Western Michigan University
ScholarWorks at WMU

Honors Theses

Lee Honors College

4-12-2009

Making an Xbox360 Video Game

Daniel Frandsen
Western Michigan University, dfrandsen@gmail.com

Follow this and additional works at: http://scholarworks.wmich.edu/honors_theses

Part of the Computer Sciences Commons

Recommended Citation

Frandsen, Daniel, "Making an Xbox360 Video Game" (2009). *Honors Theses.* Paper 1521.

This Honors Thesis-Open Access is brought to you for free and open access by the Lee Honors College at ScholarWorks at WMU. It has been accepted for inclusion in Honors Theses by an authorized administrator of ScholarWorks at WMU. For more information, please contact maira.bundza@wmich.edu.

Making an Xbox360 Video Game

Daniel Frandsen
4/12/2009
WMU Honors Thesis

Table of Contents

1. Introduction....3
2. Inspiration and Game Idea....3
3. Framework Decisions....4
4. Engine Design....5
5. Enemy Design....9
   - 5.1 Bird01, ECube, and Fish01....11
   - 5.2 Bullets....12
   - 5.3 Fish02....12
   - 5.4 Hypatia....13
   - 5.5 Serpent....14
   - 5.6 Chand, Surfer....15
   - 5.7 Pythagoras....15
   - 5.8 Tower....15
   - 5.9 Shark, Pine....16
6. Graphics....17
   - 6.1 3D Design in Blender....17
   - 6.2 XNA Animation Component Library....18
   - 6.3 2D Textures in GIMP....18
   - 6.4 Particle Effects....18
   - 6.5 Real-time GPU Shader Effects....19
   - 6.6 Bloom Shader Effect....20
7. Testing Tools, Optimization, and Source Control....21
8. Conclusion....22
9. Credits....24
10. Resources....25

1. Introduction

In the area of computer science, game design pushes the limits on what is possible in real-time graphics. The game must process the entire scene up to sixty times a second, so code techniques that normally work must be adapted to be as efficient as possible. Some challenges cannot be overcome by traditional methods and must be adapted to work in a high-demand environment. In addition to these obstacles, students have limited opportunities to develop games for console platforms due to the cost required for development kits. This paper will explore the process behind developing a game for the Xbox 360 with only the resources available to a student, and the limitations still existing in such a process.

2. Inspiration and Game Idea

Much and the style from the game (entitled "Olu") is taken from the game Rez. In Rez, you play as a hacker diving deeper into the network in search of an AI that has embedded itself within the system. However, the unique aspect of Rez is in the gameplay and presentation, not the story. As the player moves through the level, more and more layers of music are added in the background. In addition the sound effects, the background graphics and enemy movement move in time with the beat of the music. When combined with the psychedelic graphical effects, the game creates a sense of synesthesia (different stimuli associated with each other).

Olu takes these elements, changes a few things, and adds more details. Rez uses trance music, which falls under the larger genre of electronic music. Trance music is simplistic, repetitive, and does not rely much on melody. Olu's music is still electronic music, but more melody-based and less rhythm-based. Since Olu is based more around melody, and much of

the stages are build-ups to the climax of the melody both in music and action. Olu also expands on the recipe by adding weapon and enemy duality into the main gameplay. In Rez, the player fires at the enemies on the screen. In Olu, enemies are weak to the weapon of the opposite type. In other words, analog enemies are weak to the digital weapon and digital enemies are weak to the analog weapon.

3. Framework Decisions

Before diving too far into developing Olu, I wanted to make sure I chose the game framework and platform that bested suited my needs and limitations. As a proof of concept, I developed a small prototype using C++ and OpenGL (see Figure 1).

