throwing food at other kids, creating a mess of the cafeteria in the process. When different people with different playstyles play the game, they may create different stories when they play. For instance, one player may carefully throw food in order to keep the cafeteria clean and only hit the enemy players, saving food for scoring more points. Another player may not mind getting the cafeteria dirty and wasting food, ensuring that they hit the other kids at the expense of extra points.

2.2 Technical Description

We programmed our game using primarily C++ and OpenGL. C++ is a powerful programming language that is widely used in the gaming industry. Because C++ is similar to C, it was easy to work with and we were able to program in it easily since all of our team members were familiar with it. C++ also has some aspects, like classes and inheritance, that make it well suited to game development. Certain objects in the game world can be classified and sub-classified using inheritance, making game programming easier. For example, in our game, we declare an entity object, representing both the player and the enemies. Our collision logic works when comparing any two entities, and does not need to worry about colliding the myriad of different types of entities present.

OpenGL is a cross-platform graphics library API that makes developing 2D and 3D games easier. Our actual implementation of OpenGL has many parts, including OpenGL (GL)[1], OpenGL Utility (GLU), the free OpenGL Utility Toolkit (freeglut/GLUT)[4], the OpenGL Extension Wrangler (GLEW)[3], and the OpenGL Shader Language (GLSL)[2]. The OpenGL suite, consisting of GL, GLU, and GLUT, are what we used to draw and display all of the graphics to the screen. We used GLEW to determine what functionality the graphics card in the machine possessed, and then enable additional graphics functions to make them available to us. We programmed additional graphics shaders in the GLSL, and enabled graphics-approprate features (as reported by GLEW).

Another important technology we used in our program is Lua scripting.[6] Lua is a lightweight programming language that is very well suited for scripting events, information, and game actions into our game. We use Lua scripting to create and process various game content, calculate updated values rapidly, and load in user defined values at the start of the game. The flexibility of the language and the program hooks are what allow us to add new content rapidly and easily into the game.

3 Related Works

3.1 League of Legends

Figure 1: A League of Legends screenshot.

League of Legends is a strategy game developed by Riot Games, where you control a single character and move them around a wide-open battlefield, attacking other player characters and other units controlled by the computer player. [5] We looked at this game for many things, primarily the control scheme and viewing angle. League of Legends has a similar camera angle to our game, along with a similar style of gameplay. For that reason, we modelled our game controls after theirs, using a simplified version of their movement and attacking controls in our own game. League of Legends also uses Lua to store, modify, and alter game values during gameplay. We sought to emulate this model on a simpler scale.

3.2 Fat Princess

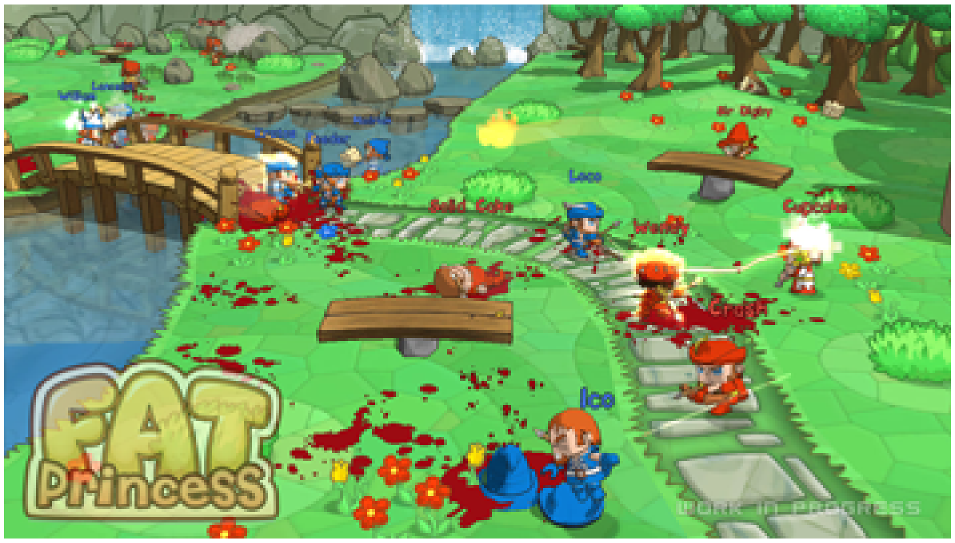

Figure 2: A Fat Princess screenshot.

