Sweet Water

Implementation of a 3D Platform Shooter

Nick Moresco

California State Polytechnic University

San Luis Obispo

Winter & Spring 2011

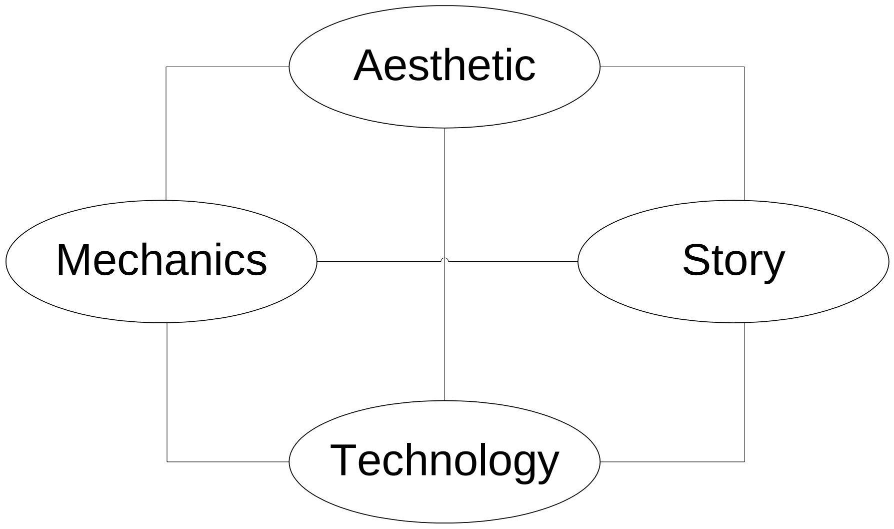
Advisor: Professor Zoe J. Wood

Introduction

Computer graphics is an area in computing that is advancing rapidly. This is largely in thanks to the explosion of the gaming industry. The huge popularity of video games is one of the largest driving forces of computer graphics. The role of computer graphics in gaming is essential. In non real time graphics applications, we have the luxury of having few time constraints. In a video game, we need to render complex scenes quickly because a game is highly interactive. The need to create highly detailed and realistic scenes in a very short time raises a lot of complex technical challenges. Shooter games, in particular, require a high level of interactivity because of the sheer amount of actions all happening at once. Users expect a game to respond to their input instantaneously. This paper will highlight some of the technical challenges of creating a 3D game and our methods of solving them.

Problem Description:

Our goal was to create a working game that not only had beautiful graphics, but was also fun to play. To that end, we had to focus on a number of game design elements.



A tetrad of related game design elements as seen in *The Art of Game Design: A Book on Lenses*

We had to keep each of these four aspects in mind while designing our game. While the primary focus of the project was on the technology side, each of these aspects is essential in creating an effective game. Without an aesthetically pleasing game, all the complex technologies we endeavored to create are lost. If there is no story or setting in the game, the player has little motivation to play our game. If the game's mechanics are too complicated or too simple, the player won't enjoy playing the game. Not only did we need to create a piece of software that rendered complex scenes quickly, we also needed to incorporate the "fun factor" into the game play. Nobody wants to make a game that people won't play. The problem or question is twofold.

1. How do we render a complex 3d scene with many special effects in a short period of time?
2. How do we encode fun into our application using 1's and 0's.

Motivation:

Video games are becoming a bigger and bigger part of human society. Many graphics technologies that exist today have been spurred on by the desire to create the next generation of games. Games require many advanced technologies that mesh together to create a specific experience. Discovering what it takes to create a game from start to finish was a great learning experience.

Previous Work/Related Work:

There are a number of games with similar elements to our proposed game. The game play mechanics and graphical styles of Halo, Doom, and Shadow Complex all influenced the creation of Sweet Water.

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 innovative first person perspective. Doom was created by Id Software features a space marine that fights demons from hell on Phobos, a moon on Mars. The latest game in the series, Doom 3, is the inspiration behind many of our design decisions within the game engine including our decision to use the .md5 model format pioneered by Id Software.

Halo

Created by Microsoft for the Xbox and PC, this first person shooter series garnered massive popularity when it was first released in 2001. You play a super soldier who leads an army to save the human race from an alien swarm with a variety of weapons. Its popularity can be attributed to its stunning scenery, ruthless AI, and polished gameplay.

Shadow Complex

Shadow Complex is a game that was released on the Xbox Live Arcade in 2010. Created by a small team under Epic Games, Shadow Complex features great graphics, and a platform based environment. Much like our game, Shadow Complex is restricted to two dimensions while still featuring fully 3D scenes. This game was responsible for many of our gameplay decisions and also drove our game’s desired graphical style.

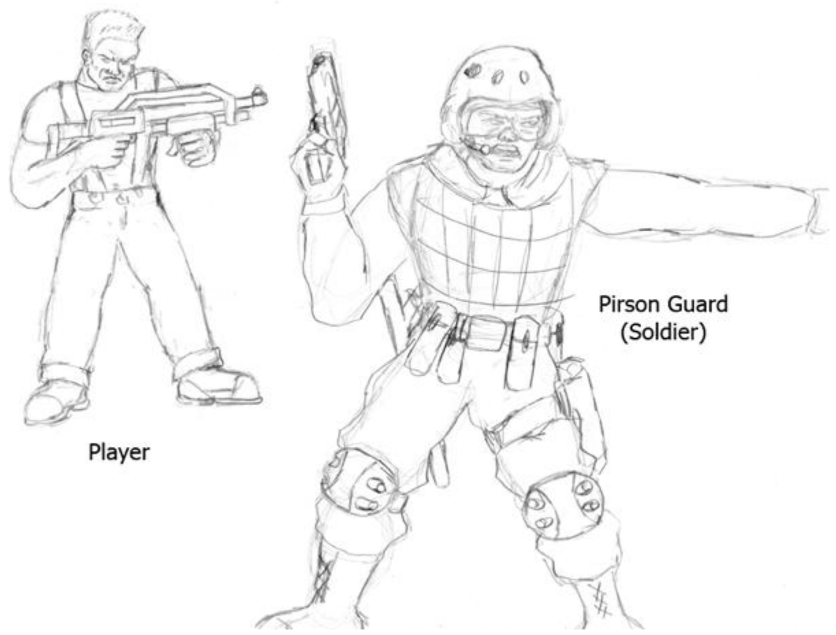
Project Overview

Story

Sweet Water is a 2d platform shooter with 3d elements. The game is set in the far future. The name Sweet Water refers to the moniker ironically given to an infamous prison ship floating out in deep space. All the worlds' toughest criminals are sent there to be isolated from society. When an unknown force causes a disruption on the ship, your character seizes his chance for escape. The objective of the game is simple enough: escape Sweet Water by fighting through the prison guards with whatever weapons you can find.

