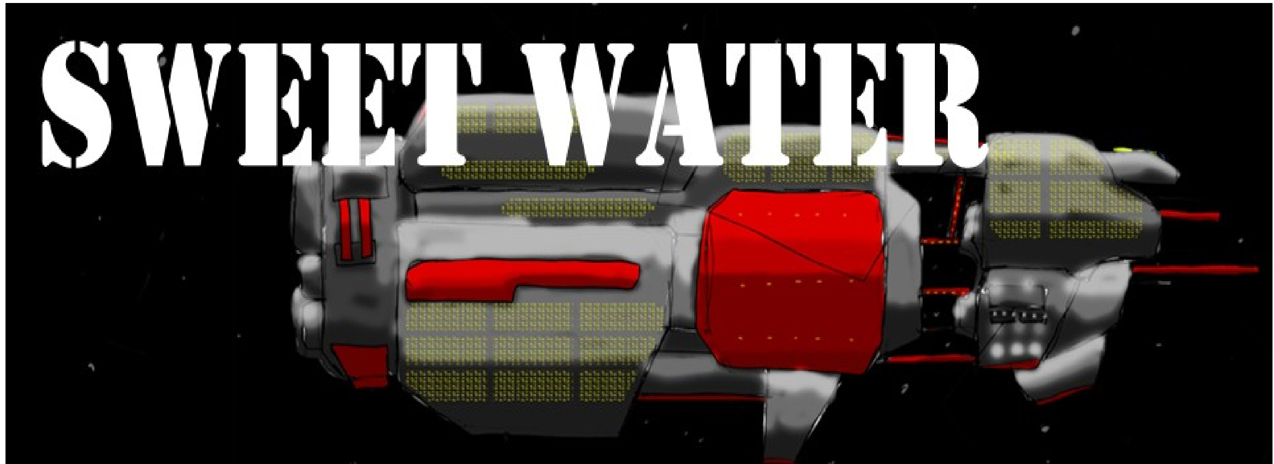
Sweet Water

3D Side-scrolling Platforming Shooter



By Steven Udall
California Polytechnic State University
San Luis Obispo
Winter and Spring 2011
Advisor: Professor Zoe J. Wood
Teammates: Robert Bernal, Nick Moresco, Ilya Seletsky

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 looking graphics. In non real time graphics applications, we have the luxury of having few time constraints. Because games need to be highly interactive, we need to render complex scenes quickly. 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. What most users don't know is the number of tricks required for implementing a real time graphics application.

Problem Description:

Our goal was to create a working running game that was also fun to play. 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 two fold.

1. How do we render a complex 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. Discovering what it takes to create a game from start to finish was a great learning

experience. Games are a complex and challenging problem, and we wanted to solve a complex problem with a unique solution.

We've played video games nearly all of our lives. We wanted to know how people make games from start to finish. Only after this project did we truly understand why it takes a large company, with millions of dollars, years of production to make a big hit game.

Previous Work/Related Work:

There have been many great shooters created since the first games came about. Halo, Doom, and Shadow Complex influenced the creation of Sweet Water.

Halo Series

Who doesn't like defending the universe against hoards of aliens called the covenant? Created by Microsoft for the Xbox and PC, this first person shooter presents stunning scenery combined with ruthless enemies. The goal is to save the human race with a variety of weapons.

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.

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.

Overview of Solution

Games are complex, therefore they require a great deal of components to work correctly. Sweet Water was created 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 a little information about each.

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: Whatever the high level game code might be

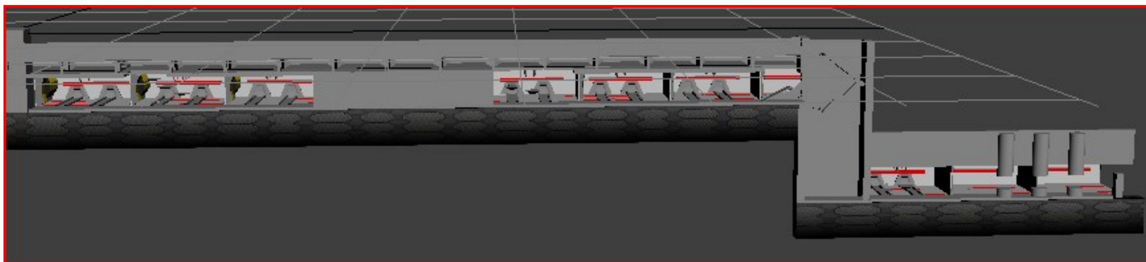
Entities

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-in functionality included:

- Geometry exporter
- Light and camera exporter
- Entity exporter



Particle Effects 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

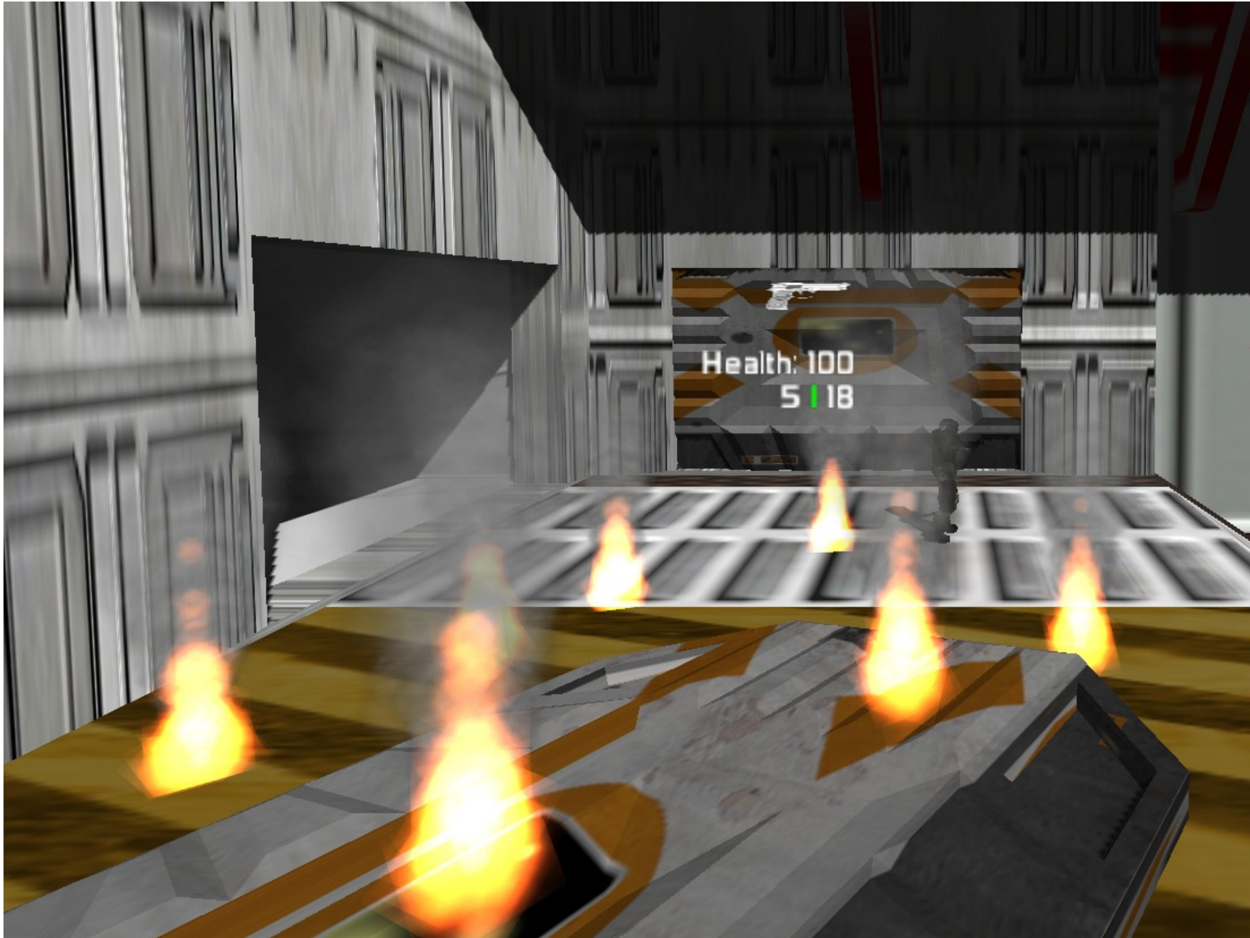
The following represents other technologies present in our game:

- Animation
- Inventory System

- Shadow Mapping
- Normal Mapping
- Per Pixel Lighting
- Particle Effects
- Physically based character movement and interaction

Algorithms:

Particle Effects System



A particle effects system (PES) is essential to the aesthetics and feel of most any game. From the dust pool coming off of a character's heel to the trail of a rocket, a PES provides a means to make an interesting world filled with life. This section explains how the particle system implemented in Sweet Water is organized and details how it functions.

