Third Degree

By: Michael Sanchez
California Polytechnic State University
Winter 2010 – Spring 2011
Advisor: Prof. Zoë J. Wood

Table Of Contents

Introduction

Why a Video Game?

Video games have been driving hardware and graphics development ever since the rise of the first video game consoles. Having a senior project that consists of making a substantial video game is an excellent way to learn about and implement many different types of graphics technologies. Building a video game also provides more experience in all aspects of real world applications development. From the processes of design to team management, the process of building a game will build experience in the many aspects of software engineering. For my senior project, I choose to implement a 3D interactive video game related to the platforming action-adventure genre.

Team and Course Structure

The *Third Degree* game began in Cal Poly's Winter 2010, CPE 476++ - Real-Time 3-D Computer Graphics Software Systems class. The focus of the class was to create a video game based within a 3-D environment, with course requirements to integrate certain graphical technologies into the game. In CPE 476++, Mark Paddon, Chad Williams and Michael Sanchez had the initial idea for a side-scrolling platformer built within a 3-D environment called *Third Degree*. Various people assisted in the project in order to provide assets and support by way of music, concept art and models, as well as general story development and voice acting. Tim Biggs, Jon Moorman and Joshua Marcelo then joined their programming team to also work on the game throughout CPE 476++, as well as into the Spring 2011 quarter as a senior project where our work on *Third Degree* continued with improvements to graphical technologies while also fleshing out the foundation for the story.

Genre & Setting

Third Degree is a 3D side-scrolling adventure game. The game takes place in the mind of the main character and the player is continually immersed with story driven game play. Third Degree combines story elements, traditional platforming and interesting game mechanics to provide a unique player experience.

Preliminary Game Decisions

Before the entire team was formed there were substantial design and game related decisions made. The group had first decided that this game was not only to be a senior project but the start of a game that hopefully could later be distributed out in the gaming industry. With this mentality in mind all design decisions were made for the "long run" of the game. The first decision that was made was the type of the game that was going to be made. Before this however the manager, Mark Paddon, thought of an excellent idea to play five different substantial, no flash games or mini-games, indie games and to come up with what was good about these games and bad. This list was discussed in great detail and it was decided that the game was to be graphically dense so as to immerse the player and hopefully make them feel as if they were actually in the game, as noted before *Trine* was the main inspiration for this aspect of our game. Also like *Trine* the basic game play was also going to be same, so our game is a platformer type game along with various objects in the environment that can be interacted with. After this decision the group decided that the most efficient and streamlined way to make such graphically inclined levels was to build a level editor. Bearing in mind that this was indeed for the long run, the level editor was to be a major focal point of some of the team members during the next two quarters. What was expected of the editor upon completion was a fully utilizable editor that could import meshes/animations, place them anywhere in the level, be able to apply any sort of translations, rotations, or scaling to them as well as modify any properties of that given mesh.

Alongside the decisions to make the game based off of *Trine* the story line for our game became very detailed and developed during this time. As the story was being developed the concept of the Mental Deterioration Bar was also proposed. The idea behind this bar was at first the simple replacement of the health bar and just dubbing it with its own special name. However as the story progressed so did the development of the MD Bar. Now the bar served as a gauge for the player to determine the mental sanity of the main character. Once this idea was solidified other ideas were built off of the MD Bar such as object switching, the focus ability and the firelegs ability. The object switching served to notify the player about several things: the reminder that the player is not in a real world, the level of the MD Bar, and the purely visual effect of having things between two completely different time eras.

Game Mechanics

The following are in depth descriptions of all of the core game mechanics to Third Degree. Some aspects of some of these are still a work in progress and most have been revised, redesigned, and/or re-implemented several times.

Object Switching

In the game every object in the scene is projected to have both a futuristic and a historical mesh. These objects will randomly change to their associated counterpart at a given MD level. Objects are categorized into three different areas Large, Medium, and Small. These categories have a set range within the MD Bar that will then assign a randomly generated MD value to that object within the constraints. This allows the flow of object switching to make some sense to the player in the game. Because if the buildings and bigger platforms were to be switched to their designated futuristic mesh before say a small barrel or cube, the perception of the level would make less sense to the player.

Mental Deterioration Bar

The Mental Deterioration Bar or the MD Bar for short is the mechanic of our game that facilitates almost all of the rest. The MD Bar is the level of “insanity” that the player is currently experiencing. The higher the insanity the more futuristic objects will be apparent in the level. Also along with a higher MD level is the amplification of player abilities. The MD Bar is able to be reduced by using the players focus ability. There is various tradeoffs’ between having a low and high MD, balancing these varying levels of MD and abilities are left to the player to decide. This allows different gamers to play in accordance with their own styles. Once the MD bar gets dangerously close to full the MD Bar will begin to flash red, as will the player.

Fire Legs

Fire Legs is a special ability that the player can use once a player has 35% of their MD Bar filled. At this point the player can then activate this ability by pressing the space bar. Once this ability is activated the cursor will change from a crosshair to a red mouse cursor. The player can then use Fire Legs by clicking the left mouse button. However to use Fire Legs the player must be standing on a surface. After the ability is used a fiery particle system will erupt from the players’ feet and the player will zoom towards the direction in which the mouse cursor was clicked at. After the use of this ability there is a cool down for the ability to reset. The higher the MD a player has the farther the character will go.

Focus

The Focus ability is used to lower the players MD level. This ability is simply activated by holding down ‘f’, once the ability is activated the players MD level will start to fall. During this time objects will switch back to their original state if their appropriate MD level is met. Once the player decides to release ‘f’ a spherical particle system will burst out and around the player.

This particle system acts as a force field and will appropriately send environment objects flying away. The force field generated is proportional to the amount of MD that was “Focused” if the entire bar is “Focused” from almost full to empty is when the biggest force field will propagate through the map away from the player.

Puzzle Objects

Puzzle objects are the main environmental obstacles that the player will be dealing with throughout the game. These puzzle objects can range anywhere from swinging platforms to swinging spike walls. Only when the player successfully interacts with these objects is when the player can proceed throughout the level.

Third Degree Story

The game follows the story of a convict kept in confinement who is essentially given a chance at redemption through a special testing program. A recent alien artifact has crash landed on the Earth’s surface, and a panel of scientists are conducting specialized experiments to find out what it does. The convict is one in a line of test subjects given a chance of freedom through experimentation. When the artifact is fitted on the convicts head, he is put in some kind of virtual environment that resembles London in the 1860s. The convict, though determined to gain his freedom, soon feels the grasp of insanity closing in around him, and the only way out is to either finish the virtual simulation, or die trying.

Project Design Specifics

As a development platform for Third Degree Microsoft Visual Studio 2008 was used. For our editor UI system we used Nokia’s Qt libraries and tools. The Visual Studio plugin was used to integrate Qt and facilitate the design of the Level Editor. The source code was under version control and the primary language was C++. The repository for our entire project was hosted on Unfuddle.com. All team members were given accounts and access to the repository to commit and update changes to their projects accordingly.

Project Design

First quarter design comprised of three projects in one Visual Studio project solution. These three projects were the Editor, the Engine, and the Game. The Game project consisted of the main game loop and all pre-game initializations. The Editor contained all of the code necessary to run the Level Editor, as well as the code for manipulating objects in the Level Editor. The Qt UI forms and initialization were also kept in the Editor project. The Engine contained all of the graphics, game play, sound, player/enemy, and higher level utilities to assist in calculations in any of the code.

