Third Degree

By: Tim Biggs

Team Members: Mark Paddon, Tim Biggs, Chad Williams, Jon Moorman, Joshua Marcelo, Michael Sanchez

California Polytechnic State University

Winter 2010 – Spring 2011

Advisor: Prof. Zoë J. Wood

Table of Contents

Introduction
Project Overview
- Story
- Look and Feel
- Game Mechanics

Related Works
- Trine
- Maya
- Doom 3

Technologies
- Character System
- Combat System
- Joints
- Puzzle Objects
- Scene Triggers
- Mesh Cooking

Results
- Design
- Teamwork
- Final Game
- Feedback

Future Work
Conclusion
Additional Credits

Introduction

In today's world, interactive 3-D games play an essential role in providing entertainment and recreation to many people. The interactive gaming industry, which began around the 1970's, has grown into a large mainstream cultural phenomenon, earning $10.5 billion in 2009 in the US alone (CDR Info). Throughout this growth, many gaming genres were created and explored, providing a wide variety of different situations and scenarios for game players to experience. Some of these gaming genres include action, adventure, role-playing, and strategy. In this way, games have been able to provide a tailored experience for everyone, and have become a unique icon in the mainstream culture. For my senior project, I chose to work with a team to develop and implement a 3-D interactive game filled with action and adventure.

Our game, *Third Degree*, began in Cal Poly's Winter Quarter 2011, as a project the CSC 476++ - a Real-Time 3-D Computer Graphics Software Systems class. The focus of the class was to create an interactive 3-D video game world, with course requirements to integrate various graphics technologies into the game. Before the program began, Mark Paddon, Chad Williams and Michael Sanchez had the initial idea for a 3-D side-scrolling platformer called *Third Degree*. Jon Moorman, Josh Marcelo, and I then joined their programming team to work on the game throughout the CSC 476++ program. Our work continued into the Spring 2011 quarter, where we made various graphical, gameplay, and storyline improvements to the game.

Project Overview

<u>Story</u>

In *Third Degree* you are a convict, who in being given a chance to gain his freedom, submits to a strange experiment with a panel of top scientists. A recent alien artifact has fallen from the sky, and it has the power to create a virtual simulation of an 1860s style London. The

player is one in a line of several given a chance to test the artifact in exchange for freedom. He is placed into this simulation to find out anything he can about this alien technology. Unfortunately, not everything is as it seems, as the convict finds himself slowly going insane throughout the simulation. He soon realizes he is stuck in a race against the simulation, trying to find the answers before his sanity crumbles away.

Look and Feel

Although the real world setting is our present day world, the game primarily takes place simultaneously in 1860s London and the futuristic world of the aliens. Because the human mind does not have the capacity to comprehend the alien simulation device, the reality around the player will phase in and out of the futuristic world. In other words, as the player becomes more insane, objects in the 1860s London world will phase out and be replaced with those of the futuristic alien world. To reduce the level of insanity, the player has the capability to focus while progressing through the simulation world. This focus will also revert the simulation back to the 1860s London setting.

Game Mechanics

The gameplay centers on progression along a stage by means of jumping on platforms, solving puzzles, and fighting off enemies. The player has two jumping abilities - regular jumping and a higher special jump - to figure out puzzles and progress through the level. Puzzles are physics based, meaning the player must use his knowledge of reality to solve them. Examples of puzzles include combinations of trap doors, swinging platforms, and swinging spikes. Additionally, the player has a weapon with which to fight off enemies that will attack him throughout the level. The player must use each of these game mechanics to progress through the level, and possibly find bonus areas or items.

The atmosphere of *Third Degree* works to give the player a feeling of light-heartedness, intrigue, and challenge. The character's sanity meter slowly goes up over time, and objects phase in and out as it fills. Once the meter fills all the way, the player "passes out" and is reset

to the last checkpoint. One may think that the best way to progress through the game is to just simply keep this sanity meter low. However, in order to use various abilities such as the higher special jump, the meter must be filled up to a certain amount. The player must balance these mechanics accordingly to get the most out of his character's actions. Platforming in the game should also provide enough of a challenge to the player so it the game will not feel too easy or unsatisfying.

