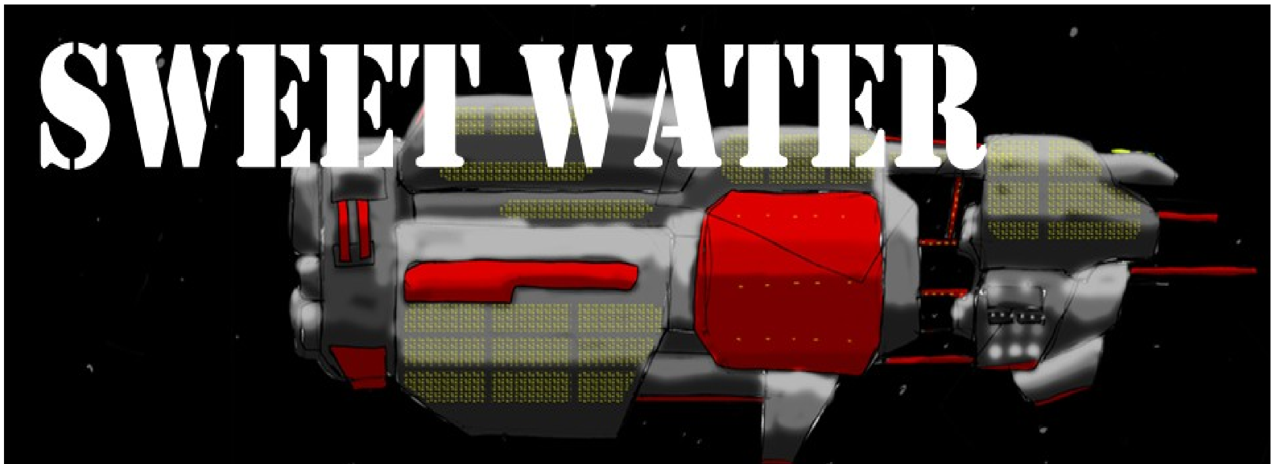
Sweet Water

3D Platform Shooter



By Robert Bernal
California Polytechnic State University
San Luis Obispo
Winter and Spring 2011
Advisor: Professor Zoe J. Wood

Introduction

The role of computer graphics in gaming is essential. As computer processing and graphics processing units improve, the limits of real time rendering are beginning to disappear. The increase in computing power has enabled us to create applications with amazing graphics. Video games have become a big part of human society. In today's world, users expect games to respond to input instantaneously. Players do not want to sit around while their computer frantically draws the next frame. Because games need to be highly interactive, about 30 – 60 frames per second, we need to render complex scenes quickly. Rendering complex scenes presents many challenges, and our papers present solutions to overcome these challenges.

Problem Description:

Creating a beautiful interactive 3D game has many smaller sub problems. We spent the last two quarters focused on finding the answers to these two questions:

1. How do we render a complex scene with many special effects in milliseconds?
2. How do we encode fun into our application using 1's and 0's?

Drawing a complex scene, with thousands and thousands of vertices, in 0.033 seconds sounds impossible by human standards. Computers have evolved to have the capability to do this for us. However, we have to find ways to tell the computer how to do this efficiently. Even though computers have become incredibly fast, they still have limits.

Imagine playing a game and you come across a rock. Sounds boring right? That rock had to go through matrix transformations, lighting calculations, texture mapping, and possibly other special effects. Lighting alone is one of the most physically complex entities in the universe. Trying to compute how light actually works can take hours for large scenes, and it is incredibly difficult to implement.

Since we don't have hours to compute one scene, we have to fake how it light works. Other effects such as physics, shadows, and animations are also computationally expensive. We have to fake some of those simulations in order to compute them in a reasonable amount of time. Complex algorithms are required to fake these simulations.

The other problem we set out to solve was making our game fun. Unfortunately, we did not get to invest as much time into improving game play. Sweet Water's game play is limited because our focus was the graphics technologies in the application.

Previous Work/Related Work:

There have been many great shooters created since the first games came about. The halo series, created by Bungie, for the Xbox and PC has been one of the most successful shooters of all time. Doom 3, created by id Software, won awards for its engine. Shadow Complex, created by Chair Entertainment for the Xbox Live Arcade, is a Metroid style game. All of these games had some influence in the creation of Sweet Water.