The second quarter design comprised of four projects. The three of the previous quarters and the new MeshCooker project which handled all of the mesh cooking required by PhysX. However unlike the last quarter the code contained in each project was drastically redesigned and restructured. The Third_Degree project (the Game project in the first quarter) now contained many more classes and newly implemented features. The project contained a StartUpMenu which took care of the preliminary splash screen that allows the game to run in different resolutions, full screen, and to enable the Debug Mode if necessary. The Menu System code which handles the beginning of a new game, loading a level or exiting is also in this project. Some classes and code that was moved from the Engine project was the player/enemy code. Some other code that was also newly implemented in project was sounds, triggers, non-rigid bodies, and cooking the latter two come with the entire project being newly ported to PhysX. The Engine also had many classes added and implemented due to the PhysX port but for the most part the code contained in the projected was the same, there was the addition of various technologies and these will be explained in greater detail in the implementation section.

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*. 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 Tr*ine*.

Fig. 1 – Screenshot for *Trine*

Maya

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

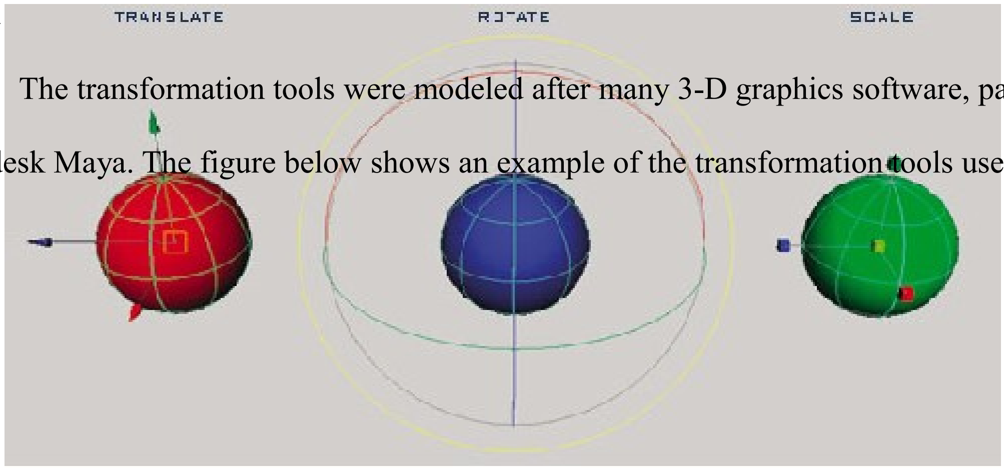


Fig. 2 – Examples of the Translate, Rotate & Scale tools in Autodesk Maya.

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*. 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*

Implementation

Overview of main game loop

The game is run through the GameApp class. This class sets up the SDL initializations including the key press events. There is a main update loop that updates the current game state whether it is the menu system or the actual game update loop. The last task the GameApp class deals with is loading a map. This function initializes the main game engine and the game class; it contains only function calls to the lower level load map call which is in the game class. The loop will then run until the exit command is raised.

Overview of 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

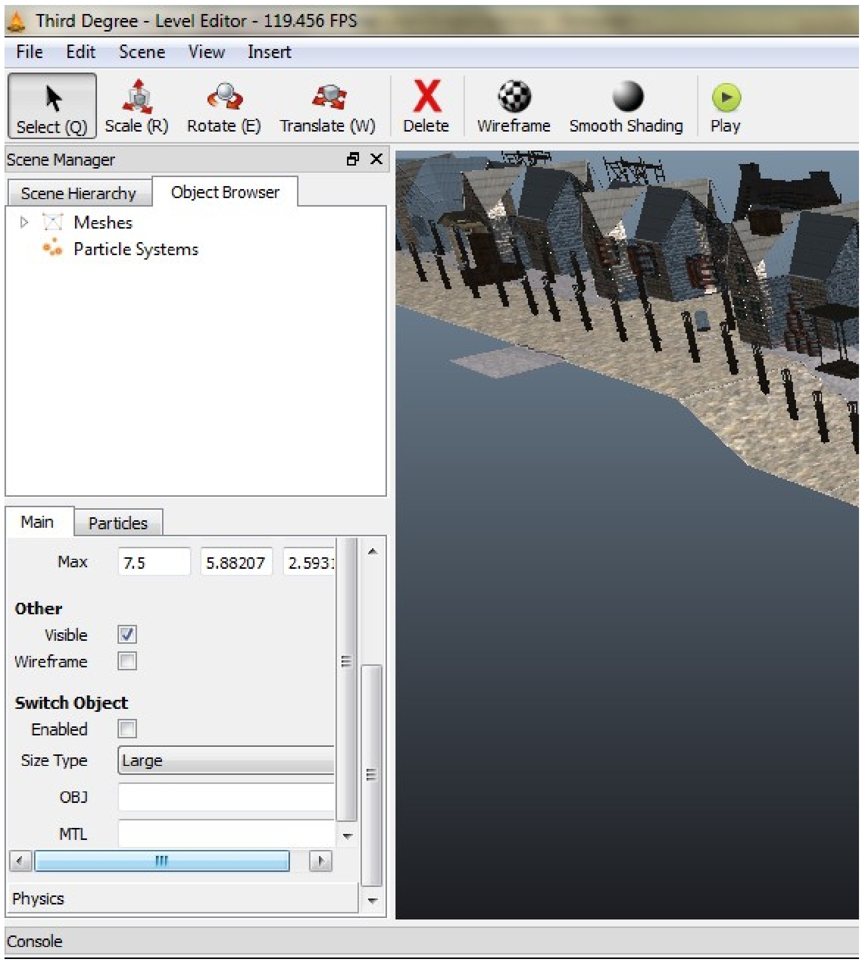
Level Editor

The Level Editor is a high scale UI system that allows anyone to make levels for the game. Every object can be imported through a simple list that is propagated in the editor. Meshes are added to this list if more are moved into the specified directory. After importing an object all attributes for position are editable. Particles trigger points, and enemy/player spawn points are also able to be placed into the editor. The switch objects are all able to be seen by clicking on the Future Preview button.

Object Switching

In the game there are objects that can be switched between a historical and futuristic mesh.

Every object that can be switched is placed in a category: large, medium, or small. Categories had to be made to place these objects in because having a building or the floor switch to the futuristic mesh while a small trash can or barrel was not switched was undesirable behavior and was thought to detract from the overall fear that was trying to be accomplished. The object would be placed in these categories in the Level Editor.