Whereas the game sets itself up to be of a more serious tone, the actual story tries to convey a more comedic feeling, feeding the player with just enough information in a satirical manner to keep him intrigued to continue on. *Third Degree* strives to keep a balance between story and gameplay in order to provide the player with an enjoyable, intriguing, and satisfying experience.

Related Works

While there were many influences for *Third Degree*, the following works both influenced the gameplay as well as helped to provide examples for how to approach certain tasks for components of the *Third Degree*.

Trine

The general mood and feel of the game were greatly influenced by this side-scrolling platformer, Frozenbyte's *Trine* (Frozenbyte). Visual inspirations, as well as overall feel of the gameplay mechanics such as movement, puzzle object interaction and elements of the combat system helped in making decisions for *Third Degree*. The Figure below shows an in-game screenshot for *Trine*.

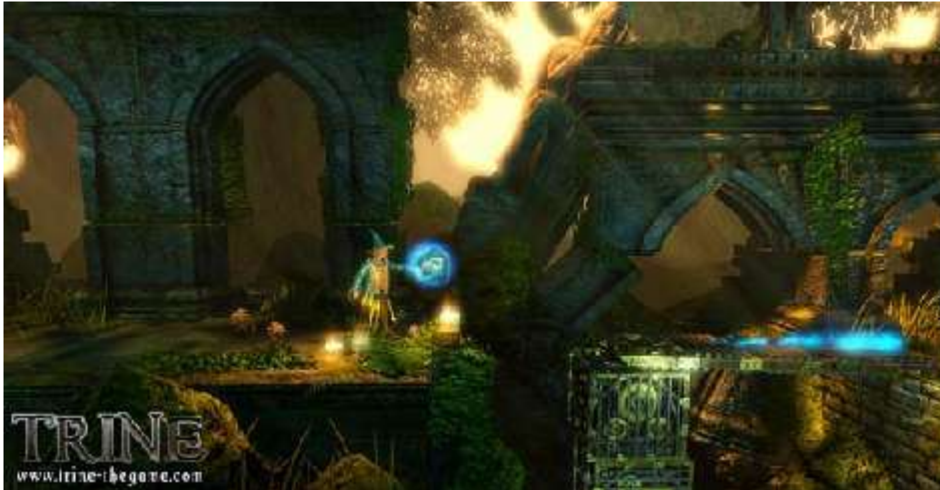

Fig. 1 – Screenshot for *Trine*

Maya

The transformation tools were modeled after many 3-D graphics software, particularly Autodesk Maya (AutoDesk). The figure below shows an example of the transformation tools used in Maya.