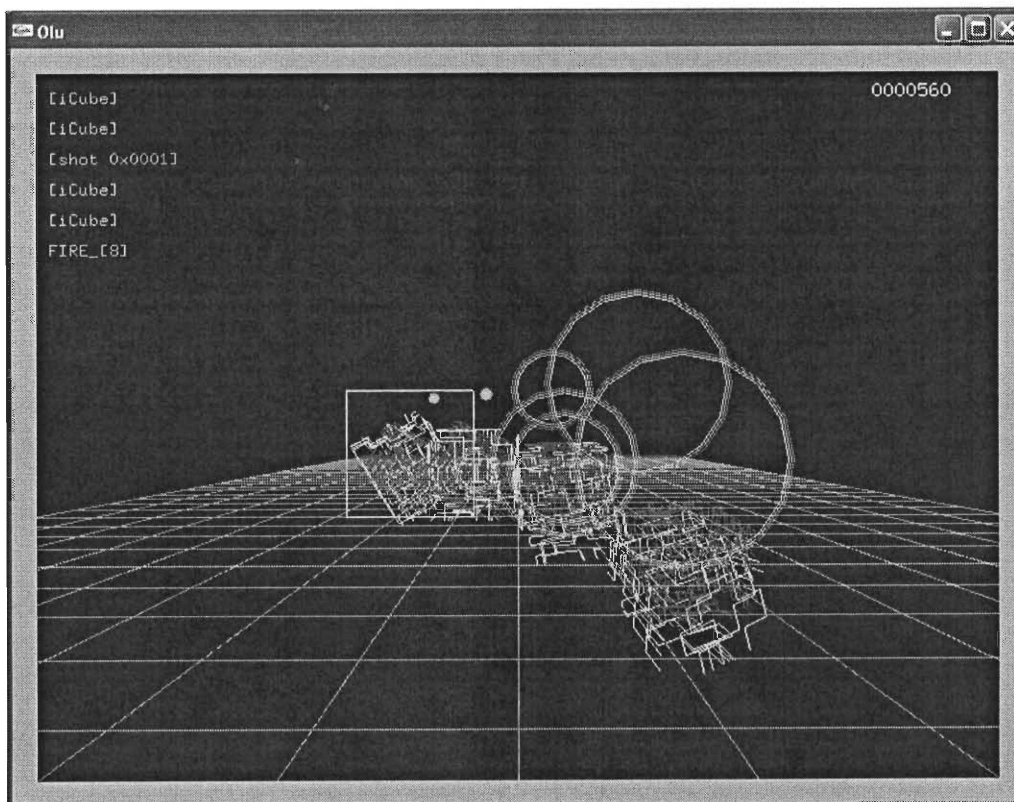

Figure 1 - A screenshot of the first prototype built using C++ and OpenGL.

The proof of concept worked fine, but I ran into issues when trying to add complex graphics and playing more than one sound file at a time. Simply put, C++ and OpenGL were powerful, but with a lone individual working on the project, there was not enough time to devote to understanding very low level operations. After building this proof of concept, I tried building the same prototype using C# and the Microsoft XNA Game Framework. I found that the XNA Framework provided an easier way to do things out of the box, such as drawing basic 3-D models and playing multiple sound files at once. In addition, XNA was built for both the PC and the Xbox 360, so the opportunity opened up to build the game for the Xbox 360. The tools were free, and the license to develop on the 360 was free to any university student. All these factors led to my decision to build Olu using the Microsoft XNA Framework.

4. Engine Design

Although the Microsoft XNA Framework makes it easy to begin writing a game, it doesn't have as many tools built into the Framework as other more specialized but limited game-making tools. It does not contain a physics engine, sprite libraries, 3D editing tools, or many other tools which are required to make a game. What it does contain is a very structured object library that represents basic elements of a game. Included are classes for games, content managers, sprite renderers, sounds, etc. Each XNA game starts with a Game object that has basic commands such as LoadContent, Update, and Draw. LoadContent will be called after the game's content (sprites, models, textures) has been loaded, Update will be called every 20-30 milliseconds to update the game, and Draw will be called every 20-30 milliseconds to draw the game. Update and Draw are automatically called on a consistent basis by the core

XNA Framework. In addition to the core game, XNA will also call Update and Draw on any Game Components that are added to the list of Game Components in the game framework. These Game Components could be menus, physics systems, or any other component that might be running alongside the game. These components can be configured to enable or disable Updating and Drawing if, for example, the game is paused.

Starting from this framework, I converted over the code from C++ that already worked. The structure I had in mind was fairly basic but let me start building the core engine of the game. Actions in the game needed to be based around the rhythm of the music. Levels would be split into smaller zones, and more layers of music would be added for each zone you played through. One piece of functionality that was present in the prototype was targeting enemies and linking their sound effects with the beat of the music. To accomplish this, I have several variables to keep track of when the beat hits and what beat it is in the measure. See below for the pseudo-code representing what is run during each Update of the game.

```
Float curTime // represents the time until the next beat
Float curBeat // represents current beat of measure
Float maxBeat // represents beats per measure
Bool firstPass // represents that this is the first pass through
Update for this beat

if (curTime < 0.001)
{
	curTime += BEAT;
	firstPass = true;
	curBeat = (curBeat + 1) % maxBeat;
}
else
	firstPass = false;
```

In addition, there is an array of 32 "channels" – float variables valued in the game between 0 and 1. Any object can read or write a value to a channel, with the purpose to control a different aspect of the game. Usually the beats control these channels, with a channel "firing"

on a certain beat. Another object, such as an enemy or background element, may be monitoring the same channel and change its behavior based on what it finds in the channel. This system is very flexible by design, so instead of trying to link two objects which are otherwise unrelated, the designer can instead chose for them to use the same channel.