Fat Princess is a game developed by Titan Studios for the PlayStation 3.[7] This game had an art aesthetic similar to our game, along with a similar camera angle. One of the things we used was the music from Fat Princess; we felt that the style of the music matched our game, and the game had a comical mischevious feel to it, two aspects that we wanted to emulate in our game. The gameplay was also slightly similar to our game, in that you control one character and attack other characters. However, because Fat Princess was only available on the PlayStation 3 at the time of our game development, we did not look at it for control styles, as it used a Playstation 3 controller, and we were programming our game for the computer. The art style was very cartoonish, with lots of solid primary colors and colors that made the game look cel-shaded.

4 Algorithms Overview

We used a variety of different technologies and algorithms to make Food Fight. Below is a list of various technologies that the team members contributed and used during the development of the game. Note that some of the final technologies described in our project may differ from the ones described in other senior projects.

4.1 Technology List

- 3D Interactive Game Environment (Chris Gilson, Annabel Hung, Eric Fong)
- Original Art and Models (Chris Gilson)
- Audio and Sound Effects (Eric Fong)
- Lua Scripting (Eric Fong)
- Occupancy Grid (Annabel Hung)
- Collision Detection (Annabel Hung)
- Enemy Artificial Intelligence (Annabel Hung)
- View Frustum Culling (Chris Gilson, Eric Fong)
- Particle System (Chris Gilson)
- Game UI Overlay (Chris Gilson)
- Food Selection System (Annabel Hung)
- Game State System (Annabel Hung)
- Collision and Game Logic (Annabel Hung, Eric Fong)
- Game Controls (Eric Fong)
- OpenGL Shaders (Chris Gilson)
- Projected Planar Shadows (Eric Fong)

5 Algorithms Details

5.1 3D Interactive Game Environment

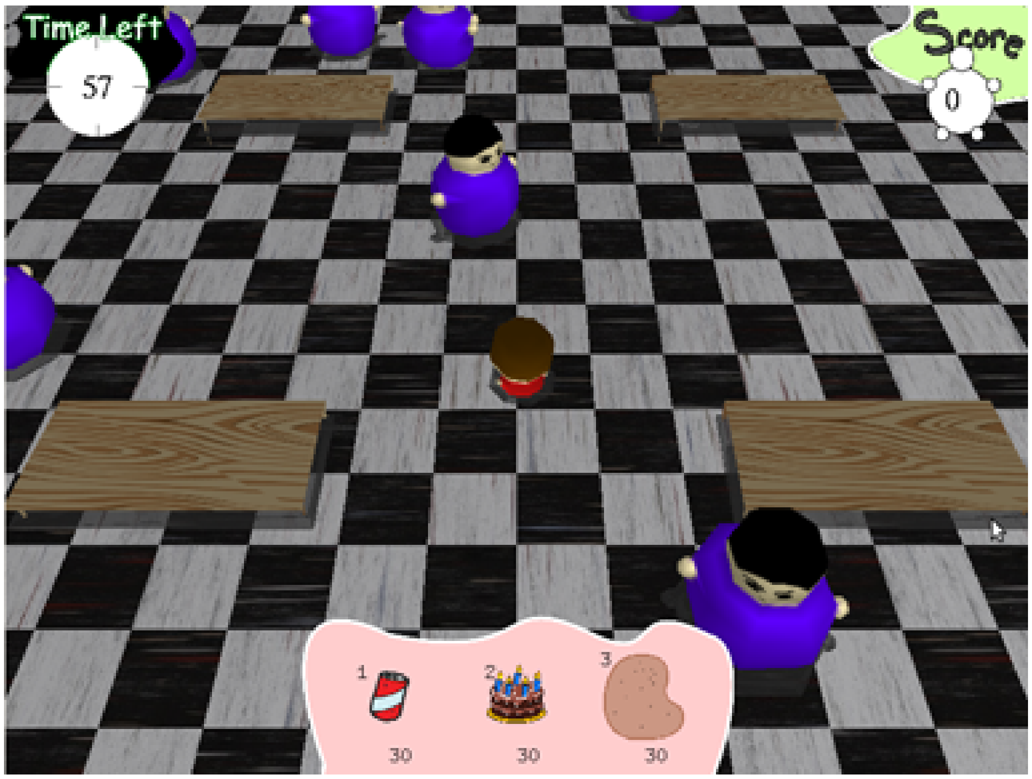

Figure 3: The basic level layout.

