ABSTRACT

GAME ENGINE ARCHITECTURE: A COMPREHENSIVE VIEW

by
Donald Kehoe

Game development is an ever growing interdisciplinary field. A variety of different skill sets need to come together to create a professional game. These range from art, audio and design through programming and development. It is difficult to isolate the different disciplines in game development since every aspect of the game is co-dependent on everything else. Even in the programming domain there are a number of sub-disciplines that a game can require such as physics programming, environmental programming, artificial intelligence and programming the core game engine. A game can even be developed in a number of different ways depending on the design of the final product.

This daunting array of required disciplines makes it difficult to offer effective instruction in game development. The tendency is to divide the disciplines according to traditional academic categories. This leads to offering courses on game development that are too narrowly focused to lead to any functional games or to courses that are purely theoretical. This thesis attempts to outline the development of a generic game engine and show how it can be implemented with minimal skills, beyond those in programming. It is intended to serve as a foundation for developers who will specialize in a particular area, but will nonetheless need to understand what a game engine is and how programmers in different fields can work with it.

GAME ENGINE ARCHITECTURE:
A COMPREHENSIVE VIEW

by
Donald Kehoe

A Thesis
Submitted to the Faculty of
New Jersey Institute of Technology
in Partial Fulfillment of the Requirements for the Degree of
Master of Science in Computer Science

Department of Computer Science

August 2009

APPROVAL PAGE

GAME ENGINE ARCHITECTURE: A COMPREHENSIVE VIEW

Donald Kehoe

7/22/09

Dr. James McHugh, Thesis Advisor
Professor of Computer Science, NJIT
Date

7/22/09

Dr. Andrew Sohn, Committee Member
Associate Professor of Computer Science, NJIT
Date

22 JULY 2009

Dr. Michael Baltrush, Committee Member
Associate Professor of Computer Science, NJIT
Date

BIOGRAPHICAL SKETCH

Author:	Donald J. Kehoe
Degree:	Master of Science
Date:	August 2009

Education:

- Bachelor of Arts in Computer Science
  New Jersey Institute of Technology, Newark, NJ, 2002
- Master of Science in Computer Science
  New Jersey Institute of Technology, Newark, NJ, 2009

Major:	Computer Science

Dedicated to my loving wife, Sandra Kehoe

ACKNOWLEDGMENT

This thesis would not have been possible without the support of Dr. James McHugh, whose guidance pushed me to gain a better understanding on the subject.

I would like to show my gratitude to Dr. Andrew Sohn and Dr. Michael Baltrush for their support and input in the development of this work.

Lastly, I offer my regards to all of those who supported me in any respect during the completion of the project.

TABLE OF CONTENTS

TABLE OF CONTENTS

TABLE OF CONTENTS

(Continued)

LIST OF FIGURES

CHAPTER 1

INTRODUCTION AND GAME ENGINE OVERVIEW

Game development is an ever growing interdisciplinary field. A variety of different skill sets need to come together to create a professional game. This stretches from art, audio and design through programming and development. It is difficult to isolate the different disciplines in game development since every aspect of the game is codependent on everything else. Even within the sphere of programming there are a number of sub disciplines that a game can require such as physics programming, environmental programming, artificial intelligence and programming the core game engine. A game can even be developed in a number of different ways depending on the design of the final product.

This daunting array of different disciplines makes it very difficult to offer useful instruction in game development. The tendency is to divide the disciplines according to traditional academic departmentalization. This has led to course offerings on game development that are too narrowly focused to produce any functional games or courses that are purely theoretical. This paper will attempt to outline a generic game engine and how it can be implemented from scratch with minimal skills beyond programming. This will serve as the foundation for developers that will specialize in an area, but will always need to have the understanding of what a game engine is and how programmers in different fields can work with it.[6]

A game engine is the core set of data structures and functions that provide the foundation for developing a full game. The engine is responsible for managing all of the resources available to the system and providing the capability to utilize any given resource. This includes loading and drawing graphics as well as handling the input of the player.

The engine does not deal with the final upper-level assets (graphics & sounds that are used in the game) or the specific configuration of the game logic. The game logic level is the part of the game that uses the capabilities provided by the game engine in order to give the final game its form and function. In most engines that are developed from scratch for a single game, the two components (engine & logic) will be intertwined. In engines that are developed for long term use (beyond a single game) the components will deliberately be broken into separate parts of the project.

In the Id Software series of game engines (referred to by the game they were developed for or simple Id-Tech 1 through Id-Tech 5{in development}) this breakdown is accomplished by an executable and library separation. The game engine compiles as a separate project into the executable, while the game logic project compiles into a run time linked library (Dynamic Linked Library or DLL for windows based architecture software object or SO for unix based architectures).[7]

CHAPTER 2

DESIGN CHALLENGES

In order to properly design a game engine, the desired capabilities need to be identified. The style of game desired, target market and platform will dictate design choices for the game engine. Games that are made for specialized hardware such as cell phones and game consoles will also need special planning to account for limited system resources.

2.1 Target Hardware Considerations

Once the target hardware has been identified the game engine can be planned to account for any limitations and utilize any features. Here is an overview of the hardware commonly used in game development.

Developing games for the personal computer is common since most people have personal computers ranging from high end custom desktops to portable laptops. PCs tend to have an abundance of memory and very fast processors as well as the benefit of hard-disks for saving and storing large amounts of information and internet access (now used as the vehicle for delivering content as well as online persistent worlds). Unfortunately PC deployment has to take into consideration the multiple operating systems that all need custom builds for the game or limit who can play the game by choosing to support only one. Since PCs also suffer from variable specifications, there is no base line capability that can be assumed for all systems, so minimum specifications must be planned for, ignoring anything with less than the desired capabilities. This thesis will focus on PC development.

With web based games there is a great advantage in that the specific operating systems are generally not a consideration. The major drawback for web based games is limited capabilities. These games generally build on the Adobe Flash or Microsoft Silverlight development environments. These platforms are limited in their access to the hardware of the computer playing the game and generally perform very slowly compared to games deployed directly on the PC.

A strong market now is the cell phone game market. Cell phones have reached a point of nearly total saturation for most parts of the world, which makes them an ideal platform for developing games. The drawback for cell phones is that they are very limited in their capabilities. They have limited memory and processor speeds, and on board storage (in the form of a solid state memory card) is not guaranteed (at the time of this writing). The size of a program that can run on a cell phone is a limiting factor as well. The code and content must all be below a given amount of memory in order to run at all. For example, very few cell phones have support for 3D acceleration.

A very popular market is the hand held console. These consoles are hardware designed for playing games and are made to be compact and portable. These have the advantage of being purely dedicated to running the game. Other platforms share system resources with other processes, but not so with consoles. Although all of the system resources are available to the developers, the system has very limited resources and many of them have little or no support for 3D acceleration.

The most competitive market at the time of this writing is the home console. Home consoles have the distinct advantages of total access to system resources as well as very good system resources. They are on par with the capabilities of the Personal Computers while having standard system capabilities. The amount of memory, processor speeds, hard-drive sizes and 3D capabilities available to the developers are known for any given console. The only real drawback is the need for the official development kits in order to take full advantage of the system resources and to distribute a game on that system.

2.2 Graphical Considerations

Once a target platform has been chosen, the graphical capabilities may be dictated by that choice. For more capable platforms, this choice may still be important. This section covers the capabilities of the different graphical engine types.

2.2.1 2D Game Engines

When 3D acceleration is not available or the game intended does not call for 3D graphics, a purely 2D game engine can be developed. Dedicated hand-held consoles usually have hardware assisted functions for handling operations related to the manipulation of two dimensional images. This includes scaling and rotation of two-dimensional images. With the growing popularity of 3D games, this 2D acceleration has been dropped from most graphics processors.

2D games use animated flat images called sprites (see section on sprites for more details) to represent all game elements that move and larger images to handle full screen images and background images. The capabilities of the game engine focus on the management of 2D resources and deal with all movement and collisions in a 2D vector space.

2.2.2 3D Game Engines

If the target platform has 3D acceleration, 3D game engines become a possibility. It is worth noting that 3D game engines are possible on systems without 3D acceleration, but limitations placed on the game because everything has to be handled in software often mean the game would be better made for other systems. In a 3D game most of the game's graphical assets will be 3D models. This includes the active entities as well as the environments. All movement and collisions are handled in a 3D vector space.

2.2.3 3D in 2D Hybrid Game Engines

When 3D acceleration is limited but high resolution images are possible, a technique called 3D in 2D is used. This method uses pre-rendered images for the bulk of the graphics. By fixing the in-game camera to the orientation of the rendered point of view the illusion of true 3D is maintained.

This method is easy to implement: Using a 3D modeling and rendering program (in this case Blender: www.blender3d.org) an artist will create a full 3D virtual environment. A rendered 2D image of the 3D environment is produced and the orientation of the 3D camera is recorded. The 2D image is loaded into the game as the full screen background.

In the game the camera is set to the same parameters (position & rotation) that the scene was rendered from. Any additional graphics that need to be animated are loaded into the game as true 3D object models and rendered on top of the background. Everything except the background image is subject to perspective rendering. The result is an effect where the entire scene appears to be 3D, even though the bulk of the scene is in actuality a 2D image.

Although this method managed to save processing power and allow our game to run on older hardware, it does cause some problems in development that will be explained in sections below.

2.3 Cross Platform Challenges

In order to ensure that a PC game can reach as many potential customers as possible, the game engine needs to be able to be ported to different operating systems. Microsoft Windows and Apple Macintosh systems are both very common, but they are very different from each other as far as software development is concerned.

Strict programming conventions and the use of cross platform development libraries are used to to make a game easy to port. The two development libraries SDL {Simple Direct-media Layer} and OpenGL {Open Graphics Library}can be used rather than Microsoft's DirectX or Apple's Xcode. SDL and OpenGL are implemented on all major operating systems so any calls to either of these libraries do not need to be rewritten when compiling for a different target system. The entire project is coded in ANSI standard C using the C Standard Libraries. This way all basic functions will be available to all target operating systems.

CHAPTER 3

GAME ENGINE STRUCTURE

The purpose of the game engine is to provide access to the system's resources and update the elements in the game world. The Main function of any game engine will have the main game loop, designed to be infinite until an exit condition is met. For exiting the program outside of the main game game loop, a call to exit(0); will work. The work of the game engine can be broken down into two basic tasks, updating and drawing (audio work is handled by a separate thread which is maintained by the SDL mixer library).

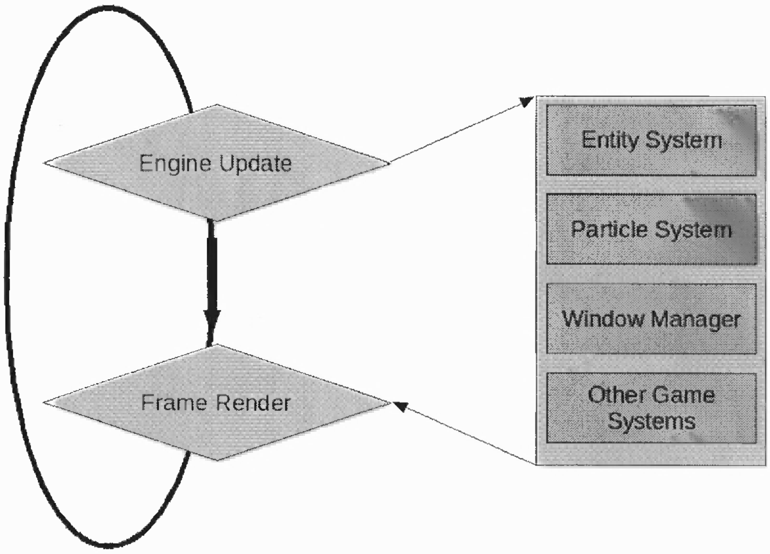

Figure 2.1 The cycle of the game engine: First the sub systems of the engine need to be updated for the new iteration and then the frame is drawn.

All elements of the game engine need to be updated frequently. This will include the entities of the system (see Section 4.7), particles (see Section 4.8) and any input structures. Everything that is updated is based on the same slice of time so that when all is updated that needs to be updated the engine can draw everything based on their new data.

Each snapshot of drawn graphics is called a frame. Each frame is drawn by the game engine during an iteration of the main game loop. The primary data structure for the draw loop is the draw surface. A surface is a generic structure used to describe any possible graphical image. This can be either a picture in memory or the memory address for the physical video surface. The physical video surface is the image that is displayed on the monitory. For the main game draw loop, the physical video surface is the one that is most important.

For the purposes of smooth animation, a secondary surface is needed as well. This second surface (often just referred to as 'screen' for simplicity) is what is drawn to directly within the loop. The initial call to ResetBuffer() will clear this drawing buffer. After everything has been drawn to the secondary video surface the entire surface is copied to the physical video memory. This technique is called "double buffering". This prevents slow drawing from causing a flicker in the animations.

Below are the functions used to create a frame:

```
    ResetBuffer();  //Clear the drawing buffer and fill it with a solid
color

    DrawWorld();  //Draw the pre-rendered back ground image to our buffer

    drawEntities2();//draw all 3D world objects to our draw buffer

    DrawHud();   //Draw the heads up display to buffer on top of the
background and entities.

    DrawMouse();  //Draw the sprite that makes up the mouse

    NextFrame(); //Place the contents of our draw buffer on the actual
video surface for the player to see.
```

The order is set based on a layered drawing approach. Everything is drawn to the same video buffer before being block transferred (Blitted) to the screen. This way anything drawn before can be overwritten by what gets drawn afterward. All of the draw functions are encapsulated by the calls to ResetBuffer(); and NextFrame(); which serve to clear the drawing buffer so it can be used again and to Blit the newly drawn buffer to the screen (physical video surface).

