Food Fight!

Christopher Gilson

June 2011

1 Introduction

Efficient real-time rendering of a graphical environment is an on-going challenge in the world of computer graphics. Video game development is on the forefront of efforts attempting to conquer this challenge, where the balance between realistic, vibrant graphics, and fast, seamless performance is paramount.

Today's production video games are created by teams of often hundreds of developers, programmers, and artists. The object of this project was to begin to understand some of the common challenges in real-time rendering, and some commonly used solutions, as well as to understand the myriad of skills required to create a finished video game.

This project was developed as part of CSC/CPE 476/476++, which is a collaborative game development course series. Games were developed by teams of 3-6 members over the two-quarter duration of the course, although some teams chose to work separately during the second quarter. Four regularly scheduled demo milestones during each quarter were used to assure that students were completing what was expected, within the expected time frame. The project team was as follows:

Quarter 1:

- Eric Fong
- Anabel Hung
- Christopher Gilson

Quarter 2:

- Christopher Gilson

2 Project Overview

2.1 Genre and Setting

Food Fight falls under the umbrella of Casual Games. Casual Games aim to provide a simple, fun environment for entertainment which requires little to no experience or time to enjoy. Development focused less on complex gameplay mechanics, and more on visual satisfaction. The ultimate goal was to give the player the same feeling of destructive, messy success and satisfaction that one gets when throwing a watermelon off of the top of a tall building.

2.2 Objectives

The objective of the game is to complete each level within the level's time limit. Each level contains a certain amount of enemies. To complete a level, each enemy must be removed from the room by first knocking them over by throwing food at them, and then pushing them to the level's exit. When all enemies have been removed, the level is complete and the next level begins.

2.3 Look and Feel

Food Fight was intended to have a silly, over the top, cartoonish look. The artistic theme could be described as messy. The major visual mechanic that achieves this is the splat system, where any food item colliding with any object adds some colorful messy graphic to the environment. This quickly adds up, making the environment incredibly colorful and messy looking. In order to facilitate the feeling of dirtying the environment, the starting environment was made to look very clean and neat, accentuating the mess that the player creates through the course of each level.

2.4 Story

Food Fight's story is incredibly simple, as the aim of the game as a whole was simple, fairly mindless, cathartic fun. Your character is essentially the outcast in a school environment, who has been picked on and tormented by everyone else in school, even the teachers. Fed up with this horrible learning environment, you've chosen to strike back, and you will not be satisfied until the school bully, Billy, has been defeated once and for all.

2.5 Technical Frameworks / External Software

- Host Language: C++
- Graphics Library: OpenGL
- Windowing: GLUT
- OpenGL Extension Library: GLEW
- Audio: Irrklang Free
- Modeling: Blender
- Texturing: Blender / Photoshop / GIMP

3 Related Work

3.1 Fat Princess

Fat Princess by Titan Studios, for the Playstation 3 was a major influence on the concept and graphical style of Food Fight. Fat Princess is an incredibly fun and involving game, with beautifully simple graphics and simple but fun gameplay.

Fat Princess' game play.

Food Fight's concept was, essentially, to create a game that would allow all the silly, graphically over-the-top fun of Fat Princess, without as much blood. One main difference between Fat Princess' concept and Food Fight's is that Food Fight is much more focused on single-player, while all the gameplay, whether it be single- or multi-player, is very much team oriented.

3.2 Animal Crossing

Nintendo's Animal Crossing for the Nintendo Gamecube, as well as it's predecessor for the Nintendo DS, were also very influential on the graphical style of the game. Animal Crossing's style of completely cartoon-ish art, without the explicit use of cel-shading, created a fun but surprisingly realistic look that Food Fight attempted to emulate.

Animal Crossing's visual style.

3.3 Pikmin

Nintendo's Pikmin, for the Nintendo Gamecube, served as an additional gameplay influence. Pikmin included a game mechanic where, loosely described, you would defeat an enemy, then bring that enemy back to your home base in order

to accumulate points, which could be used to obtain more minions that could then be used to defeat more enemies. This cycle of defeating enemies in order to get points, which were in turn used to defeat more enemies, was a good one, and was the basis for the point system used in Food Fight.