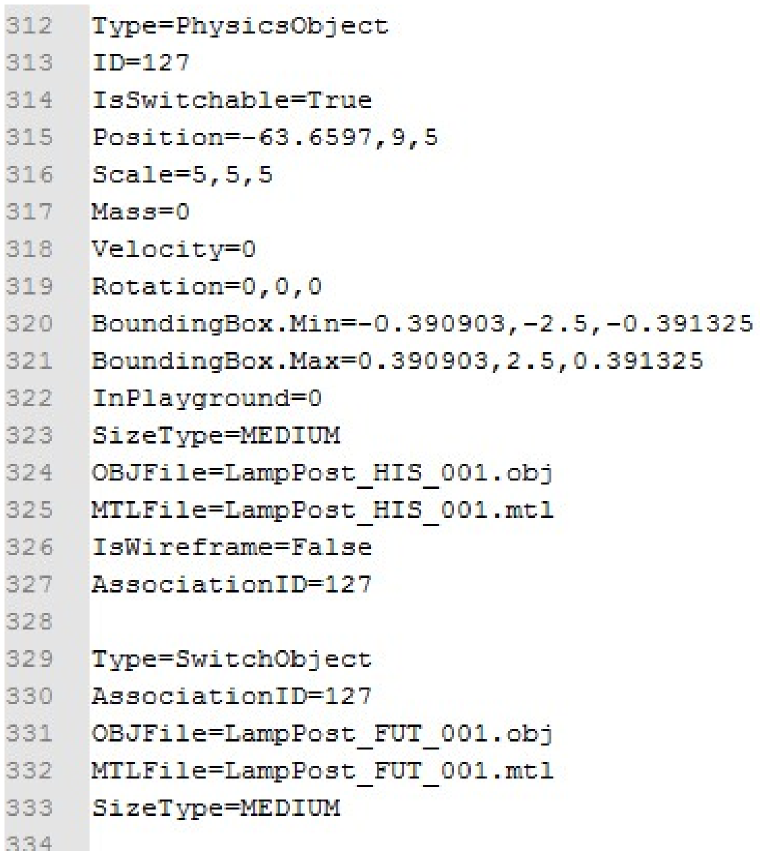

Third Degree - Level Editor - 119.456 FPS
File Edit Scene View Insert
Select (Q) Scale (R) Rotate (E) Translate (W) Delete Wireframe Smooth Shading Play
Scene Manager
Scene Hierarchy
Object Browser
Meshes
Particle Systems
Main
Particles
Physics
Console

Max 7.5 5.88207 2.5931

Other
Visible ☑
Wireframe ☐

Switch Object
Enabled ☐
Size Type Large
OBJ
MTL

When the Level Editor saves a map, the object with a switch mesh contains an association ID, the flag for denoting a switch object and the size label.



```
Type=PhysicsObject
ID=127
IsSwitchable=True
Position=-63.6597,9,5
Scale=5,5,5
Mass=0
Velocity=0
Rotation=0,0,0
BoundingBox.Min=-0.390903,-2.5,-0.391325
BoundingBox.Max=0.390903,2.5,0.391325
InPlayground=0
SizeType=MEDIUM
OBJFile=LampPost_HIS_001.obj
MTLFile=LampPost_HIS_001.mtl
IsWireframe=False
AssociationID=127

Type=SwitchObject
AssociationID=127
OBJFile=LampPost_FUT_001.obj
MTLFile=LampPost_FUT_001.mtl
SizeType=MEDIUM
```

The code for implementing this feature was contained in the GameObject and Game classes. The constructor of the Game class calls the AddToSizeVectors after the initialization of all of the physics objects. The function then loops through the entire objects array(which is already filled from the InitPhysics call and then does three checks to see if the object is switchable and of the given sizeType specified in the map file. If this is the case then the objects is added to an array of the appropriate size type. Once this is complete the GenerateMDVals function is called to loop through all three arrays and randomly generate MD switch values for the objects. Again these objects receive appropriate MD values within the given range due to the sorting of sizes. Also within game there is a function called ReadjustMDObjects, this function loops through the entire object array and determines which mesh should be drawing with the current MD level. The SDL timer function MDObjectUpdateTimer calls itself every 500 milliseconds and goes through the object array and determines if each object needs to be switched and then sets the IsSwitched boolean. An important note to the time choice is that this timer must have a faster callback time than the rate at which the MD updates itself or there may be occasional skipping of objects and inconsistent transitions. The logic of which object to draw is located in the Draw function of the PhysicsObject class. In the function there is an if else conditional statement that tests whether or not the IsSwitched flag is set and then calls the function to draw the appropriate mesh. Objects that do not have an associated switch mesh will never be switched due to the IsSwitchable flag in the GameObject super class.

Trigger Scripting System

Although this technology was not used in the game due to pressing time constraints the architecture is set up for later use in the game. Script based events are the easiest way of creating special events in the game. Some examples can be playing a certain sound, creating enemies, and changing colors of lights. The scripting language that was decided upon in a team meeting was Lua. The version of Lua that was used is version 5.1. The integration of Lua into the Trigger class required the extern "C" call to the Lua 5.1 libraries. In order to execute a Lua script from the C++ code some initialization and setting up of a Lua state is required. As seen in the ExecuteScript in the Trigger class there is a call to lua_open(), this call returns a new Lua state to be used by the C++ code. Once this is done the libraries must be loaded into this state in order to use the Lua libraries with a call to luaL_openlibs(LuaStateVar), where LuaStateVar is the state variable that was attained from the lua_open() call. There are also similar calls to open Lua's io, math, and similar such libraries. This is all that is needed to initialize Lua to begin using it, now Lua files called from the C++ code and the Lua Script can then in turn call back into the C++ code. In order to do this however for every script there needs to be a call to lua_register(lua_State * L.const, char* filename, lua_CFunction fnctName). A proper C/C++ function declaration example is static int FuncName (lua_State *L). After registering these functions with Lua to execute the script just one function needs to be called: luaL_dofile(lua_State * Lstate, char* ScriptFile). An example of a simple Lua script and expected behavior can be found in the appendices section A. With this architecture in place the game can now contain dynamic scripting events based on positional or environmental queues in the game. An important note about function placement is that all functions must be declared so as to be visible to the lua_register function calls. This requires that they be static functions and not be class member functions. Another side note about getting Lua integrated with Visual Studio is that the Lua library files and headers must be obtained via the Lua site. These libraries must then be added to the additional dependencies section under the Linker section of the project properties pane.