The level layout was designed by all of the team members. Because we only planned on having a single level, we wanted to make an interesting level. The design goal was to have two different areas; a center area that was fairly open, and an outer area that was narrower. We accomplished this by designing our single level to have four rectangular tables, positioned in a square around the room. The middle area is where players start the game, and it is open and easy to hit other kids from that location. However, it is also easy for them to hit you. Players can easily move to the outer area around the tables, making it harder for them to hit other kids, but also makes them harder to be hit.

5.2 Audio and Sound Effects

Figure 4: irrKlang, an audio engine.

For our game audio, we used an off-the-shelf audio engine. Because audio devices have not been as standardized into operating systems as graphics devices have been, it was difficult for us to use any sort of audio library like OpenAL. The simplest solution for us was to use irrKlang, an audio library designed to make playing audio easy.

For music, we used the music from an existing game called Fat Princess. Because of the nature of both Fat Princess and Food Fight, we felt the music matched both games well. Both games have a sort of michevious and comic feel, and the music fits well for both games. For sound effects, we used sounds from freesound.org to add to the game. The most common sound is the splat.

5.3 Lua Scripting

Figure 5: Lua, a lightweight programming language.

Lua was used as a scripting language in our game, which allowed us to expand and control various content in our game. Lua is used to store the information of all the foods used in the game, allowing us to make quick balance changed and even allowing players to add their own foods. Lua was used so that we could rapidly prototype content into the game and add additional game logic where necessary.

Because Lua is so flexible, we used it for many parts of our project. One of its simple uses was loading values into our game from an options file. The options file itself is stored in Lua syntax, and accessed through specific Lua function calls. It is easy for users to modify the file to change the details by hand, but it is simple to provide a front-end to change the values as well, with UI elements like a slider or number to determine difficulty. By loading all those values in Lua, we can do some things that would be more difficult if simply loading it directly through file I/O. Firstly, loading values from Lua is much simpler and easier than doing direct file I/O. Secondly, if an external source accidently modifies the file, it may end up corrupted. In Lua, it allows us to use some of its complex functions to access set program defaults. By using Lua metatables, we can specify a location for the program to access information if it is unable to access it from a specific location.

Another use of Lua in our game is to modify foods effects when they hit a target. We only use a simple modifier that does not change the food much when it hits a target, but we included Lua program hooks so that the food can be manipulated by an outside script. Whenever a food hits a target, Lua can be used to modify the food before it assigns its damage. This allows us to program additional logic into the foods, making it simple to add complicated logic like "area of effect" attacks, increasing food speeds, and modifying food damage. This complexity is what makes the game interesting. Although we kept the foods and their corresponding hooks simple, the inclusion of Lua allows us to add more engaging gameplay.

5.4 View Frustum Culling

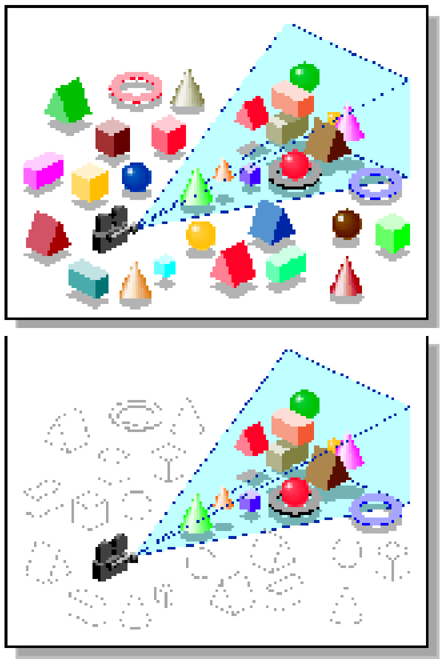

Figure 6: A visual description of view frustum culling. The first image is every shape being drawn. The second image removes shapes that are not in the camera's view.