Pikmin carry a defeated enemy back to base.

Pikmin, however, is a game with a larger, overreaching goal, which requires much more additional time investment to complete. Food Fight's aim was to create a game with a number of smaller, easily surmounted goals.

4 Feature Overview

4.1 Quarter 1

- 3D interactive game environment (Christopher Gilson, Annabel Hung, Eric Fong)
- Original Art and Modeling (Christopher Gilson)
- LUA Scripting (Eric Fong)
- Occupancy Grid Collision Detection (Annabel Hung)

- View Frustum Culling (Christopher Gilson, Annabel Hung, Eric Fong)
- Particle Systems (Christopher Gilson)
- HUD (Christopher Gilson)

4.2 Quarter 2

All Quarter 2 features were implemented by Christopher Gilson unless stated otherwise.

- Click To Throw (Christopher Gilson)
- Shadows (Christopher Gilson)
- Mess Mechanics (Christopher Gilson)
- Rolling Enemies Effect (Christopher Gilson)
- Dodge Game Mechanic (Christopher Gilson)
- Linear-interpolation based effects (Christopher Gilson)
- Player Rainbow Shield (Christopher Gilson)

5 Feature Details

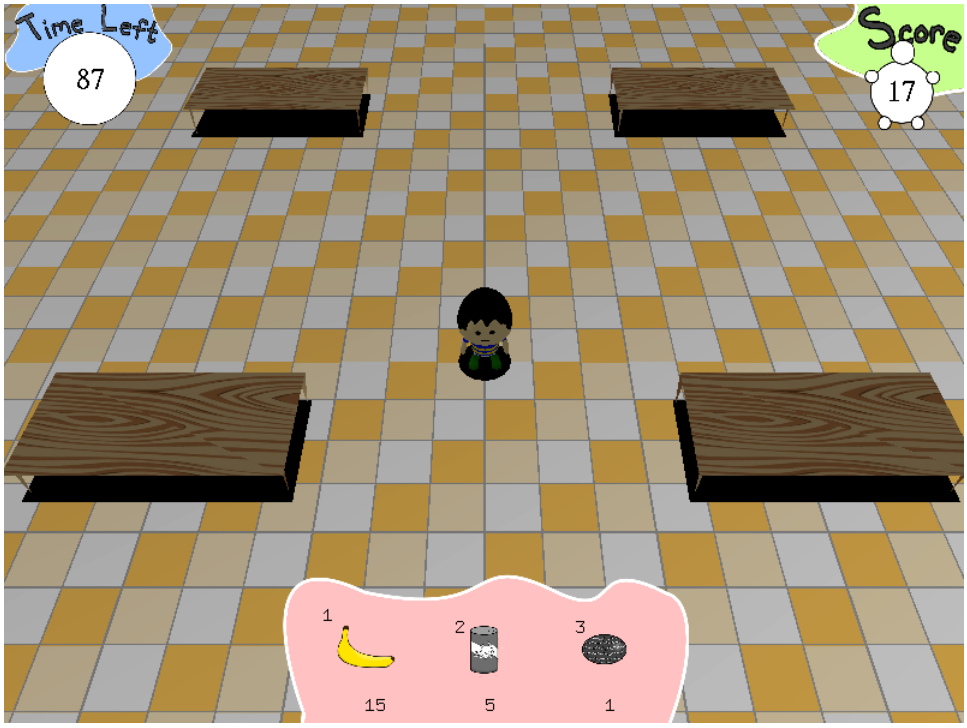
5.1 Food Selection

The food selection screen is drawn in 2 stages. First, the 2D background is drawn, including the vending machine background, the text-based buttons, and the food name, description text, and price. The depth buffer is then cleared, and the 3D foods are drawn over the already drawn 2D aspects. Click detection is handled by recognizing the absolute positioning of the mouse click, and becomes inaccurate if the window is resized.