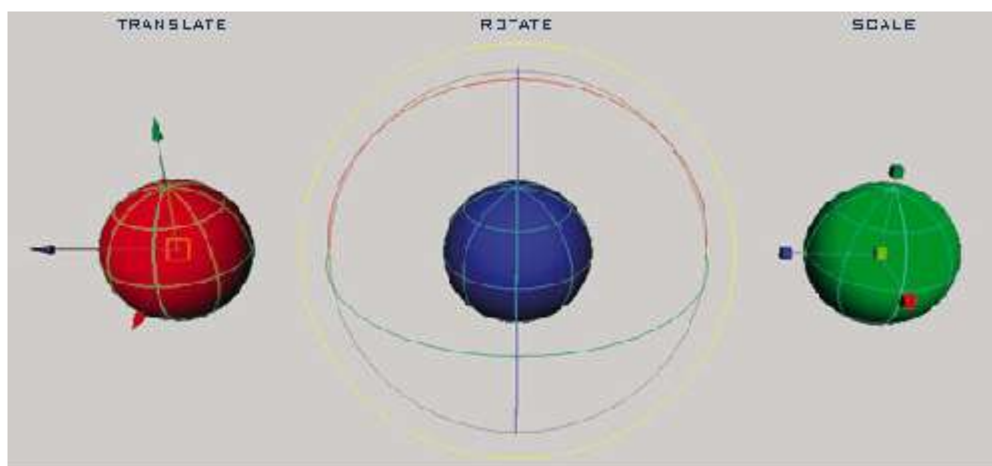

Fig. 2 – Examples of the Translate, Rotate & Scale tools in Autodesk Maya.

Doom 3

The animation in the game utilizes the md5 format used in a number of 3-D games, most notably in Activision's *Doom 3* (Activision). The MD5 structure created for *Doom 3* provides a robust and efficient way for representing animation. The figure below shows an example mesh from *Doom 3* modeled using MD5.

Fig. 3 – Example of a MD5 model used in *Doom 3*

Technologies

Technology	Authors/People involved
View Frustum Culling	Josh
Skeletal Animations	Josh
Enemy AI	Josh
Player Control/Movement	Josh, Tim
Combat System	Josh, Tim, Mark
Editor - Object Transformations	Josh
Editor - Main Functionality	Chad
Deferred Rendering	Chad, Ryan Schmitt
GLSL Shaders	Chad
Core Engine Optimizations	Chad
Particle System Implementation	Chad
High Level Design	Jon, Mark
Map Loading/Saving	Jon
Physics Engine Integration	Jon, Tim, Mark
Object/Joint System (aka Puzzle Objects)	Jon, Tim
Glow Shader	Jon, Chad
Animated Textures	Michael, Mark
Menu System	Michael
Fire Legs Implementation	Michael, Tim
OBJ Importer	Michael, Chad
Octree	Mark
Sound Design	Mark
Focus	Mark

Character System

As the game development began, one of the first things needed for a working prototype was a system to handle character movements and interactions in the 3D environment. The only viable alternative to such a system would be to restrict all character interactions, such as movement and collision detection, to the character class itself. Unfortunately, this lends itself to bad code design, which the team wanted to avoid. Also, because the level editor was being designed side by side with the game, there needed to be a way to place characters in the world, without manually specifying a starting position in the character class. With a character system, it would be possible to simply import the character location from a map file created by the editor.   The character tracking system consisted of a few key classes linked together through function calls to each other. At the top level, all characters - including the player character - were tracked through a special singleton class called the Character Manager. This class, among other things, included a step function, to advance the character simulation, and an instantiation function that, when given a position, would create a new non-player character located at that position. A player character was automatically created when the Character Manager came into existence, and could be manipulated directly through a reference provided by the manager. During the Character Manager's simulation step, each character, both player and non-player, would be updated according to the current environment state. By extracting the character management from the

Figure 4 - A Jumping Character Model

remainder of the game's logic, the code became much simpler, cleaner, and easier to use or understand.

Below the Manager, there are 2 types of character classes that each handle their interactions with the outside world: the player and non-player characters. The game creates only one instance of the player character, and uses it as the controller for the actual human player. In contrast, any number of non-player characters are created throughout the simulation, and are mainly used as enemies with artificial intelligence. When either the player or non-player collides with the world, or when some other input event happens, the character is notified through the Character Manager. In this way, both the player and non-player characters are free to roam throughout the world, and only react as appropriate when given a function callback.

Within the character classes, various functions and algorithms are used to simulate real world movement. Actions a character can use include running, jumping, grabbing objects, and firing a projectile.