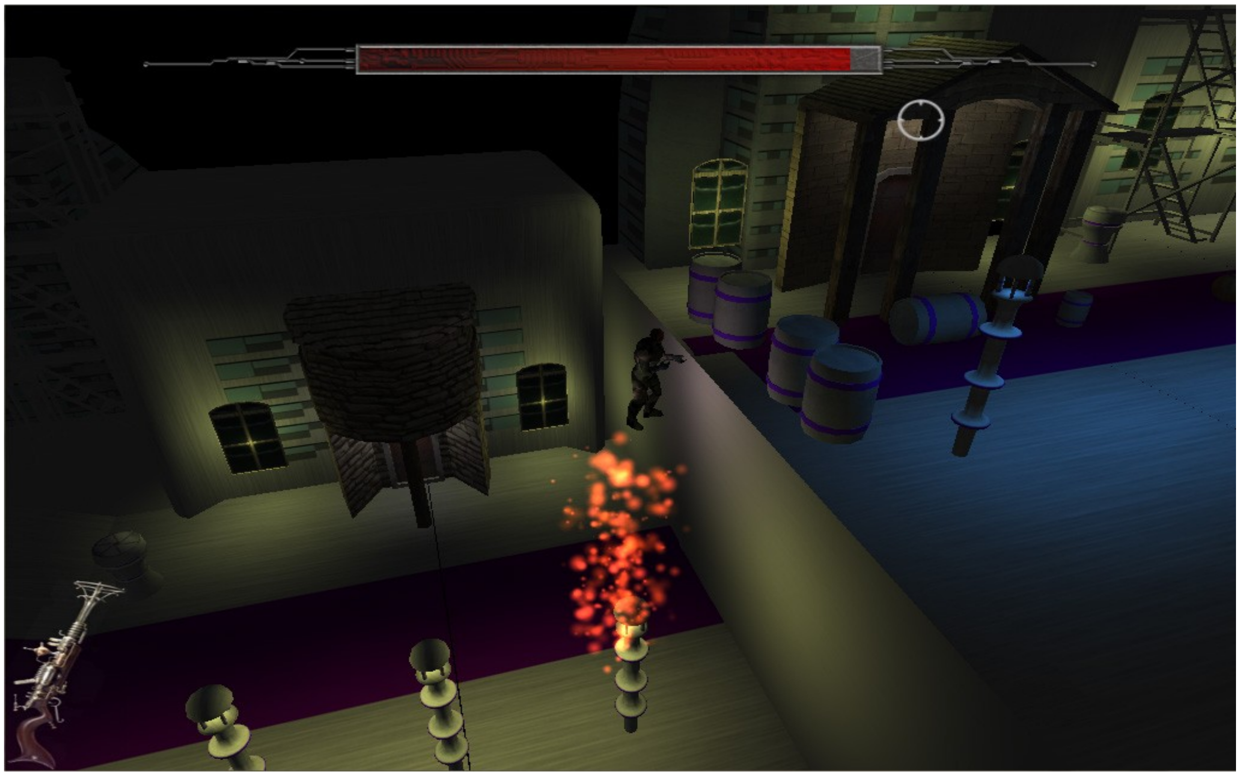
Parent/Child System

As the game developed it became apparent that objects were going to be needed to associate with each other. This is especially true with a side scroller game as objects in the foreground and background are static objects that the player never interacts with. This technology was necessary to tie lights to lamp posts, attack a particle system to the player, as well as have objects that interact with each other linked for easier access to fire events. This was another preliminary architecture setup of the first quarter of the game. This small system was to integrate into the code a simple way to associate objects with each other for purely checking positional differences between both objects or to merely keep one objects in the same relative distance to another object. To implement this simple yet useful feature an stl vector of type GameObject was added to the GameObject class. There was a method called

AddChild(GameObject* child) that would add the child to the Children vector. Then anytime the base object had its position updated, all of the children would also reflect that change. This was primarily used for attaching a ParticleSystemInstance to the player. However if the game needed particle systems for instance to follow another object or to stay within a certain distance, they would need to be part of this simple system to accurately reflect positional changes.

FireLegs

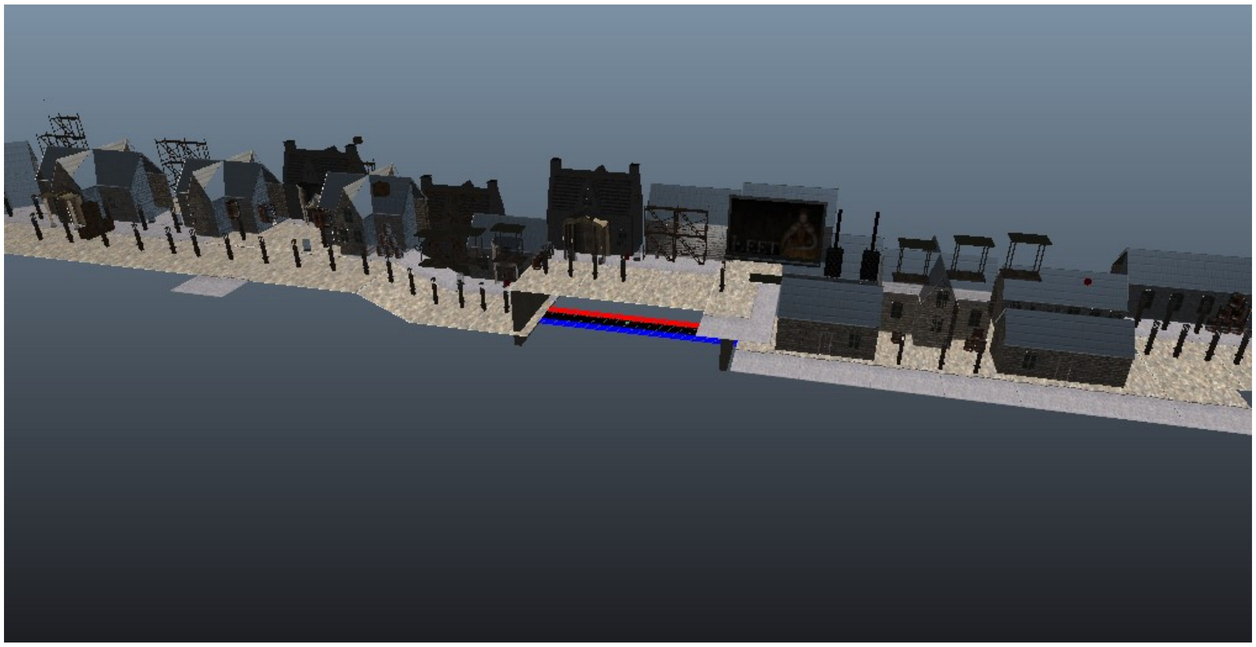
There were many design decisions regarding FireLegs in regards to behavior, shooting distance, particle systems, and behavior of the character. At first fire legs was implemented with the Bullet physics libraries. After the decision to move over to PhysX a massive code change occurred to most of the base architecture so fire legs had to be re-implemented to suite these changes. Instead of an impulse of a certain magnitude being applied to the character as in the Bullet implementation, there is now a specified distance to move over a certain time period using the PhysX CharacterControllers call to move(…). The original FireLegs implementation was all originally contained in the PlayerCharacter class. With the port over to PhysX and the re-architecture of the project the FireLegs code was re-implemented in the Abilities class. In the GameApp class there is an event created to toggle special abilities, this even then gets propagated to the Game class RegisterInput function that checks one of the restrictions of fire legs which is if the ability is ready to use again from the timer cool down. If so then the boolean for activating the players ability is negated, the FireLegs cursor position is calculated and the HUDDisplay's boolean for showing the FireLegs cursor is negated. Once the player presses the spacebar the similar key handling events happens except with a different event. This event checks if the player ability is active and if so will then proceed to check if the player is on the ground in the Player class. If all of these conditions are met then in the Abilities class the UseAbility function is called, inside this function the movement vectors for the direction in which the player will propagate is calculated, the ability and cool down start timer variables are initialized and the FireLegs particle systems' PlayBurst method is called. The initial IsActive boolean is set to false and another that is only used in the Abilities class called UsingAbility is set to true, the DisableKeys boolean is also set to true along with the CoolDown boolean. The after effect of this function results in the keys being locked, the cool down timer starting and the player beginning to boost through the world graphically with FireLegs. In the Update function of the Abilities class is where the distance and the cool down timer is being updated for FireLegs. Once the distance traveled is met the UsingAbility and DisableControls are set to false and the FireLegs ability is over. The update will continue to update and set the CoolDown to false once the arbitrary cool down time is met. This completes the cycle of one FireLegs use.



Mesh Importing

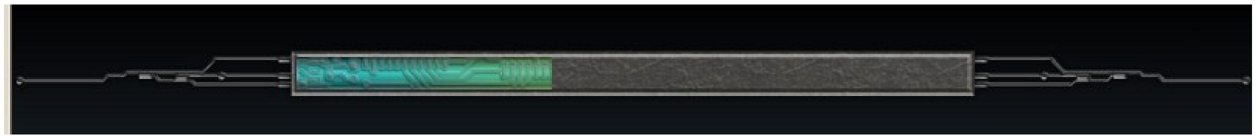
All games need some sort of way to import the 3D objects modeled by the artists into the game. The OBJImporter does just this and will import any properly formatted .obj files. The OBJImporter is a C parser for OBJ files. The code was built off of Zoë Woods basic mesh importer. The OBJ format is a very diverse format and can be presented in a numerous amounts of ways. For the game the standard that the graphical artists adhered to was one of the fuller formats. This format contained a material file name (.mtl) from which all of the material properties and textures are defined. Big groups of vertices, normals, and texture coordinates would be specified for different regions of the model. The importer would then associate all of these groups with their respective materials and textures. During this process the extants of the mesh is recorded as well as after the mesh is done loading there is a scale value generated that scales down the mesh to a standard size, this is for meshes that are made in different scaling systems.