5.2 The Basic Cafeteria

5.2.1 The Four Walls

The walls and floor of the cafeteria are sized and drawn based on values imported from LUA scripts. The door, as well as exit sign, are drawn as textured quads directly in front of / above the wall and floor surfaces so that they are visible even if the wall behind is very dirty.



The Game Begins!

5.2.2 HUD

The HUD is the last thing to be drawn into the scene, and is drawn over the game environment. First, the depth buffer is cleared in order to assure that the HUD appears on top of any already rendered geometry. The HUD is then drawn under an orthographic projection in viewport coordinates to simplify drawing. The positions of all HUD elements are drawn with respect to the window's width and height so that the HUD draws properly regardless of window size.

5.3 Effects and Mechanics

5.3.1 Click To Throw

The challenge of implementing a click-to-throw system involves correctly translating the coordinates of an actual mouse click into coordinates in world space. OpenGL uses two different matrices to keep track of and apply geometric and perspective transforms to geometry. The Model View matrix keeps track of geometric transforms such as rotations, translations, and scales, including camera

transforms (using the *gluLookAt* command). The Projection matrix maintains the state of the viewing volume, allowing for orthographic (rectangular) viewing volumes, or perspective viewing frustums. Because all geometry is transformed according to these matrices, the coordinates of a mouse click on the game window must be transformed as well, in order for the click to be accurately interpreted.

This transformation of the mouse coordinates is done using the *gluUnProject* function. The function takes in an x, y, and z coordinate of the original click, as well as the current Model View and Projection matrices. It then uses the current viewport (which determines the numerical boundaries of the window itself), and transforms the click according to the ModelView and Projection matrices. The z coordinate of the original click is an important argument to this function. A z coordinate of 0 will return the world coordinates of the mouse click on the near plane of the current projection. A z coordinate of 1 returns the world coordinates of the mouse click on the far plane. Both of these coordinates are needed for the final transformation.

Because the floor (which is where we are trying to find the coordinates of the mouse click in respect to) is always in view of the camera, it is certain that the floor is located in between the near and far planes of the view volume. The floor is (due to hard-coding) always located on the plane $y = 0$, regardless of size. So then, the coordinates of the mouse click in respect to the $y = 0$ plane are found by finding the point where the line between the mouse clicks on the near and far plane are, and the plane $y = 0$.

The advantage of this method is that it does not rely on any hard-coded values, and is always accurate, regardless of changes to the viewport, Model View, or Projection matrices, so long as the floor remains at $y = 0$ (although the algorithm could easily be adjusted to work for a floor on any plane within the view volume)

5.3.2 Shadows

The currently implemented shadows are created with overwhelming simplicity. Every game object that casts a shadow is rendered twice. Once normally, and then again in black, without any lighting or texture calculations, flattened using a *glScalef* command.

Shadows of different models

5.3.3 Making a Mess

The mess making mechanic refers to the effect where whenever a food item or particle system particle touches the ground or another player, it creates some small puddle shaped graphic (a splat) on whatever it touched. It adds a major feeling of fun and accomplishment to the overall gameplay.

Messy Walls, Messy Floors Each wall, as well as the floor, has two textures. The first texture is the base texture, (orange tiles on the floor, striped pastel colors on the walls. This base texture is static, and is always painted. Each surface also has an individual, completely transparent texture, that is painted over the base texture. Whenever a food contacts either a wall or the floor, a splat is then painted onto the transparent texture (the splat texture).

The splats are painted onto the texture using a Frame Buffer Object (FBO). First, the FBO is bound, and the splat texture of the surface to be painted on is bound to the FBO. New Projection and Model View matrices are pushed to the OpenGL stack, and cleared with *glLoadIdentity()*. The current viewport is

then stored, before being set to the exact dimensions of the actual wall texture. An orthographic projection is then set, with the bottom-left at (0,0), and the top-right at (texture width, texture height). This assures that the world and viewport coordinates are the same. The Model View matrix remains as an identity matrix. At this point, the coordinates of the food's collision with the surface are transformed into the texture coordinates, and the splat is painted directly onto the texture.