View frustum culling is a technology used to speed up the graphics processing step of our game. First, we calculate the six planes of the viewing frustum, the camera's view volume. Then, before we draw any object in our game, we compare its position to all six of the planes. If any object is outside of the view frustum, we cull it and do not draw it in the scene at all. This speeds up the performance of our game significantly because we have many objects that end up not on the screen, and not drawing them saves time during graphics processing.

One optimization that we do not do is that we do not cull shapes that are "hidden" behind other shapes. For example, in the figure above, there are some items that are actually not visible to the viewer because other objects are completely blocking their view. In our situation, we felt that this situation did not arise very often, because we use a top-down perspective, and items would rarely need to be culled if completely blocked from view by another object.

5.5 Collision and Game Logic

Because the core of Food Fight's gameplay relies on collisions and what happens when things collide, the collision and game logic is one of the most important sections of the game. Whenever an entity collides with another entity, it goes through a complicated set of logic in order to determine a final outcome. The primary collision logic that was considered was food colliding with a player or an enemy. In this situation, we compare who created the food to the target; if they are the same, we do not update any values. If the creator of the food is different from the collided target, then we update the score values of the player, either my incrementing or decrementing, depending on who was hit by the food.

The entire collision logic is separate from the core game in two ways. One is that determining collision has been separated from the logic of a collision's aftermath. Determining a collision and determining what happens once two objects have collided were kept separate during development and testing, making it easy to create the correct logic, and allowing us to pinpoint problems in collision detection. The second part of collision logic is that it uses Lua to modify collisions and alter values based on complex situations in the collisions. This also happens separate of both actually determining collisions, and even storing variables in C++. By using Lua to store values, it was easy to write external scripts to test Lua's modification of values instead of having to play the game. The scripts also make it easy for us to add more complex logic in the future, by not having the script be compiled along with the rest of the program. The script itself can be tested independently of the rest of the code, cleanly separating complex collision logic from basic graphics and math calculations that make up the base gameplay of Food Fight.

5.6 Game Controls

5.6.1 Mouse Controls

The controls for the game required two important components. The first was analyzing any given mouse click on the game window, and drawing a line through the camera's view at the mouse click's location and intersecting the one-pixel vector with the ground plane to return a location on the ground plane that we utilized for either movement or throwing food. This is accomplished by first getting the X-Y coordinates of the mouse-click in the viewport and getting both the modelview and projection matrices. We then use a GLU function called "unproject" to get two points; the near point of the vector that goes through the pixel at X-Y, and the far point of the vector that goes through X-Y. The function gives us a specified point in unit space along a given set of X-Y coordinates. We do this twice to get the near and far values, and intersect a vector that we create using those two points with the ground.

The controls for the game were based on similar real-time-strategy (RTS) games. Although our game is not an RTS, it has a similar perspective to those games, which is why we developed the control scheme this way. However, there are some core differences that have made this control scheme less than optimal for playing our game.

One difference is that, in normal RTS, the player is controlling an entire army in most cases, not just a single unit. However, because the player often wants to move and throw food at the same time, the player must use the same controller for both actions, which is difficult to do if a player is not used to that style of control. One aspect that we had not considered when using this control style is that in all of the example games we looked at, we forgot that the character(s) being controlled have subtle artificial intelligence controlling them that make them much easier to control. In League of Legends, a mouse-click on an enemy tells your character to go to that enemy, and attack that enemy when in range. It uses the same philosophy of using the mouse to both move and attack, but has the subtle change of adding intelligence to the controls. If a character clicks on an enemy, the game assumes they want to attack that enemy, and even moves the character into range to attack if necessary. The opposite is also true; League of Legends assumes that if you move into range of an enemy that you will want to immediately attack it, and automatically makes the character perform attacks on nearby enemies. Because our game does not provide these AI assistants, mouse control becomes much more difficult to use in practice. It is much more difficult to move and attack at the same time if each action has to be performed identically.