The PES consists of two parts, the particle emitter and particles it emits. Two different classes provides modularity and therefore ease of particle management.

Particle Emitter

The particle emitter is an invisible object that ejects and manages particles. It takes in six

arguments to the constructor: the emitter type which determines what attributes particles will have, a map controller which directs and provides access to the camera and what is happening currently on the game map, the position of the emitter in world space, the three dimensional direction it's emitting in, the lifetime of the emitter, and a scale factor which scales each particle type appropriately. Note that the direction passed in was originally a single angle since OpenGL's rotation function takes an angle about an axis (which works fine for a textured quad and two-dimensional gameplay); however, this was changed in favor of a vector, allowing the particle effects to not always be emitted in the x and y axes. Particles are emitted normal (retrieved from the physics engine) to the surface that is hit for projectiles. Also, the scale factor must affect more than, for example, the quad size in order to scale the effect properly. Sweet Water's fire is scaled by its particle speed and quad size.

All particles inherit their attributes from the emitter. However, a particles attributes are a bit different from the emitter. The emitter must store the minimum and maximum of randomized attributes since the exact value for the particle is calculated upon creation and any time we need to update the attributes. Also, the emitter has its own lifespan which should not be inherited by particles.

The constructor retrieves the proper attributes for the emitter type and allocates the maximum number of particles that the emitter can possibly ever show at one time so that allocation never needs to be done during emission. Particles are managed using two different lists: an alive list and a dead list. All particles begin in the dead list and are pulled into the alive list as needed based on the birth rate, which is limited by the lifespan of each particle and the maximum number of particles. Particles that reach the end of their lifespan are swapped from the alive list to the dead list. Note that the particle emitter should wait for all particles to die before

killing itself.

All particles in the alive list are drawn by the emitter rather than the game engine. This is done to simplify and optimize view frustum culling (not processing stuff that's not within the view frustum) on particles. There is a significant hit to performance when testing for view frustum collision on all particles. So, as long as the bounding box used in testing for intersection with the view frustum is large enough, only one box around the entire emitter and its particles is needed.

Particles

The particles in the system are simply textured quads. Since quads are two-dimensional, the player needs to always see the quad at the same proportions. We can achieve this by billboarding the quad; in other words, the quad should always face the camera in all three dimensions.

Billboarding should be done in a vertex shader just before every draw of the particle with two rotations. The first rotation is done about the y axis to get the quad to face the camera on the x-z plane. Recall that the dot product of two vectors can be represented as the magnitude of the first vector multiplied by the magnitude of the second vector multiplied by the cosine of the angle between them (v1 dot v2 = v1.magnitude * v2.magnitude * cos(angle between v1 & v2)), so the magnitude of this rotation is simply the arccosine of the quad's normal x-z component dotted with the quad to camera vector x-z component (angle = acos(QuadNormalXZ dot QuadToCameraXZ)). It's important to remember to normalize both vectors since we are assuming the magnitude of each is equal to one. All that's left is to get the quad to face the camera in the y axis. Using the same method for rotating about the y axis we know that the angle of rotation should be the arccosine of the camera's view vector dotted with the view vector's x-z

component (angle = acos(Camera dot CameraXZ)), and the axis of rotation is just the vector perpendicular to, or cross product of, these two vectors (axis = cross (Camera, CameraXZ)). For the case where these two vectors are equal, we do not need to do the second rotation and in fact will cause an error since the cross product of two parallel vectors is unknown.

The constructor of the particle object takes only one argument: the parent emitter. This allows easy access to the attributes of the parent, which are copied from the parent to the child.

The update method for particles receives the time passed and uses this to follow the classical physics formula for velocity (v* = v + timePassedSinceLastUpdate * acceleration), which is then used in the classical physics formula for position (position = (average velocity) * time). Recall that the update function is called by the emitter rather than the game engine, so we can return true when a particle is still alive after the update and return false if it has died. This allows us to easily check to see if a particle needs to be dispatched to the dead list before we do somewhat costly physics calculations.