The OBJImporter is used as a purely static class and is therefore put into the utilities folder of the project. The class takes in a pointer to a Mesh object and an .mtl and .obj files. It then opens the files fills the Mesh objects face, texture coordinates and normal vectors. This data is then used in the VBO for the mesh. The parser will scan each line in the .obj file and do various actions when different data is encountered. When a ‘v’ is encountered this signals the parser that the next three chunks of data are going to be vertices and handles them by adding them to the vertices vector. If a ‘vt’ is encountered then the parser knows that the next two chunks of data are going to be texture coordinates. If a ‘vn’ is encountered then the parser knows that the next three data chunks are vertex normal’s. If an ‘f’ is encountered then we know that the next nine pieces of data are three faces with format “f v/vn/vt v1/v1n/v1t v2/v2n/v2t” and will be added to the faces vector. The previous four data formats consist of most of the .obj file for most meshes. There are other key words that indicate different properties for the mesh such as “usemtl” and “g”. The “usemtl” keyword indicates the texture (if any) and material to associate with the current segment of the mesh and can be located in the .mtl file. The “g” keyword is the group name of the current segment of the mesh that is being processed. These flags usually directly preceed the faces data which makes sense for this format is usually used during run time in some editors and needs the textures and materials to associate to the faces that will be given directly after. The .mtl file contains three keywords that are important for material specification. They are “Ka”, “Kd” and “map_Kd” which represents the ambient, diffuse and texture properties and filename respectively. The Level Editor and most of the game uses only the OBJImporter to import meshes.



Mental Deterioration Bar

The Mental Deterioration Bar (MD Bar) is one of the main components of the game. A lot of the story and the game play mechanics correlate directly with the MD Bar. The first iteration of the MD Bar started off as mostly just the health of the player and was originally located at the bottom of the screen instead of the top where it currently resides. The MD Bar is implemented inside of the MDBar class. This class handles the updating and drawing of the inside of the bar only. The external “case” of the MD Bar is located in the HUDDisplay class. The MD updating is contained in an SDL timer. The rationale behind using the timer and not an update loop was because at this point it was believed that using the SDL timer would not affect frame rate or performance in any way. However upon completion of the first quarter profiling of the project showed that the timers were indeed taking away from the framer rate. Original positioning of the MD Bar was hardcoded, but was then readjusted to work with any screen resolution.

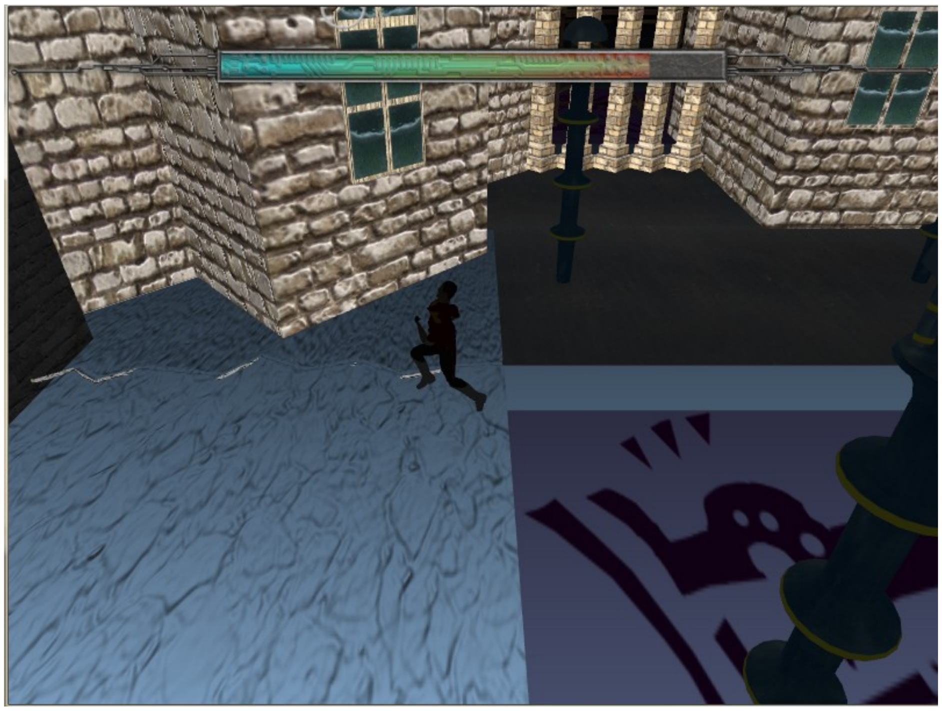


Animation

Animation proved to be the most difficult task for both quarters. The main reason for this was because of trying to use an animated mesh that was made for us by our art assests team. The reason this was difficult was because of the specific options of the format needed to be set when exporting the mesh. Trying to identify these options proved to be too difficult for the programmers and artists. There were many iterations of animation implementations as well as the various animation format used. The animation formats that were explored and failed were FBX and Collada. The final animation format that is used in the game is MD5 animation.

The first attempt in using animation in the game was in incorporating Collada. The Collada format had a parser already set up for animation, the Collada DOM (Document Object Model) is a tool for parsing documents in Collada. Incorporating all of the libraries into Visual Studio and setting up the framework for the Collada animation proved to be extremely difficult. One of the main problems was that the Collada DOM came with numerous external libraries, some of which happened to be already included in the game. One such library was the Bullet libraries, however instead of including the entirety of them there was only a small portion. This was fine until it was discovered that these partial Bullet libraries were from an older Bullet release, and on top of that these libraries were customized especially for use with the Collada DOM. Unfortunately there was an inescapable linker issue that could not be resolved which had to do with duplicate Bullet functions being included and therefore had to abandon the prospect of using Collada.

The reason for choosing Collada first was because of the really simple methods of calling different animations of the character and leaving everything else to Collada to handle. The second just as important reason was that the art assets team was making all of the assets in Maya and one of the export formats from Maya was Collada. The second implementation of animation was using the FBX format. There was successful implementation of this format however FBX can have animations exported in takes or by keyframe. Example animations from the web that was in the takes format worked successfully in the game and the proof of concept was even used for one of the milestone demos. The only snag that was encountered with this format is that when exporting to the FBX format from the maya file the artists must know how to do so. Unfortunately there was not enough documentation for the artists to learn how to export to takes so the animated character model was exported by keyframe. All of the FBX animation was handled in the FBXLoader class. This class was adapted from the animation example that came with the FBX sdk, and modified heavily to work in the game. Once again integrating the sdk into the game proved to be quite difficult. Unfortunately the FBX animation example used lots of global variables in their code. Not only were these variables in some of the source files but were used in the headers themselves too. In the end five global variables had necessitated themselves into the FBXLoader and the game. The extern C call was once again needed to solve linker issues from some of the header files. Once all of the linker issues were dealt with, the modification of the animation loader took place. Since this example was for demonstrating lots of features of the FBX sdk that could be used a lot of code had to be stripped out. Once it was fully tailored to the game the animation that was made for the game was introduced. The character originally had four animations that was exported into the FBX format, the artists provided a key frame key for the animations as well. Once these keyframes were set in a higher level function created in the FBXLoader class to facilitate the switching of the animations, only one animation seemed to execute properly. Eventually it was discovered that the SDL timer was not exactly consistent and the Draw loop was being called from within the game update loop. These two did not work well together as sometimes one would execute twice while the other has only executed once before or after that double execution. This eventually led to very undesirable behavior.