Another problem with using the mouse control scheme was that our game was not actually an RTS game. Even though our game is played from the top-down perspective of an RTS, our game has action game concepts that do not mesh well with RTS game concepts. One example is attacks and misses: in RTS games, attacks do not “miss.” The opposing characters cannot move out of the way of an attack, they will simply be hit. This makes RTS AI easier to program; when in range, it will simply hit successfully, and animate a projectile to indicate an attack is in place. For our action game, we cannot use this method of simple attacks to determine an action for our player because we wanted to use collision detection and player decisions to throw foods, forcing the player to aim manually. It is this manual aiming combined with movement on the same control scheme that causes problems; with no additional AI to help support the player, the mouse control scheme loses some of its potential and becomes increasingly complex.

Using the mouse control scheme has its benefits. One of them is simplicity, which is in line with our small team goals. The controls of the game are all vector based; whenever any point is clicked on the screen, a vector is made from the character's position to the point of the food. This vector is then used appropriately by the object it is associated with; if the vector is used for food throwing, it is attached to a food and used for animation and positioning. If the vector is used for movement, then it determines the character's movement and next position in the world. Because of collision detection, it is important to have a movement vector in place in order to check the "next" position of an object. The mouse control scheme lets us calculate simple vectors.

Another advantage of mouse control is that is analog instead of digital. The mouse control allows us to alter the speed of a player depending on how far away they click from their character. Because the vector has a higher magnitude, the player animates more quickly to their destination. If the player clicks closer to their character, they will move more slowly. One strategy that arose during playtesting was players clicking close to themselves to move slowly, allowing them to dodge incoming foods but still throw accurately.

5.6.2 Alternate Control Schemes

One alternate control scheme we experimented with was using the keyboard to move as well as select which food you used. Keyboard controls felt similarly restrictive as mouse controls, mostly due to the fact that the keyboard controls felt very stiff with movement limited to only four directions. Although we did not experiment further with keyboard controls, some improvements would probably have made them more suited for our game. One improvement we could have made was to make keyboard controls smoother, by adding smooth animation rotation and eight-way controls. Also, when we added keyboard controls, we used a different animation system, translating the model and camera per-point instead of using vector transformation.

Another change to keyboard controls that would have been effective would be to add a movement vector to the player depending on what combination of buttons are pressed. This way, we could keep the existing animation/positioning system but calculate the movement vectors depending on which buttons were being pressed. Also, it is difficult to vary the acceleration of a character realistically using only a keyboard with digital 8-way controls; although mouse controls are not realistic, it allows for full 360 degree movement, along with varying speeds, that would be very difficult to simulate or replicate with a keyboard. Even trying to make the simplest shadow is difficult, but it adds a simple aesthetic to the game that improves the cartoon aspect of the look. Because of the way our graphics are drawn, adding these kinds of simple shadows are simple, but not the most realistic. However, because our game was not a realistic game, we decided to use the simple algorithm, keeping our shadows simple and easy to program.

5.7 Projected Planar Shadows

Even trying to make the simplest shadow is difficult, but it adds a simple aesthetic to the game that improves the cartoon aspect of the look. Because of the way our graphics are drawn, adding these kinds of simple shadows is easy, but not the most realistic. However, because our game was not a realistic game, we decided to use the simple algorithm, keeping our shadows easy to program and debug.

5.7.1 Shadow Algorithm Details

Because we draw every model directly by using OpenGL primitives, we can use a simple matrix transform to draw the model a second time, except flattened against the ground and colored grey, to imitate a flattened shadow. This is accomplished by finding a matrix transform that will flatten a model and draw it in only two dimensions along the ground.

$$M = \begin{pmatrix} l_y & -l_x & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & -l_z & l_y & 0 \\ 0 & -1 & 0 & l_y \end{pmatrix}$$

Figure 7: Projected Planar Shadow Matrix, projecting onto the X-Z plane.

We provide the position of the light in the scene and put it into the matrix. When the OpenGL primitives are multiplied by the matrix above, it transforms the model to a flat plane, along the floor. We use a light source directly above each model so that the shadows are always drawn directly below the model. Because this calculation is so simple, it is a simple matrix transformation to draw the shadow. By disabling texture mapping and coloring the “model” grey, it makes a simple shadow. Because our game is graphically simple, the style of projected planar shadows fits well, and because the computation is simple, it was easy to integrate and program into our game.

6 Results

6.1 Accomplishments

Despite missing enhancing features, *Food Fight* has progressed from a simple idea to an actual game. The first step was to ensure the application functioned properly, and then to find a balance between the game's difficulty and entertainment value.