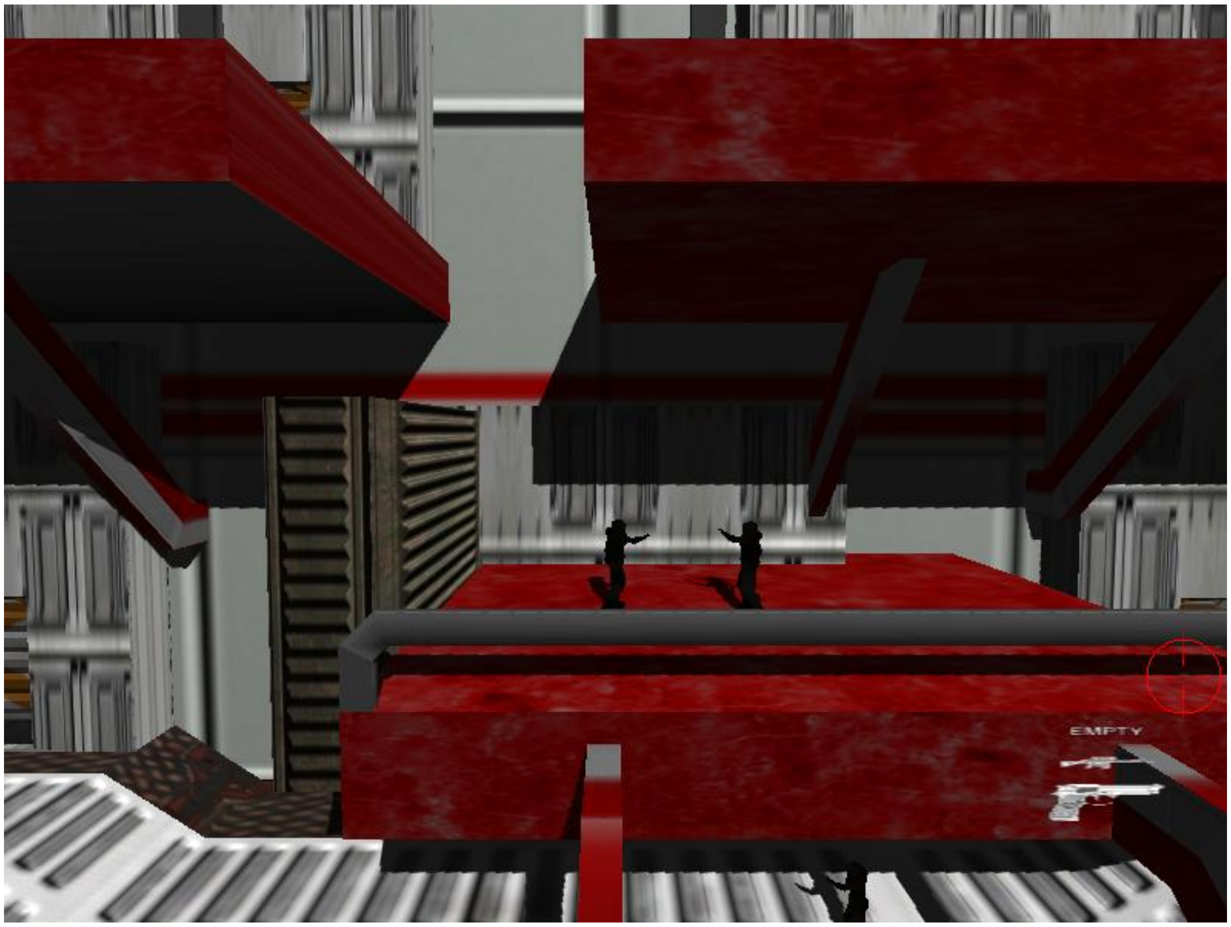
Visual Style

Our graphical style is influenced by games like Shadow Complex and Mirror's Edge. The characters in the game are humans who have been living on a prison ship for a number of years. Our initial concept art portrayed them as very tough and battle-hardened.



Initial Concept Art Drawn by Group Member Ilya Seletsky

We wanted a clean industrial look with lots of cool neutral grays. To offset this neutral color scheme, important elements of the word are very bright primary colors.



The bright red balconies guide the player to jump on them.

Mechanics

The emphasis of the gameplay is on strategic combat, ammo management, and exploration. We didn't want the player to run blindly through the level constantly firing in random directions. We wanted each action by the player to be deliberate. In order to accomplish this, we decided on a number of gameplay decisions early on in the development process.

The player starts the game out with a weak pistol. As the player defeats enemies, he is able to pick up their weapons, enabling access to 3 more powerful weapons. Each weapon has its own strengths and weaknesses associated with it. The shotgun fires multiple low damage pellets at the enemy. It is very powerful from close range, but weak in long range situations. The sub machine gun fires very quickly and does a lot of damage, but is very inaccurate. The rifle is very accurate and does a large amount of damage, but fires and reloads very slowly. It is best used in long range situations.

One of our goals with the game was to encourage the player to use each weapon as the situation requires. We found in play testing that players would often choose their favorite weapon and stick with it the rest of the game. In order to force the player to switch weapons, we made the ammo for each weapon scarce. The player has more than enough ammo to finish the level between all of his weapons, but one weapon does not have enough ammo to defeat all the enemies in the level by itself.

Our first level design consisted of mostly straight corridors with a few platforms that lead to raised areas. Almost immediately, we found that players would just run forward and shoot in a straight line, killing enemies before they even had a chance to appear on screen. Our later level design featured much more dynamic levels with multiple paths and forced the player to move in all four directions (up, down, left, right) in order to complete the level. This made the player much more cautious in approaching each new area as enemies could be in more unpredictable locations. This also helped us develop the exploration aspect of our gameplay as the player searches for the correct route through the ship.

Technical Details

Games are complex and require a great deal of components to work correctly. Sweet Water was created using C++. We used OpenGL for rendering, SDL for our windowing libraries, FMODex libraries for sound, and Nvidia's PhysX libraries for in-game physics. To support our game development, 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 a little information about each.

Game Engine

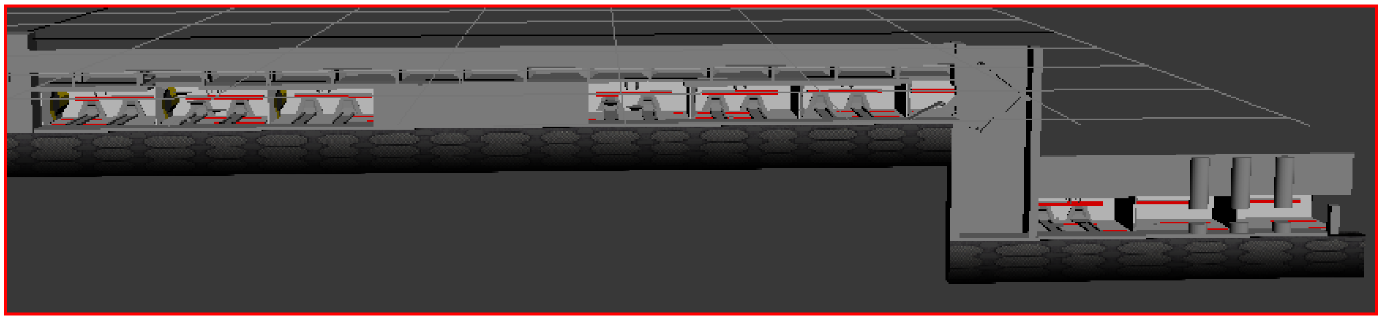
The main components of our engine are:

- Application Controller: Application initialization and start of the game loop
- Audio System: FMODex Initialization and sound instances
- Video System: OpenGL initialization and render state manager
- Event: Inputs
- Logger: Error reporting and debugging
- Resource Manager: Manages textures, sounds, etc...
- Settings: Screen resolution, sound volume, etc...
- Game: Whatever the high level game code might be

Level Editor

Since we had such a specific setting for our game, we needed a level with a lot of geometry to convey the setting properly. Instead of writing our own level editor, we chose to use 3ds Max to create the levels for our game world. We wrote our own plug-in that allowed us to transform 3ds max into a level editor that exported into our own simple format we could use to load the level directly into our game. This allowed us to harness the full ability of 3ds Max to create our levels instead of devoting time to creating our own level editor with reduced functionality. The plug-in functionality allowed us to export the following:

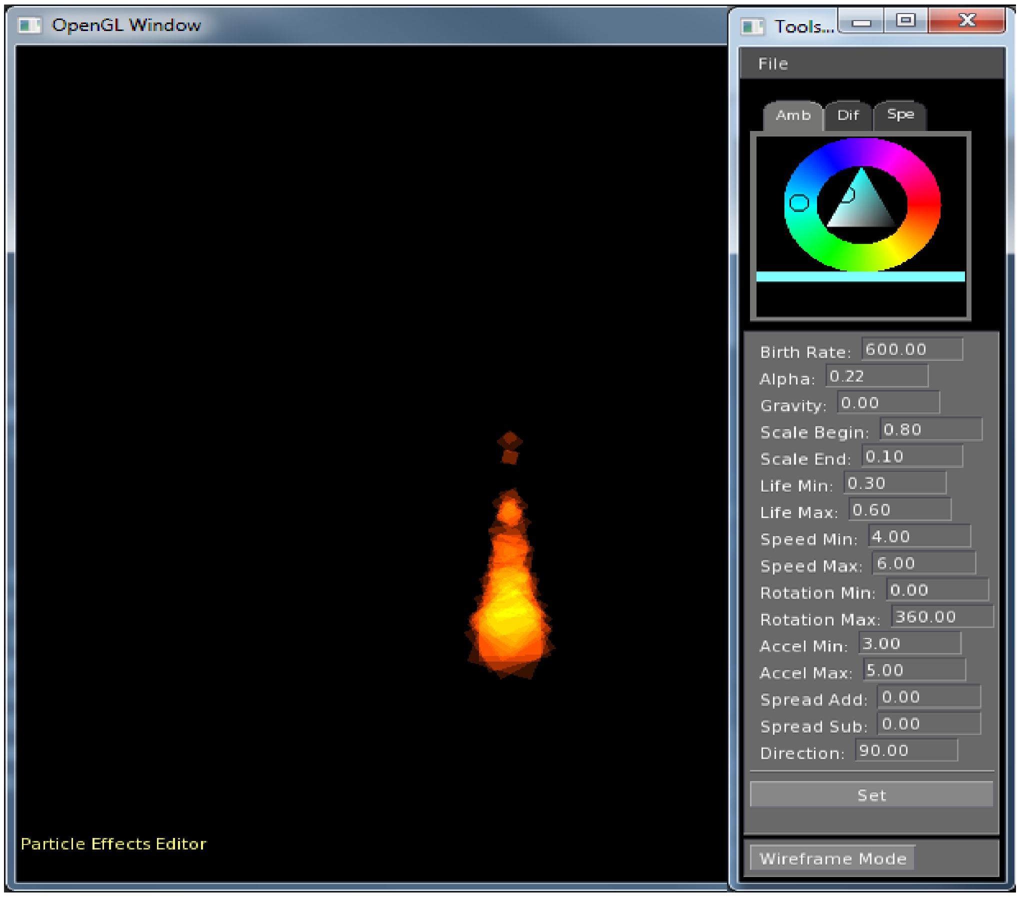
- Level geometry
- *uv* texture coordinates
- Entity locations (the player, enemies, particle emitters, etc)
- Light locations
- Camera locations.



A view of one of our levels from a distance

Particle Editor

One of the first things we decided this game needed was a robust particle system. Particle systems would allow us to model effects like fire, blood, or the sparks of bullets hitting metal. In order to create these effects, we set about making an editor for particles that could export directly into our game. This way, we could edit particle effects in real time and iterate on the settings until we found the effect we were going for. Our particles are texture-mapped 2d sprites. We implemented bill-boarding on the particles to ensure they look right regardless of camera orientation.



Birth Rate:	600.00
Alpha:	0.22
Gravity:	0.00
Scale Begin:	0.80
Scale End:	0.10
Life Min:	0.30
Life Max:	0.60
Speed Min:	4.00
Speed Max:	6.00
Rotation Min:	0.00
Rotation Max:	360.00
Accel Min:	3.00
Accel Max:	5.00
Spread Add:	0.00
Spread Sub:	0.00
Direction:	90.00

The Particle Editor in Action

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

Algorithms:

This paper's primary focus will be on the implementation of Per-Pixel lighting, Normal Mapping, and Shadow Mapping using GLSL shaders. 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 Robert Bernal, Ilya Seletsky, and Steven Udall.

Shaders in OpenGL

In OpenGL, rendering is done in a series of discrete steps. This is known as the graphics pipeline. Shaders allow you to take control of a specific step in the graphics pipeline and insert your own code in its place. Originally, this code had to be written in assembly but more recent video cards support specific shader languages. Shaders have become extremely popular in graphics applications because they allow you to create effects that would otherwise be impossible in the normal OpenGL pipeline. All of the shaders in this paper have been written in GLSL because of it similarity to C and compatibility within OpenGL.

Vertex Shader

The vertex shader takes control of the vertex processing stage of the graphics pipeline. The code in a vertex shader is executed per vertex. Since you are taking control of the graphics pipeline, you have to handle all transformations, lighting, texture mapping, etc that OpenGL normally does automatically. A trivial vertex shader that does the same thing as the graphics pipeline would be:

```
void main() {
  //transform the vertex to clip space
  gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

GLSL gives you access to a number of things from OpenGL to simplify writing your code. For example, gl_Vertex represents the vertex's current location while gl_Position represents the final position of the vertex after it is transformed. gl_ModelViewProjectionMatrix, unsurprisingly, represents the model view/projection matrix.

Fragment Shader

The fragment shader operates on every pixel sent to the shader after rasterization. Like the vertex shader, anything that OpenGL does automatically for you must be handled manually in the shader. A simple shader that just colors a pixel based on an object's material properties is outlined below.