The draw method follows this order of operations: set up the GL states and material in the particle emitter's draw method just before, translate the quad to its position, scale the quad based on its interpolated value, billboard the quad to face the camera, rotate the the quad in the x-y plane, determine any other interpolated values (in this case, color and alpha) based on the particle's lifespan and life remaining, set the quad's color, and draw the quad. Note that, for the x-y plane rotation, the angle between (1,0,0) (this is the zero degrees vector in the x-y plane) and the emitter direction vector x-y component should be added to whatever other rotations are desired so that particle effects that have a specific direction on top of a random direction, such as the sparks in Sweet Water, will face the correct way.

Each particle has a position, velocity, acceleration, gravity component, direction, scale

begin and end, life span, color begin and end, alpha, and x-y rotation. Beginning and ending attributes are interpolated between, while minimum and maximum attributes from the emitter attributes are randomly chosen between. The minimum and maximum attributes available are particle life span, speed, x-y rotation, and acceleration.

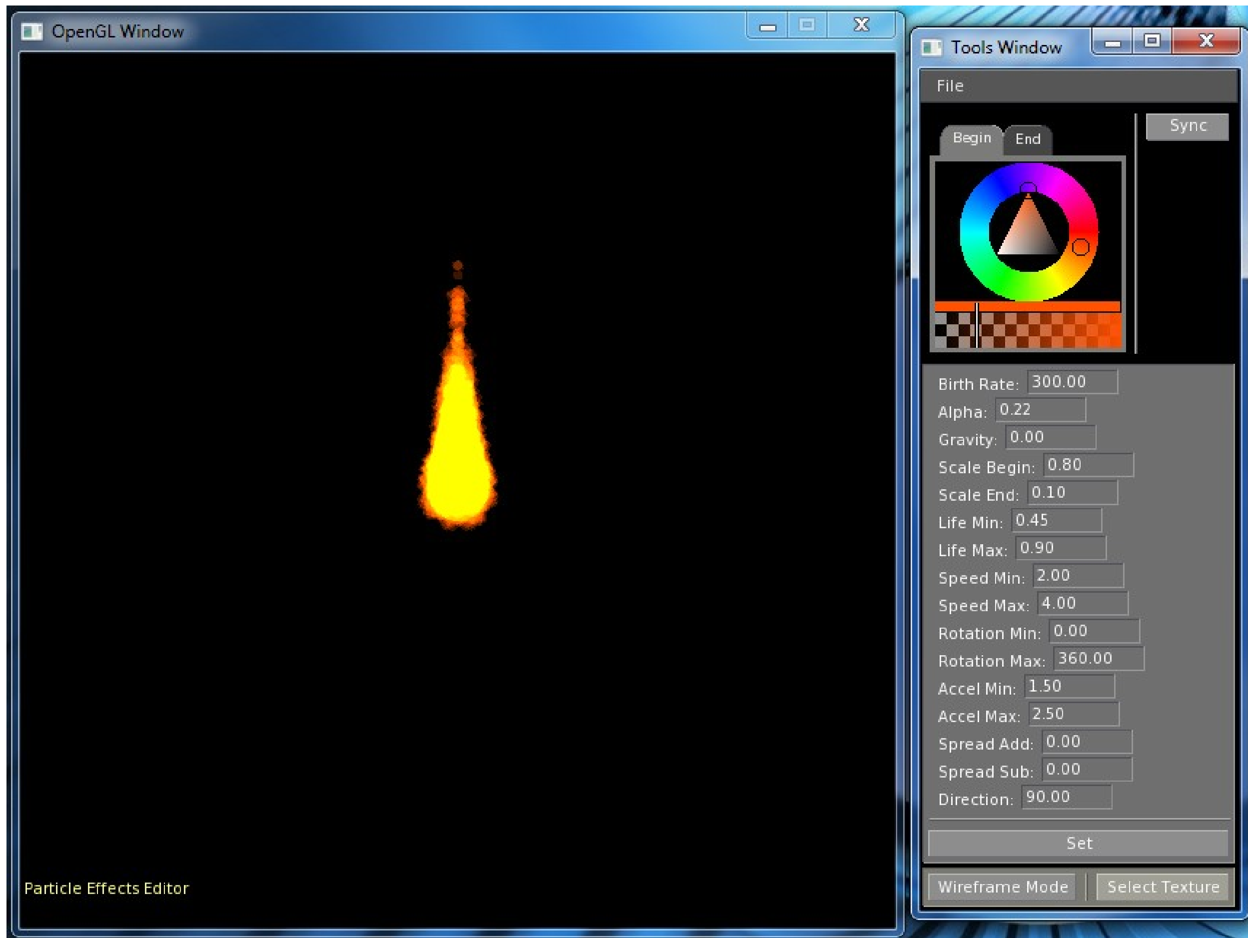
Particle effects greatly benefit from blending to aesthetically fit into the world. The two blending methods implemented for the Sweet Water PES are additive color blending and alpha blending with an alpha map. Additive color blending is where textures, when superimposed, add their color values together. Black areas of a texture count as zero and are transparent where white areas count as one. So, a color like orange (about (255, 127, 0)), for fire, when superimposed will add up to a light yellow color toward the inside of the flame while darker along the outside; however, add a small amount of blue to the orange and it will add up to a much whiter yellow where the particles are most dense. Additive color blending requires that depth writing be off for OpenGL while drawing the particles and that the particles be drawn after all other parts of the scene. Alpha blending is where the texture contains an alpha map or mask that indicates the transparency of each pixel of the texture. This type of blending is useful for something like blood, where you would not want the red to add up to be a brighter red when superimposed but you still want transparency. For the best blending results, alpha blending requires sorting and drawing particles from back to front with depth writing turned off and drawing all particles after everything else in the scene.