We believe that we have developed a game that is fun and challenging with a good level of replayability. Perhaps the best feature about *Food Fight* is that many players have found it quite amusing to throw food and make a mess while trying to earn points. Because our game has very simple rules and controls, a new player would not need to deal with a steep learning curve and can immediately begin enjoying the game. We also feel that the art and music selection both compliment the game, adding to its cute and funny style. Overall, we are most proud of creating a playable game that people can actually enjoy.

6.2 Game Images

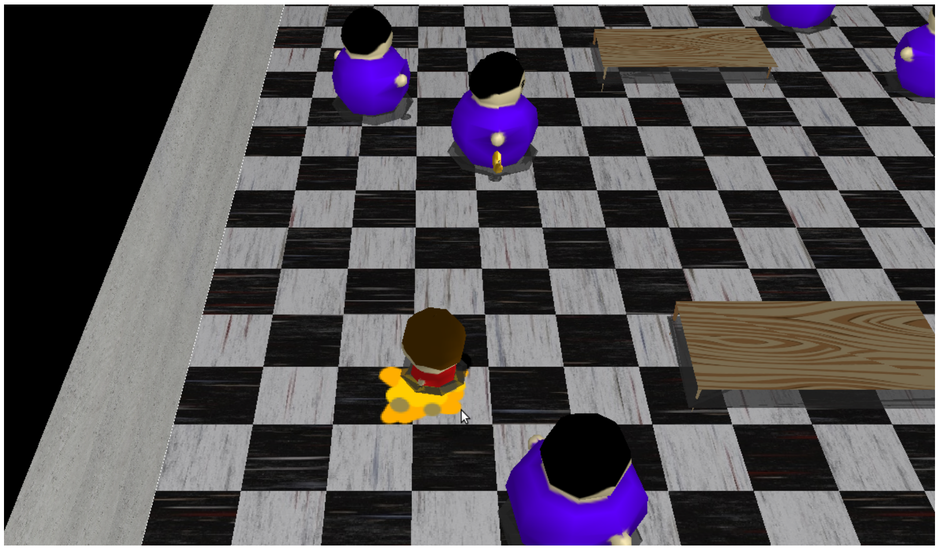

Figure 8: Enemies attacking the player.

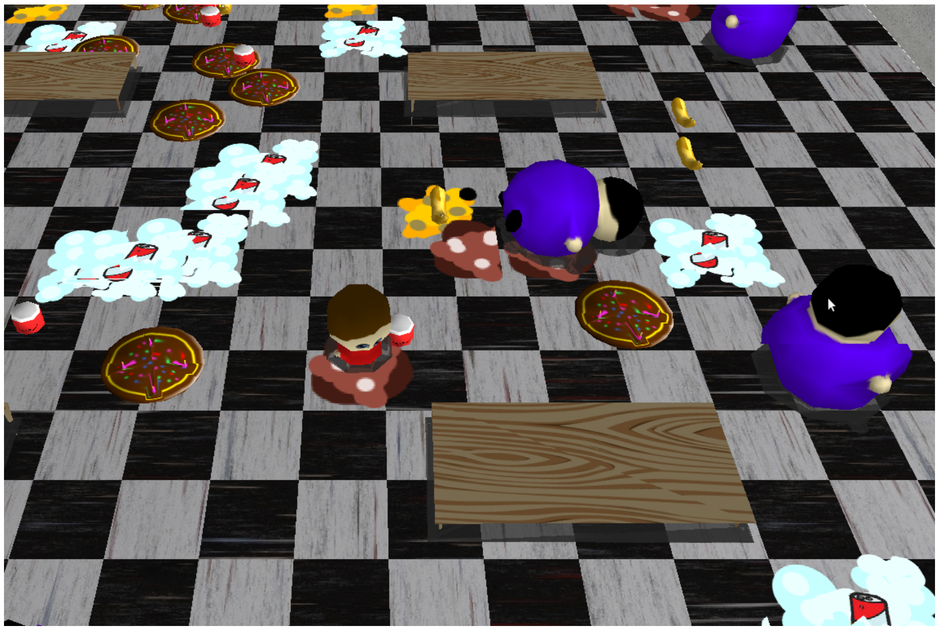

Figure 9: Enemies fallen down after hitting food.