Halo Series

Who doesn't like defending the universe against hoards of aliens called the covenant? This first person shooter presents stunning scenery combined with ruthless enemies. The goal is to save the human race with a variety of weapons. Halo uses the

Doom

Considered by many to be the best first person shooter of all time, Doom popularized the first person shooter genre with its amazing graphics and game play. Doom features a space marine who fights demons from hell on Phobos, a moon on Mars.

Shadow Complex

Shadow Complex features great graphics, and the platform based environment. Although only 2.5D, Shadow Complex allows for 3D movement of entities other than the character. Shadow Complex inspired Sweet Water's 2.5D adventure theme. The Unreal Engine 3 powers Shadow Complex.

General Description

Sweet Water is a 2.5D platform shooter. You begin the game as a soldier of war on military prison ship Sweet Water. A mysterious event occurs that gives you the opportunity to escape. Your must fight your way through prison guards to find a way to escape the ship.

Games are complex. They require a large number of components to work correctly in unison. We created Sweet Water using C++. We used OpenGL for rendering, OpenAL for sound, Physx for in game physics, and we made our own tools for level editing, particle effects editing, and resource loading. This section will list the major components in our solution and provide general information about them. All of the tools used for Sweet Water, were created using C++. Sweet Water, and the graphics related tools utilize SDL as a multimedia library. We also use several shaders for complex effects such as per pixel lighting.

Game Engine

- Application Controller: Application initialization and start of the game loop
- Audio System: OpenAL Initialization and sound instances
- Video System: OpenGL initialization and render state
- Event: Inputs
- Logger: Error reporting and debugging
- Resource Manager: Manages textures, sounds, etc...
- Settings: Screen resolution, sound volume, etc...
- Game: Game Logic

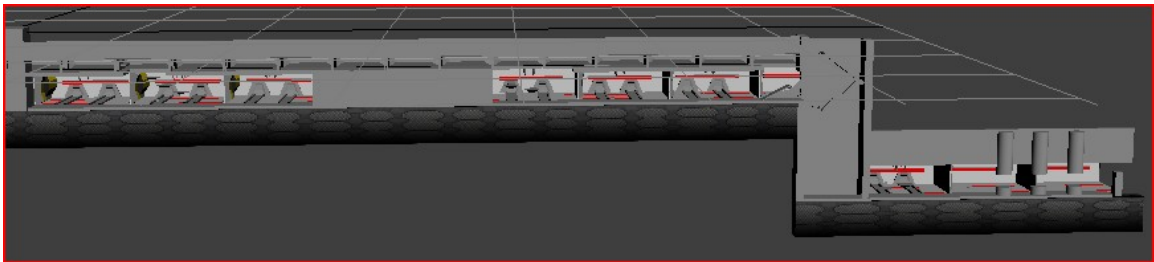
Entities

An entity class represents the player, enemies, and other objects in the game. All of these entities have their own classes, which then derive from the entities class.