The other main function for the game loop is to update all of the elements important to the game. This is accomplished by a call to UpdateAll() which will call the update functions for all of the game's input structures, entities, particles, and lights.

As the main loop iterates the game engine will update all information, handle all of the collisions, artificial intelligence updates and draw the next frame. This main loop needs to be able to run on the average of 60 Frames Per Second(FPS) (the standard rate for most video games). The more action is involved in the game, the more a constant 60 fps is necessary.

The game engine is responsible for initialization of services as well as the updating and drawing of the content in the game. Initialization will vary based on the specific needs of a game engine. Shown here are the core needs for most games.

```
        Init_Graphics(1,1);       // Sets up the display for drawing and
creates draw buffers for blitting.

        InitAudio();              // Sets up the audio system and audio
resource management

        InitSpriteList();         // Sets up the sprite resource manager
```

Each of these calls either initializes a subsystem of SDL / OpenGL or loads content needed by the game. Init_Graphics() is the function that initializes SDL & OpenGL, tests the system for viable graphic modes, creates draw buffers, acquires a pointer to the video draw buffer and sets the default state for OpenGL.

Init_Graphics() is the primary initialization function for the game engine. Within this function, the libraries used (SDL & OpenGL) are initialized as well as the graphics subsystem and initial stats of the 3D rendering system are set.

```
void Init_Graphics()
{ ...
    if ( SDL_Init(SDL_INIT_AUDIO|SDL_INIT_VIDEO |
SDL_INIT_JOYSTICK) < 0 )
     {
        fprintf(stderr, "Unable to init SDL: %s\n",
SDL_GetError());
        exit(1);
     }
     videobuffer = SDL_SetVideoMode(S_Data.xres,
S_Data.yres,S_Data.depth, Vflags | HWflag);
     screen = SDL_DisplayFormat(temp);
     /*sets the viewing port to the size of the screen*/
     glViewport(0,0,S_Data.xres, S_Data.yres);
     glMatrixMode(GL_PROJECTION);
     glLoadIdentity();
     /*view angle, aspect ratio, near clip distance, far clip
distance*/
     gluPerspective( 60, 4.0f / 3.0f, .01, 1500.0f);
     glMatrixMode( GL_MODELVIEW );
     /*Clear color for the buffer*/
}
```

Some sections of this code have been omitted to reduce redundancy.

The call to SDL_Init sets up the SDL system and gives graphic control to this program. SDL_SetVideoMode gives access to the physical video buffer memory address. For use within SDL functions, this is treated like any other SDL_Surface, which is a data structure that holds information on 2D images.

The calls to functions with a "gl" prefix are used to configure the OpenGL 3D rendering system. These will set the screen resolution to the desired range (determined in some of the omitted code or through user specified options), determine the rendering type (in this case perspective view, where objects farther away from the camera appear smaller, as opposed to orthogonal view, where depth from camera has no bearing on the rendering), and sets the perspective ratios. All of these numbers are standard to most OpenGL implementations.

3.1 Environment Structure

Most games take place in specific environments. These environments, often referred to as "maps" or "levels", provide the setting for the game, unique challenges for the player, structure to the game play experience as well as additional requirements for the game engine. The creation and application of the different environments for a game have a great impact on the way many of the other resources are designed and implemented. Entities will need to have updating functions redesigned to work with the different ways the environments are modeled in data. This section covers the most common methods for implementing environments as well as the means with which entities can interact with the environment.

3.1.1 2D Layered Backgrounds

In simple 2D games, one or more backgrounds can function as the environment. In earlier games, but the background served only aesthetic purposes, in other games the background can be interacted with on other levels. When more than one background image is used (usually with a transparent or “key” color) the images are layered in a specific order. Entities that lie between layers need to be assigned to a layer so that the game engine can draw them in the correct order.

When working with multiple layers, additional special effects can be implemented. One is referred to as the parallax effect. The effect attempts to replicate that aspect of perspective wherein objects farther away from the view appear to move slower than they actually do. An example can be found while driving, where a mountain can be seen off in the distance. As the driver speeds down the road, it is obvious to the driver that he or she is moving very fast based on the observed motion of the ground near to the driver. However, the mountain appears to remain the same relative to the driver's perspective.

The motion in the parallax effect can be simulated when two or more layers (more is better) are used. All of the images, save for the image farthest away (and the first drawn), need to be progressively larger than the camera's bounding box. The largest layer, which is the closest to the camera and the last to be drawn needs to be the size of the playing environment. As the camera follows the player's movement and gets closer to the edges of the top layer, each layer drawn is offset by a smaller and smaller percentage of the actual camera position.

In the code sample below, screen is the SDL_Surface pointer to the drawing buffer. An SDL_Rect is a structure with two signed integers x & y and two unsigned integers w & h. This is used to define the rectangle that is used to copy and paste between surface buffers. The data structure level contains information about the game level. Here the only element referred to is layers, which is the number of layers used by the level. BGLayer is an array of surface pointers that contain references to the images to be drawn. Before this function is called, the last background image (BGLayer[0])is already drawn to the screen.

```
void DrawLevel()
{
  int i;
  int maxw;
  int maxh;
  SDL_Rect frame;
  frame.w = screen->w;
  frame.h = screen->h;
  maxw = BGLayer[level.layers-1]->w;
  maxh = BGLayer[level.layers-1]->h;
  for(i = 1;i < level.layers;i++)
  {
```

/*the next two lines calculate the offset of the layers based on the size of the largest layer. Note that when the largest layer is encountered the division equals 1, which creates no offset from the camera's position.*/

```
   frame.x = (( BGLayer[i]->w - Camera.w )/ (float)(maxw - Camera.w) )
*Camera.x;
   frame.y = (( BGLayer[i]->h - Camera.h )/ (float)(maxh - Camera.h) )
*Camera.y;
      SDL_BlitSurface(BGLayer[i],&frame,screen,NULL);
```

```
    }
}
```

The map file can contain information about the background image(s), such as where exits (to other levels are), starting positions for entities and map objects, and other level controls. These controls are usually put into place by a map editing tool, which would also be created by the programmers of the game engine (or with programmers working closely with the engine programmers).

In order for entities to interact with the background correctly, additional information may sometimes be provided. One type of interaction is a walk mask. Walk masks are images, often times simple black and white) which correspond (same width and height) to the visual image being used for the background. Areas that the player (or other entities) can walk on are designated by being a set color (white) and areas that cannot be walked on are another color (black). The walk mask is never shown to the player, it is only used for calculations.

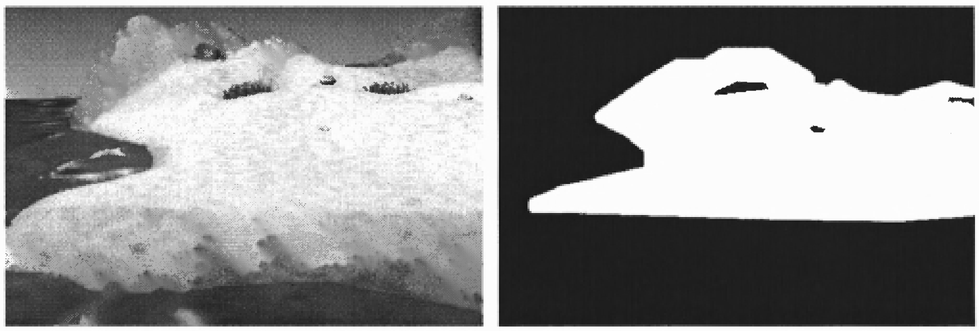

Figure 3.1 Background Image (left) and corresponding walk mask (right).

3.1.2 Tile Maps

Another common method of environment structure is the use of tile maps. Tile maps are created by a series of tiles (often represented by a sprite). The structure of the tile map is only an array (two dimensional) where the elements of the array relate to which tile is to be drawn. When the level is loaded by the game engine, a single surface is created where the dimensions are set equal to the number of tiles multiplied by the dimensions of an individual tile (the height and width of the sprite). Then the tiles are drawn to the surface for later use. The background image is pre-computed to speed up drawing time, since most systems are optimized for large blits (block transfers of information, this case it is copying one block of image data over another) over number of blits.

Tile maps have a number of advantages over background images. Levels based on tile maps will take up less space on the hard disk or media that the game is distributed on. This is because the image information for the tile set (the sprite containing all of the tiles for a given level) is generally going to be smaller than the level that is created with those tiles. Furthermore the same tile set can be used for multiple levels.

Tile maps inherently have any needed walk masks associated with the tile data. Different tiles can be determined (in the planning stage of development) to be able to be walked upon. Others can be designated as blocking which prevents entities from walking on or through the tile. In this setup, the path finding algorithms can be applied very easily. Additional information can be associated with different tiles based on the design of a specific game, such as healing tiles or tiles that act as exits to other levels.

Layered backgrounds do have an advantage over tile maps. Layered background images can be created in more artistic ways (through painting, rendering of a 3D scene or photography) which leads to more pleasing results. Tiles, even well-made tiles, result in obvious grid patterns in the environment.

Within tile mapped games, some calculations are very simple and fast. One example is the location of an entity relative to a tile. In this style of game, even though the world is broken up into tiles, movement is generally allowed to remain free. The tile that an entity lies over is easily determined by dividing the position of the entity by the size of the tile (ent.x / tile.w), or if the size of the tile is a power of two , bit shifting can be employed to make the calculation even faster (ent.x >> 6, where the tiles are 64 by 64 in size).

There are games that are made with a combination of both methods to provide the best of both styles. In this style of game, the world is defined by a tile map. The tile map image is exported from the game and given to an artist. The artist can then paint over the tiles to make a more fluid natural looking environment. The game does all calculations based on the tile map information while the players see the work of the artist.

Another style of game can use tile maps and layered backgrounds together with the parallax effect. The foreground (closest layer to player) is defined by a tile map. In this method, a clear or empty tile can be designated so other layers can be seen in the "distance". Most "platformer" style games use this method.

3.1.3 3D Height Maps

Simple 3D games that need only the most basic of environments to interact with can use what is called a height map. Height maps are generated by the game engine from adjusting the 'z' values of the vertices in the mesh that defines the level. The level itself is usually a flat series of connect “quads” (quadrilaterals, or 4 sided polygons). The height and width of the level is determined by the map editing tools, as well as the changes in heights.[1]

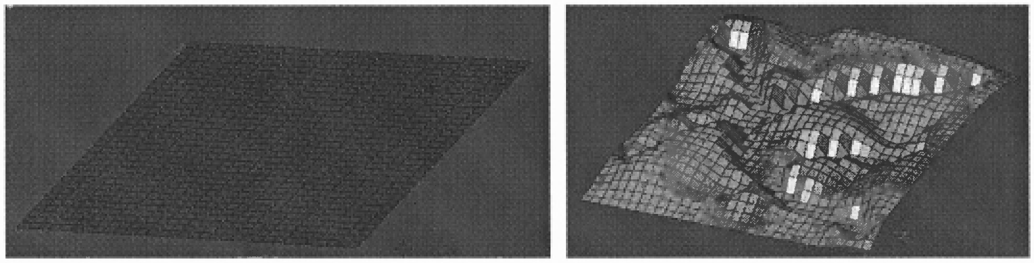

Figure 3.2 Flat grid of quads (left) and changed height map (right).

Working with height maps allows easy use of tiles in 3D games. Tiles are often referred to as textures in 3D games. Height maps also allow for very simple game mechanics for the engine. Entities that walk on the ground (as opposed to flying entities) have a very simple means to test what height the ground is at. Using the ray tracing functions (see section 7.5) the game can find the exact triangle (or quad) from the map model that the entity is above and then find the exact point of intersection with the polygon. The entity's height can then be aligned to this point of intersection.

Boundaries can be determined based on the size of the map. Entities cannot move past these arbitrary borders. For boundaries within the playing field, blocking entities can be placed (see entity collision section above) or extreme height differences or extreme normals can be used to determine if a pathway is blocked. For instance, if the Z value of the normal defined by a plane is lower than a set threshold, then the plane is considered too steep to traverse.

3.1.4 Model Objects / Walk Masks

Model objects are environments that are made up of a Model (see resource sections above) or multiple models. If the model is simple enough, the height map methods of environment collision can be used, but more often than not the complexities of the world model object is great enough to warrant the use of Binary Space Partitions (see below). When the game does not require the speed and complexity afforded by a BSP the use of a model object and a walk mask is possible.

A walk mask in 3D is a simple model object that aligns to the complex model object showing the environment. Where the environment model object is complex due to the design of the game, the walk mask is just a simple mesh overlaying the areas of the complex environment that can be traversed. Like, the walk masks in 2D, the 3D walk mask object is never drawn, it is simply kept in memory so calculations can be done to keep everything running quickly since there is less information to search through in the walk mask. Since the walk masks are inherently simpler than the drawing model, it takes up far less space than the rendered model (no texture information is needed either). The walk mask will function like a height map, providing a small set of triangles with which to search for collisions.