Particle Effects System Integration

The PES is integrated into the entity system created to distinguish and manage individual elements of the world. The entity system manages depths, GL states, updates, and draws for all parts of the scene. It was decided that the PES be an entity subclass where the entity constructor

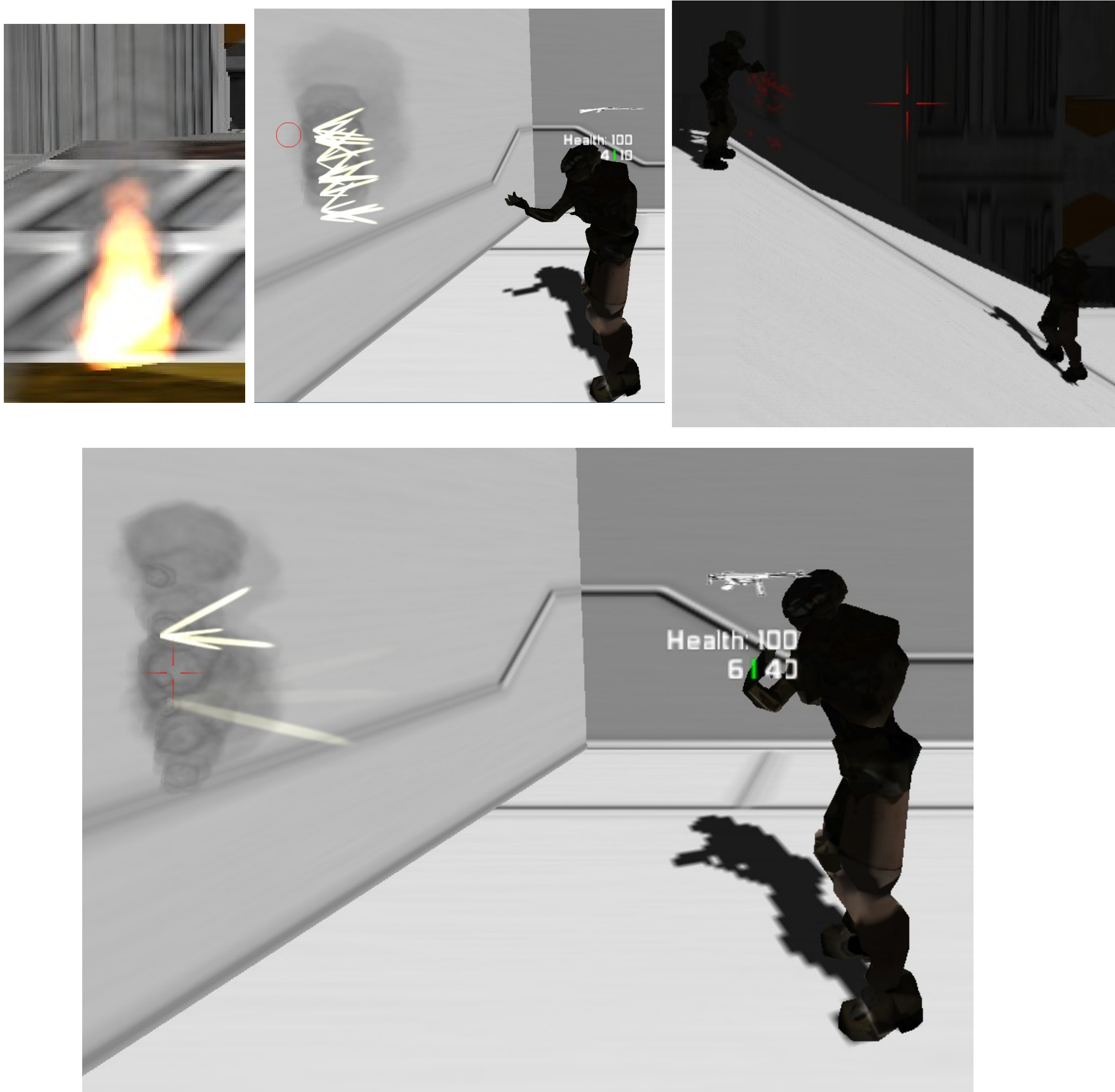
is passed the necessary values. This allows particle emitters to be allocated on the fly and will automatically be put into the entity list to be managed by the game engine.