Level Editor

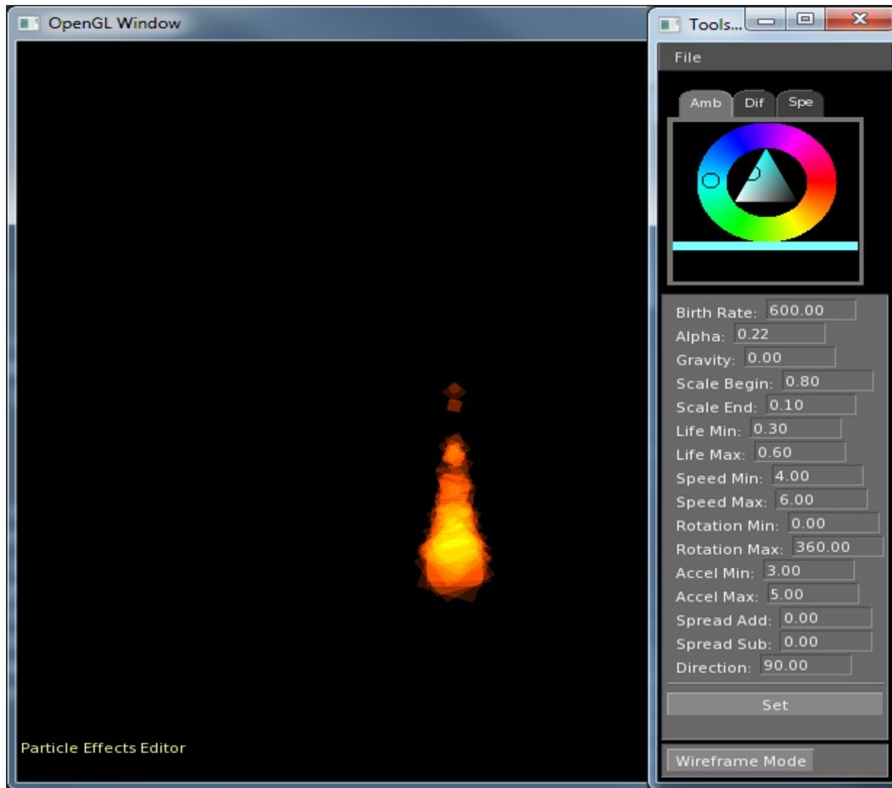
3ds Max plug-ins allowed us to transform 3ds max into a level editor. We wrote and used plug-ins to turn 3ds max into a level editor. The plug-ins were created in C++ using the 3ds Max SDK. The plug-in functionality included:

- Geometry exporter
- Light and camera exporter
- Entity exporter



Particle Editor

We created the particle editor from scratch because we wanted to customize the effects in our game. The particle editor allowed us to create custom effects and then integrate the effects into our game quickly.



External Features Seen By the Player

The following represents other technologies present in our game:

- Animation
- Inventory System
- Shadow Mapping
- Normal Mapping
- Per Pixel Lighting
- Particle Effects
- Physically based character movement and interaction (collision detection)

Algorithms:

Our solution contained many algorithms, but this paper will focus on the animation. If you would like to find more detailed information about the other algorithms used in Sweet Water you can read the papers of my teammates Nick Moresco, Ilya Seletsky, and Steven Udall.

Character Animation

There are several ways to do character animation. Procedurally generating movement is best suited for movements that can be easily described using mathematic equations. Another way is to have an artist create the animations from scratch using a model and some software such as 3ds max. We used 3ds max to modify existing animations for our character movement. We then applied an algorithm to combine these animations. We wanted to avoid spending a large amount of time combining animations for the game through modeling software.

Set Up

In order to use the md5 format, we wrote an md5 importer that reads md5anim files. The importer retrieves the model and animation data for use in the game. The importer uses common parsing techniques to obtain the information and store it in the associative data structures. The number of data structures required to accomplish this becomes numerous. Each data structure builds off the lower level structures to create one structure that contains all the data.

Data Structures

Md5 animations define movement of a character using key frames. Each key frame contains offsets from a single base frame. The offsets of each key frame are contained in the joints defined for the character using a modeling software such as 3ds Max. The joints have a position and orientation, and we can manipulate these values to obtain the desired look. Every manipulated bone contains a non-zero flag value.

The data structures we used were stripped down from other solutions we encountered in our research. We did not need some of the data present in md5 files, so we used lighter structures. Using lighter structures saved memory, and computation time in our application. I introduce these structures in order to provide some information required to explain other parts of the animation process.

Animation

```
struct Anim {
   int numFrame;
   int numAnimatedComponents;
   std::vector<Frame> frames;
   std::vector<Joint> joints;
   std::vector<JointInfo> jointInfo;
};
```

The Animation structure holds all of the necessary information from an md5anim file. It contains a list of frames, joints, and information about the animation itself. Once this structure filled, you have all the data you need to start basic animations.

Frame

```
struct Frame {
   std::vector<float> animatedComponents;
   std::vector<Joint> joints;
};
```

The frame structure holds information about the number of components that are animated, and which components are animated. The frame lists only the joints that differ in position and/or orientation from the base frame.

Joint

```
struct Joint {
   NxReal pos[3];
   NxQuat quat;
   int flags;
   bool prioritized;
};
```

The joint structure holds the position and orientation of the bone. The NxReal and NxQuat fields are the Vector and quaternion used with Physx. The joint also holds flags that reveal which components have been modified. The prioritized variable allows for the mixing of animations without changing the md5 file format. Prioritizing the animations will be discussed when we go over the animate algorithm.