3.1.5 Binary Space Partitions

A Binary Space Partition, or BSP for short, is a way of organizing the geometric information of the model object in a way that makes calculations and rendering easier and faster for the game engine. Applying a BSP to a model object is a complex and time consuming process. This is done when the level designer is finished making the map and is applied by the map editing tools. The resulting BSP file can be loaded quickly (it is in a binary format) and used directly by the game engine.[1]

Creating the BSP works by finding the triangle that is the closest to the midpoint of the level. It uses the plane described by the triangle to bisect the world into two halves. The triangle is made the parent and then each half is broken in half again and again, until every triangle has been added to the binary tree. The binary tree is organized by spacial relations, so that every triangle on one half of the tree is one side of the parent and all of the others are on the other side of the tree.

The BSP speeds up collision detection since the tree can be traversed to the nearest possible triangles before any real computation needs to be applied. As calculations are applied, by either going deeper into the tree or back up the tree, only the most likely triangles are checked first until a collision is detected or the triangles being checked are out of the range of the collision.

The BSP also speeds up rendering since only the triangles that can possibly be in the camera's view (all on one side or above the nearest triangle) need to be drawn. Furthermore, the triangles can be fed to the renderer based on the distance from the camera (depth), which is how most rendering systems (like OpenGL) are optimized.

3.2 Definition File Parsing

A definition file is a plain text file containing configuration information or other data in a textual format. This kind of file is very useful for defining data needed for a project outside of code such as the initial state of a level or map. This is especially relevant to the 3D models exported from modeling tools since many export scripts utilize a text based format to ensure maximum compatibility. It is also useful for defining common elements in a game, such as player statistics which need to be tweaked often throughout development and often by someone other than the programmer.

The most basic functions used for text file parsing are made available in the ANSII standard C header file "stdio.h". Using the standard I/O file helps insure that the source code remains platform independent. A sample file parser designed to load in a list of item data is illustrated below, interspersed with explanatory comments:

```
void LoadItemListText()
{
  FILE *file;                          /*pointer to the file to open*/
```

Here the file "itemlist.cfg" is being opened for reading in text mode.

```
  if(file == NULL)
```

The pointer needs to be checked for validity. If the file does not exist or if there is some other external problem, the game will give an error output and terminate the function before any more serious problems can occur.

```
 memset(ItemList,0,sizeof(Item) * MAXLISTITEMS);  /*zero all information
in the list*/
```

The array ItemList is an array that stores all of the information related to items in this game. The function will put information into this list for later use.

```
  while((fscanf(file, "%s", buf) != EOF)&&(i < MAXLISTITEMS))
   {                                        /*go through the entire file,
line by line*/
      if(strcmp(buf,"#") ==0)
      {
        fgets(buf, sizeof(buf), file);
        continue;/*ignore the rest of the line.*/
      }
```

This code allows for comments in the file that is being parsed. Any line that begins with a '#' symbol will be ignored.

```
     if(strcmp(buf,"end") ==0)
      {
        ItemList[i].index = i;
        i++;
        continue;
      }
```

Whenever the keyword "end" is encountered, the index is updated, ending any data entry for the previous item index.

```
     if(strcmp(buf,"uname") ==0)
      {
        fgetc(file);  /*clear the space before the word*/
        fgets(ItemList[i].uname, 80, file);
        c = strchr(ItemList[i].uname, '\n');
        if(c != NULL) *c = '\0';    /*replace trailing return with
terminating character*/
        continue;
      }
```

In order to read a string in, some special steps are required to ensure the string is formatted correctly for use with the program. fgetc will read a single character from the buffer (in this case the return value is ignored). fgets will read the remaining string from the file. This will work for single words or whole lines. The file is formatted with each line ending in a return character '\n', but the string in the game needs to end with the terminating character '\0', so strchr is used to locate the occurrence of '\n' so it can be changed to '\0'.

```
        if(strcmp(buf,"swidth") ==0)
         {
           fscanf(file, "%i", &ItemList[i].swidth);
           continue;
         }
```

For integers and other standard data types, fscanf can handle feeding information into program variables with ease. It uses the same conventions as the printf family of functions, except fscanf reads information from the stream, as opposed to writing to it.

```
      }
      fclose(file);
      NumItems = i;
    }
```

Once all the information is read from a file, it is important to close it. Note that parsing information from text files is only done when a game is being developed. The text files are easy to edit, but take time to read, and can grow to very large sizes. When a game's development is almost completed and ready to be released, it is common to convert the files into a binary format that can be loaded quickly with a single “fread” call in binary mode.

CHAPTER 4

RESOURCE MANAGEMENT SYSTEMS

One of the main responsibilities of a game engine is the fast management of the resources needed by the game. A game will utilize a number of different resources to help craft the virtual world. Games will need access to window systems, 2D animated images called sprites, 3D models, sounds and descriptors for the elements of the game.

The resource management systems of the game engine are classes that interface with the rest of the game engine systems. Other systems will request a resource and then the manager will load it into memory and pass a reference to it on to the requesting system. When the resource is needed, the manager will utilize the resource for instance, drawing an image on the screen or playing an audio file. When the resource is no longer needed, it has to be freed from memory.

Implementation of resource managers is not difficult. Starting with section 4.1 below, different subsections will be detailed. Common to all systems is the need for efficiency. One key feature that will improve performance of a resource manager is designing the system to handle large numbers of resources in as fast a way as possible.

There are two ways that a large, variably sized list is maintained in a program: one is to use a linked list, the other is to use a large static sized array. Linked lists are popular since it allows for new resources to be allocated on the fly as needed by a program, but the linked list method has a serious drawback that makes them too costly to use for a game engine.

Linked lists carry with them an overhead that is costly in time. Traversing a linked list is slower than referencing an array element by its index. Additionally, adding new nodes or deleting nodes from a linked list is even slower. Every time a program requests more memory from the operating system or gives memory back to the operating system, there is a delay as the request is added to the operating system's queue of things to do.

With a static array the overhead for allocating and deallocating memory is gone, but the cost is flexibility and memory. With a static array there is an upper limit to the amount of a given type of resource. A good design will set this at a safely high upper limit, but not too high. This also provides a safety net for the game. With a linked list, it is possible to run out of memory while a game is running. With a static array the upper limit is what is requested when the program is launched. The game will not run on a system that does not have enough resources available. For these reasons static arrays are used for most game engines.

For each type of resource managed, there is a data structure (see below) used to describe it. Additional variables are added to these data structures for the resource manager. The filename of the resource and the reference count.

The filename of a resource is important so redundant copies of the same resource are not loaded into memory. It is possible for multiple calls to load the same resource. By first checking to see if the resource being requested (by filename) has already been loaded, a pointer to the existing resource can be returned instead of loading a new copy.

This also helps to make the greatest use of the limited amount of a resource. The resource maximum applies to the maximum number of unique instances of a given type.

The reference count, often abbreviated RefCount or uses, is a tally of the number of objects or entities in the game using a particular resource. When a resource is no longer used by an object or entity it will call the Free function for that resource. If the reference count drops to zero, only then is the resource actually freed.

Another method for streamlining the resource management of a game engine is to pre-cache and hold resources. Any given level, the map or stage of a game will have a list of known resources that are used in that area of the game (such as the interior of a building, an open field or an asteroid belt). The game engine will load a copy of every single resources that may be used by the level. This way, there is an initial up front load time, but while the game is running, no additional disk loads are needed, except in rare cases where a resource was overlooked. With this initial load of each resource, the reference count will never reach zero. The resources are purged only when the level is closed.

4.1 Sprites

Sprites are animated 2D graphics in a game. They are used to represent anything in a 3D game that cannot be represented by 3D graphics. This can be due to the subject being inefficiently represented by 3D objects in the case of complex things such as foliage on trees or impossible things such as perfect spheres. Sprites are the only type of graphics in 2D games.

Sprites are animated. To show animation a sprite is composed of more than one image, each representing a different snap-shop of motion called a frame. Games use two common methods for representing sprites : individual images representing each frame of animation, or a sprite sheet containing all of the frames of animation on one image. The sprite sheet method is the one implemented here for a few reasons: It keeps the frames of animation easier to track, it lets compression work on the entire sheet which may be smaller than the individual files, and there is less to keep track of in code.

In the data structure below, the full sprite sheet of a sprite is stored with the information needed to gain access to an individual sprite.

```
typedef struct
{
  char filename[80];
  SDL_Surface *surface;   /*for use with SDL*/
  int w, h;         /*the width and height of the frames of the sprites,
not the file itself*/
  int framesperline;        /*default is 16*/
}Sprite;
```

The filename of the sprite is stored so that multiple calls to load any given sprite will not result in redundant copies of the same sprite in memory. All will work from the same sprite.

Within the data structure of SDL_Surface, the surface's height and width (in pixels) is stored. For the sprite to work, the height and width of each frame of animation is needed as well. The draw functions will also need the number of frames per line. By default in this game engine, it is set to 16, but it can be changed as needed.

When a single frame needs to be drawn the position within this surface is determined by:

$$\text{X coordinate} = \langle\text{frame}\rangle \,\%\, \langle\text{framesperline}\rangle * \text{width} \qquad (4.1)$$

$$\text{Y coordinate} = \langle\text{frame}\rangle \,/\, \langle\text{framesperline}\rangle * \text{height} \qquad (4.2)$$

The individual frame can then be copied from the sprite sheet to wherever it is needed.

Below is an example of a sprite sheet:

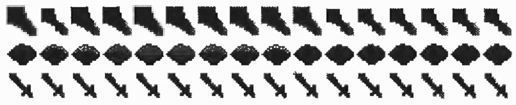

Figure 4.1 Sample sprite sheet.

4.2 Models

Models are the collection of 3D information for an object in a game. Models are made with artistic tools such as 3D Studio Max or Blender. A model is composed of a list of triangles. The triangles are created and arranged by an artist to represent a 3D object needed for the game. The object can be something simple like a rock or desk, or more complex like trees and people.

Model Objects, like sprites, can be animated. There are several ways of implementing object animations. The most common are animation by frame where there is a different model representing each frame of animation. There is Key Frame animation where only select frames (Key Frames) are loaded and extra frames needed by the game are interpolated between key frames. There is Armature animation where a skeletal system is created which maps triangle vertices to bones in the armature. In armature animation only the skeleton is animated (with key frames) and the positions of a model object's vertices are computed based on the poses of the armature. This game engine uses Key Frame and Armature animation.[3]

4.3 Triangles

The Model objects in the game engine are composed of triangles. Triangles are the simplest geometric shape and cannot be folded by rearranging the points of the triangle. The points of the triangle are represented by vertices (below). Each triangle is composed of the three vertices that make up its shape as well as the three vertices that map to its texture.

Triangles are mapped to a 2D plane called a UV map. The UV map allows the game engine to align the triangle's face to a triangular image while the triangle is being drawn (rendered) by the game engine. This allows for the 3D objects to have more texture than a single flat color.

```
typedef struct
{
  GLuint vindices[3];          /* array of triangle vertex indices */
  GLuint tindices[3];          /* array of triangle texcoord indices*/
} Triangle;
```

In the game engine, a triangle needs to keep track of which vertices in the model vertex list are used by the triangle and which texture coordinates in the model are used by the triangle. Since the mesh of an object is composed of a continuous mesh of vertices connected by a triangles, most vertices are part of many triangles. The same may be true for texture coordinates, but due to the needs of the layout of a 2D image, the same triangles that may share vertices and edges on the mesh, may not share the same vertices and edges on the texture. It is possible to have different textures applied to different triangles on the same mesh as well.

4.4 Vertices

Vertices are points in space. They are represented by 3 floating point values which describe the position of a vertex along the X, Y and Z axes. Depending on developer preference a vertex is sometimes represented by an array with 3 elements and sometimes as a structure (named Coord) with 3 members (x, y & z). The array method is more compatible with matrix operations.

Vertices are a basic building block for 3D constructs. 3D models are composed of triangles, which are in turn composed of vertices. Movement and animation in 3D models is accomplished through the transformation of the positions of the vertices that make up the triangles.

4.5 Armatures

Armatures are frameworks of bones that mimic the function of a model's skeleton and muscle structure. Each bone is associated with some number of vertices in the model. As a bone is moved and posed, the linked vertices are moved along with the bone. This is used in animation extensively to make the job of the animating artist easier.

In the game engine, the use of an armature has a number of advantages over simply using a series of models for animation. It is the armature that is animated, not the model object. With an armature, a model only needs to have one pose, the rest position. For other frames of animation, the pose of the model is calculated based on the poses of the armature.

In all cases, the armature will have less information associated with it than the model itself, so having many frames of animation (each stores the position and rotation of each bone) takes up far less memory than an animated model (which stores the positions of every single vertex in the model). Whereas a typical model will have at least 5000 vertices, an armature will have on average 50 to 60 bones, a significant savings on memory.

Armatures exist as separate data structures from models. This means a single armature can be applied to all models that fit a basic size and type. For instance, a human armature can service any human shape model, be it a soldier, a villager, a robot, or even a giant. A single armature can be created for a horse or other pack animal. This cuts down on the development time of the artist so only a single animation sequence needs to be created.

Armatures also provide meaningful information about the model in question. Bones are named and have position, rotation and scale maintained separately. This can be useful for putting additional objects on a model, like putting an apple in the hand of a player, or a helmet on the head of a knight, or a glowing light special effect over the eyes of a monster. This can cut down on the memory needed and time for an artist as well since a new model does not need to be created for every possible combination of items being held or equipment worn.

Many game engines will use the information of the armature to create special effects as well, such as "Rag Doll Physics" where an unconscious character can be moved and bent according to the armature. Other effects made possible with armatures are clipping and snapping to real word objects. Hand bones can be placed directly on the rungs of a ladder, foot bones can be placed directly on the steps of stairs.