A big, big mess.

This approach yields two very beneficial qualities. Because every splat is kept track of in one solid texture, as opposed to each splat being it's own quad, there is no risk of z-buffer related flickering when splats overlap. While this problem could also be solved by z-ordering the splats before each render, this approach solves the problem much more elegantly and efficiently. Additionally, this approach creates very little overhead for drawing the splats each frame. Each individual splat is only processed once, when it is drawn onto the splat texture, and after that the splats are all kept track of and drawn by one single texture, which speeds up processing greatly.

Messy Enemies, Messy Friends All enemies, as well as the player model, do also get dirtier as they get hit with more and more food. Because the texturing coordinates on each model are different because of shape and size, the exact position of the food impact on the model is not calculated. The splat is painted onto a texture that is then multi textured over the model's original texture in the same way that it is in the case of the walls and floors. Each model has it's own unique texture reserved for splats.

5.3.4 Rolling Enemies

The semi-realistic rolling effect that occurs when the player bumps into an enemy that has fallen down is achieved via some simple vector math.

We begin with the object's velocity, which is represented as a vector of the form (x,y,z), representing the magnitude of the vector along each axis. Because the enemies are (as of yet) incapable of flying, the y magnitude of the vector will always be 0. This allows us to, mathematically, deal with the vector as though it were a 2D vector on the X-Z plane, which will greatly simplify calculations later on.

Now we have a velocity vector of the form (x,z), representing the enemy's 2D movement along the ground. In order to achieve a realistic rolling effect, the enemy must be rotating at a speed proportional to the magnitude of the velocity, around an axis that is perpendicular to the velocity. Basic geometry tells us that the perpendicular vector of (x,z) will be (-z,x). The object keeps track of it's current rotation, and on each update, the rotation angle is increased by some base value multiplied by the velocity's magnitude. When drawn, the model is first rotate 90 degrees from it's current orientation around the y-axis (so that the model looks as though it is rolling around it's middle, not over it's head), then it is rotated the current rotation amount, around it's velocity's perpendicular (-z, x). Because the velocity is decreased at each update, the resulting effect is a smooth roll which slows down to a stop.

One disadvantage to this approach is that it treats the object like a cylinder, and in the case of more triangularly shaped models, the rolling effect does look somewhat unnatural. However, overall it creates a very nice rolling effect which requires little processor overhead.

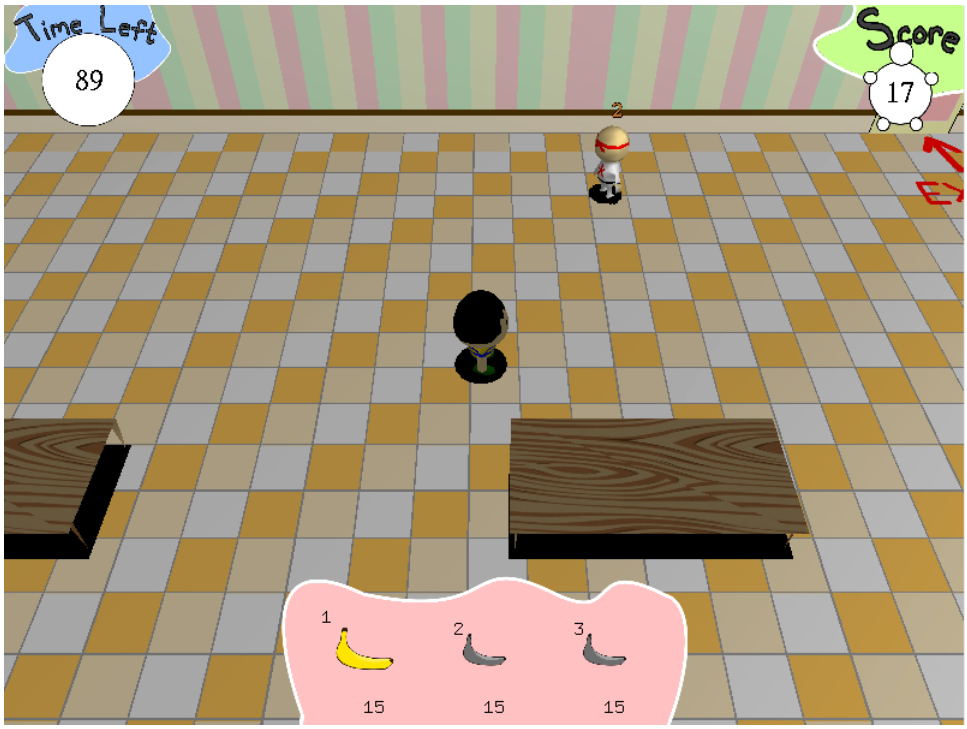
An enemy is knocked over.