Weight

```
struct Weight {
   int joint;
   NxReal weight;
   NxReal pos[3];
};
```

The weights describe how a vertex depends on a bone. More weights can increase the accuracy of rendering an animation. Each vertex can have more than one weight.

Vertex

```
struct Vertex {
   int weightIndex;
   int weightCount;
   NxReal pos[3];
   NxReal textCor[2];
   NxReal normal[3];
};
```

The vertex structure contains fields you would typically find in any vertex container. The outstanding fields in this application are weightIndex and weightCount. The weightIndex retrieves the corresponding weight to vertex from the list of weights, and the weight count is how many weights a vertex depends on.

Animation Algorithm Details

Data Structures

```
struct AnimPer
{
   bool m_loop;
   bool m_running;
   int m_priority;
   int m_curFrame;
   int m_animNum;
};
```

The AnimPer structure separates animation data between instances of characters. We do this to save memory usage. Characters with the same model share model data, but an instance structure is required to keep every model running animations separately.

- m_loop – Determines if the animation repeats after it ends.
- m_running – Set to true as long as the animation needs to run. Determines whether the animation displays in the current frame.
- m_priority – Level of priority for this animation. The application currently has five levels of priority ranging from 0 to 4. An animation's prioritized bones will draw over an animation with a lower priority.
- m_curFrame – The current frame of the animation. The animating algorithm builds an interpolation frame using this frame.
- m_animNum – Index into the list of animations an entity has available to it.

Algorithm

The algorithm responsible for animating a character breaks down into four different steps. This solution allows for modularity and the ability to break down the code into

multiple functions that can be tested separately. The animate function has the following signature:

```
void animate(float dt){}
```

The only parameter to the animate function is the change in time since the last frame. The change in time allows the function to run independently of the frame rate of the application. A great deal of initialization is required per frame before animation can take place. After the animate, but before drawing the character, the character's vertices and normals are built. The following portions of the application all contain animation related code.

1. Update the character.
2. Set animations to Running or Not Running.
3. Call animate function.
4. Build the vertices and normals before rendering.

Updating the Character

The state of the character updates from the previous frame. Anything that has happened since the last frame is reflected in the character's state. For example, if a user presses the move left key then the character's position should be updated and the animation changes.

Set Animations to Running or Not Running

The animations that the character displays are set to running or not running based on the user input. If the user presses the fire button, the firing animation is set to run. When the fire animation ends, the fire animation is set to not running. Distinguishing running from not running animations allows multiple animations to run concurrently.

Animate

The animate function itself can also be broken up into several distinguishable steps.

1. Determine what animations need to advance or end based on the change in time.
2. Sort the animations based on priority.
3. Set up interpolation frame.
4. Build interpolation frame by adding prioritized joints.

Animation Advancement

The running animations will advance if enough time has passed. Our solution supported using the change in time from the last call, or it could combine the time with a framerate set by the md5 file. The algorithm needs to check which animations currently run, and it should advance the frame of the animation if necessary. The following pseudocode gives one example of how we did this.

```
animationTime += change in time

for every animation
	set tempTime to animationTime
	if animation is running
		if tempTime > 1.0
			while(tempTime > 1.0)
```

```
                tempTime -= 1.0
            nextFrame = curAnim’s current frame + 1 || 0
        if curAnim’s current frame is the last frame in animation
            set curAnim’s current frame to the beginning of animation
            set curAnim to not running
        push the animation to the end of the running vector
```

The algorithm should be adapted to fit the needs of your particular application. A data structure that can easily sort should be used. At the time of writing this portion of the code, correctness was more important than speed. The algorithm can be optimized based on the requirements of the application.

Sorting Animations

Sorting the animations based on priority can make further computation easier. Because the number of animations running at one time is limited for most character, any basic sort will suffice. Even if a character runs 10 animations at a time (for our application the most we use is three), the computation time is trivial.