Particle Effects Editor



In order to make quick and effective use of the particle effects system created, it was decided that an editor be created. The particle effects editor is composed of three parts: the particle effects system (mostly the same as the one in the game), a particle effects system manager as a miniature engine for running the system in the editor and interfacing with the GUI, and an Agar GUI which comes with built in OpenGL built on top of the SDL windowing system.

Particle Effects Generation and System Use



This section describes a decent way of creating a good looking particle effect (like the ones above). The recommended steps are as follows:

1. Choosing a texture – Ideally you want to choose or create something with a large amount of detail that is generally non-uniform. This will give the appearance that a single texture is actually a bunch of different textures.

2. Choosing attribute values and realism – This is something you just have to play around with until you find something you like, then tweak it over multiple days so that you have a fresh look at it each time you work on it and so that you have time to decide what you do or don't like about it. Iteration and refinement is key here. Also, try to think about how the real world effect appears to move. For example, real fire appears to have an upward acceleration.

3. Giving the emitter a static direction – An effect can easily be given a static direction by creating a texture with its bottom most point at the center. An example of this is the sparks in Sweet Water.

4. Making effects look organic – Randomness adds a huge amount of cool organic look to an effect. Give the static direction as much randomness as possible.

5. Deciding on maximum number of particles – Usually, you should not need to go over 100. Fire in Sweet Water maxes out at 30 particles.

6. Choosing a birthrate (smooth emission) – If you want emission to be very smooth, use this formula: birth rate = lifespan max / max particles. Where birth rate is seconds per particle and life span is in seconds.

7. Choosing between additive color blending and alpha blending - Additive color blending is great for a particle effect that appears to or actually emits light. An alpha mask or map is useful for everything else. Alpha masks can easily be added to an image file using a photo editor such as Photoshop and by turning the texture into a grayscale image and pasting that into the alpha channel. White will be full alpha, while black will be no alpha.

Inventory System

The inventory composed of two types of weapon objects in the game. There are two base classes with subclasses for each weapon.

One of the types is designed to interface with the entity system and game world. These are represented by a model on the ground that is dropped from a character when he swaps it for another one or when an enemy dies, as seen above. It holds the weapon model and texture data for weapon item on the ground, other physics details (physics shape, etc), and ammo. The shape is tested for collision with the character's capsule shape and is then picked up in the update method of character's structure.

The other type interfaces with character structures in the game and represents what the character is carrying. It holds animation of the weapon, weapon specifications (ammo, clip size, firing rate, accuracy), weapon model and texture data for character's hands, and weapon sounds.

Because the two types converge in the character's structure, swapping data between the two is simple.

Conclusion:

Working with a team to build Sweet Water was an awesome experience. I learned a great deal about video games, their graphics technologies, and the kind of structure and organization a

team needs to create one. I highly recommend taking up or joining a video game project for anyone interested in real time computer graphics.