```
void main() {
        gl_FragColor = gl_Color;
}
```

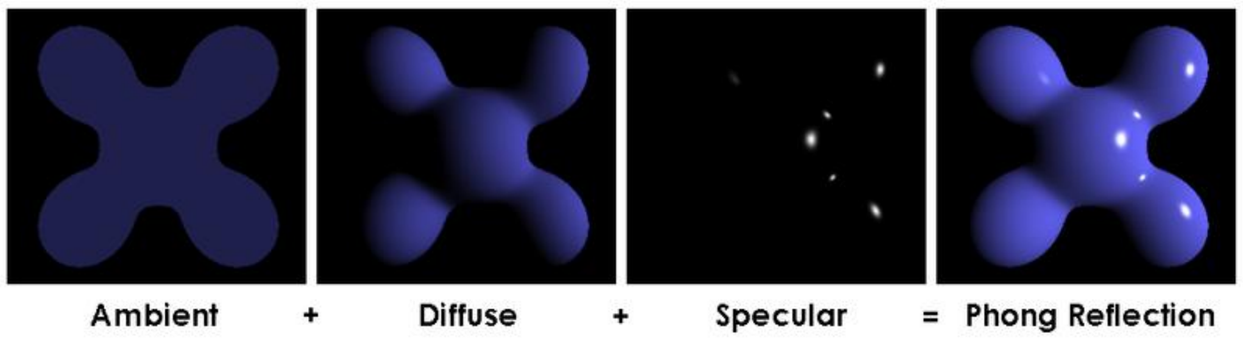
In this case, gl_FragColor represents the final color of the pixel and gl_Color is the rgb value specified by glMaterial in OpenGL.

Per-Pixel Lighting

In the OpenGL fixed-function graphics pipeline, all lighting computations are done per-vertex. Each pixel is lit based on an interpolation across these vertices. This was an unacceptable lighting solution for us because much of our geometry is very planar. The vertices on an object were often far away from each other. This would result in a very flat shaded world with few specular highlights because of the small number of vertices in our level. The solution to this is to implement a per-pixel lighting algorithm with a shader. Instead of calculating the lighting at each vertex and interpolating the value for each pixel, we calculate the lighting equation for all pixels. This is more expensive to compute, but since it is done in a shader, the graphics hardware does most of the heavy lifting and the performance hit to our game was negligible.

Modeling the Shading Equations

In order to approximate lighting in our scene, we implemented the commonly-used Phong model. The phong model breaks up lighting into three different components: ambient, diffuse, and specular.



Taken from the Wikipedia Commons from the Article on Phong Shading

Diffuse represents the light reflected off a dull or matte surface, specular represents the light reflected off a shiny surface, and ambient is an approximation of global lighting to prevent areas that are not hit directly by light from appearing pure black. Most objects have a combination of diffuse, ambient, and specular lighting components. Note that while ambient and diffuse components look the same no matter what angle the camera is at, the specular lighting changes depending on the orientation of the camera. According to the Phong model, the color of a pixel at I_p is:

$$I_p = ambient + (diffuse * (N \cdot L)) + \ (specular * (R \ \cdot V)^{\alpha})$$

Where:

N = normal of the pixel

L = light vector

V = viewing vector

R = reflected vector

α = shininess component

Since the reflected vector R is difficult to compute, a common optimization is to use the half-angle approximation. This gives a vector that is similar to the reflected vector without computing an additional dot product. The equation is as follows:

$$H = \frac{L + V}{2}$$

Substituting R for H gives good results and speeds up the calculation of this equation. Since we need to do it for each pixel, this is very important.

Implementation

All that is necessary now is to implement this equation using a shader. Luckily this is very straight forward thanks to GLSL. The vertex shader looks like this:

```
varying vec3 normal;
varying vec3 view;
varying vec3 vertex_to_light;

void main() {

        // transform the vertex into world space
        vec4 pos = gl_ModelViewMatrix * gl_Vertex;

        // The normal matrix represents the inverse transpose model-view matrix.
        normal = gl_NormalMatrix * gl_Normal;

        // Calculate the viewing vector
        vec3 view = vec3(gl_ModelViewMatrix * gl_Vertex);

        // Calculate the light vector
        vertex_to_light = gl_LightSource[0].position.xyz – pos;

        // transform the vertex to clip space
        gl_Position = gl_ModelViewProjectionMatrix * gl_Vertex;
}
```

In this example, the label “varying” means that the variables *normal* and *vertex_to_light* are calculated per vertex and interpolated per pixel before being passed to the fragment shader. The actual code here just calculates the transformations necessary to pass the relevant data to the fragment shader. The light’s vector is taken from OpenGL’s own light system that was specified before the shader code. The fragment shader is where all the actual shading will happen. It looks like this:

```
varying vec3 normal;
varying vec3 vertex_to_light;

uniform sampler2D diffuseMap;

void main() {

        // Get the base texture color
        vec4 base = texture2D(diffuseMap, texCoord);

        // get the ambient contribution from the light
        vec4 ambient = gl_LightSource[0].ambient;

        // normalize the interpolated normal and light
        vec3 normalVec = normalize(normal);
        vec3 lightVec = normalize(vertex_to_light);

        // calculate the diffuse contribution
        float diffuse = gl_LightSource[0].diffuse * clamp(dot(lightVec, normalVec), 0.0, 1.0);

        // calculate the half-angle
        vec3 h = (lightVec + view) / 2.0;
        h = normalize(h);

        // calculate the specular contribution
        float specular = gl_LightSource[0].specular * max(dot(normalVec, h), 0.0);
        specular = pow(specular, 10.0);