Another problem solved early on dealt with determining whether an enemy was within the player's target reticule. I chose to have enemies consist of one or multiple point target(s) in space. If the single target point was viewable inside the box the player was using, then the player could target that specific target. To accomplish this, I projected the 3D point in space where the target exists to determine where it appears to be on the screen. If the point was within the player's target box, then the target is selected. Once there is a list of particles, the game chooses the closest one to fire at. Since this needed to be on the beat, the game only attempted to lock onto enemies on sixteenth-note increments of the measure. When the player releases the locks on the enemies, the enemies will be fired at in the order in which they were locked on, and the sound effects will be delayed to be in rhythm according to how the enemy defines the pattern. In addition, your score is multiplied by how many enemies you are locked onto, so it is beneficial to lock onto as many enemies as possible before firing.

When duality between the two enemy and weapon types was added, I revisited how the locking on system worked. I changed the system to reinforce using the opposite weapon type. Mechanically, the opposite weapon type did more damage and gave you more energy towards a super attack; however, this was not obvious to the user. To reinforce these actions, effects were changed based on whether you used the opposite weapon. If the opposite weapon is used, there is a louder sound effect, a background visual, and a stronger rumble feedback from

the controller. If the similar weapon is used, there is a softer sound effect, a weaker rumble, and no background visual.

For many sounds to play on-beat at the same time, a structure needed to be built with specific requirements. These requirements included scheduling sounds to play on specific beats, many sounds to be created over the course of the game, and to be properly disposed after they are finished playing. To accomplish this, a list of Cues (sound objects in XNA) are kept, along with an object representing which beat each can be played on. When the Cue is ready to be played, it is moved to a different list of active Cues and played. Every second or so, the game checks the active Cue list and properly disposes of any Cues that are done playing.

As mentioned earlier, there is rumble functionality built into the game. The Xbox 360 controller contains both a high-frequency and low-frequency motor for rumbling. The high-frequency motor has a quick response time but is not as powerful, and the low-frequency motor has a slow response time but is more powerful. Each motor is controlled by one of the 32 channels, and is updated by a rumble Game Component. The slower motor is the motor that rumbles when an enemy is targeted, while the faster motor is the motor that rumbles along with the beat.

At this point in the paper much of the basic functionality has been explained: the engine marks when there is a beat, channels control actions of objects within the game, music and rumble goes along with the beat. However, the music and rumble changes between each zone of each level. To control this, I used an xml-based structure for the levels, and nodes in each zone marking when certain musical cues should play, and when certain channels should be updated. Under each "zone" node in the xml file, there are both "music" and "cue" nodes

("cue" representing the channel modifications). The music and cue nodes are similar in structure. Both nodes have a beat, a measure loop amount, and a play measure. Put simply, the music or channel is only activated on the beat of the play measure. For example, if the measure loop amount is 4, the play measure is 1, and the beat is 2; the music will play on the second beat of the 1st, 5th, 9th, 13th, etc measures. Additional elements of the xml structure will be revisited later.

To split these sounds into different files, I used an audio editing program which was based around the creation of music using a construction of loops. Fruity Loops Studio (FL Studio) constructs music using a playlist enabling and disabling loops of music. These loops can be exported into individual sound files, which is exactly what I needed. By copying the pattern of the song in FL Studio, I was able to reproduce the song pattern in my xml file.

As I began adding background elements to the xml file, I needed additional interaction between the 3D models and the music. To accomplish this, I made objects representing various 3D transformations (movement, rotation, and scaling). I allowed any of these transformations to be controlled by channels, whose actions were defined in the same zone in the xml file. I also allowed objects to change color based on channels. This allowed background objects to "pop" out along with the beat of the music.

5. Enemy Design

The enemies in Olu range from large and complex to a simple sphere, so the Enemy class had to account for all these possibilities. There were functions of enemies which needed to exist in every enemy in the game. Some of these functions included the target framework, a

path-finding system, being added or removed from the system, the ability to update and draw, and the enemy type (analog or digital). Each enemy has more actions and attributes beyond this, but all enemies have this core functionality.