Running is a fairly simple action for any character. Once a character is instructed to run, either from input provided by the user, or from an AI subroutine, a function call will apply a force to the character in the given direction. Simulating running by treating the character as another physics object does work, but unfortunately it comes a few problems. It does not allow the player (in this case, the user playing the game) very fine-grained control over character movement, as applying a force as a physics object would cause some slipping in character movement, since physics objects have no feet and will therefore slow down only by friction. Additionally, since the character is treated as a normal object in the scene, it is much harder to include certain movement enhancements such as auto-stepping (having the character automatically step upwards through small bumps) and removal of undesired physical behavior (such as resetting a tilted box while still on it).

Character jumping consists of two main stages: preparing the controller and actually jumping. Before a character can jump, the controller must determine that the character is on the ground and not in the air already. This is accomplished through a ray cast. In the implementation, three rays are cast from the center, left and right of the character's position toward the ground. These rays are then passed to the physics engine to perform the ray cast. When any of the rays collide with an object, the engine reports a hit via a callback. Because the rays are short, any hit will assume that the character is on the ground and can therefore jump.

Allowing a character to grab objects works similarly to jumping, but a contact test is performed instead of a ray cast. Whereas a ray cast will create and extend rays into the 3D world, a contact test will just simply check if two collision shapes have actually collided. When a character wants to grab an object, this contact test determines whether the character is touching the object it wants to pick up. Once picked up, the object can be thrown, in the direction of the character's velocity. Unfortunately, due to a buggy implementation and the need for different gameplay design, the feature has since been removed from the game.

Finally, a character can choose to shoot a projectile. When this action happens, a callback to the Weapon Manager class, described below, spawns the projectile into existence, in the direction the character specifies. For the player, this direction is created based on where the mouse points in the game window. The character can launch any number of projectiles into the scene

Combat System

One of the core game elements revolves around the ability to fight off enemies. In order to accomplish this, a system was needed to handle combat interactions between the characters and the environment. Without devoting combat management to a separate system, the combat

could not really exist, and the game would have been probably been less challenging, fun, or both.

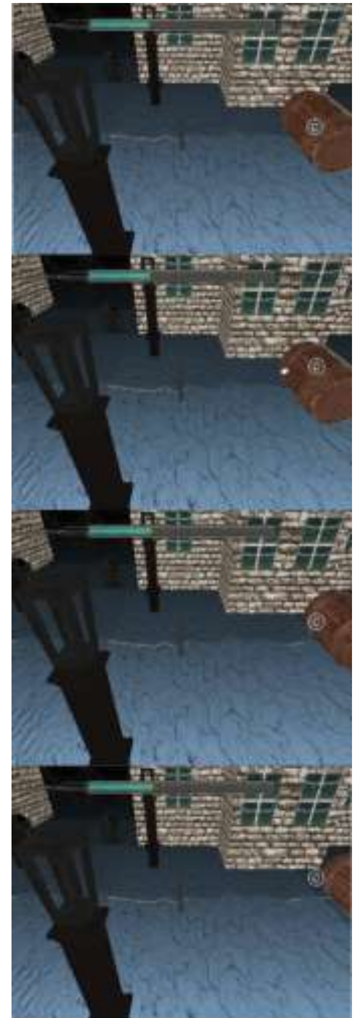

Figure 5 - Projectile Sequence

The initial version of the combat system was designed similarly to the character tracking system described above. A main singleton class, Weapon Manager, tracked all of the objects in the scene related to weapon interaction. The Weapon Manager stores a list of every combat-related object in the scene, and updates that list every frame. Any weapons, represented internally as either lines or points, are moved forward according to their trajectories, and are removed from the list when their simulation terminated. Ranged weapons are spawned as points that travel along a trajectory, and melee weapons are spawned as lines that extend from a character's position in a certain direction. When a new weapon is used, it is created and then registered through the Weapon Manager, which will track the weapon for the rest of its lifetime in the simulation. The weapon lifetime is set up to expire after a collision, or a certain distance or time are attained.