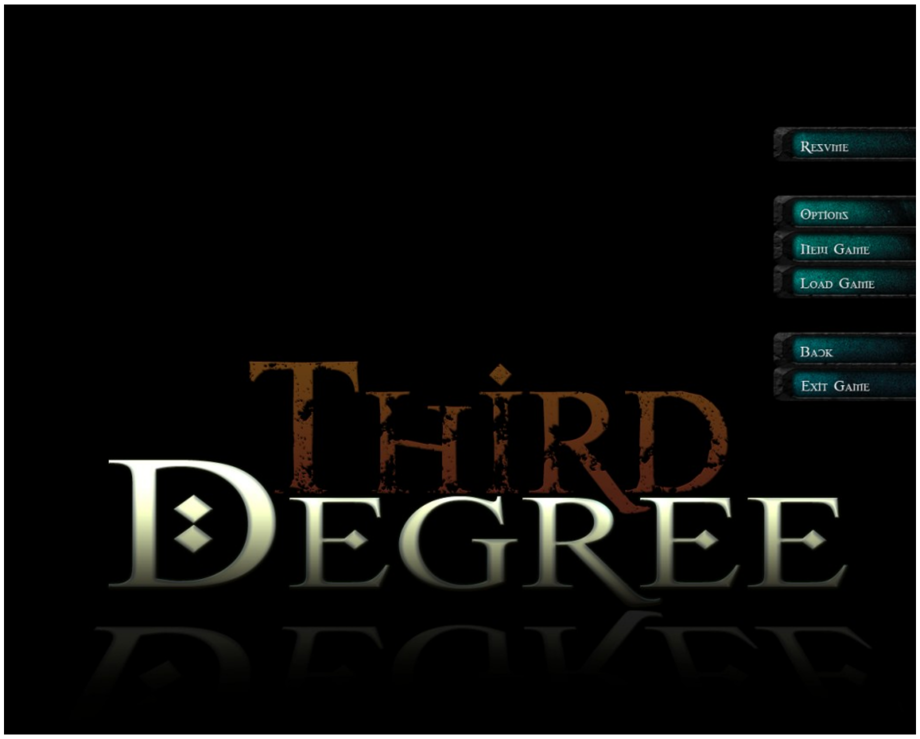


As is easily distinguishable the character has a very warped left arm and the entire body is leaning to one side, which should not be happening in the normal run animation. The final animation format used was the MD5 format. Unfortunately the assets team has not figured out how to export to this format so there is a temporary animated character for now.

Menu System

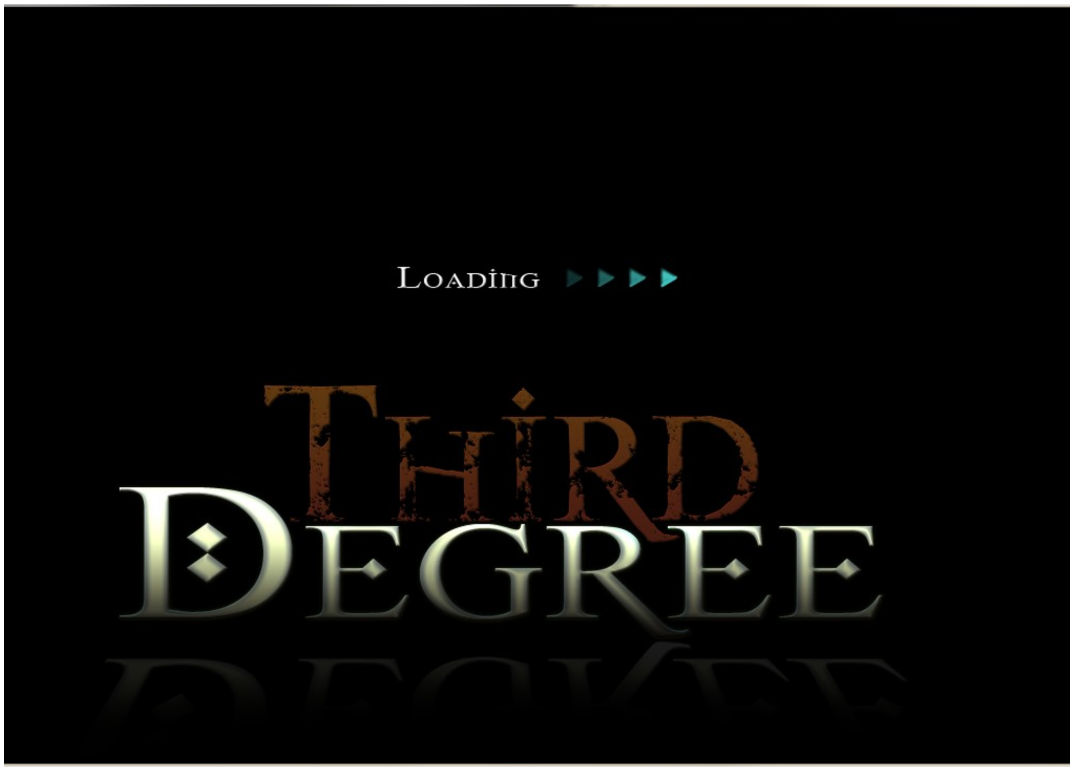
All games need a menu that will handle starting new games, setting appropriate options, and loading some sort of saved state. Without a menu the game visually lacks the components necessary to convince players to play the game. The menu system for third degree was implemented as a separate system away from the Engine code. It is a part of the higher level of the overall architecture of the game. The menu

system can load a new game or a new map. The load map functionality is in place, however, at the current time there is a lack of maps to load for game play so there are no maps to load if this screen is chosen it will be blank. The menu system is screen resolution independent. No matter what size the screen or the textures, if stretching were to be graphically permissible, the menu system will accurately display the menu and handle button input correctly.



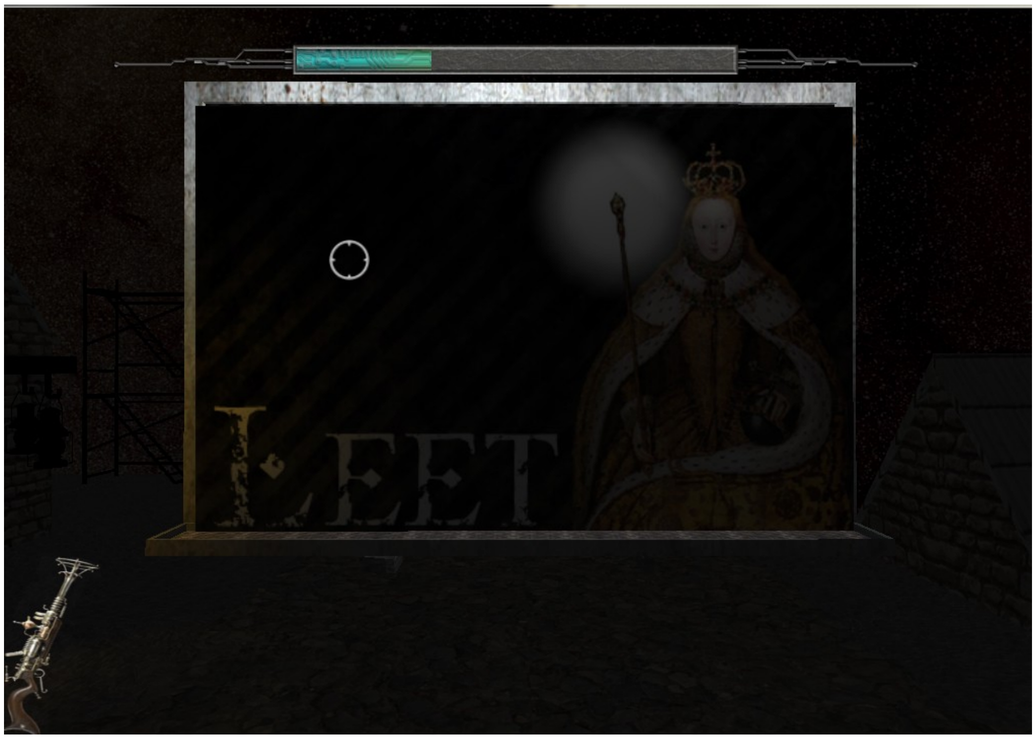
This is done by mapping texture coordinates from pixel to world space coordinates and using the screen width and height and the button width and height. By specifying a relative position using these variables, and the desired layout any size screen and button textures will correctly map to the desired position every time. There is a slight difference however once the full screen flag in the SDL setup for window is raised. A bug that is known with this method is that when full screen is active the specified game screen size is attempted to fit onto the users display. When this happens it is possible for the coordinates for registering the mouse input for button clicks becomes slightly skewed. For highlighting the buttons all that is needed is to draw the highlighted button instead of the dark button. All button events and handling happens in the