Below is the data structure of the armature. It makes use of the Bone structure and the Pose Structure.

```
typedef struct
{
  char    name[80];                 /*name of the armature*/
  Bone    *root;                    /*pointer to the bone that is the
root.  An armature cannot have more than one root bone.*/
  Bone    *bones;                   /*the list of bones in this
armature*/
  Pose    *frame;                   /*the list of poses for each frame*/
}Armature;
```

This basic structure for the armature is simple. It has a list of bones and a list of posses. This way the bones can be accessed for their initial states and posses can be accessed by frame for the difference from the rest state. Together these are used to transform the associated vertices from their initial states.

Next is the data structure for the bones. It is used to keep track of the relationships between the bones and the bone's initial state.

```
typedef struct
{
  char    name[80];                 /*name of bone = name of associate
vertex group*/
```

```
  char    pname[80];                /*name of parent bone*/

  char    cname[MAX_KIDS][80];      /*names of children*/

  float   location[3];              /*base of this bone*/

  float   matrix[4][4];             /*the matrix for the initial
rotation*/

}Bone;
```

/*some entries omitted for simplicity*/

The name of the bone is important. It is used to search by, referenced by the bone poses, and to determine which vertices are associated with the bone. Bones are organized in a tree structure, with each bone having a single parent and the possibility of many children. The transformations of the parent are passed on to the children. Location is the X, Y & Z location of the bone in space relative to the overall armature. The matrix is the quaternion rotation matrix. For every pose frame of the armature, each bone has its own associated information:

```
typedef struct

{

  char    name[80];                 /*seams redundant, but needed to
search by*/

  float   loc[3];                   /*offset location from original
state*/

  float   rot[4];                   /*quaternian rotation of bone*/

  float   scl[3];                   /*the scale of this bone*/

}BonePose;
```

Bone poses are simple. The pose frame contains the change in location, rotation and scale from the bone's initial rest state (given by the bone information).

To help organize the armature, some systems employ what are known as Bone Groups. Bone Groups are sets of bones that can be set to different frames than other bones in other groups. Some bones can be members of multiple groups. This allows the animators and designers to create custom animations and to blend them naturally.

For example, the lower body can be added to a "legs" group and the upper body can be added to a torso group while the head can be made its own group. The legs could be set to cycle through the run animation, while the torso can be set to a firing or action animation, while the head can be given a point to focus on. If there are bones shared between the groups (such as the hips and neck) then these bones would take the average position dictated by both groups.

Another technique available for armature animation uses the concept of layering. Separate actions can be "layered" on top of each other with different priorities to get an even greater range of possible armature positions. In the previous example, additional layers can be applied to the running and attacking player. If the player is taking damage, then the pain animations can be layered on top of the other actions, so the player can react to damage (with the effected bones spread out among several groups) while still performing other actions. In some modern game engines it is possible to add layers that react to environments (syncing with the group, terrain or stairs), include facial animations, and interact with other entities (shouldering past someone in a crowd) all at the same time.

In order to have bones take into consideration multiple groups and layered actions, the same process is applied for a vertex that has more than one bone as a parent. The weights for all poses need to be normalized (like a vector would be normalized) before the deformation (process of changing the vertices based on the bones) takes place. All transformations that are applied are scaled according to their own component, which is a percentage of the originally intended motion.

4.6 Actions

The frames of animation for an armature (or model or sprite) generally equate to specific actions as set frames. For example, a walking sequence can be performed between frames 40 and 56 of the animation. These sequences need to be planed out before-hand so the programmer can call the appropriate frame of animation at the desired time. As projects grow, this hard-coded pre-planning becomes tedious and time consuming to update as project parameters change and adjust. In order to accommodate variations in the plans and artistic interpretation, actions can be used to help organize the animations.

When using actions as a descriptor or definition file, each file usually corresponds to a single sprite, animated model or armature. When absent, the game engine will fall back on a generic template. The action file will list actions by name, the starting and ending frames for that action, the frame rate for stepping through that action and how that action is handled when the last frame is drawn. When the animation is complete the game engine needs to know what to do next (assuming a different action has not been called that short circuited the action in question). Here are some possible actions that the engine will perform:

- Loop – This tag will have the animation restart when completed until a new action is specified.
- Oscillate – This tag will have the frames begin to count back to the beginning once the end is reached and then back to the end once the beginning is reached until a new action is specified.
- Idle – This tag will set the next action to an idle state when finished.

- Hold – This tag will have the animation hold on the last frame after the sequence is complete. This tag is useful to end a dying animation, where the last frame is the dead pose.
- Chain <action> - This tag will switch the animation to the specified action after this one is complete.

With actions and definition files (see section below), the controls of the game begin to be broken out of the source code for the game engine. As custom actions can be referenced in a definition file and the appropriate animations can be added to the action file and animation, the game can produce new content without the need to update the source code.

4.7 Entities

Entities are any and all objects in a game that can act or be interacted with. This distinguishes entities from other objects in the game that are there for purely aesthetic purposes such as particle effects (fire) and immovable objects (the world itself). In different game engines, entities are sometimes referred to as "Objects", "Actors", "Tokens" or any number of alternate terms. These terms are all functionally synonymous.[2]

Entities are similar to other game resources in that they can be created and destroyed on the fly and need to be maintained quickly. Entities themselves use many of the other resources in the game. A typical entity will use a Model or a Sprite to represent itself. Entities will use different sounds throughout the life of an entity.

Unlike other resources, entities are unique. A single entity cannot be shared between multiple entities. The information contained within an entity may be redundant or equivalent to another entity. For example, there may be several bullet entities that all look the same and move the same, but which all have different positions in space and different entities that fired the bullets. Entities do not use reference counts, but a single "used" flag. This flag is set to one if the slot in the entity resource array is in use or zero if it is available.

The other main distinction is that entities can function differently. All entities have a function pointer called a "think" function. All think functions have the same parameters and return nothing. In this standard C implementation all entities need to be passed a pointer to itself. In a C++ implementation this would be accomplished with a virtual function which inherently has access to its member data. The think function is used to give unique behavior to different entities.

Every entity in a game is structurally the same thing. They are all described by the same data structure. Each entity gains its distinction via different data and different think functions. The various types of entities can either be hard-coded or gained from a descriptor or configuration file. All think functions need to be either hard coded or extended through an embedded script system (such as Python or Lua). For the sake of speed this game engine hard-coded the few entities that were used.

Every entity has the opportunity to update itself once every engine update. This is done by calling the think function of the entity (if it has one). The specifics of a think function will vary, but most entities will have their movement evaluated, collisions with other entities checked for, and animations of the model or sprite object incremented.

Below is a sample entity structure. Only portions of the entity structure that are relative to this section are included. Inconsequential maintenance information has been omitted as well as information that would pertain to a specific style of game.

```
typedef struct Entity_T

{
```

For many games, different entities can have relationships to other entities. The two most common relationships are owner and target. The owner of an entity can be the entity that created it (such as in the case of a projectile being created by the weapon that fired it), or a submissive relationship (such as in the case of a Leader being the owner of subordinate officers).

The target pointer can point to another entity to describe the focus of this entity. The focus can mean the literal target for a projectile (for instance, a seeking missile) or an entity that should be followed. The exact meanings of these two useful pointers will change (or be ignored) as needed by individual entities.

```
 struct Entity_T *owner;              /*pointer to the owner entity*/

 struct Entity_T *target;             /*pointer to the target entity*/

 int used;                            /*for reference in resource
management*/
```

The model and texture or the sprite image (depending on whether this entity is a 3D or 2D object) is the reference to the model or sprite resource. These are passed to the drawing functions for each entity.

```
  Model *model;             /*the model for the entity, if it has one*/

  Sprite *texture;          /*the texture for the entity*/
```

The think function is a function pointer which can be changed on the fly as needed by an entity. The think function is what gives each entity its' unique behavior as described above.

```
 void (*think) (struct Entity_T *self);      /*called by the engine
every so often to handle input and make decisions*/
  Uint32 NextThink;                                  /*used for how often
the entity thinks*/
  Uint16 ThinkRate;                                  /*used for
incrementing above*/
```

Each entity references the graphical information in the graphical resource managers (models & sprites). Since many different entities can all reference the same graphical information, some information related to the graphics needs to be maintained by individual entities. Frame information (for animation) needs to be maintained separately by different entities. This is necessary since different entities that may use the same model will be at different points in an animation based on different actions.

```
float frame;                /*current frame to render*/
int fcount;               /*used for animation, the loop variable*/
float frate;                /*how often we update our frames*/
Uint32 framedelay;        /*ammount of delay between frames*/
```

The position, rotation, scale and the vectors for changing position, rotation and scale will vary on an entity by entity basis. Here the physics convention for position (s) is used.

```
Coord s;                    /*screen coordinates*/
Coord a;                    /*acceleration*/
Coord v;                    /*vector values*/
Coord rotation;         /*the rotation for the 3D*/
Coord rotVect;          /*the vector for rotation*/
Coord scale;             /*the scale of the 3D model*/
Coord scalVect;        /*the scaling vector*/
```

To handle collision detection, an axis aligned bounding box is maintained for each entity. The bounding box has the X,Y &Z offsets from the location of the object (s) with the Width, Height and Depth dimensions of the box to describe the quadrilateral.

```
 BBox box;                                   /*the size and offset of an
axis aligned bounding box*/
```

Health (how much damage that an entity can sustain before dying) is just the first of many variables that are in place for keeping track of game specific data.

```
 int health;                                 /*this is the entity's
health*/
. . .
}entity;
```

4.8 Particles & Temporary Entities

In any game engine, anything that can act or be acted upon is designated an entity. Entities can be made up of one or more sprites or 3D models. Entities require more space and processing power than is required for some game visuals. To achieve special effects in games, temporary entities (sometimes referred to as Particles) are often used in place of entities.

Particles are a variation on Entities. Whereas entities can exhibit complex behaviors based on individual think functions and update functions, all particles follow the same update function. This cuts down on the processing time for particles. For particles to be effective, the game engine needs to be able to produce large numbers of them. Particles also differ from entities in that all particles exist on a timed lifespan. As soon as the lifespan decrements to zero, the particle is freed and made available for redefinition. Examples of particles in games include ambient weather effects such as rain or snow, fire and smoke as well as sparks and additional detailing for explosions.

The data structure for particles needs to be set up to allow for all possible particle types supported by the game engine. Here is an example taken from a 2D game engine:

```
typedef struct Particle_T
{
  int  type;                                    /*type of particle*/
  float sx,sy,sz;          /*coordinates of where the particle is getting
rendered*/
  float vx,vy,vz;         /*vector of particle*/
  float ax,ay,az;         /*how is the particle accelerating*/
  Uint32 Color;          /*what color is the  particle?*/
  Uint32 Dcolor;        /*What color does the particle need to be before
it dies.*/
  Sint16  R,G,B,A;      /*Color vector for red, green,blue and alpha
respectively*/
  Sprite *sprite;
  int used;
}Particle;
```

Note how this particle data structure allows for the color of the particle to be defined with a color vector or color destination. The color vector is the amount of change applied to each channel of a color (red, green, blue and alpha) in each frame that the particle is updated. The color destination is the target color that the particle needs to be by the time the particle's lifespan expires. This is accomplished by a simple linear interpolation based on the remaining lifespan of the particle at each frame.

This 2D game engine particle system allows for the 3rd Dimension. Using simple tricks for drawing the particles (allowing the z value to effect the y values in a set ratio) a 3D effect can be generated with the motion. In some 2D games that simulate a 3D environment (while not actually being 3D) this effect is very helpful for maintaining that illusion. If this were a 3D game engine particle structure, additional pointers to 3D models would be available.

All particles are updated from one function. The UpdateParticles Function is called once a game frame to update and draw all of the particles in the game. This function checks each particle in the massive static array for activity, updates the coordinates and color (and any other factors of the particle system), and then draws each particle before moving on to check the next one.

Particles gain their form and function from various particle emitters. A particle emitter can be one of two types, an instant particle emitter or an entity particle emitter. Instant particle emitters are functions that get called only once for any particle generating event. These are useful for sparks, debris and explosions: events that originate at a single frame and are not sustained. For lasting effects, such as rain or a burning camp fire a particle emitter entity is needed. These are entities that incorporate particle generation into their update functions.

The way particles are emitted can define the overall shape and form of a particle system. Particles can be emitted from a single point, a plane or area. Particles emitted from a single point can create the effect of a spray (with a single direction, slight random variation and an acceleration down (gravity), explosions (with a completely random direction, slight acceleration due to gravity), or a drip (very slight random vector down, with gravity playing a part). Planes (where the starting point of a particle can be any random point lying within the plane) can generate weather effects (where the plane is in the "sky" of the game, particle acceleration is set to gravity, and a random "wind" vector is applied to lateral movement). Areas (some volume or enclosed shape where particles can be generated from any random point within the area) are used for localized effects where a single point will not suffice such as a torch (area is the top of the model, particles accelerate away from gravity) or a fog effect (particles are given a random vector and are slow moving, not effected by gravity).