Interpolation set up

The interpolation frame needs to have room for all the joints in the model. If you accidentally overlook this step, as I found out during coding, you will have strange behavior and crashes in your application. A simple if statement will prevent any surprises relating to the size of the interpolation frame.

```
If interpolation frame’s joints != number of joints in the model
    Resize interpolation frame’s joints to be the number of joints in the model
```

Build Interpolation Frame Joints

In order to interpolate the frame, we need to decide what the positions and orientations of the joints. A joint is added into the interpolation frame based on its priority in the current animation. The priority scheme behaves the same as a priority queue. Joints are added in order of the priority of the animation. The algorithm adds joints from lowest to highest priority so that only one pass is required. The following pseudocode presents a general solution.

```
If there are running animations
       for every joint in the lowest priority animation
              set interp frame’s joint to running animations running frame joint
       for every other running animation
              for every joint in current animation
                     if the joint is prioritized
                     set interp frame’s joint to current animation running frame
                     joint
```

Describe solution into overall solution of project

Even though we have all this data that describes how the character moves, we still need a way to get this information onto the screen. The characters vertices are built before calling the draw methods that every entity has.

Building the Vertices

Rendering a character’s triangle mesh from md5 format requires building the vertices before rendering. As stated before, a number of weights are associated with each vertex. Md5 gives you the capability of having multiple meshes in order to split your model into multiple components. Splitting your model could reduce computation if you have some part of the model only needs to display part of the time. Some of the meshes may be used

for special effects therefore do not need to be visible all the time. By giving the mesh's internal representation a boolean variable, we can control the visibility of the mesh.

The following pseudocode describes a general solution.

```
For every mesh in the model
	If mesh is visible
		For every vertex in mesh
			For every weight per vertex
				Get weight from vertex weight index
				Get joint from current running frame and weight's joint
				index
				Build quaternion q with weight
				Build quaternion res from q * joint.quat *
				inverse.joint.quat
				vert += joint.pos + res * weight
```

After building the vertices, they may need scaling. We scaled the vertices by a small factor in order to scale the models to game size. One might also scale the vertices using a graphics api function call. In the latter case you would need to isolate your drawing of the character meshes from everything else.

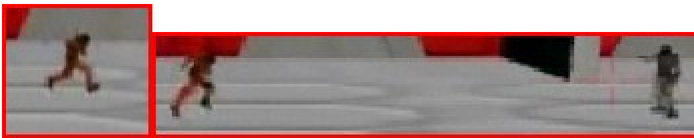
Building the normals

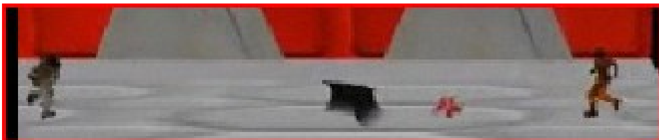
The lighting model you use will dictate how to build the normals of the entity. The normals can be built depending on what lighting model you are using. In any case, building the normals requires computed vertices. Our solution creates the normals per vertex. The normal for each vertex is summed from each contributing face. The time needed to compute the normals can be reduced by culling out any unseen meshes.

Animation Results

The following pictures portray the animations before they could interpolate with each other.

The figure on the left shows the character firing in a normal mapped environment. The figure on the right shows the character firing without the normal maps.

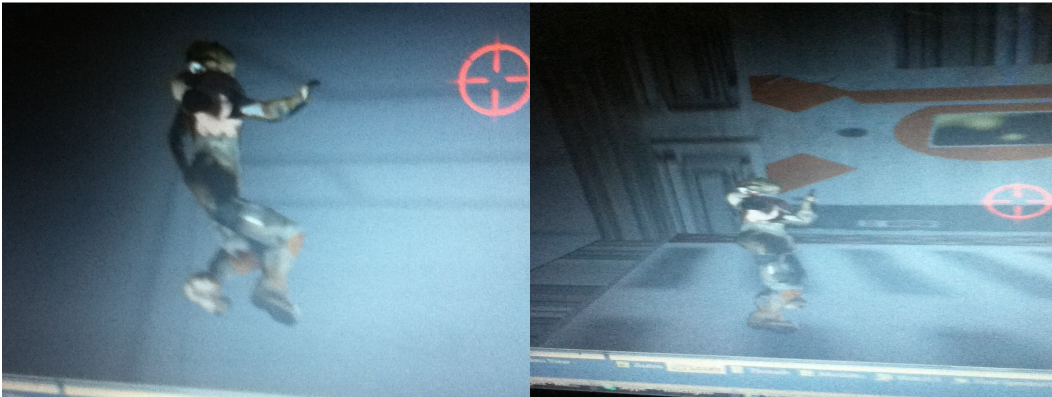




These figures show the character in different stages of movement including moving backwards.