CheckButtons function. Switching from one screen to another screen is also handled in this function, by setting up various enums for buttons, screens, and mouse states the menu system becomes easily manageable when handling events. Adding or deleting certain events is also relatively easy to do and does not take much coding or re-analyzing of any code. By having an active screen variable different sections of this function are easy to identify and modify. The use of enums makes determining what happens when a button is clicked very trivial.



Animated Textures

Animated textures are used in two places in the game currently. One is the Loading Screen in the menu system and the other is an animated billboard in the game.



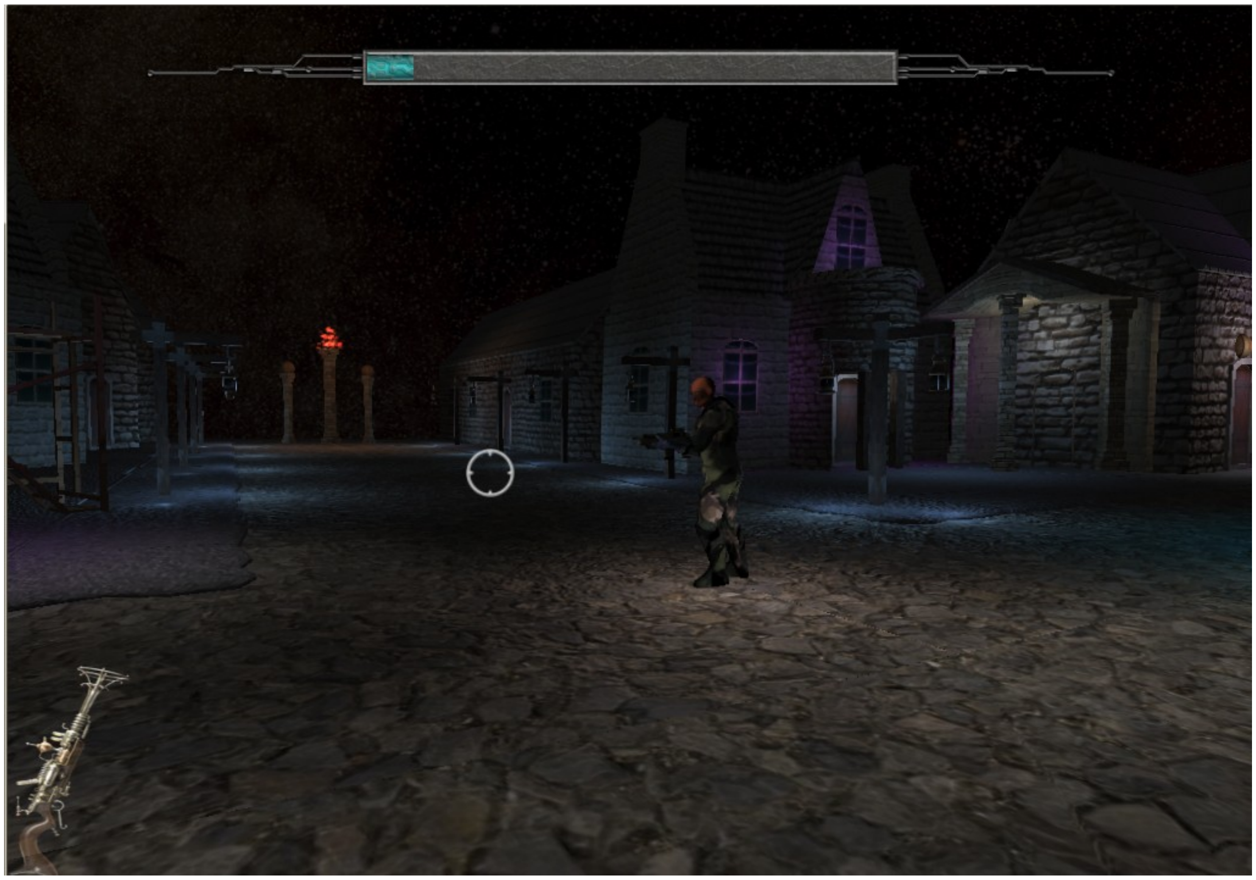
The current updating of the billboard texture is done in correlation to the MD Bar. Each time the MD Bar increases the animated textures is advanced one step in its animation. The textures are set to loop forewords and backwards through their respective animations. Traditionally a timer would make more sense to control the flow of the update more smoothly however as mentioned before timers cut down on performance and were therefore taken out of the game completely. With other textures in the future the game will use the update timer to be able to accurately control the speed of any animated texture.