Temporary entities are special case versions of particles. They are intended to exist for a set amount of time and do not think on their own. They just move according to some preset parameters. There are many types of temporary entities (particles are a subset of temporary entities), but the most common is a link. Links require two positions to be linked between. They are created by drawing some effect over the distance between those two points. The most common example is a bolt of electricity. Electricity will arc between two points. To simulate that effect in a game, an electricity graphic is either stretched between the two end points or a smaller graphic is tiled between two end points. Stretching works well in a 3D game engine, where the graphic (almost always animated) has a resolution that is close to the distance between the two end points. Where the results of stretching would produce ugly results, tiling (placing a series of sprites end to end in succession) can produce better results. With tiling, the sprite needs to be made with the intent in mind by the artist. Some artistic programs have tiling tools to facilitate this.

CHAPTER 5

3D MODEL RENDERING

One of the most basic challenges for any 3D game is to create the method for rendering the 3D information. This function "DrawModel" shows the simplified method for feeding OpenGL the model information so it will look correct. Note that the information for the mesh is loaded from a file exported from an artist's tool (blender).[3]

```
void DrawModel(Model *model,Sprite *texture, Coord pos, Coord rot,Coord
scale, float alpha, int frame, int drawMode)
{
  int i;
  GLMgroup* group;
  GLMtriangle* triangle;
```

These pointers are used to create a shorthand access to the part of a complex data structure that is needed for rendering.

```
 glEnable(GL_LIGHTING);

  glBlendFunc(GL_SRC_ALPHA,GL_ONE_MINUS_SRC_ALPHA);

  glEnable(GL_BLEND);
```

These enabling functions tell OpenGL how the rendering of the information that is about to be sent to it should be handled.

```
 glPushMatrix();
```

OpenGL works on a stack principle. Any operation ordered on level of a stack is automatically applied to lower levels of the stack, but not on higher levels.

```
if(texture != NULL)
{
  glEnable(GL_TEXTURE_2D);
  glBindTexture(GL_TEXTURE_2D,texture->image);
}
```

The above code checks to see if there is an associate texture to use when rendering the object. The below code will move and rotate and scale our object to the dimensions determined by the entity that owns this object. This way multiple entities can use the same object model but show them in different positions.

```
glTranslatef(pos.x,pos.y,pos.z);
glRotatef(rot.x, 1.0f, 0.0f, 0.0f);
glRotatef(rot.y, 0.0f, 1.0f, 0.0f);
glRotatef(rot.z, 0.0f, 0.0f, 1.0f);
glScalef(scale.x,scale.y,scale.z);
```

The section below will step through the object by groups of triangles. The groups are organized into a linked list (a structure that is normally too slow to use in rendering and has been removed in more processor intensive games).

```
group = model->object[frame]->groups;
while (group)
{
  glBegin(drawMode);
  for (i = 0; i < group->numtriangles; i++)
  {
    triangle = &model->object[frame]->triangles[group->triangles[i]];
```

This loop traverses every triangle in the model that is a part of the group. It then feeds OpenGL the information with the three functions glNormal3fv (which describes the normal of. a face at the vertex, used for smooth or hard shading), glTexCoord2fv (which gives the texture UV coordinate to use) and glVertex3fv (which gives OpenGL the vertex in question). This is done three times, once for each vertex that makes up the triangle.

```
    glNormal3fv(&model->object[frame]->normals[3 * triangle-
>nindices[0]]);
      if(texture != NULL)glTexCoord2fv(&model->object[frame]->texcoords[2
* triangle->tindices[0]]);
      glVertex3fv(&model->object[frame]->vertices[3 * triangle-
>vindices[0]]);
... /*the other two triangle points omitted due to redundancy*/
    }
    glEnd();
    group = group->next;
  }
  glPopMatrix();
```

Once OpenGL has been given the mesh information, the stack is popped so that any changes made for this object do not corrupt any other objects. For this reason, all features that have been enabled are disabled.

```
  glDisable(GL_LIGHTING);
  glDisable(GL_BLEND);
  glDisable(GL_TEXTURE_2D);
}
```

CHAPTER 6

ENTITY SYSTEM FUNCTIONALITY

To create the effect of a ballistic projectile entities need to be updated based on multiple factors. Game engines are active, so everything needs to be updated continuously. For the most basic movement of a projectile, an entity needs a vector, the x, y and possibly z components of a vector, or an array of three floating point values. For every update loop of the game engine, every entity will have its vector's value added to its' position. This creates the motion.

Basic motion with a single velocity vector works fine for many projectiles and other simple entity motion types. This also works well for flying and hovering objects, since the think functions for the entities can determine the vector for motion. The think function can calculate the direction that the object should be moving in and then set the vector based on the intended velocity.

For self propelled projectiles such as rockets, this still works since a rocket would move so fast that simple directed motion is perfectly acceptable. Gravity can be ignored since any drop off due to gravity may be negligible. For slower objects, or any entity that would have a noticeable drop due to gravity, an additional step is needed.

Acceleration vectors can be applied to the motion vector. By adding an acceleration vector to the velocity vector before it is in turn added to the position of the object, acceleration can be emulated. When dealing with multiple forces, such as gravity and propulsion, accelerations can be summed before adding them to the vector. Often times, in order to maintain more meaningful control over the motion of the objects in the world, gravity is maintained as a separate, global variable (In games, not only is the magnitude of gravity variable, but the direction of gravity is also not a fixed constant). This way, if gravity ever gets adjusted or turned off, it has a single point where it needs to be adjusted.

6.1 Bouncing

Some types of projectiles need to be able to bounce off walls and objects that do not take damage. There are different approaches to doing this depending on the setup of the world and how accurate the game needs to be. The simplest way to create a bounce effect is applicable when the world is laid out in a grid, or all walls are axis aligned (planes fall in only two of the three possible axis). In this case, a simple reflection of one of the vector components is needed. For an example in 2D: a projectile that is moving in the positive Y direction hits a ceiling or floor, then the Y component of the vector is negated (multiplied by -1).

For more accurate reflections, the game engine will need to perform more calculations. The following formula describes how to calculate the resulting reflected vector R from the initial (normalized) vector V of the object and the normal vector N of the intersected surface.

$$R = V - (2 * V \text{ [dot] } N) N \tag{6.1}$$

[dot] is the dot production operator.

The code for calculating the reflected vector follows. Note that the vector created will be a normal vector.

```
ReflectVector(Coord V, Coord N ,Coord  R)
{
    Coord temp;
    VectorScale (N, (2 * DotProduct(V,N)), temp);
    VectorSubtract(V,temp,R);
}
```

The Coord data type is an array of three signed floating point numbers. This can represent a point, a vector or other circumstances where three floating point numbers are appropriate.

6.2 Tracking

For some types of entities, being able to track and follow other entities is essential. Homing missiles and computer controlled players may need to be able to adjust their course based on the position and motion of a target entity. This simple problem can become very complex depending on the desired behavior.

For an AI opponent tracking a player, the AI agent will need to have a pointer to their target. The simplest means to hone in on the player (or other target), is to calculate the desired vector of movement by comparing the positions of the target and the tracking entity. This works well in open space, but falls short when walls and other obstacles potentially lie between the target and tracker. The following is a simple algorithm for getting a very direct path to the target, regardless of the obstacles in the world. This algorithm requires that the past few positions of the target be maintained by the target, creating a trail that can be followed. The longer the trail, the more likely the hunter will not lose track of the target. The algorithm is:

1. Start with the current position of the target.
2. Perform a direct trace to see if any obstacles obstruct the direct path to the target.
3. If the path is unobstructed then calculate movement vector to the target.
4. If the path is obstructed, set the next most recent position as the new target position and perform step #2 again.
5. If all previously positions are obstructed, then break off the pursuit.

This method works well for direct tracking, but if the motion of a guided missile is desired some additional work must be done. In missile guidance, there is a limit to how drastically the trajectory can be altered for a given update. This threshold is maintained as a variable for the entity (thus allowing degrees of effectiveness for homing).

Following the same steps as shown above to get the target location, the missile agent then needs to perform additional computation. The positions are used to calculate the optimal (direct) vector to the target. The angle of rotation is calculated that would need to be applied to the missile's vector to match the new vector. If this angle is greater than the threshold then the threshold is used instead. The vector of the projectile is then rotated by the desired angle. For very smart missiles, the game can even allow the missile to track the trail of a target.

CHAPTER 7

COLLISION DETECTION

In any active game entities in the game will need to directly interact with other entities and the game world. This problem has various solutions for different situations depending on the style of the game and how accurate the collision detection needs to be.

7.1 Distance Collision

The most basic method for collision detection between two entities is based on using a size. In this method, entities need to have a center point and a radius. Collision between two entities occurs when the distance between the two entities is less than or equal to the sum of the radii. This is based on the usual distance calculation : sqrt((x1 – x2)^2 + (y1 - y2)^2) <= r1 + r2. The formula needs to be modified for the purposes of efficiency since it contains square roots. Since the actual distance does not need to be known, We can modify the formula to make the calculation simpler by squaring: (x1 – x2)^2 + (y1 – y2)^2 <= (r1 + r2)^2. Note that in the case of a 3D world representation the component (z1 – z2)^2 is added to the equation. In cases where this much accuracy is not needed, but additional efficiency is needed, the squares can be completely ignored. The result is less accurate, but good enough for many game types. This method is most useful when dealing with spherical or very small objects.

7.2 Axis Aligned Bounding Boxes

For most game purposes, an axis aligned bounding box works best. A bounding box is a close approximation of the size and shape of an entity in the form of a box. This box is never rotated in any way regardless of the rotation and actions of the entity. This way assumptions can be made about the orientation of the bounding boxes and the math for collision calculations can be made much simpler.

The bounding box is represented by a structure with four values, x, y, w(witdh) & h (height). When dealing with 3 dimensions z and d (depth) are added as well.

```
int Box_Collision(AABBox box1, AABBox box2)
{
  if((box1.x + box1.w >= box2.x) && (box1.x <= box2.x+box2.w) && (box1.y
+ box1.h >= box2.y) && (box1.y <= box2.y+box2.h) && (box1.z + box1.d >=
box2.z) && (box1.z <= box2.z+box2.d))
    return 1;
  return 0;
}
```

The above function only works when two entities can be represented by axis aligned bounding boxes. If an entity is just represented by a size-less point, a bounding box can still be used, but the w, h and d values are left at zero.

It is important to note that in the entity data structure, the position of the entity is maintained separately from the x,y,z values of the bounding box. The x, y and z values of the bounding box represent the offset of the box *from* the entity position value. This way, duplicate information does not need to be maintained.

It would seem that the size of a bounding box can be calculated from the maximum size of the entity graphic (3D model or 2D sprite). For many games the relative size of an entity's bounding box can be determined by a design choice. In some action games dealing with large number of hazardous entities (such as mines or bullets), it is common to make the player entity's bounding box smaller than the player and therefore make the game more forgiving for the player's mistakes.

7.3 Complex Entity Collisions

It is not uncommon for games to incorporate complex entities or entities that are actually made up of multiple entities. An example of a complex entity may be a large snake. The snake can take on a variety of shapes as it coils and bends. The shape of the snake will change drastically as it does this. This makes the use of distance collisions or bounding boxes very inaccurate most of the time (since most of the bounding box will be empty space). In cases such as this, more complex arrangements can be made.

For snake-like objects, a series of bounding boxes or a series of empty entities can be used. Each segment in the list can have its own bounding box or its own entity, where any collisions and actions are passed on from / to a parent entity which does the thinking for all of the entities and handles collision events.

These types of complex entities are very important for games where specific body parts need to be identified, such as shooting games (where shooting a leg can disable) and fighting games (where the collision detection needs to be very precise). In a fighting game, each segment of a fighter's body (hands, forearm, upper arm, body, head, thigh, shins and feet) all need their own separate collision checks. This handles cases where both players are attacking. Thus the "fist" bounding box will collide with different bounding boxes on the opponent which results in different effects, such as a block or hit, with more or less damage.

With the use of armatures, the motion of the complex series of bounding boxes can be established by the poses of the bones in the armature. Additionally, the control points (the head and tails of the bones) can be used as center points for distance collision detections.

7.4 Path Finding

In any game where computer controlled entities need to determine their own maneuvers, or in games where the player will designate a target location for their units to move to, it is important to have fast and reliable path finding. There are many methods for determining a path for an entity to follow, from simple crash & turn algorithms that will result in a viable path but without reliability, to more exact algorithms such as the commonly used A* (pronounced ay-star) algorithm.[5]

Crash and turn algorithms have the entity move in the vector towards the goal location. When an obstacle is encountered, the entity will change direction until its path is no longer obstructed and continue to move. At some point when the desired direction (towards the goal) is no longer obstructed, that path is resumed. If the new path does not result in an opening towards the goal, then the entity will need to backtrack until a path opens up.

This method will result in a viable path in many cases, but there are possible layouts where it becomes inefficient. The method is used best when there are only a few obstacles in an open area and the obstacles have a convex shape. Concave shapes and maze-like layouts will result in needless backtracking. Depending on the threshold of how long an entity will traverse a path before backtracking, some complicated layouts may be almost impossible to traverse. When this method is used, it tends to be used only on small scales and often in conjunction with other more complicated algorithms (such as when the crash-and-turn method fails).

The A* method is the basis for most path finding algorithms. The A* works best in a tiled environment (such as a chess board) where each tile is connected to only the 4 adjacent tiles (or less when it is on an edge of the board). Here is a simple explanation of the A* method, explained plainly, with descriptive variables instead of the ones traditionally used (p, g, h, etc):

Two lists, Unchecked and Checked, are required:

Unchecked – A list that contains tiles that will be checked for the path. The list contains the tile (x, y), its parent (x,y), the number of steps it took to get to this tile and the estimated number of tiles left to check before the goal.

Checked – The same type of list as unchecked except this one keeps track of which tiles have already been checked.