        // combine terms to make color
        gl_FragColor = diffuse * base + specular + ambient * base;
}
```

The fragment shader computes the lighting for the current pixel according to the Phong equation given above. The uniform sampler2D variable *diffuseMap* is a texture that is passed into the shader to represent the base color of the object. Note that for this implementation, we assume that the specular component of all objects is 1 and the shininess is hard coded to 10. Instead of hard coding these values, we could have passed them to the shader in a specular map and a gloss map to control the values of these parameters. However, because of time limitations, we decided to hard code these values. Since most surfaces in our game are metal, we felt this was a good compromise in the face of creating specular and gloss maps for each material in our game.

Normal Mapping

In order to have realistic visuals in our game, we need to show lots of detail in our objects. Unfortunately, lots of detail means highly complex geometry that is expensive to compute. One technique for lessening this geometry is called normal mapping. In normal mapping, an additional texture is passed to the shader that specifies the normals of an object. This texture is called a normal map. This additional texture data is used to shade the object instead of the normals supplied by the geometry.

Diffuse Map
+
Normal Map

If we model an extremely high polygon model, we can export a normal map from that model. We can then use a low poly version of the same model combined with the normal map to shade the object. If we use very simple geometry that has complicated normals stored its normal map, we can give the illusion of a complex 3d object with lots of details when in reality the object is very low poly and easy to compute.

Implementation

A normal map's data is expressed in tangent space. In order to use this data we must transform all our light and camera vectors into tangent space. The matrix to do this is known as a TBN matrix and is as follows:

$$\begin{bmatrix} normal.x & normal.y & normal.z \\ biNormal.x & biNormal.y & biNormal.z \\ tangent.x & tangent.y & tangent.z \end{bmatrix}$$

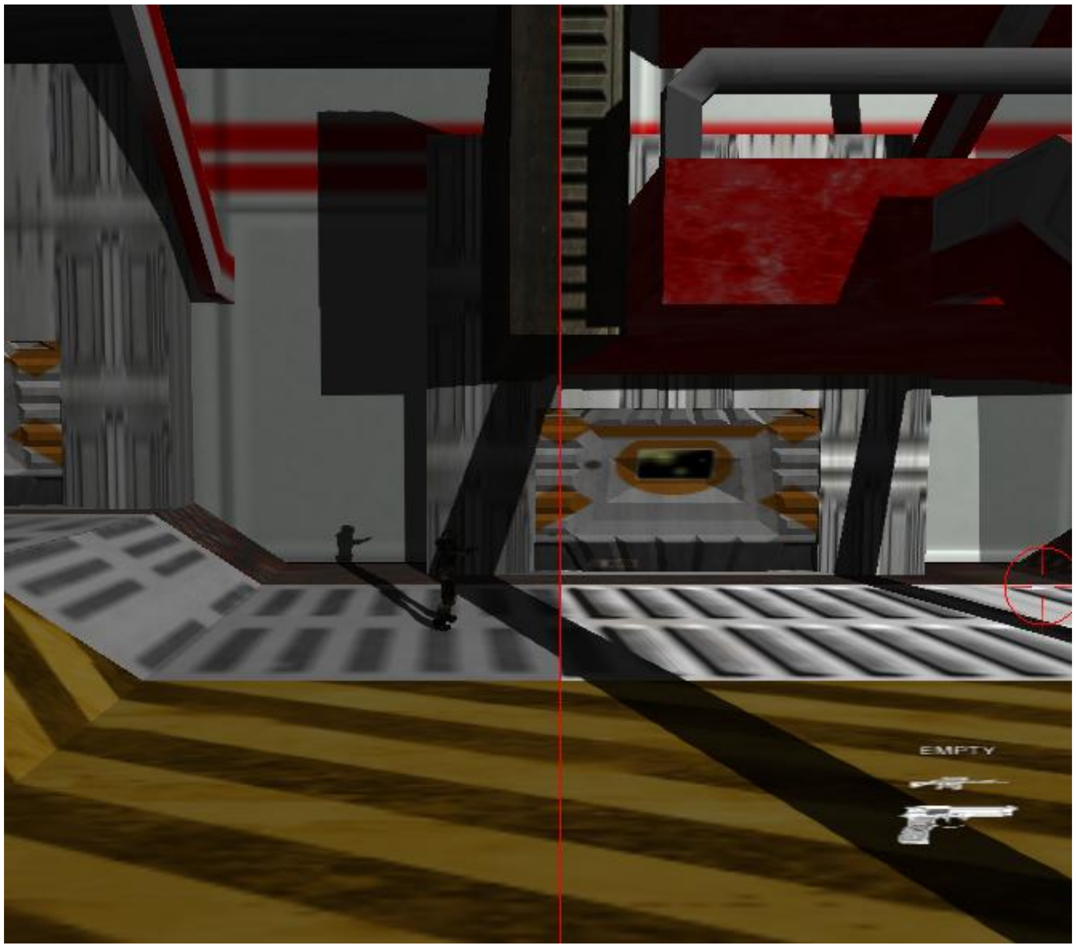
In GLSL, we are given the normal vector and pas in the tangent vector. You must transform both of them by the inverse transpose of the model-view matrix (just like we did for the normals in the per-pixel shader). To get the biNormal vector, take the cross product of the two vectors. The Vertex shader code looks like this:

```
varying vec3 lightTangent;
varying vec3 eyeVec;
varying vec2 texCoord;

attribute vec3 vTangent;

void main() {

        // Calculate the TBN matrix
        vec3 n = normalize(gl_NormalMatrix * gl_Normal);
        vec3 t = normalize(gl_NormalMatrix * vTangent);
        vec3 b = cross(n, t);

        …

        // Transform the viewing vector
        eyeVec.x = dot(view, t);
        eyeVec.y = dot(view, b);
        eyeVec.z = dot(view, n);

        // Transform the light vector
        lightTangent.x = dot(vertex_to_light, t);
        lightTangent.y = dot(vertex_to_light, b);
        lightTangent.z = dot(vertex_to_light, n);
        …
}
```

As previously stated, vTangent is passed in from OpenGL. The view and light vectors are calculated as before. The only difference is that they are transformed by the TBN matrix before being passed to the fragment shader. The fragment shader also looks very similar to the per-pixel shader. The only real difference is you get the normal from a texture lookup instead of an interpolated vertex.

```
varying vec3 lightTangent;
varying vec3 eyeVec;
varying vec2 texCoord;

uniform sampler2D diffuseMap;
uniform sampler2D normalMap;

void main () {

        // Normalize incoming vectors
        vec3 lVec = normalize(lightTangent);
        vec3 vVec = normalize(eyeVec);

        // Get normal from texture lookup
        vec3 normalVec = normalize(texture2D(normalMap, texCoord).xyz * 2.0 - 1.0);