5.3.5 Dodging Mechanic

The "Karate Kid" enemy has the ability to dodge any foods thrown at him from the front. This mechanic was also achieved using simple vector math.

Every game object has a velocity vector, which determines how fast, and in what direction, the object is traveling. It is possible to determine the orientation of two objects in relation to each other using these two velocity vectors. For the purposes of this calculation, "in front of" is assumed to be any direction within 90 degrees on each side of the velocity vector, making a full 180 degree "front".

First, the angle of each velocity vector is determined in relation to the vector (0,0,1) which is an arbitrary reference vector, in the range of 0 to 360 degrees using the dot product vector operation



The Karate Kid, unsuspecting

The 0-360 degree range is achieved by using a second reference vector (1,0,0) in order to determine whether or not the angle between the vectors is obtuse. Once the angle of each vector is known, a simple calculation tells us whether or not the angles are within 90 degrees of each other, and consequently whether the colliding object is approaching from the back or the front.

Note: The method used to determine the angle of the velocity vector relative to a reference vector is also used in the calculation that makes all the models face forward along their velocity vectors.

5.3.6 Healing and Haste Effect

The effects accompanying the "heal" (green) and the "haste" (red) abilities are both accomplished using the same method, using a different function to control particle displacement.

A wizard casting haste on a fatty

The distance between the two objects is calculated. The distance between each particle is then calculated according to the total number of particles in the effect. A cursor is placed at the origin object, and is then moved incrementally along the line, placing particles as it goes. Each particle's position is displaced by a pattern function before being drawn. The pattern function for the healing effect is a simple sin function, while the function for the haste effect is a translation upwards by a set radius, followed by a rotation by an increasing angle around a vector equal to the effect's original line, creating a spiral effect.

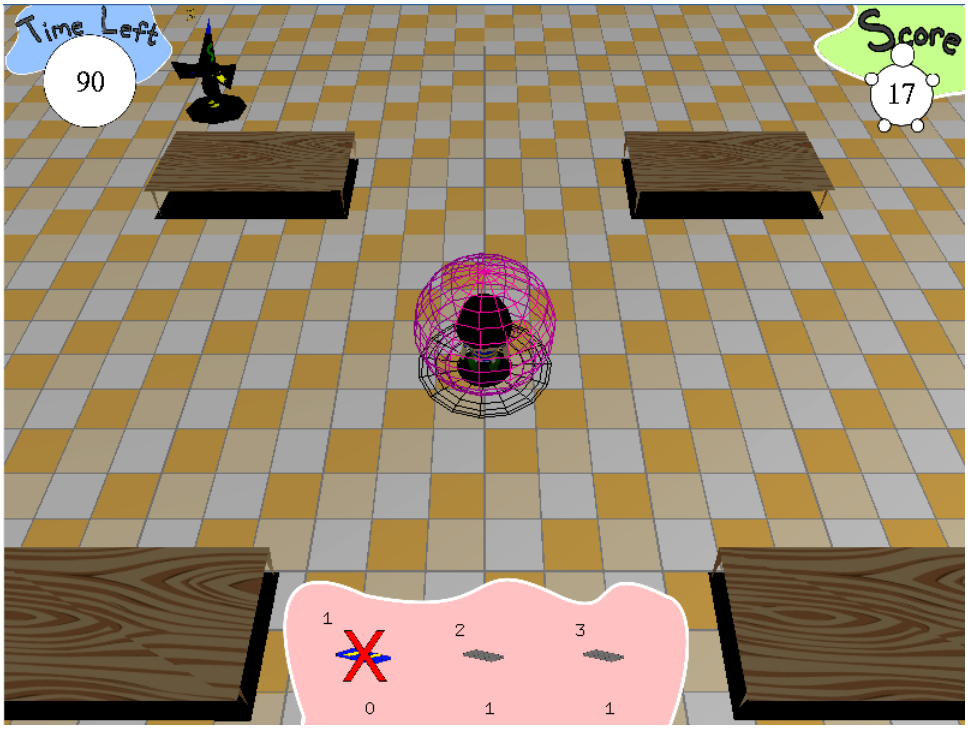
5.3.7 Player Shield (Color)