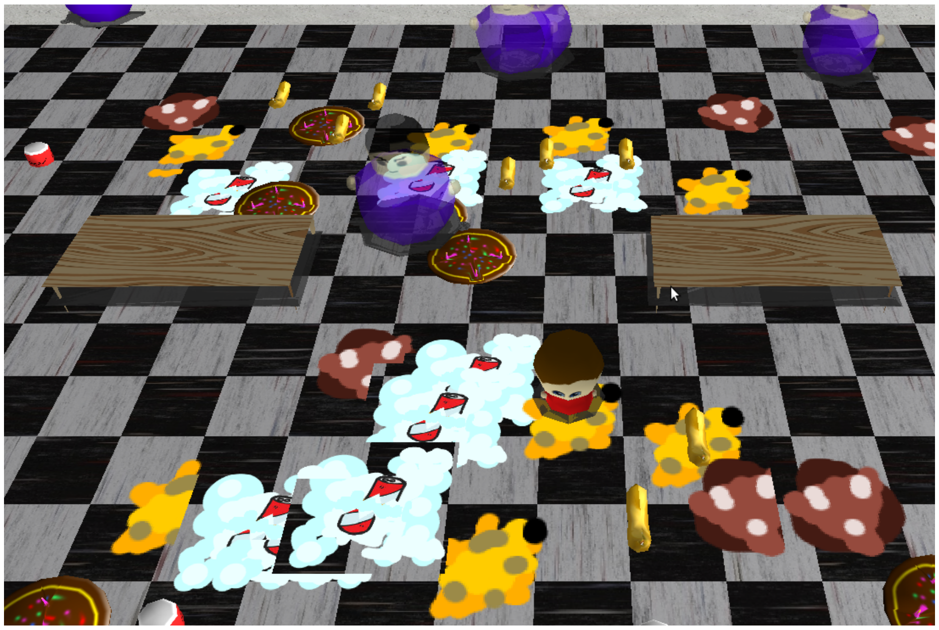

Figure 10: Enemies are slightly faded for a few seconds after getting up.

6.3 Player Feedback

We asked players to provide feedback in a few specific ways. One of them was difficulty; we wanted to know how to tune our game to make it more accessible to players. Another area we wanted feedback in was how the controls were and if the player was able to play the game. The last area we asked people about was how stable the game was and if it seemed error-prone and buggy.

Tester	Stability	Fun	Difficulty	Suggestions
1	5 of 5	4.5 of 5	too hard with many enemies	temporary invincibility after getting hit
2	No crashes	Cute, funny idea	Really hard, controls are weird	Make it easier
3	stable	Sort of fun	Hard, too easy to keep getting hit	Make the enemies not throw so much
4	stable	Pretty fun, funny idea	Controls fine; AI is hard at first	Differentiate foods
5	good	Cute	Hard controls, game is too hard	Make enemies attack less

Figure 11: Summary of player feedback.

6.3.1 Difficulty

Our game started out more difficult than intended, with too many enemies and "smart" enemy AI. It was simple for us to edit the number of enemies and let the testers play a second time, and that was based mostly on skill. People who were familiar with games and gaming did not have a challenge with more enemies on the screen, while players who were not familiar with games had an easier time with only three or four enemies. Players also believed that the score was a poor reflection of their performance; scores below zero did not make sense to most players, and made the players feel like they were doing more poorly than they were. In our second round of testing, we set the number of enemies to a medium between hard and easy, and did not allow the score to drop below zero. This led to a more consistent experience between testers. After player feedback, the gameplay was explicitly changed in a few ways:

- The player will no longer fall down when hit. Players were frequently hit and thus spent most of the game time unable to run away.
- The amount of time the enemy between enemy attacks has been increased.
- The distance considered within range between an enemy and the player has been increased.
- An enemy is worth no points while he is fallen.
- An enemy is worth no points for a few seconds after getting back up.