In this system, weapon shapes are updated manually (not via the physics engine), and are programmed to react to a collision through a callback system. When a weapon shape collides with another object, that object's callback gets run, and the object will then react appropriately to the collision. As the implementation only supports ranged projectiles, shapes are checked per frame via a point-in-box collision check - a method similar to checking two axis aligned bounding boxes. Instead of two bounding boxes, only a point and a single bounding box are tested. However, doing a single collision test is problematic, as a shape's position could be updated beyond the position of the object it needs to collide with. To

provide a more accurate collision response, the system uses several point-in-box collision checks extrapolated along the weapon shape's trajectory.

Because it would be very inefficient to check a weapon shape for collision against all objects every frame, the Weapon Manager makes use of the Octree data structure. This data structure spatially groups every object in the scene, such that when object interactions are queried, not all objects in the scene have to be checked every time. Once a weapon shape comes into existence, it is only tested for collision around nearby objects.

Once a weapon shape collides with an object, the Weapon Manager notifies the object via a callback, and the object can react appropriately. Characters, such as the player, react by taking damage from the weapon. Other objects, such as boxes, react by applying a force on itself based on how hard and where the weapon shape hit it. If, however, a shape were to go beyond a certain distance, or finish its trajectory, the shape would just simply be removed from the scene.

Joints

A joint in the Nvidia PhysX physics engine (Nvidia) is essentially a special relationship between two physics actors in the simulation scene. Examples of joints include hinges, pulleys, and ropes. Because the game centers around platforming and puzzle solving, there needed to be an efficient way to include joints in the level so a designer had more freedom and mobility in creating a challenging puzzle. The best way to specify properties of these joints in the simulation for this project was to make them configurable in the map file itself.

Each type of joint in the PhysX engine has several different properties that can be modified in order to make it behave a certain way. The challenge, then, was to allow the map file to specify which joint it wanted, along with whatever properties it needed to set. This was accomplished through a separate parser in the map loader.

The parser will first check what type of joint the map wants to use. It will then parse the respective properties of that joint. In the actual implementation, this simply means that all properties were checked all of the time, but certain properties could only be applied if the joint was of the correct type. Also, not all properties are parsed as strings; some are parsed as vectors, integers, or enumerations. Once all of the properties of an entry finish parsing, the joint will be added to a list to later become created as part of the physics scene.

Puzzle Objects

The primary use for the joint system, described above, is to allow for the creation of puzzle objects. With the help of the joint system, puzzles do not have to be hard-coded as classes in the project and have more flexibility in how they can be defined to behave. For instance, if a designer wants to try a new puzzle, he does not need to contact any programmers to make it, and can just instead create a unique configuration file, as described below.

Figure 6 - Swinging Platform

Puzzle objects are made up of special configuration files that, when referenced by the map file, will dynamically load in extra physics objects and joints at load-time. The configuration file is a script that uses a set of inputs and variables to define various physics objects and joints that can be loaded through the map loader. The grammar of this scripting system is as follows:

<Program> = <Input> <Variables> <Begin> <Script> <End>
<Input> = input <Type> <String> = <Value>
<Variables> = var <Type> <String> [= <Value>]
<Type> = int | vec3 | string
<Begin> = BeginLoad
<End> = EndLoad
<Script> = <String> [= <Value>] | ""
<Value> = <Integer> | <Float> | <Vec3> | <String> | <Expression>
<Expression> = <Addop> [+ <Addop>]
<Addop> = <Subop> [- <Subop>]
<Subop> = <Multop> [* <Multop>]
<Multop> = <Divop> [/ <Divop>]
<Divop> = <Integer> | <Float> | <Variable>
<Variable> = $<String>
<Integer> = <Digit> <Integer> | <Digit>
<Float> = <Digit> <Float> [.<Digit> <Integer>]
<Digit> = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9
<String> = <Character> <String> | <Character>
<Character> = A-Z | a-z | <Character> <Digit>

The rule labeled "Script" is what gets passed to the map loader to load into the physics scene. From this rule, any number of physics objects and scene joints can be defined and dynamically loaded during map load time. This provides the advantage that when using a puzzle object, the map needs to only know which puzzle to reference, and not what the inner workings of that puzzle are. As the grammar implies, this scripting system borrows from concepts behind programming language design and implementation.