The player's shield effect, triggered by using the "candy" food item, is drawn using a simple *glutWireSphere* call. The rainbow effect however, is created using a specific sequence of color alterations.

Every color value is broken into three parts, representing the red, blue, and green components of the color. In order to achieve the oscillating effect seen in the player's shield, the color values are changed in the follow order, beginning

with a bright 100% green:

```
Increase Red
Decrease Green
Increase Blue
Decrease Red
Increase Green
Decrease Blue
```



The player's sugar-induced shield

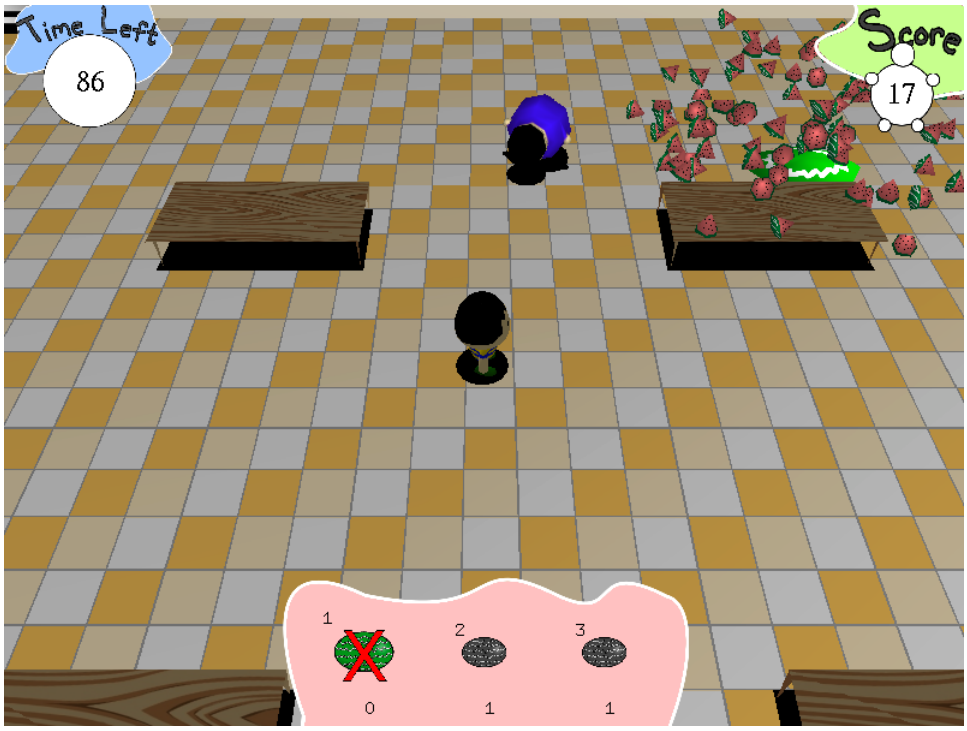
This causes a full rotation through the color spectrum without creating colors too close to black or white.