6.3.2 Controls

Controls were a big part of our testing because of the feedback we had received all throughout the game's development. When new players used the mouse, they did not have a problem; they understood clicking one button to move and one button to throw food. Players who were more familiar with gaming in general, but not the top-down perspective or RTS controls, immediately tried to use keyboard controls to move. Even during play, these testers kept continuing to use the keyboard even after it did not work, and only remembering after a few presses that the mouse was used for movement and throwing food. Experienced gamers who were familiar with other RTS games had no trouble controlling the game itself, but had expectations for certain things to happen automatically, such as automatically attacking enemies in range or moving to an enemy automatically if clicked on.

6.3.3 Stability

Although we tested for stability and bugs, all aspects of the game functioned as intended. Players instead commented on visual bugs or things that looked out of place. One example was texture z-fighting, where textures on the same plane are drawn strangely. A few players commented on that, asking what it was or what it meant. Other than that, the game was stable and did not crash during testing.

6.4 Development Process

Because our team consisted of only three (and then reduced to two) members, we did not really have a team manager. For both quarters, the project was divided into two parts, graphical and logical. Because we had to begin writing code soon after our idea was approved, there was little time for planning, such as project design and work assignments. As a result, the team ended up with code that was difficult to maintain and an imbalanced work distribution.

The end of the first half of development saw changes, both good and bad. The good effect was a massive code reorganization, which made the second half of development significantly easier. The bad effect was the loss of the team member who was most experienced with graphics. The team dealt with this reduction by focusing on improving the gameplay portion of the game, which was their strength. Their hope was that even though the game did not look as well as it may have, it would still be a very fun game to play.

7 Conclusion

One of the most important conclusions that I came away from this project with is that content is very important to a game. I have been a personal believer that gameplay is more important than content, and that a good game that has good mechanics will become greater than its lack of content. That may be the case, but for a majority of projects, good art and a good aesthetic will improve a game significantly. Another aspect is the type of content that is in a game; by producing a game using abstract art that can be programmed, or very simple art that is improved by complex graphics shaders, less art content is required to make a good visual experience.

I gained a lot of experience with small-team development as well. Because we began as a team of three, and then developed into a team of two, we found that developing the basics of our game was difficult, and additional content was hard to make. Because we had planned for this, we have a game that feels relatively light on content, but with a system that is easy to add content to. Developing a program this way, especially a video game, is difficult because it does not feel like a lot of progress is being made. However, increasing stability and testing the inclusion of simple models increased my programming confidence; it was really rewarding to know that it is possible to integrate future content, even if we are unable to make that content ourselves.

One of the most rewarding aspects of the project was seeing how good programming practices improved the program overall. In classes, we are taught a lot of maintennance and convenience tricks that are almost never worth using on small assignments. However, when working on a project as large as this, finally putting those programming tricks to use and having them increase your productivity and the quality of the product is very rewarding. I truly felt like I confirmed the knowledge that I learned from my other classes by using it and implementing it in our program.

8 Future Work

Our project was developed with future work in mind. Because most of the members of our team are not artists, adding artistic content is one of our future goals. We developed our project knowing that we would have sparse art assets. By hiring artists or acquiring new art, we hope to improve the aesthetic of the game by increasing the number of models and textures in the game.

Another aspect of the future is to expand on the scripting in the game. Because we considered complex logic part of "content," we avoided programming complex gameplay that could unbalance the game and make it not fun, and instead focused on stability and smoothing out content integration instead of making the content itself. Also, the existing models and art that we have did not mesh well with some of the more complex art we had in mind. By increasing the complexity of our game logic, we can make the game more engaging as a whole. Adding an entirely new level of strategy to the game would be the next step, increasing the knowledge of the game required and allowing experienced players to use their knowledge and improve at the game.

References

[1] Khronos Consortium. Opengl.
http://www.opengl.org/.

[2] Khronos Consortium. Opengl shader language.
http://www.opengl.org/documentation/glsl/.

[3] Milan Ikits, M. M. Opengl extension wrangler.
http://glew.sourceforge.net.

[4] Olzsta, P. freeglut.
http://freeglut.sourceforge.net/.

[5] Riot Games. League of legends.
http://www.leagueoflegends.com/.

[6] Roberto Ierusalimschy, Luiz Henrique de Figueiredo), W. C. Lua programming language.
http://www.lua.org/.

[7] Titan Studios. Fat princess.
http://us.playstation.com/games-and-media/games/fat-princess-ps3.html.