The target framework is built in a similar fashion as the enemy framework, in that each type of target inherits basic actions and attributes from a base target class. Each target has a position, hit points, the enemy it is attached to, whether it is targeted or not, and the type of target (analog or digital). With this information, the routines can be written to determine whether a target is in the reticule without knowing how the target actually works. The different types of targets used are a basic target attached to an enemy, a target attached to an enemy's arm or leg (which causes the arm or leg to disappear when it is destroyed), or a target attached to a single face on the 3D model (which causes the face to disappear when it is destroyed).

The path framework is also built in an abstract fashion, feeding general information back to the enemy (such as position, direction, start and end). Each Enemy has access to a Path List, which contains a list of Paths (which may be of different types). As one path finishes, Path List determines what the next Path to follow is and continues the operation on that Path. Some Path types include a line, a Bezier curve, and a circle. The Bezier curve is most flexible, providing a Bezier curve which supports oscillations along a trigonometric function around the curve. Since a Bezier curve can also be a line, this supports trigonometric movement along a straight line as well.

All other base enemy actions are pretty straightforward. When an enemy is created, it is move into a list of active enemies. When an enemy dies or leaves the area, it is removed from the list of active enemies. The enemy type determines the type of target when the enemy does

not explicitly define it. The enemy type also defines how the enemy should be drawn – most enemies can be either analog or digital. The difference could be as simple as drawing a wireframe instead of a solid object, but usually is much more complicated than that. Usually the digital versions are glowing with a brighter color and emit brighter particles. The rest of this section will cover the unique behaviors of specific enemies.

5.1 Bird01, ECube, and Fish01

All of these enemies are fairly basic; they have no animation, they are not textured, and use very simple 3D models. The first obstacles encountered with these enemies were important to solve early on. First, both Bird01 and Fish01 had to face the direction they were moving, as if they were flying through space. To accomplish this, they get the direction they're pointing from the path they are currently following, and adjust their "up" vector in space to appropriately match. This is accomplished by taking a cross product of the current up and direction vector to get the "right" vector. The right vector eliminates the vertical component, and then is crossed again with the direction vector. What is returned from this operation is the new up vector, which is at a right angle to the new direction vector. This calculation is important to help prevent graphical issues from arising as the model is transformed to point a specific direction. Throughout the development of this game, I ran into many issues as I ran through different revisions of this function. Some of these issues included skewing of the model, twisting (where the model would "spin" along its direction vector), and the model sometimes disappearing altogether. Second, ECube is one of the few enemies in the game whose graphics are generated in memory. Each side of the cube consists of a network of right-angled zigzag lines,

which are generated randomly. In the original prototype, the basic structure was generated in the beginning, and a transformation was calculated *every* frame to properly determine where the effect should be drawn. As I soon realized as I added more effects, it was much too expensive to calculate every frame, especially since the effect could easily be pre-calculated without any penalty to presentation. I created a vertex buffer that contained the finalized information about where the points are in each cube face, and this action is done once when the cube is created. Rather than doing this operation up to 60 times per second, I ran the operation once when the enemy was created at the cost of a little more storage space.

5.2 Bullets

Bullets were not more difficult, but once again provided a new obstacle. They had to track towards the player, using the path framework. In other words, they had to use a path that was not already pre-defined in an xml document. Another obstacle that appeared later showed that it was more efficient to draw all bullets at the same time. Rather than calling the Draw function for every enemy, a BulletCollection object set up the graphics settings and then called the lowest-level draw function for each bullet. This helped cut down on up to a hundred Bullets accessing the graphics settings on the graphics card (which is expensive to do).

5.3 Fish02

Fish02 was the first enemy to use animations, and the first enemy to have more than one target. In this section we will not discuss the creation of the animated model (see Section 6.1), but instead describe how it was used in the program. Fish02 was drawn using four separate 3D

models, and did not use an armature (a technical name for a type of skeleton used in 3D design) and instead only modified the location and rotation of each model. Until this point in the project, each enemy shared one global graphics object containing the model. However, this strategy did not work when using animations because each enemy was at a different point in the animation. My solution was to keep the model information shared, but have individual animation information for each enemy instance. This would help later as other enemy types would run into similar issues with sharing assets.