The following steps are then taken:

1. Given the starting tile, add it to the Unchecked list with its estimated distance to goal [d = sqrt(dx^2 + dy^2), for example].
2. Go through the Unchecked list, searching for the first tile with the smallest estimated distance to the goal.
3. Check to see if this tile is the goal tile (if so, then finish). If not, add the children

of this tile (that have not already been placed in the Unchecked or Checked list) to the Unchecked list. If a child is already on one of the lists and the number of steps it took to get here (this tile's steps + 1) is less than the steps of the child in question, it will then be re-added to the Unchecked list with its new information.

4. Place this tile in the Checked list.
5. Repeat steps 2 through 5 until finished (that is, tile is the goal tile).
6. When finished, build the path based on the parents of each tile, backwards from the goal tile back to the start tile (the tile without a parent).

This method will find the most efficient path to the goal from any location if there exists a path. Unfortunately, this method can be very slow in application. A few points of efficiency can be added to increase the speed, but they reduce the accuracy, though this can be compensated for later rather easily. This game engine uses the following modifications to improve performance time: The dx^2 and dy^2 are added together to determine the relative estimated distance to the goal. This aves a division operation (part of square root) which is a costly computation. Children are never added back into the list if a faster method is later found.

Now the algorithm will produce a valid path every time if one exists, except that path is not necessarily the most efficient. To overcome the loss of efficiency the resulting path is cut down to a series of way-points for the entity to follow. This is accomplished by this simple modification, for each node in the path check to see if a line can be traced from this node to the next node (for each node after this node) without encountering an obstacle. If an obstacle is encountered, set the previous node as a way point and delete all nodes between them.

This way only important nodes are kept track of and superfluous tiles are ignored. The entity can then follow its path by walking towards each node in succession until the goal is reached.

This game engine does not use a tile map to keep track of walkable terrain. The game uses a walk mask 3D model object to define the area that can be traversed. This 3D model is composed of a linked triangle mesh that defines the terrain. The above algorithms still apply, but this time, the tiles are individual triangles in the mesh and the children are adjacent triangles (a triangle is adjacent to another triangle if two of the vertices are shared between them). This style of game is very popular in the professional games industry. Most prominent is the game Starcraft, by blizzard entertainment.[8]

7.5 Ray Tracing

The process of ray tracing is one used frequently in 3D applications. It is used for many tests such as collision detection between two objects, the establishment of boundaries and the placing of objects on the "floor" of the world. Ray Tracing can also be used for advanced lighting techniques.

Ray tracing is done by projecting a ray from a single test point and checking to see if the line created will intersect with any of the triangles in the model being tested. For some cases the exact point in space where this intersection takes place needs to be returned.

In general the functions that use ray tracing need to provide a set of information that will be tested to determine if a collision between the ray and the model has occurred.

The ray is modeled with a starting position and a direction. The magnitude of the direction is important for many applications of this test such as when an entity is moving and needs to stop if it hits a polygon in the game world. The functions will also need the triangle in question. This is generally gained by using a set of triangles in a 3d model or through traversing the binary space partition that makes up the level of the game.

This problem is broken down into two general tests. First the ray is checked against the plane described by the vertices of the triangle in question (the RayInPlane Function below). If the ray intersects the plane defined by the test triangle then the point of intersection (returned through the contact pointer) is tested to see if it lies within the boundaries of the triangle. If the point is within the triangle then RayInTriangle will return 1 (true) and give us the point at which the ray intersects the test triangle. The RayInPlane function will return failure if the ray is parallel to the plane, if it points away from the plane, or is too short to intersect with the plane (meaning that the entity is headed towards the plane, but will not intersect it during this frame).

```
int RayInTriangle(Coord start, Coord dir, Coord t1, Coord t2, Coord t3,
Coord *contact)
{
  Coord normal;
  int n;
  n = RayInPlane(start, dir, t1, t2, t3, contact, &normal);
  if(n == 0)return 0;
  if(PointInTriangle(*contact, t1, t2, t3,normal) == 1)return (1 * n);
  return 0;
}
```

RayInPlane is the function that will determine if the plane is intersected by the ray. It first needs to calculate the normal of the plane defined by the three coordinates in our test triangle. This function first calculates the definition of the plane from the coordinates of the three vertices that make up the triangle. It then checks the ray against the plane for intersection.

```
int RayInPlane(Coord start, Coord dir, Coord t1, Coord t2, Coord t3,
Coord *contact, Coord *normal){

  float A,B,C,D,t;

  float denom;

  A = (t1.y * (t2.z - t3.z)) + (t2.y * (t3.z - t1.z)) + (t3.y * (t1.z -
t2.z));

  B = (t1.z * (t2.x - t3.x)) + (t2.z * (t3.x - t1.x)) + (t3.z * (t1.x -
t2.x));

  C = (t1.x * (t2.y - t3.y)) + (t2.x * (t3.y - t1.y)) + (t3.x * (t1.y -
t2.y));

  D = -((t1.x *(t2.y * t3.z - t3.y * t2.z)) + (t2.x *(t3.y * t1.z - t1.y
* t3.z)) + (t3.x *(t1.y * t2.z - t2.y * t1.z)));

  denom = ((A * dir.x) + (B * dir.y) + (C * dir.z));

  if(denom == 0)return 0;/*can't divide by zero.*/

  t = - (((A * start.x) + (B * start.y) + (C * start.z) + D) / denom);
```

The above code calculates the definition of the plane described by the triangle passed to the function. It then calculates the intersection of the vector and the plane. If the denominator of the formula is zero (can't divide by zero) then the ray is parallel and will never intersect the plane. By checking the result (t) it can be determined if the ray will intersect the plane within the range of the vector (between zero and 1), if the ray is pointing away from the plane (less than zero), or if the ray will intersect with the plane beyond the range of the vector.

```
  if((t > 0)&&(t <= 1)){
        contact->x = start + (dir * t);
        normal->x = A;
        normal->y = B;
        normal->z = C;
        return 1;
      }
      contact->x = start + (dir * t);
      normal->x = A;
      normal->y = B;
      normal->z = C;
      return -1;
    }
```

With the point of intersection with the plane known (or failed) the resulting point needs to be tested to see if it is within the bounds of the triangle. To determine if a point lies inside a triangle or outside, a ray can be cast in any direction from the test point. If the number of triangle edges crossed is odd, then the point lies within the shape, if the number of edges crossed is even (or zero) then the point is outside the object. This is ultimately based on the Jordan Curve theorem for 2D curves.

This cannot be applied to the triangle as is. If the two lines (the ray and edge of the triangle) were solved for, the result may not be accurate due to a limitation in the precision of floating point numbers. Additionally, the math for doing this calculation in 3D space is unnecessary and may be prohibitive for a game.

By projecting the triangle's 3D coordinates onto a 2D plane the math can be simplified. PointInTriangle first determines which of the 3 planes (XY, XZ, or YZ) would give the best result by checking the magnitude of each component of the plane's normal (given by the function RayInPlane). The largest absolute component value is the one used. It then uses an arbitrary ray (ex: a larger vector in the positive x or y direction) created to see if it crosses the three lines (now 2D and simple) of the test triangle, ignoring one of the components of the coordinates (either the X, Y or Z component).

```
int PointInTriangle(Coord point, Coord t1, Coord t2, Coord t3, Coord
normal)
{
  int MaxAxis;
  float MaxVal;
  Coord temp;
  Coord dir = {10000,0,0};
  MaxAxis = 0;
  MaxVal = fabs(normal.x);
...
  if(fabs(normal.z) > MaxVal)
  {
    MaxAxis = 2;
    MaxVal = fabs(normal.z);
  }
```

Now check how many edges are intersected by the direction vector.

```
  switch(MaxAxis)
  {
    case 2:
      if(RayCrossEdgeZContact(point,dir,t1.x, t1.y,t2.x, t2.y, NULL) !=
0)count++;
      if(RayCrossEdgeZContact(point,dir,t3.x, t3.y,t2.x, t2.y, NULL) !=
0)count++;
      if(RayCrossEdgeZContact(point,dir,t1.x, t1.y,t3.x, t3.y, NULL) !=
0)count++;
```

```
            break;
      }
      return (count % 2);
  }
```

With the count now known, the function original function RayInTriangle can give an accurate and fast check to see if any ray (or vector) intersects any triangles. This function is required for most 3D game engines to calculate collisions.

The above function calls the function RayCrossEdgeZContact to determine if the test vector intersects with the edges of the triangle. Since this function ignores one of the three dimensions of the point and triangle this function is relatively simple. It uses the formula for lines to check if they intersect. If they do, the function returns 1, which is added to the count of intersections in the above function.

```
int RayCrossEdgeZContact(Coord point,Coord v,float x1, float y1,float x2,
float y2, Coord *contact)
{ ...
  Uden = ((y2 - y1)*(endx - point.x)) - ((x2 - x1)*(endy - point.y));
  if(Uden == 0)return 0;/*parallel, can't hit*/
  Ua = (((x2 - x1)*(point.y - y1))-((y2 - y1)*(point.x - x1))) / Uden;
  Ub = (((endx - point.x)*(point.y - y1))-((endy - point.y)*(point.x -
x1))) / Uden;
  testx = point.x + (Ua * (endx - point.x));
  testy = point.y + (Ua * (endy - point.y));
  testx2= x1 + (Ub * (x2-x1));
  testy2= y1 + (Ub * (y2-y1));
```

If the function that called RayCrossEdgeZContact needs the point of contact information, testx and testy will contain the points of contact.

```
 if(contact != NULL)
```

```
  {
    contact->x = testx;
    contact->y = testy;
  }
  if((Ua >= 0) && (Ua <= 1) && (Ub >= 0) && ( Ub <= 1))
    return 1;/*line segments intersect*/
  else
    return 0;/*line segments do not intersect*/
}
```

CHAPTER 8

ARTIFICIAL INTELLIGENCE

The notion of Artificial Intelligence in games is very different from the academic pursuit of developing a truly intelligent agent. In games, the goal is to simulate the behavior of another player or generate a desired behavior for another entity in the game. The application of AI in games can vary from the very simple, to the very complex, to systems that will have the AI learn from history.

8.1 Rules Based AI

The most basic example of artificial intelligence in games is illustrated in the Pac-Man example. The original pac-man game had the player control the pac-man while the computer controlled 4 ghosts that roamed a simple maze. To the average player, the ghosts appear to move in a complicated pattern as they search for the player's entity. The reality is that each ghost follows a very simple algorithm that dictates their behavior. [4]

One ghost would always turn left when faced with an intersection.

One ghost would always turn right when faced with an intersection.

One ghost would turn left, then right, then left, then right, etc.

One ghost would turn in a random direction at each intersection.

These very simple rules made the behavior of each ghost very predictable, but the overall result with each of these ghosts following their own patterns created the impression that the ghosts were actively searching for the player. This most basic example showcases how an algorithm or rules based "artificial intelligence" can be implemented.

8.2 Script Based AI

More complex AI controlled entities will follow a complicated context based script. These agents will check a series of conditions to determine their behavior. The conditions will determine the best preconceived course of action for a given situation. An entity in a shooting game will have these types of conditions to check against:

- If Health is below a set value then search for health.
- If Ammo is below a set value then search for ammo.
- If Being attacked then search for cover.
- If Enemy target is in sight range then set enemy as active target.
- If Active Target is set then move into range of Target.
- If in range of Target then attack.
- Else Search for a valid Target.

These rules guide the AI agent's behavior in order to create a challenge for the human players. The rules can be adjusted to increase or decrease the effectiveness of the AI and therefore, the difficulty for the player. This model can often be implemented based on a finite state machine, with each state organizing the conditions into a different priority. Here is the same entity broken into different states:

- Searching - Roam map looking for health, Ammo and Enemies.
    - If enemy is found, set active target and change state to Engaged.
    - If health is reduced to level below set threshold then set state to Danger.
- Engaged - Move into range of enemy. Attack enemy, find cover.
    - If enemy is defeated change state to Searching.
    - If health is lowered below a set level set state to Danger.
- Danger - Run from enemies. Search for health.
    - If health is found and enemy is still active then set state to Engaged.
    - If health is found and enemy target is lost, then set state to Searching.

8.3 Adaptive AI

For games with a very deep and complex experience, the previous examples may not be enough to provide adequate challenge for human players. Players are very good at learning the patterns that the AI will fall into when adhering to rules, however seemingly complex. For the player to be continuously challenged by the computer controlled entities the intelligent agents need to be able to adapt and anticipate the actions of the player.

This style of Artificial Intelligence is often employed in fighting games. These games commonly place two combatants against each other. Both players will have a wide variety of moves all with different pros and cons. Most of the moves in a fighting game can be boiled down to a very complex adaptation of Rock/Paper/Scissors: rock beats scissors, paper beats rock, scissors beats paper. For any given situation, it is known which of the three is the best decision. This is where the predictions are needed.

The adaptive AI will need to track the history of previous player decisions. This history is checked against for possible patterns. In practice, most people will tend to fall into easily predictable patterns. The better players will make those patterns harder to determine and more complex. The better AI's will keep track of larger histories and examine larger chunks of the history for pattern checking. This problem is exactly the same type of problem in branch prediction for processors, except this one is more complex.

CHAPTER 9

CLOSING

The themes and topics discussed in this paper each have a crucial role to play in a game engine. Each will work together to provide the basic structure that one can use to create a fully functional electronic game. Students will be able to take what is developed from this paper and combine it with individual design-specific functionality and assets to create a cohesive well integrated game.