        …

        // combine terms to make color
        gl_FragColor = vDiffuse*base + vSpecular + vAmbient*base;
}
```

Normal mapping goes a long way toward increasing the detail of a scene with only a minimal decrease in performance. In the picture below, the right side is normal mapped while the left side is not.

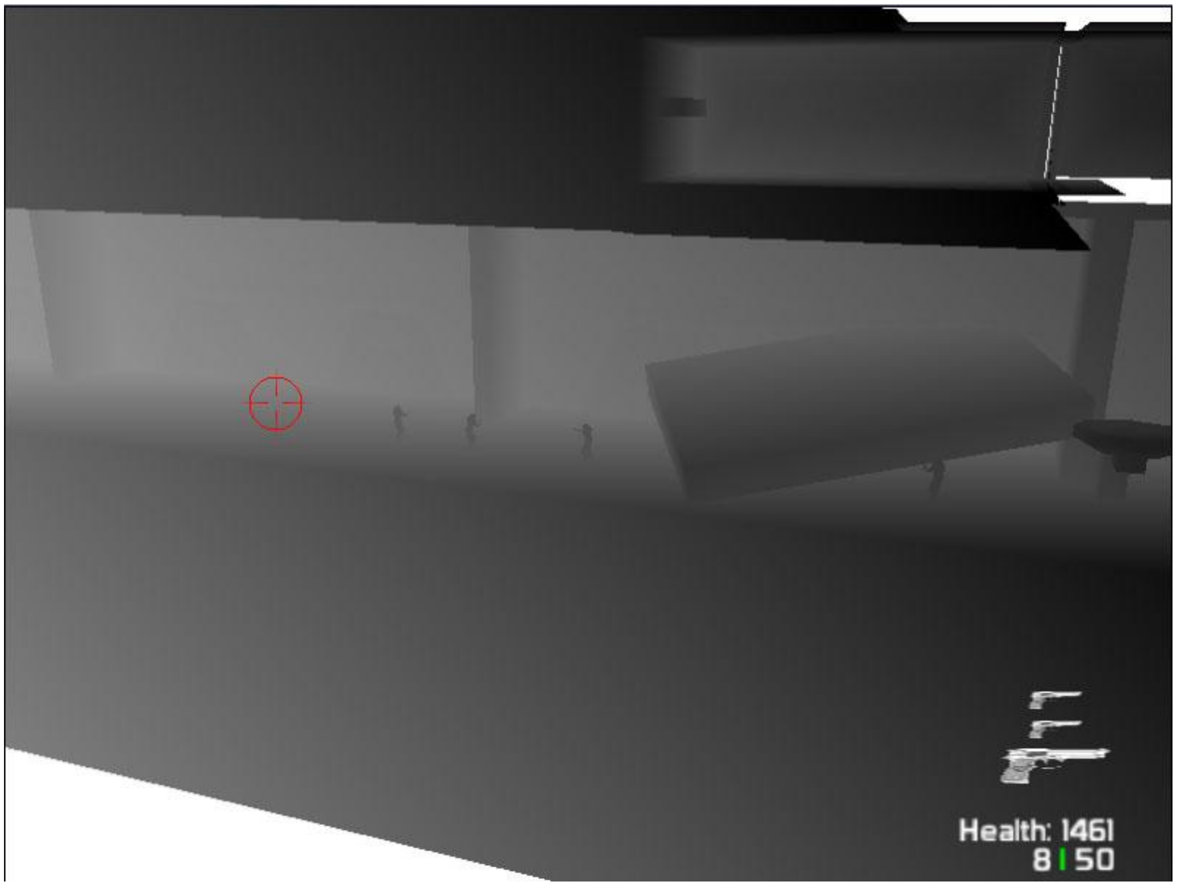


Normal mapping is especially apparent on the grating on the floor and the wall.

Shadow Mapping

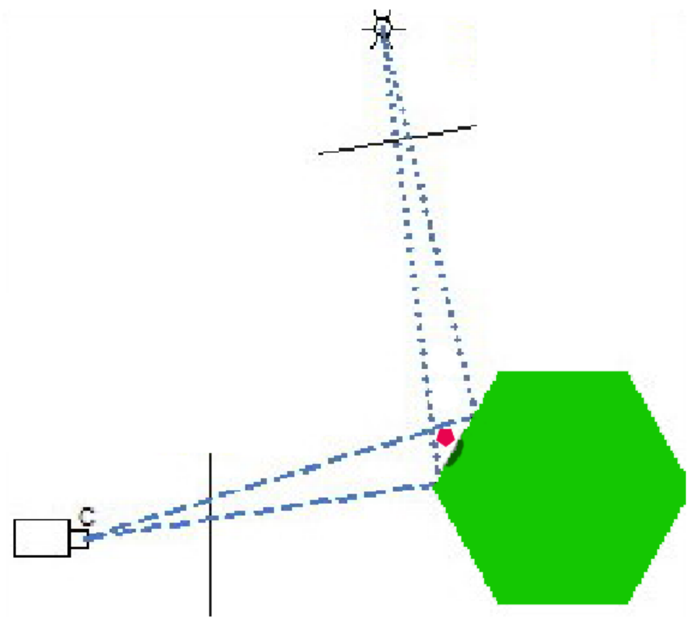
One of the most important ways to add realism to a scene is to render shadows. Shadows add depth to a scene and convey a lot of perspective information to the player. Unfortunately, shadows are very difficult to simulate in OpenGL because every triangle is rendered independently. There is no way to know if one triangle is blocking the light of another in the fixed function pipeline. Drawing a ray from the current pixel to the light and testing for intersection with each object, a technique done in ray-tracing, is far too slow a process for a real-time application. Our solution to this problem is an algorithm known as shadow mapping. Shadow mapping is a two-pass approach to shadows.

1. Render the scene from the perspective of the light and store the depth information in a buffer.



This is a depth-only render

2. Render the scene normally and transform each point into the light's coordinate space.
3. Compare the depth of the pixel with that in the light's depth buffer to determine if the current object is the closest object to the light (not in shadow).



The area in shadow failed the depth test against the pink hexagon.

Implementation

In order to store the depth buffer from the light's render pass, you need to utilize a feature in OpenGL known as frame buffer objects. Frame buffer objects allow you to render a scene to a texture instead of to the screen. They are pretty simple to use, but require a bit of setup before they can be utilized. For more information on initializing them, I recommend the tutorials on gamedev.net.

Before rendering the first pass, the first thing to do is turn on the newly created frame buffer object. We must also turn off color writing since we only want to write to the depth buffer.

```
// Turn on the frame buffer
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, m_shadowFrameID);

//Disable color rendering, we only want to write to the Z-Buffer
glColorMask(GL_FALSE, GL_FALSE, GL_FALSE, GL_FALSE);

// Clear previous frame values as usual
glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

Next, we must transform the view into the light's perspective.

```
// Change the viewport to the texture's dimensions
glViewport(0, 0, shadowBufferWidth, shadowBufferHeight);

// Reset the projection matrix
glMatrixMode(GL_PROJECTION);
glLoadIdentity();

// Set the field of view, aspect ratio, and near/far planes
gluPerspective(45, shadowBufferWidth / shadowBufferHeight, 0.1, 500);

// Reset the model view matrix
glMatrixMode(GL_MODELVIEW);
glLoadIdentity();

// Place the light at lightPos looking at the player
gluLookAt(lightPos.x, lightPos.y, lightPos.z, playerPos.x, playerPos.y, playerPos.z, 0, 1, 0);
```

Now we can draw the objects in the scene and turn the frame buffer back off.

```
drawObjects();
// Turn off the frame buffer object
glBindFramebufferEXT(GL_FRAMEBUFFER_EXT, 0);
```

In the second pass, we must transform each point into the light's coordinate frame. A convenient way to do this is to store the current model-view projection matrix in one of OpenGL's supplied texture matrices. This makes it very easy to pass to the shader.

```
// Shift units from [-1, 1] to [0, 1]
const GLdouble bias[16] = { 0.5, 0.0, 0.0, 0.0,
                            0.0, 0.5, 0.0, 0.0,
                            0.0, 0.0, 0.5, 0.0,
                            0.5, 0.5, 0.5, 1.0 };

 // Grab modelview and transformation matrices
 glGetDoublev(GL_MODELVIEW_MATRIX, modelView);
 glGetDoublev(GL_PROJECTION_MATRIX, projection);

 glMatrixMode(GL_TEXTURE);
 glActiveTexture(GL_TEXTURE7);

 glLoadIdentity();
 glLoadMatrixd(bias);

 // concatenate all matrices into one.
 glMultMatrixd(projection);
 glMultMatrixd(modelView);

 // Go back to normal matrix mode
 glMatrixMode(GL_MODELVIEW);
```

The second pass is rendered as normal. Our per-pixel lighting shader can be modified to accommodate shadows fairly easily. The vertex shader only requires one additional line to pass the new texture coordinate lookup.