5.4 Hypatia

Hypatia was the first boss enemy, and presented many problems unseen before. First, it used animations using an armature. This caused the need for more additions to the graphical framework for displaying a 3D model, and for additional modifications to the animation library that was used (see Resource 15). Second, it had many different states where it would act differently, draw differently, and be structured differently. To handle this, I made an EnemyState class which had delegate functions for updating, drawing, removal testing, and the next state to move into. Then I created functions which would return the EnemyState for each different state. For example, the PhaseOne state handled the actions for the time before the backmost element of the enemy was destroyed. The test for removal was once all the hit points for that area of the enemy were below zero. Once this happened, the enemy was given the next state, which moved the enemy further towards the player, and caused the next series of events. This occurred until the last state, which triggered the enemy's death. The EnemyState class was re-used in many other enemies.

In addition, the last phase of Hypatia allowed individual faces to break off of Hypatia's 3D model and swarm, soon after attacking the player. This was a serious problem to account for. It would be very expensive to rebuild the vertex buffer after each face was removed, but each individual face needed to be removed. To solve this problem, I changed the model drawing framework to use a list of faces to draw. When a model was first imported, it would draw faces 0 to n-1 (assuming n is the number of faces in the model). The data would consist of a list of two numbers, the first start and first end point. If a face was removed (and the vertex data copied into a different object), then the two numbers would split to two. The elements would then be 0, a-1, a+1, n-1 (where a is the index of the removed face). By using this method, the vertex buffer would stay the same, and different parts of the model could be removed cheaply at any point in time.

5.5 Serpent

Serpent was the first enemy to have a texture. Although there was no animation, it required writing additional code for the graphics card to handle properly displaying and lighting these textures. The serpent also spawned additional enemies behind it, where only the "head" of the serpent was of the Serpent class, and the rest of the segments were generated by that first object. This required the segments after the first to move in a fashion to not disconnect the segments from each other. To accomplish this, the new segments of the Serpent were added after a delay relating to how fast the enemies were moving along the path.

5.6 Chand, Surfer

Chand, named because it roughly resembled a chandelier, was the first enemy to have a texture and animated. This caused an issue since the content processer for animated models handled textures differently than the content processor built into XNA. This caused a headache until the source was caused, and the process was adapted to properly display the textures.

Surfer was also textured and animated. The desired effect was to have the enemy lean to either side as it "slid" down the level, similar to how a snowboarder rides a snow-covered hill. To accomplish this, a property was added to the enemy to represent which way it was leaning. The value would range between 0 and 1, and this number would determine how far the Surfer was in the animation, and how many particles to emit on each side.

5.7 Pythagoras

Pythagoras was the second boss, and built on most of the knowledge from previous enemies. Pythagoras was textured, used animated models, spawned enemies, and made individual faces into targets. The biggest change made to the framework to account for this enemy was that the face targets did not move after they were spawned, unlike Hypatia's targets.

5.8 Tower

Tower was the first "giant" enemy in the Hall of Giants level. The new problem presented with this enemy was face targets which stayed attached to the textured enemy as the enemy was moving. In previous enemies, the face targets were spawned at their current location and

did not need to continue tracking the enemy which it was attached to. In the case of Tower, every face of the moving enemy was a target, and could be knocked off at any point in time. This caused the need to make a target which tracked the location of any given face driven by an animation at any point in time. Once the face was knocked off, the face also spawned a "digital water" type effect when it passed a certain height. This meant that the face needed to be tracked after it was hit, so the tracking mechanism could not be a target. To solve this, I used the FallingObject class to create a face that would trigger the effect as it passed the water height (this class was also previously used with ECube, when it fell apart after dying).

5.9 Shark, Pine

Both Shark and Pine presented a more advanced problem similar to the problem presented by Serpent. Shark and Pine both had segments that follow the head, but the head followed a non-defined path based on what actions are required at any given state. To solve the problem, the head Shark passed the information on which velocity the tails should travel and how long they should travel in that direction. The tails then checked the current speed of the head Shark and advanced forward in the queued directions appropriately to properly follow the Shark in a pattern similar to a chain or centipede, where the action in the tail segments is delayed slightly. Pine re-used this idea in the same way. Pine was also the first enemy to utilize a new function of the line-based particle system, where it "sucks in" particles to create a ball to shoot at the player.

6. Graphics

XNA provides some functionality in displaying and processing graphics, but does not provide any functionality in generating them. Throughout the development process, many different tools were used for many different purposes, including 3D modeling and animation, texturing, exporting data, generating real-time effects, and post-processing graphical techniques.