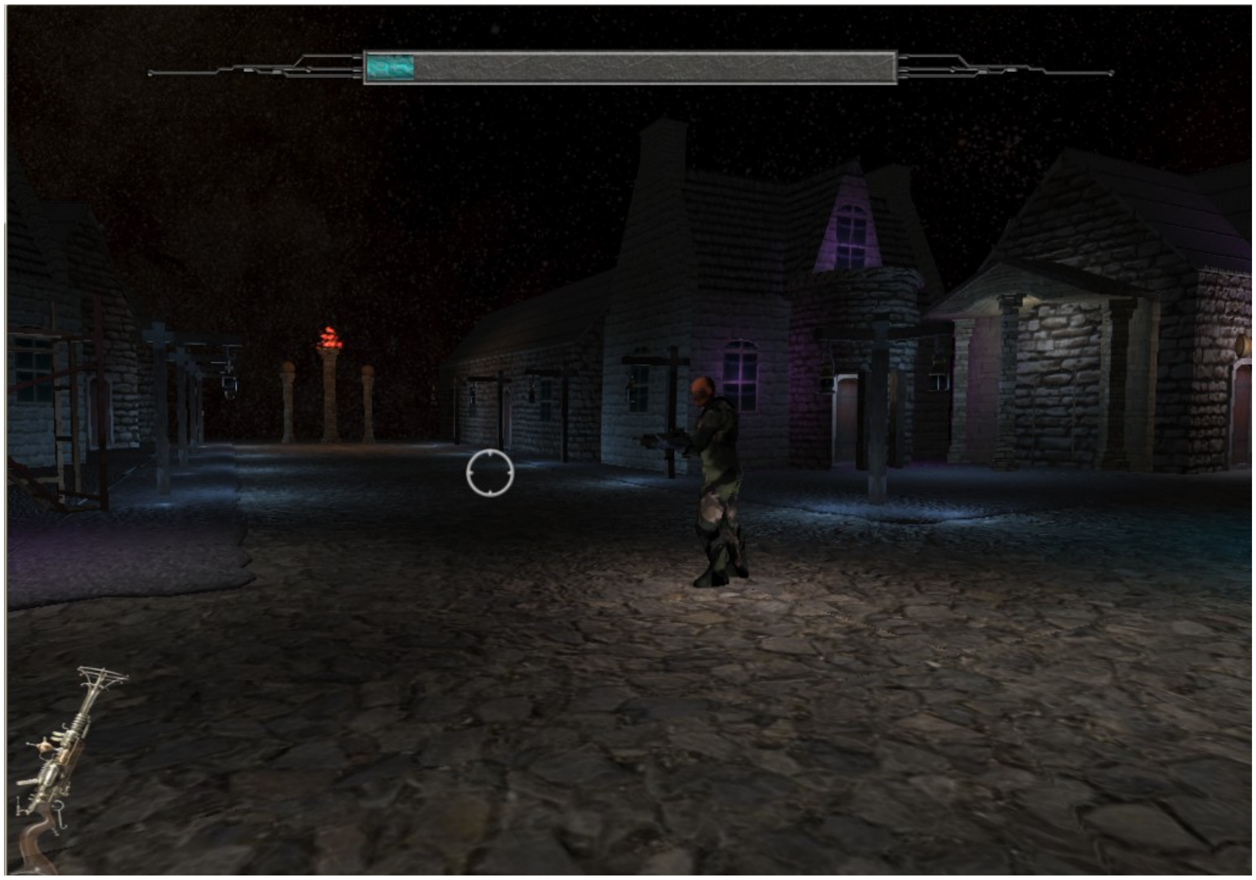
Results

The end product of the two quarters ended with a fairly stable game with high amounts of implemented graphics technologies and basic game play. All of the graphics technologies previously mentioned were all achieved and the game itself was very graphically pleasing for most who played the game. The game play consisted of basic enemy combat and basic environment traversal. There were no highly complicated environment puzzles or bosses of any sort in the game. The first level of the game is going to be used as a tutorial to intro the game and its' story to the player.

Biggest Accomplishment

The overall graphical goal of our game was to set an appropriate ambient mood for the world such as is done in Frozenbytes' *Trine.* This is the greatest accomplished of our finished product:





By interesting manipulation of the graphics pipeline the game was able to have various graphics technologies such as Normal Mapping and Deferred Rendering added to the game. These two technologies are the most essential in providing the mood and feel from the game. Along with the highly reliable editor, making amazing levels becomes easily possible.

Testing and Feedback

During testing there was lots of feedback given to help improve the game and determine what was working and what was not. Some of the aspects that people liked in the game were the graphically rich environment, with emphasis on the lighting. People also liked the physics effects of the environment objects and the game actually being a sidescroller. Some aspects that people did not like included not being able to see the bullets, better control of the camera (there was too much movement from the camera), inconsistent frame rates in certain areas of the map and the gun of the player being too small. Also some testers noticed that there was not enough choices or for the player to do in the level. Testers varied in responses on whether the game was easy, difficult or just right.

Team Development Process

Working on a programming team of six people can be quite a challenge to manage at times. Even with a good ticketing system and good communication there can still be some issues in assigning or giving out work to do. There are certain tasks that may take longer than others and some tasks may also not involve working on the game directly. This was the case with implementing the editor and working on tasks such as sound, or a menu system. What does not work with big groups unfortunately is the ability to keep everyone busy with something that the person would not mind working on. The good thing about a bigger team is the ability to get input and feedback about architecture and game design choices. The first quarter architecture was only provided with input from a few of the team members and towards the end of the quarter there were large parts of code that architecturally were out of place. Also another mistake was the use of the Bullet Physics Libraries, only one person researched physics libraries and not enough care was taken on thoroughly researching the libraries. Unfortunately Bullet had very little documentation and examples, there was a point where progress was seeming to halt because of this. So the decision for the second quarter was to switch to the PhysX Libraries and to re-architecture the entire project. The end result was that most of the code that was

previously written had to be re-implemented, and therefore cost the team a lot of implementation time for other game components. The game continues to hold a firm architecture and there have been no real problems with this or PhysX.

Conclusion

There was lots of new knowledge gained from working on this two quarter long project. Good communication and group dynamics are definitely something that I learned from working in a large group. As soon as communication is lost with someone or there is a group conflict, there begin to arise issues within the group and the amount of work getting done. Also analysis of code as well as technical specs of code was also that I learned more of. When implementing a game mechanic it is important to know exactly what the expected behavior as well as any post or pre-conditions. There were also many times where I would have to look at others code to determine either how to integrate my code or to find a similar solution to a similar problem.

Time management was also something that was improved upon during the two quarters. When making a game is very easy to pour countless hours at a time into implementing a new feature or something that would make the game run faster. This does not work for the work from other classes will begin to pile up and a reflection of bad grades may occur.

New development processes as well as coding techniques were also learned. Since I did not know C++ very well the amount of learning I did about the language itself greatly increased. I also learned that despite the fact that you may be working on something completely foreign to another group member, having someone look at your code when stuck is a very useful strategy.

Future Plans

The ultimate goal for this game is to ship it out into the market using Steam. Before this can be accomplished the game must be bug free and have all features mentioned working correctly. After submitting the game to Steam it may not be approved, if it is not the game will be fixed according to what was wrong. If it is approved then Steam will put the game on their online game store.

Appendices

A. A simple Lua Script file:

```
          print("Going through LuaScript!!!");
          average();
print("Passed through average function call to C++ code!");
```

The above code would print "Going through LuaScript!!!" to the console run the function average() in the C++ code and then print "Passed through average function call to C++ code!".

Credits

Ryan Shmitt – Deferred Rendering

Character Controller Set Up – Salome Navarette, Jordan Hand, Alex Hill

AI/Animation – Joshua Marcelo

Lead Graphics/Editor – Chad Wiliiams

PhysX Integration – Tim Biggs, Jon Moorman

Octree Implementation/ Team Manager – Mark Paddon

Art Lead, 2D Artwork – Josh Holland

3D Modeling – Ben Funderberg

3D Modeling – Tom Funderberg

3D Modeling – Mikkel Sandberg

Splash Screen, Game Modes – Hector Zhu

Voice Acting – Mitch Epeneter

Sound Lead – Sam Thorn

References

FBX – technology and file format owned and developed by Autodesk

Collada DOM – A Collada C++ tool for parsing through Collada Xml documents

MD5 Animation – proprietary but still open animation format specific to Doom3

PhysX – Nvidia's physics library.

SDL – Simple Direct Media Layer, windowing system used in applications worldwide

Trine – A Frozenbye side-scrolling action puzzle game

Super Smash Brothers – A Nintendo fighting game

Mass Effect2 – An action role playing game by BioWare

Assassins Creed – An ubisoft action game

Casual Gamer – A gamer that usually plays games on their leisure time with no commitment to the game.

Hardcore Gamer – A gamer that plays very intensly, for long periods of time, and puts lots of commitment into a game.

Windows Directory Reading - Referenced from http://msdn.microsoft.com/en-us/library/aa365200(VS.85).aspx and http://sandip132.blogspot.com/2009/01/c-convert-string-to-wchar-and-vice.html which also references MSDN