The figure on the left shows the character jumping down an elevator shaft. The figure on the right shows a character jumping in the opposite direction.

The following pictures portray the animations after interpolation was possible.

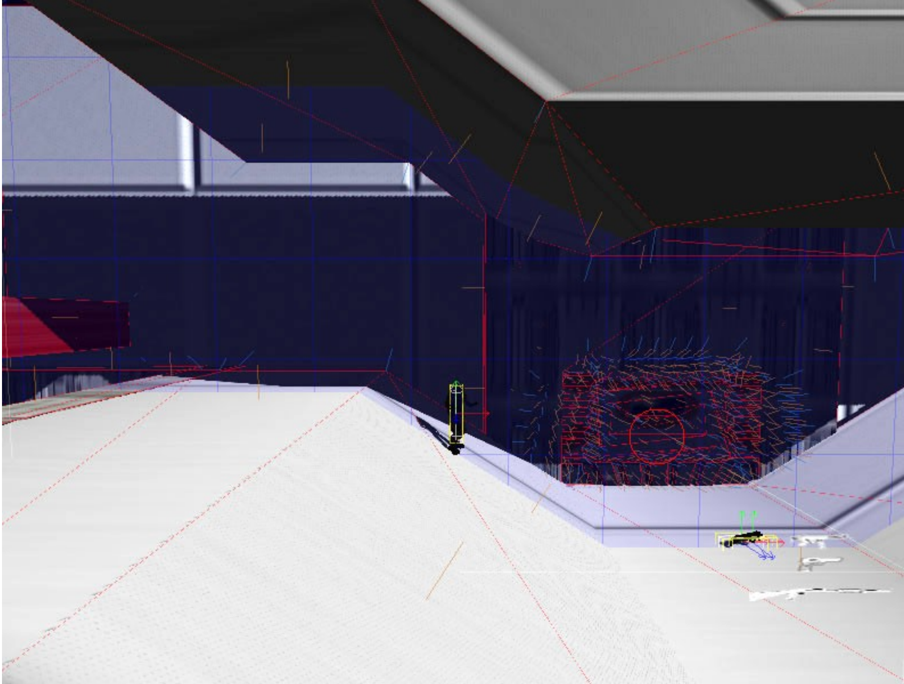


The figures above show the character performing multiple actions at the same time. They also show new animations that were implemented during the second quarter.

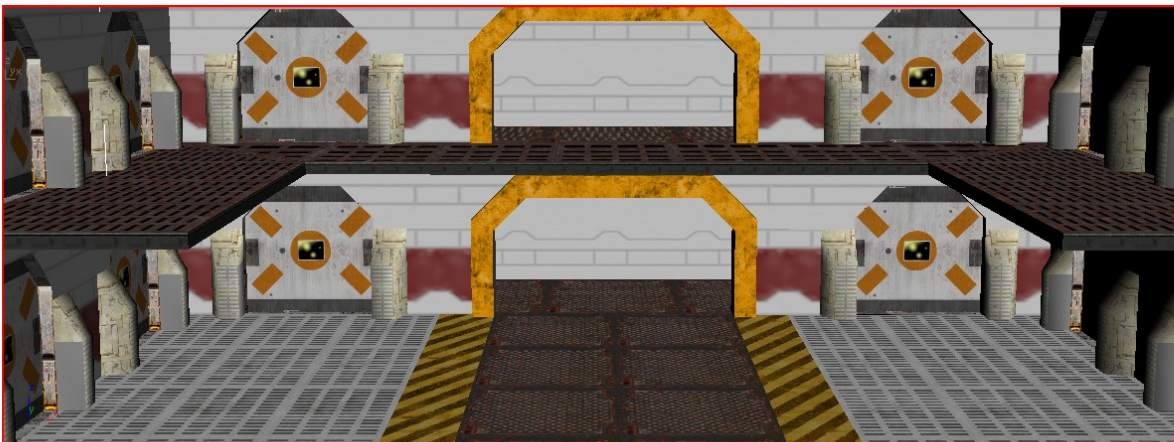
Results:

A fully playable platform shooter resulted from the last two quarters. Sweet Water has gone through play testing, and it received positive reviews. We reached our goal of

creating a fun highly graphic intensive real time application. Below are some pictures of our efforts.