In the script, variables and inputs are tracked through a string to structure mapping. That is, a variable name will map to a structure containing the type and value of the variable. For instance, a variable named "Object" can have the type of string and a value of "some object." Variables must have a value which, once initialized, cannot change for the remainder of the script. Inputs, by contrast, can optionally have default values. If an input does not have a default value, a value must be provided by the map file.

Variables can be one of three different types: a string, a numerical value, or a vector. A string is simply just a series of characters in sequence, treated as one single unit. A numerical

value, while labeled as an integer in the grammar, is actually treated as a floating point value internally. This allows for higher precision when using numbers in the script. Finally, a vector is a series of three floating point values, labeled as x, y, and z. These are typically used for specifying position, scaling, and rotation within the script.

Additionally, the script provides some simple mathematical operators to apply to variables or numeric values. Calculations for these operators are performed at run-time. As suggested by the grammar above, the order of operations is preserved for these operators. However, because each operator is considered separately, each will have its own precedence. This means that, according to the grammar, division will have a slightly higher precedence than multiplication, and similarly subtraction with addition.

In order to use a configuration script, an entry in the map file must provide the proper input for the script to configure correctly. This will include the script name and any inputs that the script requires. Once a script name is parsed, the stream will be passed to a special interpreter class. This class will parse out the remaining inputs, add them to the mapping, and then parse the referenced script. The output is sent to a separate I/O stream, which is passed back to the map loader. The map loader will then parse this stream as if it were from an actual map file.

The system is set up this way because it allows for more flexibility on the part of map designers and creators. Because puzzle objects can be dynamically added through this scripting system, the choice of what type of puzzle to make or put together is not limited to a small set of items from which designers can choose. In other words, designers are not limited to a given set of puzzle object classes in code, and are instead limited only by what the script will let them accomplish. Also, since these objects are not hard-coded into the game, the project does not need to be re-compiled every time a new puzzle is added.

Scene Triggers

Scene triggers are special objects in the physics simulation. As the name implies, they are meant to cause, or trigger, certain events to happen. They are very useful for creating player checkpoints, special in-game events, and reaching the end of a level.

Because only one trigger callback can be registered in the PhysX engine, every callback to a trigger must go through a single class. Also, since the engine does not keep track of any information other than the trigger shape, each trigger shape contains a special reference to an associated trigger object that holds the game state information necessary for a trigger to function. Each trigger object keeps track of things such as a list of objects to notify, and whether it can be activated by the player or not.

Once a trigger is activated, the callback class extracts the trigger object information and performs any appropriate actions. All objects associated with the trigger are notified of the activation through a different callback, and any joints (see above) are applied a number of pre-set actions.

Triggers can be loaded in from the map file via the map loader. Additional properties allow the map to specify which objects are attached to a trigger, and what actions to take upon activation.

Mesh Cooking

The PhysX software development kit (Nvidia) provides a helpful API interface for a service called mesh cooking. Mesh cooking, as defined by PhysX, is the process of converting, or cooking, a triangle mesh from its original form into a format the engine can load and use in the scene during simulation. The primary advantage of mesh cooking is that it allows objects to have more accurate collision detection. Instead of fitting each object to a bounding box, it is

possible to "cook" each object into a more accurate collision model. In order to use this tool to its full advantage, a separate project came into development.

The project is a command line utility meant to read in mesh data, and through PhysX, convert it to a format for the engine to use during simulation. When this utility is run, it will load in the mesh data, normalize it, prepare it as otherwise necessary, and send it to the mesh cooking function. From there, the data will be cooked and written out to a separate file to be loaded later by the game.