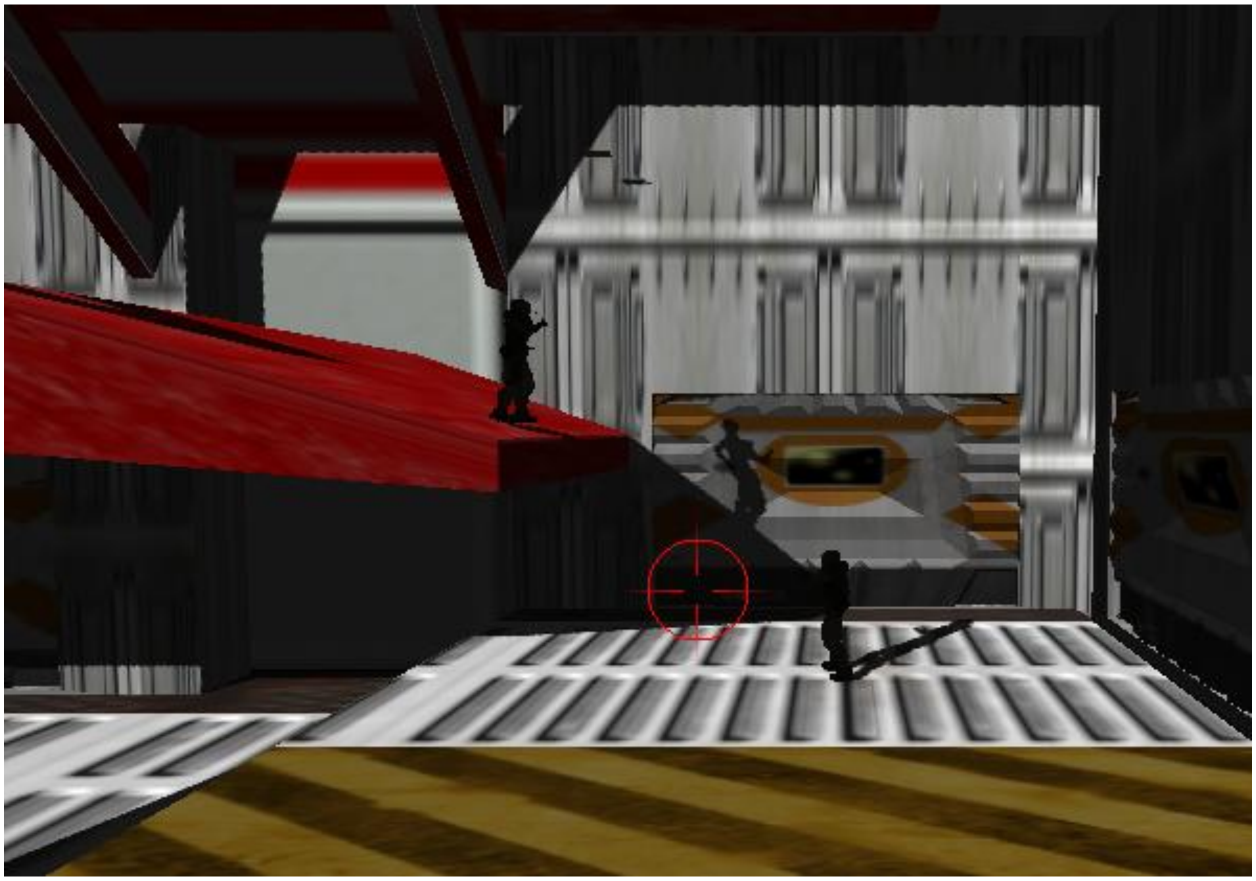
```
varying vec4 shadowCoord;
…

void main() {
        // Transform the shadow texture coordinate by the light's
        // transform matrix
        shadowCoord= gl_TextureMatrix[7] * gl_Vertex;
…
}
```

The fragment shader is a little more interesting.

```
// The depth buffer from the light's render pass
uniform sampler2DShadow shadowMap;
…
varying vec4 shadowCoord;

void main() {
        …
        float shadow = 1.0;
        // Avoid sampling values behind the light's view frustum
        if (ShadowCoord.w > 0.0)
                shadow = shadow2DProj(shadowMap, shadowCoord).r;

        //combine terms to make color
        gl_FragColor = ((vDiffuse*base + vSpecular) * shadow) + vAmbient*base;
}
```

The function shadow2DProj handles the depth comparison between the supplied depth buffer and the current pixel. It will return 1 or 0 accordingly. Notice that no matter if we are in shadow or out of it, we still add in an ambient term to prevent shadows from being 100% black.

Shadows not only make a scene look more realistic, they also convey depth information that is crucial to conveying the player the spatial orientation of everything in the scene.