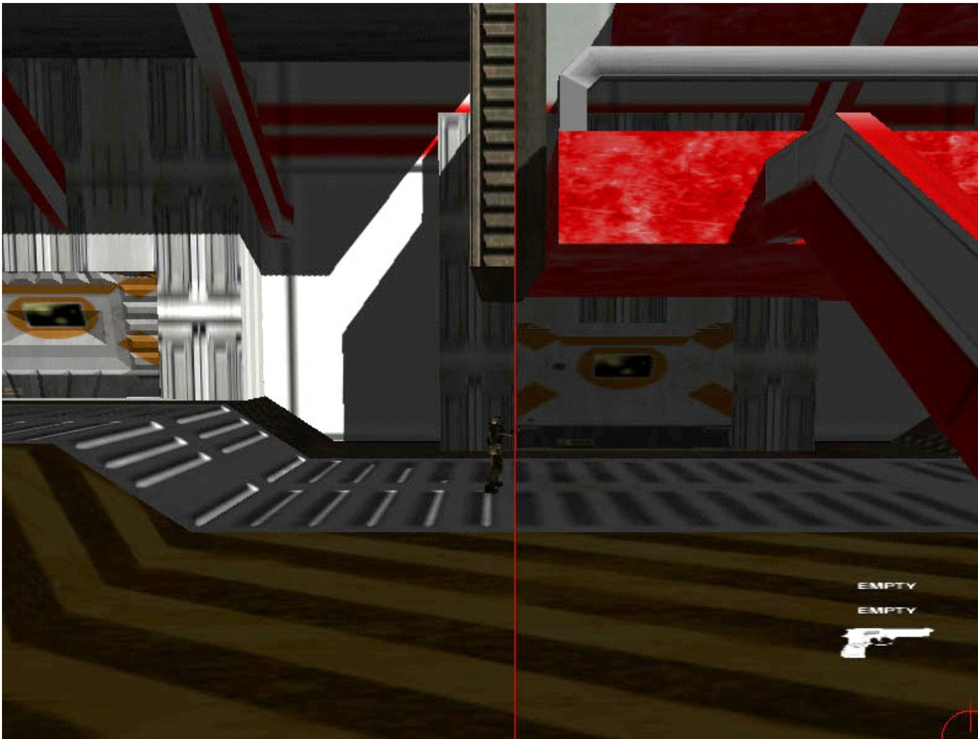


Debug mode showing normals, physics objects, and collision grid.



Geometry in level 2 created using 3ds Max.

Fire particle effects and shadows in game.



Shadows shift with character movement.

Future Work:

Due to our time constraints and small team, we did incorporate some features that would enhance the game play. We strived to add more weapons, and better firing effects. We would've like to add more weapons, and better firing effects. We also wanted to add Screen Space Ambient Occlusion, but we ran out of time.

Today, many games are released and constantly improved upon. We would have liked to optimize some of the computational intensive algorithms we used. For example, moving the building of character vertices per frame from the cpu to the gpu using a shader. There's always room for improvement, and maintaining a video game can be just as fun as making it.

Conclusion:

Though we ran into many bumps during the last two quarters, we ended up with a playable game. We have two nearly complete levels, basic game mechanics, and many graphics technologies. Creating a game in two quarters was a great learning experience. A good team can improve the experience. Having a good team helped make my experience enjoyable.

Though I enjoyed the experience, there were a few things I would have done differently with a second chance. Though we did some initial design, the majority of design was

finished during coding. Some more initial design may have reduced the time we spent redesigning and reworking our code. In addition, I would have listed all the features we wanted to implement at the beginning. This would have provided a solid framework for development throughout the two quarters.

Overall, the project was a success. We have a working playable game.

References:

http://tfc.duke.free.fr/coding/md5-specs-en.html

http://en.wikipedia.org/wiki/Doom_(video_game)

http://youtube.com (3ds max tutorials)

http://usa.autodesk.com

3ds Max SDK Documentation