The mesh cooker uses a data stream interface, which needed to be implemented in order for the utility to work. This resulted in two implementing classes: a file input/output stream and a memory input/output stream. While the file IO stream is fairly straightforward (it just uses standard library file function calls), the memory stream is more difficult to understand. Though it may seem confusing, the memory stream class is very efficient in both execution speed and memory footprint.

The memory stream keeps track of the underlying data through a byte array that will dynamically expand as necessary. Size is tracked in terms of number of bytes, and read or write positions in the array are remembered through pointers. When multi-byte data is read or written, these pointers are decremented or incremented according to the number of bytes the datum uses. Whenever data is either read or written, the size is also modified accordingly.

After the cooked mesh is saved out to a file, it will be read in by the game once it has been started. At load time, the game will look for a cooked mesh file associated with each mesh, and load it if the file exists. When the cooked mesh data is loaded, it is re-scaled according to the scale of the associated mesh.

Results

Our team developed and implemented an interactive 3-D environment during the 20 weeks of the CSC 476++ program. We managed to accomplish a large number of graphics technologies, including deferred shading, normal mapping, vertex buffer objects, and model animation with shader support. On the gameplay end, however, we did not make nearly as much visible progress as we wanted. This is explained in more detail below.

Design

One of the first obstacles in designing the game was actually creating an architecture for the game. This was probably the most difficult, and yet the most essential aspect of the project. Without proper design, the implementation could fall apart very easily. Unfortunately, we did not have nearly enough time to design the code structure the way we wanted to, and had to eventually settle on what we thought was reasonable game architecture design.

Unfortunately, this proved to be a problem later, as we ran into a number of issues once we actually started development. Due to our lack of research and preparation, we had to re-structure our entire game and switch to a different physics engine halfway through development. If there is any one thing that is important to take away from this, it is that in order to create a good code base, we need to take the extra time to researching and planning to avoid having to do a major code re-write.

Teamwork

Another part of the learning experience for this game was our ability to work well as a team. For the most part, our group of 6 seemed to get along and work together well. We even had some people outside the group helping us with creating art assets and gameplay design. However, this did not come without its own set of problems.

One problem we experienced involved the use of our ticket system. A ticket system, which is essentially a global to-do list for the team, is very important when working in a group. It facilitates the work that all team members do, and helps to avoid major code conflicts. While this system did work for us most of the time, it was not perfect. Throughout development there were a number of code conflict issues, and occasionally times where a ticket was not honored completely.

Another problem we experienced involved working with others outside of our main group. At the start of our development, we tried to recruit a number of people to work on various other assets for the game, such as story writing or 3D model creation. Unfortunately, we found that only a small handful of those we recruited did actually stay with us for the duration of the project.

Our final problem was, in general, communication. Our team had pretty good communication amongst ourselves, but that unfortunately did not extend to outside the group. Perhaps the best example of this involved our artificial intelligence (AI) team, a group of 3 we had asked to work on the behavior for the enemies in our game. Although this team did a lot of great work, we did not know much about it due to a lack of communication. In addition, their AI did not actually make it into the final version of our game, because we had such trouble trying to contact their team.

Despite these problems, however, our team actually did fairly well in development, and we learned a lot from the CSC 476++ program.

Final Game

Because our team chose to focus more on graphics and back-end tools, we did not have as much gameplay as we wanted for our end goal. The final product of this 20 week program is a game in which the player can do some simple platforming, and fight off simple enemies. The

player also has a sanity bar, a focus ability, and a super jump ability as described at the beginning of this paper. While these key elements are in the game, other embellishments such as more level design, puzzle design, and harder enemies did not get added due to time constraints.

Although gameplay was lacking to a degree, we did complete a fully functional level editing tool. The editor features object placement, scaling, and rotation, as well as mesh importing and dynamic lighting integration. This tool allows a game designer to more easily create and test a level without needing to know too much technical information about how to create a map file. It provides a good visual interface so the user can quickly see what his changes will do for the level design. With this tool, we were able to more rapidly create new map ideas for our final version of the game.