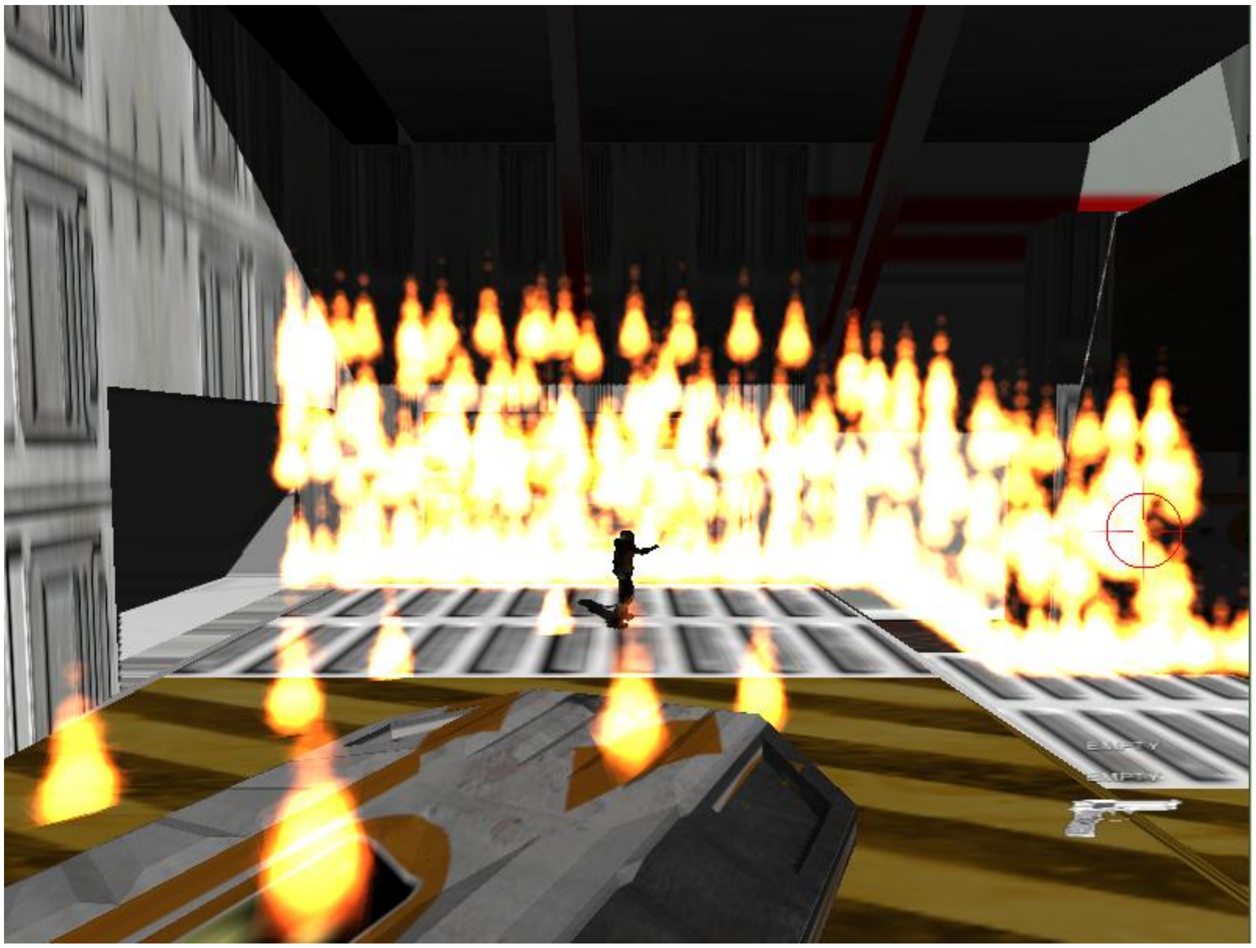
Results

The result is a fully playable single level with many advanced technologies under the hood. Although we had no dedicated artists on our team, we finished one level that has the graphical style we wanted in a way that both looks aesthetically pleasing and serves the gameplay.

The Importance of Tools

Spending the time we did to create tools to aid in development was very helpful. It did not take very long to get basic versions up and running and the tools were continually improved over the development of our game. The importance of being able to quickly see the results of our efforts is something that cannot be overstated. Tools allowed us to quickly create content and iterate over it to find the design we wanted.

Our particle editor allowed us to create some really great looking particle effects (particularly the fire) using very few particles. This reduced the computation time necessary to render particles and allowed us to have over 100 emitters on screen without slowdown.



A wall of fire using 150 emitters at a very playable frame rate

The 3ds Max plug-in was also extremely effective. The ability to create complicated geometry, swap textures on the fly, place lights, alter the camera's zoom level per-room, and position enemies all in the same editor gave us a lot of flexibility in the creation of our level. In fact, the true potential of our plug-in is not fully utilized as our single level only has two camera positions and one global light. In later places in the level, our plans were to have claustrophobic areas with a tighter camera and darker environment. Unfortunately, time constraints forced us to cut this idea.

Technologies That Worked Well

One of the technologies that dramatically improved the fluidity of our game is the animation blending system. In our game, we needed the player to be able to run, jump, crouch, fire, and reload. Each of these actions has an associated animation with it. Since we did not have time to create separate animations for each possible combination of these actions, our two options would have been to either display no animation (and thus leave the player confused as to what is happening on screen) or to arbitrarily restrict actions to the player could not reload or fire unless they were standing still. Thanks to our animation blending system, we are able to seamlessly blend the animations for running and reloading or jumping and firing without the need to create separate animations for each possible combination of actions.

The technology that I personally am most proud of is the real-time shadows. Not only do the shadows make the scene much more realistic, they also aid in the gameplay. Without shadows, the player looks as if he is floating above the floor and it is difficult to tell what is in the background versus what is in front of the player. With shadows, the necessary depth information is present and it is easy to tell where objects are with respect to one another.

The Gameplay

From a gameplay perspective, our game was mostly well received by play testers. Most players enjoyed the feel of fighting (and beating) the enemy soldiers and using the different weapons. The most common criticism we got was that the game was too hard. We spent a lot of time tweaking the stats of the player, the enemies, and the weapons. By the end of the quarter, we had multiple players finish our game without dying, so we feel that the game's difficulty level is the right level of challenging without frustration.

Working in a Team

For the most part, our team had a unified idea of what we wanted our game to be. We knew we wanted a game that had a realistic tone that would cater to more experienced gamers. We wanted to avoid an "arcadey" game that consisted of blindly running and shooting everything. Everyone in our group had a different aspect of the project they were interested in. I was personally most interested in the shaders and visual aspect of the game. The other group members found their interests in animation, engine development, and particle systems. These differing interests came with their advantages and disadvantages. They offered our group a certain independence in working. It was easy to avoid stepping on each other's toes when everyone was working on a separate thing. Unfortunately, it also resulted in certain features becoming very developed while leaving other areas by the wayside. Since each person had a designated area to work in, the other areas were left under developed.

In the second quarter, our team decided to rewrite parts our underlying engine. We were under the impression this rewrite would only include a few classes and would be finished within one or two weeks. Unfortunately, this rewrite spiraled out of control and lasted nearly eight weeks instead. While the new engine was undeniably better organized, more efficient, and less buggy, the time spent redoing old work was probably not worth these improvements. It would have been better to devote these eight weeks to improving the gameplay and adding new mechanics such as different enemies or weapons.

Future Plans

Our biggest constraint in this project was time. We had a large number of ideas that never made it into the final game. The original concept for the game featured multiple enemy types and a climatic mech battle with the prison warden. Unfortunately, these concepts had to be scrapped in favor of time. There was also a much deeper story component to the game as the player discovered the true purpose of Sweet Water. There was going to be an emphasis on "visual" story telling in which we use the environment to tell the story rather than cutscenes with dialogue. Unfortunately, without a dedicated artist, these plans were scrapped very early on. We also had plans for one other major graphics technology: screen-space ambient occlusion. This coupled with the shadows would have helped give the spaceship a brooding, dark, and foreboding atmosphere. Sadly, a team of four people coupled with an engine rewrite halfway through the project restricted our options. We decided the best thing to do was to set these ideas aside and possibly revisit them at a later date if we continue working on the game beyond the classroom.

Conclusion

This project was a huge learning experience for me. It has truly demystified the game development process for me. I now have a tremendous respect for the amount of work that goes into the technologies in games. That new sense of perspective has allowed me to notice many details in games that I would never have noticed before. I can tell what technologies and new innovations each game brings to the table.

Beyond game development, the graphics concepts I have learned in this project can be applied to many 3d applications. The most important skill I have been able to exercise is the ability to use my technical ability to create something aesthetically pleasing. This eye for creating beauty out of a mess of vector math and matrix multiplication is critical for creating quality computer graphics applications.

References

- Akenine-Moller, Thomas, Eric Haines, and Naty Hoffman. *Real Time Rendering*. Natick: AK Peters, 2008. Print.
- Schell, Jesse. *The Art of Game Design: A Book of Lenses.* Morgan Kaufmann, 2008.
- PhysX — A collision, rigid-body, and soft-body physics simulation library made by Nvidia.
- SDL — Cross-platform multimedia library to support OpenGL apps.
- Fmodex — Programming toolkit for audio playback.
- PhysicsFS —Library to access abstract archives.
- DevIL — Cross-platform image loading library.
- ATI RenderMonkey — A Windows tool for prototyping GLSL shaders.b
- Soundsnap — A archive of various royalty-free sound effects