It is clear that environments are a crucial component of a game engine due to the fact everything else in the game needs to interact with the environment. In order to build a basic environment one needs to apply the fundamental concepts discussed in section 3. The environments themselves warrant further study due to their complexity, but without this basic understanding a student would be lost.

With the understanding of how an entity system is designed and developed, a developer will be able to create the specific form and functionality for an entity that will be used to make the content that exists in the final product. An entity will use the engine resources such as 3D models and artificial intelligence to give life to the characters in a game. A further study of entities would go into higher level concepts and specific algorithms that will tailor entity behavior, such as group behavior. The foundation given in this paper will make such study more fruitful.

The area of artificial intelligence for games is a massive field. The topics covered in Chaper 8 cover the concepts needed for basic game functionality. This is absolutely essential for any efforts in game development. Further study will build on non-player character entity behavior such as learning and emulating emotion. AI's pervasiveness is as extensive as the environment in its level and variety of interaction within the game engine. Indeed, more than the environment, AI has a greater effect on the game play experience and a near unlimited level of variability within the area of programming. This high level of variability is what warrants a more in-depth treatment of AI in future work.

From this work a larger body of resources for assisting with the eventual specialization that comes from game development should be developed. Currently NJIT offers three separate tracks within game development including game programming, 3D modeling, environment design and development. Each of these will benefit from further resources on each specialty.

APPENDIX A

SAMPLE SPRITE RESOURCE MANAGER

Below is s sample of a full sprite management system. This system is only dependent on the SDL (www.libsdl.org) media library. All data structures and functions prefixed with"SDL_" or "IMG_" belong to the SDL library.

This system manages a large array of potential sprites. The system provides the ability to load sprites, destroy sprites that are not longer needed and render sprites to different video surfaces. Optional functions provided allow the sprite to loaded with color correction (LoadSwappedSprite) and draw them in different styles (Black & White and flipped).

```
typedef struct Sprite_T
 {
          SDL_Surface *image;
          char filename[80];
          int w, h;  /*Width and height of the individual frames, not the whole
 image*/
          int framesperline;
          int numframes;
          int color1,color2,color3;
          int used;
 }Sprite;
 #define MaxSprites    511
 Sprite SpriteList[MaxSprites];
 int NumSprites;
```

```
/*
  InitSpriteList is called when the program is first started.
  It just sets everything to zero and sets all pointers to NULL.
  It should never be called again.
*/

void InitSpriteList()
{
  int x;
  NumSprites = 0;
  memset(SpriteList,0,sizeof(Sprite) * MaxSprites);
  for(x = 0;x < MaxSprites;x++)SpriteList[x].image = NULL;
}

/*Create a sprite from a file, the most common use for it.*/

Sprite *LoadSprite(char *filename,int sizex, int sizey)
{
  return LoadSwappedSprite(filename,sizex, sizey, -1, -1, -1);
}

Sprite *LoadSwappedSprite(char *filename,int sizex, int sizey, int c1, int c2,
int c3)
{
  int i;
  SDL_Surface *temp;
  /*first search to see if the requested sprite image is already loaded*/
  for(i = 0; i < NumSprites; i++)
  {
      if((strncmp(filename,SpriteList[i].filename,80)==0)&&(SpriteList[i].used
>= 1)&&(c1 == SpriteList[i].color1)&&(c2 == SpriteList[i].color2)&&(c3 ==
SpriteList[i].color3))
```

```
    {
      SpriteList[i].used++;
      return &SpriteList[i];
    }
  }
  /*makesure we have the room for a new sprite*/
  if(NumSprites + 1 >= MaxSprites)
  {
        fprintf(stderr, "Maximum Sprites Reached.\n");
        exit(1);
  }
  /*if its not already in memory, then load it.*/
  NumSprites++;
  for(i = 0;i <= NumSprites;i++)
  {
    if(!SpriteList[i].used)break;
  }
  temp = IMG_Load(filename);
  if(temp == NULL)
  {
        fprintf(stderr, "FAILED TO LOAD A VITAL SPRITE : %s.\n",filename);
        return NULL;
  }
  SDL_SetColorKey(temp, SDL_SRCCOLORKEY, SDL_MapRGB(temp->format, 0,0,0));
  SpriteList[i].image = SDL_DisplayFormatAlpha(temp);
  SDL_FreeSurface(temp);
  /*sets a transparent color for blitting.*/
//                SDL_SetColorKey(SpriteList[i].image,        SDL_SRCCOLORKEY,
SDL_MapRGB(SpriteList[i].image->format, 0,0,0));
  SwapSprite(SpriteList[i].image,c1,c2,c3);
   /*then copy the given information to the sprite*/
  strncpy(SpriteList[i].filename,filename,80);
```

```
    /*now sprites don't have to be 16 frames per line, but most will be.*/
  SpriteList[i].framesperline = 16;
  if((sizex) && (sizey))/*as long as neither x and y are zero*/
  {
          SpriteList[i].numframes  =  (SpriteList[i].image->w  /  sizex)  *
(SpriteList[i].image->h / sizey);
  }
  else SpriteList[i].numframes = -1;
  SpriteList[i].w = sizex;
  SpriteList[i].h = sizey;
  SpriteList[i].color1 = c1;
  SpriteList[i].color2 = c2;
  SpriteList[i].color3 = c3;
  SpriteList[i].used++;
  return &SpriteList[i];
}

/*
  * When we are done with a sprite, lets give the resources back to the
system...
 * so we can get them again later.
 */

void FreeSprite(Sprite *sprite)
{
  /*first lets check to see if the sprite is still being used.*/
  sprite->used--;
  if(sprite->used <= 0)
  {
    NumSprites--;
    strcpy(sprite->filename,"\0");
     /*just to be anal retentive, check to see if the image is already freed*/
    if(sprite->image != NULL)SDL_FreeSurface(sprite->image);
    sprite->image = NULL;
  }
 /*and then lets make sure we don't leave any potential seg faults
  lying around*/
}

void CloseSprites()
{
  int i;
   for(i = 0;i < MaxSprites;i++)
   {
     /*it shouldn't matter if the sprite is already freed,
     FreeSprite checks for that*/
      FreeSprite(&SpriteList[i]);
   }
}

void DrawGreySprite(Sprite *sprite,SDL_Surface *surface,int sx,int sy, int
frame)
{
  int i,j;
  int offx,offy;
  Uint8 r,g,b;
  Uint32 pixel;
  Uint32 Key = sprite->image->format->colorkey;
  offx = frame%sprite->framesperline * sprite->w;
  offy = frame/sprite->framesperline * sprite->h;
  if ( SDL_LockSurface(sprite->image) < 0 )
  {
      fprintf(stderr, "Can't lock screen: %s\n", SDL_GetError());
      exit(1);
  }
  for(j = 0;j < sprite->h;j++)
  {
    for(i = 0;i < sprite->w;i++)
    {
      pixel = getpixel(sprite->image, i + offx ,j + offy);
      if(Key != pixel)
      {
        SDL_GetRGB(pixel, sprite->image->format, &r, &g, &b);
        r = (r + g + b)/3;
        putpixel(surface, sx + i, sy + j, SDL_MapRGB(sprite->image->format, r,
r, r));
      }
    }
  }
  SDL_UnlockSurface(sprite->image);
}

void DrawSpriteFlipped(Sprite *sprite,SDL_Surface *surface,int sx,int sy,int
fx,int fy, int frame)
{
  SDL_Surface *temp = NULL;
  SDL_Surface *temp2 = NULL;
  int i,j;
  int tx,ty;
  int offx,offy;
  SDL_Rect rect;
  Uint32 pixel;
  Uint32 Key = sprite->image->format->colorkey;
  if((sprite == NULL)||(surface == NULL))return;
  if(frame > sprite->numframes)return; /*lets not draw past our sprite*/
  offx = frame%sprite->framesperline * sprite->w;
  offy = frame/sprite->framesperline * sprite->h;
       temp2    =    SDL_CreateRGBSurface(SDL_HWSURFACE,    sprite->w,sprite->h,
S_Data.depth,rmask, gmask,bmask,amask);
  temp = SDL_DisplayFormat(temp2);
  SDL_FreeSurface(temp2);
  rect.x = sx;
  rect.y = sy;
  rect.h = sprite->h;
  rect.w = sprite->w;
  if(temp == NULL)
  {
    fprintf(stderr, "Can't make temp surface: %s\n", SDL_GetError());
    return;
  }
  BlankScreen(temp,Key);
  if ( SDL_LockSurface(sprite->image) < 0 )
  {
    fprintf(stderr, "Can't lock sprite: %s\n", SDL_GetError());
    return;
  }
  if ( SDL_LockSurface(temp) < 0 )
  {
    fprintf(stderr, "Can't lock temp surface: %s\n", SDL_GetError());
    return;
  }
  for(j = 0;j < sprite->h;j++)
  {
    for(i = 0;i < sprite->w;i++)
    {
      pixel = getpixel(sprite->image, i + offx ,j + offy);
      if(Key != pixel)
```

```
          {
            if(fx)tx = (sprite->w - i) - 1;
            else tx = i;
            if(fy)ty = (sprite->h - j) - 1;
            else ty = j;
            putpixel(temp, tx, ty, pixel);
          }
        }
      }
      SDL_UnlockSurface(sprite->image);
      SDL_UnlockSurface(temp);
      temp2 = SDL_DisplayFormat(temp);
      if(temp2 != NULL)
      {
             SDL_SetColorKey(temp2,  SDL_SRCCOLORKEY  ,  SDL_MapRGB(temp2->format,
    0,0,0));
        SDL_BlitSurface(temp2, NULL, surface, &rect);
      }
      SDL_FreeSurface(temp);
      SDL_FreeSurface(temp2);
    }
    void DrawSprite(Sprite *sprite,SDL_Surface *surface,int sx,int sy, int frame)
    {
        SDL_Rect src,dest;
        if((sprite == NULL)||(surface == NULL))return;
         if((sprite->numframes != -1)&&(frame > sprite->numframes))return; /*lets
    not draw past our sprite*/
        src.x = (frame%sprite->framesperline) * sprite->w;
        src.y = (frame/sprite->framesperline) * sprite->h;
        src.w = sprite->w;
        src.h = sprite->h;
        dest.x = sx;
    dest.y = sy;
    dest.w = sprite->w;
    dest.h = sprite->h;
    SDL_BlitSurface(sprite->image, &src, surface, &dest);
}
```