Feedback

We had some good sources of gameplay and play testing feedback. For gameplay and project design, we demonstrated our game to industry professionals at Dreamworks and the Industrial Advisory Board, a group of volunteer professionals who come to Cal Poly and provide feedback for programs in the engineering department. Our presentation at Dreamworks went well, and the people there seemed to be fairly impressed with our work. We did not receive any direct feedback from the IAB group.

We also had some important feedback from other individuals play testing our game. One of the aspects that people liked in the game was the graphically rich environment, with an emphasis on lighting. Several testers also liked the physics simulation of the environment objects. Some aspects we were told we could improve upon included the general lack of visual feedback when performing various actions, such as shooting projectiles, a camera that had

some very strange and awkward movement behavior, and an inconsistent frame rate while playing through the level.

Future Work

The team plans to continue to develop and eventually release this game. The goal is for it to be released on the Steam platform (Valve Corporation) as an Indie Game. This involves creating a solid, stable, and presentable game with a few hours' worth of gameplay. Getting the game to this state, though, does require a lot more coding, play testing, and cooperation from our team.

Conclusion

Throughout this process, I learned a lot about software design and architecture. Although there were some specific requirements, the actual design and implementation of our project were left for our group to decide. For instance, when trying to figure out how to implement combat, I talked with the team, and eventually decided on a system I thought would work, which is what I described in the technologies above. Later in the development cycle, this system got re-implemented, and the new version retained a lot of the same concepts from my initial design.

I also got to experience just how important good software design is, when we had to re-implement a large portion of our project halfway through the process. As I found out first-hand, it is a very good idea to take extra time to research and think through code design before actually writing anything. It takes far longer to fix a design fault in a code base than it does to fix a design fault in a project architecture idea.

I also gained a lot of experience in working with a team. Although this concept wasn't entirely new to me, I did learn about useful group tools, such as subversion and a ticketing system. I did learn, however, that communication is probably the most important aspect in working with a group. Whenever problems did arise, we discussed them and were usually able to reach a reasonable solution.

Overall, this was certainly a great experience. The CSC 476++ program does take a lot of discipline, not only within our group, but within each individual. However, from this dedication comes a product that is just as meaningful as it is memorable. Without this discipline, our final product would have been far less impressive and memorable than it was. I would highly recommend a program like this to those looking for a dedicated graphics experience. It has definitely been the highlight of my education at Cal Poly.

Additional Credits

Josh Holland - Art Lead, 2D artwork
Ben Funderberg - 3D modeling
Tom Funderberg - 3D modeling
Hector Zhu - Splash Screen, Game Modes
Mikkel Sandberg - 3D modeling
Mitch Epeneter - Voice Acting
Ryan Schmitt - Deferred Rendering
Sam Thorn - Sound Lead

Works Cited

Activision. Activision Doom 3. 2011. 7 June 2011 <http://www.activision.com/index.html#gamepage|en_US|gameId:Doom3&brandId:DOOM>.

AutoDesk. AutoDesk Maya. 2011. 7 June 2011 <http://usa.autodesk.com/maya/>.

CDR Info. CDR Info. 15 January 2010. 5 June 2011 <http://www.cdrinfo.com/Sections/News/Details.aspx?NewsId=26708>.

Frozenbyte. Trine. 2009. 6 June 2011 <http://trine-thegame.com/site/>.

Nvidia. Nvidia Developer Zone. 2011. 17 March 2011 <http://developer.nvidia.com/physx>.

—. Nvidia PhysX. 2011. 17 March 2011 <http://www.nvidia.com/object/physx_new.html>.

Valve Corporation. Steam. 2011. 6 June 2011 <http://store.steampowered.com/about/>.