6.1 3D Design in Blender

Blender is an open-source free piece of software which supports a plethora of functionality based around 3D modeling and animation. Since both XNA and Blender are free, there are other people in the online community who have already experimented with using the two together. Blender has a steep learning curve, but I had used it in the past so it was the first program I turned to when I needed to generate 3D assets. Inside Blender, I had support for everything I needed; it had modeling, texturing, and armature animation. However, I needed to export the data into a DirectX file so XNA could properly process it. The DirectX exporter that came with Blender had many bugs and glitches; there were hidden conditions required that I needed to learn before I could modify the export script to properly export the assets I needed. Some of these limitations included making sure that there were no bones without vertices, that there was only one object parented to an armature, and that there was only one armature in the scene. As I began to understand better how the DirectX file was structured, I was able to better utilize the assets in Olu.

6.2 XNA Animation Component Library

XNA provides functionality to draw models easily, but provides little guidance on how to draw animated models. After a little searching, someone had already built a limited but still useful animation library. The XNA Animation Component Library worked with little additional effort, but as the project continued limitations arose. The XNA ACL attached the models to their animations, and did not allow them to be independent of each other. Luckily the source code was published under a license that made it free to use and revise. Some of the modifications made to the library over the course of the project included making animations and models independent of each other, enabling and disabling the calculation of bones, and the ability to specify where the animation definition file is for a given asset.

6.3 2D Textures in GIMP

The General Image Manipulation Program (GIMP) is an open-source free program with functionality competitive to Photoshop. It has functionality including multiple layers, transparency, filters, effects, etc. This program was used to help generate the textures needed for the 3D models as well as most of the 2D visual effects.

6.4 Particle Effects

The particle effects system went through many revisions. Originally a set of particle generators updated every frame to generate new particles which were added into a vertex buffer, whose vertices were being updated each frame. However due to how many particle generators and how many particles were being made, it soon became clear that a change had

to be made. The load was taken off of the CPU and put onto the graphics card (GPU). A custom vertex declaration was made that contained all the information needed to define a particle at any point in time, when the current time was given to the GPU. This data included particle color, the center of the burst, the velocity vector of the particle, the start time, and the end time. Then when the GPU processed the vertex buffer, it took that information and calculated where the particle should be. This effect was efficient and worked well, but it was improved upon by changing the point particles into textured sprites.

Building on the previous idea, texture coordinates are added into the vertex buffer, and instead of drawing points, the GPU draws triangles. The GPU calculates each point of the triangle such that the triangle faces the camera. The texture that the GPU uses to draw the particles uses an alpha channel, so when the triangle is drawn, it looks like a circle rather than a triangle.

6.5 Real-time GPU Shader Effects

Although the particle system is the most complex real-time GPU effect, there are others that play a crucial role. Fog was the first effect made, and is a linear-based fog using a start and end distance based on the distance to the camera. In the vertex shader (the function which is run on every **vertex** in the scene) it gets a normalized value to represent the fog amount (but does not clamp it between 0 and 1). In the pixel shader (the function which is run on every **pixel** in the scene, and gets data from the vertex shader), it interpolates the fog amount by using the information from the vertices. After this it clamps the value and mixes the normal

color and fog color based on the fog value. This creates a per-pixel fog effect, and prevents sides disappearing into the fog unevenly.

Most of the other real-time shader effects are similar. The "Shine" effect only shows part of an object; so the fog value is generated in a way that the same pixel shader can be used. The "Water" effect only shows a model above or below the water height, once again utilizing the fog value.

The most complicated real-time effect is the Bezier transform. This effect warps an object around a Bezier curve, representing a tentacle-like limb in the game. To accomplish this, two matrices are pre-calculated so that when multiplied by the vector { t^3, t^2, t, 1 } (where t is the progress between 0 and 1 along the Bezier curve) it gives the position of that vertex along the Bezier curve. The second matrix works in the same way, giving the instantaneous velocity of the Bezier curve at that point; in other words, the direction the curve is pointing. Using that vector and a guess at a perpendicular vector, the GPU can determine the two axes perpendicular to that point on the Bezier curve. With the position and orientation on the curve, the GPU can place the vertex at the correct place on the curve.