5.3.8 Particle System

Particle systems are used to render all of the fountains of particles when food contacts the ground, such as the explosion from the watermelon, or the spray from the soda. The mechanics and rendering of the systems are dealt with separately, so the soda spray, which is rendered as billboarded quads, are processed

in the same way as the watermelon explosion is, which is rendered as 3D models. **System Mechanics** Each particle system has a gravity, maximum radius, maximum height, total number of particles, a maximum number of displayed particles, and a list of particles. Each particle object has a position, a velocity, and a rotation angle. On each update, the particle list is iterated through. Each particle's position is updated based on it's velocity, and each velocity is updated based on the system's gravity. Additionally, each particle's rotation angle is updated by a fixed amount +/- a random amount. Any particles that fall below the floor (y ¡= 0) are removed from the system, and particles are then added at the source position with a random velocity within the maximum height and width constraints, until the total number of active particles is equal to the maximum, and any added particles are deducted from the total number.

This is beneficial in that it allows control of the density, size, and duration of the particle system, and it keeps the rendering of the system entirely separate from the processing. **System Rendering** System rendering is very simple. Each particle system contains a list of particles, and a specific model or texture to use to draw each particle. The list is iterated through, and each particle is drawn. Particles that are to be drawn as billboarded quads are rendered as point-sprites, which automatically create billboarded quads. Otherwise, a model is drawn at each particle's position, rotated by the particle's current rotation amount.



Watermelon mid-explosion.

6 Results

The result of this project is a fully functioning Video Game, featuring an interactive, fully 3D environment powered by OpenGL. Although many technologies are present and fully functioning, the development team is particular proud of some particular aspects of the project.

6.1 Resource Independence

With the exclusion of sound effects, all visual assets used in the game were created internally. All models present in the game were built, textured, and animated by the development team, using a combination of Adobe's Photoshop CS4 and Blender, a free 3D modeling program. The team artist had no prior experience with Blender, or any 3D modeling software, prior to the beginning of the course. We are very proud that we were able to produce quality assets and simultaneously learn the tools necessary for building those assets.

Title Screen

Food Selection Screen

6.2 Particle Systems

Food Fight's particle systems were designed to be especially generic. Although due to time constraint, there is only one "type" of particle system in the game currently, the particle system was designed to be as general as possible. The current model for particle systems could be used to model fountain-like effects, water-fall or shower effects, explosions, and even smoke clouds, with little to no modification to the system's internal representation. The main changes required would be in particle rendering, which was purposely abstracted away from the particle's mechanics.

6.3 Mess Mechanics

The technology used to create the food mess is the team's major accomplishments. We are especially proud of this because the method used to implement the effect never becomes more complex or costly, regardless of the number of splats that have occurred. The inspiration for the rendering of this effect came

from the mechanics of reality, where if a food were to really leave a mark on a wall, it would literally be attached to (or, graphically, painted on) the wall. The usage of FBOs and textures reduced the rendering costs of the mess mechanic drastically, and made the overall complexity of the rendering method considerably more effective.

6.4 Player Feedback

Food Fight was tested throughout the development cycle, and results of player testing have been summarized by development phase.

Phase	Pros	Recommendations
End of Quarter 1 (100%)	• Click to Throw / Move is new / fun • Good food models • Toon Shader improves look / feel	• More mess • More clear objective • Way to win/lose • Way to select foods • More variety of foods / enemies • Fix minor glitches

Phase	Pros	Recommendations
Quarter 2 Week 5-7 (150%)	• Removing enemies mechanic good / fun • Much messier, more fun • Food purchase mechanics	• WASD movement • More enemies • Display enemy health • Visual effects for enemy abilities/ food effects • Walls need to get dirty as floor does • Small glitches/bugs

Phase	Pros	Recommendations
Final Version (200%)	• Good visual effects for foods.enemies • The entire environment gets messy	• Multiplayer? • Small glitches/bugs • More varied level designs • Stationary game objects don't get messy • Balancing needs work

The most common comment from player feedback was that the visual effects were fun, and contributed greatly to the game's experience. The addition of WASD controls was the most well received addition to the game.