6.6 Bloom Shader Effect

Unlike all the other effects, the bloom effect in Olu is a post-processed effect. In other words, after everything else in the scene is rendered, this effect takes the texture and makes modifications. Only certain objects in the world are bloomed, so throughout drawing each frame, the pixels that must be bloomed are marked as such in a stencil buffer (a stencil buffer is a small buffer that keeps small number value for each frame). Once the frame is done drawing,

the game grabs a texture of the scene. The scene is rendered using the stencil buffer as a filter, and renders only the bloomed areas to a render target. A render target is an object in memory that captures the render results from the GPU before it displays it to the screen. It takes this result and shrinks it down to half size (1/4 of the memory size, since it is a 2D texture). It then blurs the bloomed areas, and when done reapplies the blurred texture to the original world. This creates the illusion that those areas are "glowing", and is relatively inexpensive for the effect generated. However, due to how much is passed between the CPU and GPU and the fact that every pixel is being processed, this is **the most expensive** effect in the game; and took the most work to get the effect as efficient as it is.

7. Testing Tools, Optimization, and Source Control

Many of the tools I used have not fit into categories already covered. This section will cover nprof, CLRProfiler, Subversion, Apache, No-IP, Fraps, and VirtualDub.

nprof and CLRProfiler are both free code profilers built to monitor the .NET Framework. I used these tools to help narrow down bottlenecks that were wasting CPU time, as well as sources of memory leaks. nprof excelled in showing what the CPU was doing. After the program finished running, the nprof broke down each function into what percentage of CPU it was using, as well as what functions it called and what functions called it. Using this information, I could track down very easily what algorithms needed revising. CLRProfiler on the other hand tracked memory usage. It kept track of every object allocated, and summarized by type, when the object was generated, when it was disposed, and more. All I had to do was look at the types that had the most memory allocated when the program terminated to find objects

that were not being disposed properly or were being generated every frame and didn't need to be.

Subversion, Apache, and No-IP together allowed me to have a remote source control server cheaply (both software packages were free). Subversion is a source control system which tracks changes to projects and allows recovery if a workstation loses data or if a previous version is needed. Apache allowed Subversion to be exposed to the world through a web browser, while No-IP forwarded requests to a public domain name to my local IP. These allowed me to access my source control server at the domain name set up through No-IP, and have access to my code wherever I was. It was exceedingly helpful as I used two separate computers throughout this entire development process, and allowed them to remain in sync.

Fraps and VirtualDub allowed me to capture video of my game and convert it into a format that I could upload to the web and to put into presentations. Fraps was not free, but cost a small amount that was worth the functionality. Fraps allows you to record video and take screenshots as you run a game. VirtualDub is able to decode and encode with various different codecs, and is one of the few video editing programs that can properly view videos recorded in the raw Fraps format.

8. Conclusion

Over the past 15 months, I have learned a lot about game design, game programming, and project management. Microsoft's XNA Game Framework allows students to learn core game development skills and environment where they can succeed. However, in the past few months Microsoft has not given much support to market the games made with their

framework. Although anyone on Xbox LIVE can purchase XNA games through Xbox LIVE Community Games, there is no rating system and it is hard to determine what games are good or bad. Sales numbers have been reported for the first batch of XNA game sales, and the numbers are not promising. Although XNA is a great platform to develop for, the market has not matured to a point where it is viable to expect profit from making an XNA game.

In addition, it is very clear to me that making a game individually causes many issues. First, you have only one set of skills. I may be a good programmer, but I am only a decent designer and a horrible artist and musician. If I had a good mix of these skills on a team, I would have come out with a much better game. Second, aside from skills you need different perspectives on every element of the game. Many of my ideas for ways to change the game came from people who had seen or tested it.

Finally, I now understand the value in the experience of completing a game. I truly understand every aspect of how my game works, because I was forced to troubleshoot and understand every problem I ran into. Writing only a technology demo or physics system only gets you halfway, because you don't truly understand how every piece interacts with each other.

9. Credits

Advisors

Dr. Karlis Kaugars
Mr. Kevin Abbott
Dr. Robert Trenary

Software

Microsoft Visual Studio 2008
Microsoft XNA Framework 3.0
Blender
Gimp
Fraps
VirtualDub
Nprof
CLRProfiler
Subversion
Apache
No-IP

Audio Editing

Robert Hutchison

Vocal Talent

Mary Snow

Testers

Mr. Kevin Abbott
Keith Furry
Brenna Halpin
Justin Hostetler
Hannah Houseworth
Sasson Jamshidi
Dr. Karlis Kaugars
Simon Tower
Dr. Robert Trenary
And many others

10. Resources

XNA Overall

1. XNA Best Practices - http://creators.xna.com/en-US/education/bestpractices
2. XNA Stress Testing - http://forums.xna.com/forums/t/19525.aspx?PageIndex=1
3. Gamer Services - http://msmvps.com/blogs/mykre/archive/2008/03/01/using-gamer-services-in-your-xna-game.aspx

Blender Techniques

4. Fake Subsurface Scattering in Blender - http://mke3.net/weblog/fast-fake-sss/

Post-processing

5. Creating a Post-Processing Framework - http://www.gamasutra.com/view/feature/1812/creating_a_postprocessing_.php?print=1
6. Accelerated Decimation Using Anisotropic Filtering - http://developer.download.nvidia.com/SDK/9.5/Samples/DEMOS/Direct3D9/src/AnisoDecimation/docs/AnisoDecimation.pdf
7. Real-Time 3D Scene Post-processing - http://ati.amd.com/developer/gdce/Oat-ScenePostprocessing.pdf
8. HLSL Guassian Blur - http://www.gamedev.net/community/forums/topic.asp?topic_id=427696

Render Targets

9. Purple Screen of Death - http://blogs.msdn.com/tmiller/archive/2008/02/21/why-purple.aspx

Threading

10. Pensive Gamer: Game Loop and Threading 1 - http://blogs.spouting-tech.com/thepensivegamer/2007/04/developer_diary.html
11. Pensive Gamer: Game Loop and Threading 2 - http://blogs.spouting-tech.com/thepensivegamer/2007/04/developer_diary_1.html
12. Pensive Gamer: Game Loop and Threading 3 - http://blogs.spouting-tech.com/thepensivegamer/2007/04/developer_diary_3.html

Particles

13. Particle System Optimization - http://forums.xna.com/forums/p/16153/85159.aspx

Animations

14. Getting Matrix Palette working - http://www.gamedev.net/community/forums/topic.asp?topic_id=395581
15. XNA Animation Component Library - http://www.codeplex.com/animationcomponents/Wiki/View.aspx?title=Making%20a%20Dwarf%20Realistically%20Walk%20and%20Run&referringTitle=Tutorial%20Home
16. XNA Animation Component Library (revision list) - http://www.codeplex.com/animationcomponents/SourceControl/ListDownloadableCommits.aspx
17. Custom Content Importer - http://www.ziggyware.com/readarticle.php?article_id=166
18. XNA and Blender Example - http://video.aol.com/video-detail/xna-and-blender-part-iv/3728927951

Fonts

19. Blambot - www.blambot.com

Screen sizing

* Always use 720p - the Xbox 360 will properly scale for any resoluation

20. Xbox 360 Screen Sizing - http://ziggyware.com/readarticle.php?article_id=66

Optimization

Programs: nprof, CLRProfiler

21. Software Efficiency And Optimization - http://swampthingtom.blogspot.com/2007/04/software-efficiency-and-optimization.html
22. Garbage Collection Inspection - http://blogs.msdn.com/shawnhar/archive/2007/06/29/how-to-tell-if-your-xbox-garbage-collection-is-too-slow.aspx
23. .NET Compact Framework Optimization (1) - http://blogs.msdn.com/netcfteam/archive/2006/12/22/managed-code-performance-on-xbox-360-for-the-xna-framework-1-0.aspx
24. .NET Compact Framework Optimization (2) - http://blogs.msdn.com/netcfteam/archive/2006/12/22/managed-code-performance-on-xbox-360-for-xna-part-2-gc-and-tools.aspx
25. Improving Managed Code Performance - http://msdn.microsoft.com/en-us/library/ms998547.aspx
26. Xbox 360 speed problems - http://forums.xna.com/forums/p/9914/51731.aspx#51731

Fog effect

27. Distance Fog - http://www.riemers.net/Forum/index.php?var=638&var2=0
28. Refractions Using Fresnel - http://www.riemers.net/eng/Tutorials/XNA/Csharp/Series4/The_Fresnel_term.php
29. Fog Using Shader Model 3.0 - http://forums.xna.com/thread/17102.aspx
30. Simple Fog Shader - http://shadevampire.extra.hu/bugs/fog_try.txt

Render targets

*Multiple render targets do not support anti-aliasing

31. Render Target (MSDN) - http://msdn.microsoft.com/en-us/library/bb975911.aspx

DirectX Render States

32. DirectX Render States - http://www.toymaker.info/Games/html/render_states.html

Compatibility

33. Running XNA on old PCs http://www.riemers.net/eng/Tutorials/XNA/Csharp/ShortTuts/Reference_device.php