6.5 Development Process

The development process was unfortunately not nearly as effective as it was intended to be. A major problem with the development process as a whole was that unfortunately the experience and work ethic was concentrated mostly in one member of the team, who ended up shouldering most of the work for the first quarter. This was solved during the second quarter by segmenting the team.

However, working on a project of this magnitude by oneself comes with certain disadvantages. Being responsible for so many different resources, including modeling, art, and coding, leads to a sacrifice in quality, especially in the structure and readability of the code.

I believe the development process would have been considerably more successful if my team's experience and work output was more balanced.

7 Conclusion

This project was a major learning experience for all involved in a number of ways. To begin with, although there was a wealth of experienced developers, nobody on the Food Fight team had any experience with creation of artistic resources, such as textures or models. Additionally, although the development team was experienced in general, there was a lack of experience in OpenGL

graphics programming. A major part of the development process was researching and learning new programs and processes.

In addition to technological difficulties, learning to effectively work in a group was another important learning experience. Balancing work loads between group members, as well as effectively using version control, were both important skills learned in the first quarter of the development cycle.

Overall, this project was a better learning experience than any other singular project that I have completed during my higher education. I would highly recommend this experience to others in the Computer Science/Software Engineering major, and would go as far as to say that a similar experience should be required for all students.

8 Future Work

Although much has gotten done with Food Fight, there are many goals left for the future.

8.1 Clean Code Base

A disadvantage of attempting to create a game and all associated resources including models and art in a small time frame is that the rush can cause some unorganized or inefficient code. A major goal for future improvement would be to drastically clean and re-factor the code base to be more extensible and readable.

8.2 Variable level design

Currently, the cafeteria is drawn based on a hard-coded square design with four tables and one vending machine. Another important addition would be the ability to design levels, that would be stored in a simple script file (or something equivalent). A level editor might be another important addition.

8.3 Additional Enemies and Foods

Although there are currently 9 foods and 5 enemy types, there is drastic room for improvement. A modular system to add new types of foods and enemies could be integrated into the current code base, making it easy to add new foods or enemies.

8.4 Enemy AI

Currently, the enemies have no actual intelligence. They are simply capable of executing an ability if the player is within a certain range. An intelligent enemy AI could be a very interesting addition to make the game more fun and challenging.

8.5 Multiplayer

The single player mode of food-fight is a perfectly workable game concept on it's own, but the game concept could be modified slightly to accommodate multiple players, either cooperatively or competitively.

8.6 Mobile Device Compatibility

Although click-to-move was removed because of controllability issues, the original concept of food-fight, using only mouse controls, could easily be ported to a touch-screen based system. Additionally, because food-fight is so simple in concept, it would be a game ideal for mobile devices, where people require an uncomplicated, fun, momentary distraction, instead of a complex game requiring time investment.

9 References

1. http://www.naturewizard.com/tutorial08.html
2. http://library.forum.nokia.com/index.jsp?topic=/S60_5th_Edition_Cpp_Developers_Library/GUID-441D327D-D737-42A2-BCEA-FE89FBCA2F35/OpenGLEx/Shadows/doc/index.html
3. http://www.bluevoid.com/opengl/sig00/advanced00/notes/node199.html
4. http://www.songho.ca/opengl/gl_fbo.html
5. http://en.wikibooks.org/wiki/Blender_3D:_Noob_to_Pro
6. http://psd.tutsplus.com/
7. http://planetmath.org/encyclopedia/LinearInterpolation.html
8. http://www.ambiera.com/irrklang/

9. http://www.lighthouse3d.com/opengl/glsl/

10. http://www.opengl.org/resources/faq/technical/glu.htm

11. http://pyopengl.sourceforge.net/documentation/manual/

12. http://www.gamedev.net/topic/388298-opengl-hud/

13. http://www.wikihow.com/Find-Perpendicular-Vectors-in-2-Dimensions

14. http://nehe.gamedev.net/