APPENDIX B

SAMPLE PARTICLE SYSTEM

This section provides a fully functional sample particle system. This system is dependent on the SDL library, the sprite system (see appendix A) and the entity system described in the thesis. Note that only persistent particle effects require the use of the entity system. Particle effects that are generated at a single instance do not need persistent updating and can be handled by the particle maintenance function DrawAllParticles, which handles both drawing and updating all particles.

```
typedef struct Particle_T
 {
         float sx,sy,sz;            /*coordinates of where the particle is
 getting rendered*/
         float vx,vy,vz;       /*vector of particle*/
         float ax,ay,az;       /*how is the particle accelerating*/
         Uint32 Color;         /*what color be I?*/
         Uint16  flags;         /*so far the only flag set is wind effected or
 not.*/
         Sint16  R,G,B;         /*Color Direction*/
         int lifespan;       /*how many updates before death*/
               /*Some sprites may not be points, but sprites*/
         int frame,loop;          /*the frame it is on, and when to loop it*/
         int delay,delayrate;    /*how much time to pause between updating the
 frame*/
         Sprite *sprite;
```

```
        int fx,fy;              /*if we flip the sprite at draw*/

        int used;

}Particle;

Particle           ParticleList[MAXPART];

int                NumParticles;


float  OffSet(float  die)    /*this  will  give  a  random  float  between  die  and

-die*/

{

  return (((rand()>>8) % ((int)(die * 20))+ 1)* 0.1) - die;

}




void ResetAllParticles()

{

  memset(&ParticleList,0,sizeof(Particle) * MAXPART);

  NumParticles = 0;

}


Particle *SpawnParticle()    /*give a pointer to newly created particles.*/

{

  int i;

   if(NumParticles  >=  MAXPART)return  NULL;        /*all  out  of  them  darn

particles*/

  for(i = 0;i < MAXPART;i++)

  {

    if(ParticleList[i].used == 0)   /*lets find the first available slot*/

    {

      ParticleList[i].used = 1;

      NumParticles++;

      return &ParticleList[i];

    }
  }
  return NULL;/*catch all*/
}

void DrawAllParticles()    /*draw and update all of the particles that are
active to the draw buffer*/
{
  int i,done,killed;
  Uint8 r,g,b;
  killed = 0;
  done = 0;
  for(i = 0;i < MAXPART;i++)
  {
    if(done >= NumParticles)break;
    if(ParticleList[i].used == 1)
    {
     done++;
     if(ParticleList[i].flags & PF_SPRITE)
     {
       if(ParticleList[i].sprite != NULL)
       {
         if((ParticleList[i].fx == 0)&&(ParticleList[i].fy == 0))
           {  /*drawsprite is faster than drawspriteflipped, so don't use it
unless we need to*/
             DrawSprite(ParticleList[i].sprite,screen,(int)ParticleList[i].sx -
Camera.x - (ParticleList[i].sprite->w* 0.5), ParticleList[i].sy - Camera.y -
(ParticleList[i].sprite->h* 0.5),ParticleList[i].frame);
         }
         else
         {
                                   DrawSpriteFlipped(ParticleList[i].sprite,screen,
(int)ParticleList[i].sx  -  Camera.x  -  (ParticleList[i].sprite->w*  0.5),
ParticleList[i].sy      -      Camera.y      -      (ParticleList[i].sprite->h*
0.5),ParticleList[i].fx, ParticleList[i].fy, ParticleList[i].frame);
         }
       }
       ParticleList[i].delay--;
       if(ParticleList[i].delay <= 0)
       {
                         ParticleList[i].frame   =   (ParticleList[i].frame   +
1)%ParticleList[i].loop;
         ParticleList[i].delay = ParticleList[i].delayrate;
       }
     }
        else   putpixel(screen,  (int)(ParticleList[i].sx  -  Camera.x),  (int)
(ParticleList[i].sy - Camera.y), ParticleList[i].Color);
     ParticleList[i].lifespan--;
     if((ParticleList[i].lifespan <= 0)||(ParticleList[i].sz < 0))/*if we have
expired or fallen out of sight,...*/
     {
       killed++;
       ParticleList[i].used = 0;
     }
     else
     {
       if(ParticleList[i].flags & PF_WINDED)
       {
         /*         ParticleList[i].sx += GameMap.wx;
         ParticleList[i].sy += GameMap.wy;*/
       }
         ParticleList[i].sx += ParticleList[i].vx;    /*apply velocity to all
positions*/
       ParticleList[i].sy += ParticleList[i].vy;
       ParticleList[i].sz += ParticleList[i].vz;
```

```
         ParticleList[i].vx += ParticleList[i].ax;    /*apply accelleration to
all velocities*/
        ParticleList[i].vy += ParticleList[i].ay;
        ParticleList[i].vz += ParticleList[i].az;
         if((ParticleList[i].flags & PF_SPRITE) == 0)/*as long as we are not a
sprite, adjust the color*/
        {
          SDL_GetRGB(ParticleList[i].Color, screen->format, &r, &g, &b);
          if(r + ParticleList[i].R > 255)r = 255;
          else if(r + ParticleList[i].R < 0)r = 0;
          else r += ParticleList[i].R;
          if(g + ParticleList[i].G > 255)g = 255;
          else if(g + ParticleList[i].G < 0)g = 0;
          else g += ParticleList[i].G;
          if(b + ParticleList[i].B > 255)b = 255;
          else if(b + ParticleList[i].B < 0)b = 0;
          else b += ParticleList[i].B;
           ParticleList[i].Color = SDL_MapRGB(screen->format,(Uint8)r,(Uint8)g,
(Uint8)b);
        }
      }
    }
  }
  NumParticles -= killed;
}


Entity *SpawnParticleEnt(Uint32 Color,int mx,int my,Uint8 time)
{
  int i;
  Entity *PartEnt;
  PartEnt = NewEntity();
  if(PartEnt == NULL)
  {
    fprintf(stdout,"Unable to allocate room for a particle generator\n");
    return NULL;
  }
  for(i = 0;i < SOUNDSPERENT;i++)PartEnt->sound[i] = NULL;
  PartEnt->owner = NULL;
  PartEnt->target = NULL;
  PartEnt->think = NULL;
  PartEnt->update = NULL;
  strcpy(PartEnt->EntName,"Particle"); /*the name of the unit or building*/
  PartEnt->sprite = NULL;
  PartEnt->NextThink = 0;
  PartEnt->ThinkRate = 0;  /*used for incrementing above*/
   PartEnt->NextUpdate  =  0;  /*used  for  how  often  the  entity  is  updated,
updating is merely animations*/
  PartEnt->UpdateRate = 0;  /*used for incrementing above*/
  PartEnt->shown = 1; /*if 1 then it will be rendered when it is on screen*/
  PartEnt->frame = 0; /*this will be random later on...*/
  PartEnt->s.x = mx;
  PartEnt->s.y = my;
  PartEnt->health = time;
  PartEnt->healthmax = time;/*Resources use it for how much of a resource it
has left*/
  PartEnt->Color = Color;
  return PartEnt;
}

/*                            Specific  instances  of  particles  lie  below
*/

void FountainUpdate(Entity *self)
{
```

```
int i;
Uint8 r,g,b;
float k;
Particle *newparticle = NULL;
if(self->health == 0)
{
  FreeEntity(self);
  return;
}
if(self->health > 0)self->health--;
for(i = 0;i < 8;i++)
{
  newparticle = SpawnParticle();
  if(newparticle != NULL)/*lets always make sure we have a particle*/
  {
    SDL_GetRGB(self->Color, screen->format, &r, &g, &b);
    k = OffSet(0.5);
    k += 1;
    r = (Uint8)(k * r);/*shift the colors slightly*/
    g = (Uint8)(k * g);
    b = (Uint8)(k * b);
    newparticle->Color = SDL_MapRGB(screen->format,r,g,b);
    newparticle->sx = self->s.x + OffSet(0.2);
    newparticle->sy = self->s.y + OffSet(0.2);
    newparticle->sz = 6;
    newparticle->vx = OffSet(0.5);
    newparticle->vy = OffSet(0.5);
    newparticle->vz = 10 + OffSet(0.5);
    newparticle->ax = 0;
    newparticle->ay = 0;
    newparticle->R = 0;
    newparticle->G = 0;
```

```
      newparticle->B = 0;

      newparticle->az = -1.5;

      newparticle->lifespan = 80;

      newparticle->flags = PF_NONE;

    }

  }

}


void SpawnFountain(Uint32 Color,int mx,int my,Uint8 time)

{

  Entity *fount;

  fount = SpawnParticleEnt(Color,mx,my,time);

  if(fount == NULL)return;

  fount->update = FountainUpdate;

  fount->NextUpdate = 5;

  fount->UpdateRate = 10;

}


void  SpawnInstantSpray(Uint32  Color,int  mx,int  my,float  vx,float  vy,Uint8
time,int volume)

{

  int i,k;

  Uint8 r,g,b;

  Particle *newparticle;

  for(i = 0;i < volume;i++)

  {

    newparticle = SpawnParticle();

    if(newparticle != NULL)/*lets always make sure we have a particle*/

    {

      SDL_GetRGB(Color, screen->format, &r, &g, &b);

      k = OffSet(0.5);

      k += 1;
```

```
        r = (Uint8)(k * r);/*shift the colors slightly*/
        g = (Uint8)(k * g);
        b = (Uint8)(k * b);
        newparticle->Color = SDL_MapRGB(screen->format,r,g,b);
        newparticle->sx = mx;
        newparticle->sy = my;
        newparticle->sz = 0;
        newparticle->vx = vx + OffSet(2);
        newparticle->vy = vy + OffSet(2);
        newparticle->vz = 0;
        newparticle->ax = 0;
        newparticle->ay = 0;
        newparticle->R = 0;
        newparticle->G = 0;
        newparticle->B = 0;
        newparticle->ay = 1.5;
        newparticle->lifespan = time;
        newparticle->flags = PF_NONE;
      }
    }

  }

  void DripUpdate(Entity *self)
  {
    int i;
    Uint8 r,g,b;
    float k;
    Particle *newparticle = NULL;
    if(self->health == 0)
    {
      FreeEntity(self);
    return;
  }
  if(self->health > 0)self->health--;
  for(i = 0;i < 4;i++)
  {
    newparticle = SpawnParticle();
    if(newparticle != NULL)/*lets always make sure we have a particle*/
    {
      SDL_GetRGB(self->Color, screen->format, &r, &g, &b);
      k = OffSet(0.5);
      k += 1.2;
      r = (Uint8)(k * r);/*shift the colors slightly*/
      g = (Uint8)(k * g);
      b = (Uint8)(k * b);
      newparticle->Color = SDL_MapRGB(screen->format,r,g,b);
      newparticle->sx = self->s.x + OffSet(0.2);
      newparticle->sy = self->s.y + OffSet(0.2);
      newparticle->sz = self->healthmax;
      newparticle->vx = OffSet(0.1);
      newparticle->vy = OffSet(0.1);
      newparticle->vz = 1.5;
      newparticle->ax = 0;
      newparticle->ay = 0;
      newparticle->R = 0;
      newparticle->G = 0;
      newparticle->B = 0;
      newparticle->az = -0.5;
      newparticle->lifespan = 80;
      newparticle->flags = PF_NONE;
    }
  }
}
```

```
void SpawnDrip(Uint32 Color,int sx,int sy,int sz,Uint8 time)
{
  Entity *drip;
  drip = SpawnParticleEnt(Color,sx,sy,time);
  if(drip == NULL)return;
  drip->healthmax = sz;
  drip->update = DripUpdate;
  drip->NextUpdate = 5;
  drip->UpdateRate = 200;
}

void FireUpdate(Entity *self)
{
  int i;
  Uint8 r,g,b;
  float k;
  Particle *newparticle = NULL;
  if(self->health == 0)
  {
    FreeEntity(self);
    return;
  }
  if(self->health > 0)self->health--;
  for(i = 0;i < 150;i++)
  {
    newparticle = SpawnParticle();
    if(newparticle != NULL)/*lets always make sure we have a particle*/
    {
      SDL_GetRGB(self->Color, screen->format, &r, &g, &b);
      k = OffSet(0.5);
      k += 1;
      r = (Uint8)(k * r);/*shift the colors slightly*/
      g = (Uint8)(k * g);
      b = (Uint8)(k * b);
      newparticle->Color = SDL_MapRGB(screen->format,r,g,b);
      newparticle->sx = self->s.x + OffSet(30);
      newparticle->sy = self->s.y + OffSet(20);
      newparticle->sz = 2;
      newparticle->vx = OffSet(0.2);
      newparticle->vy = -10 + OffSet(0.2);
      newparticle->vz = 16 + OffSet(0.5);
      newparticle->ax = 0;
      newparticle->az = 0.1;
      newparticle->R = -4;
      newparticle->G = -8;
      newparticle->B = -4;
      newparticle->lifespan = 10 + OffSet(6);
      newparticle->flags = PF_WINDED;
   }
  }
}

void SpawnFire(Uint32 Color,int mx,int my,Uint8 time)
{
  Entity *fount;
  fount = SpawnParticleEnt(Color,mx,my,time);
  if(fount == NULL)return;
  fount->update = FireUpdate;
  fount->NextUpdate = 0;
  fount->UpdateRate = 30;
  FireUpdate(fount);
}
```

```
void RainSpotUpdate(Entity *self)
{
  int i;
  Uint8 r,g,b;
  float k;
  Particle *newparticle = NULL;
  if(self->health == 0)
  {
    FreeEntity(self);
    return;
  }
  if(self->health > 0)self->health--;
  for(i = 0;i < 16;i++)
  {
    newparticle = SpawnParticle();
    if(newparticle != NULL)/*lets always make sure we have a particle*/
    {
      SDL_GetRGB(self->Color, screen->format, &r, &g, &b);
      k = OffSet(0.5);
      k += 1;
      r = (Uint8)(k * r);/*shift the colors slightly*/
      g = (Uint8)(k * g);
      b = (Uint8)(k * b);
      newparticle->Color = SDL_MapRGB(screen->format,r,g,b);
      newparticle->sx = self->s.x + OffSet(16);
      newparticle->sy = self->s.y + OffSet(16);
      newparticle->sz = 20;
      newparticle->vx = OffSet(0.1);
      newparticle->vy = OffSet(0.1);
      newparticle->vz = -5 + OffSet(0.5);
      newparticle->az = -0.1;
      newparticle->R = -1;
      newparticle->G = -1;
      newparticle->B = -1;
      newparticle->lifespan = 10 + OffSet(6);
      newparticle->flags = PF_WINDED;
    }
  }
}

void SpawnRainSpot(Uint32 Color,int mx,int my,Uint8 time)
{
  Entity *fount;
  fount = SpawnParticleEnt(Color,mx,my,time);
  if(fount == NULL)return;
  fount->update = RainSpotUpdate;
  fount->NextUpdate = 5;
  fount->UpdateRate = 10;
}

void ItsRaining(Uint32 Color,Uint8 time,int volume,float decent)
{
  int i,k;
  Uint8 r,g,b;
  Particle *newparticle;
  for(i = 0;i < volume;i++)
  {
    newparticle = SpawnParticle();
    if(newparticle != NULL)/*lets always make sure we have a particle*/
    {
      SDL_GetRGB(Color, screen->format, &r, &g, &b);
      k = OffSet(0.5);
      k += 1;
      r = (Uint8)(k * r);/*shift the colors slightly*/
```

```
          g = (Uint8)(k * g);
          b = (Uint8)(k * b);
          newparticle->Color = SDL_MapRGB(screen->format,r,g,b);
          newparticle->sx = Camera.x + (Camera.w/2)+OffSet(Camera.w);
          newparticle->sy = Camera.y + (Camera.h/2)+OffSet(Camera.h);
          newparticle->sz = 60;
          newparticle->vx = OffSet(0.1);
          newparticle->vy = OffSet(0.1);
          newparticle->vz = -1;
          newparticle->R = -1;
          newparticle->G = -1;
          newparticle->B = -1;
          newparticle->az = decent;
          newparticle->lifespan = time;
         newparticle->flags = PF_WINDED;
        }
    }
  }
```

REFERENCES

1. Dalmatia, Daniel Sanchez-Crespo. Core Techniques and Algorithms in Game Programming. New Riders Games, 2003.

2. Morris, David and Rollings, Andrew. Game Architecture and Design. New Riders Games, 2003.

3. Shreiner, Dave Woo, Mason Neider, Jackie Davis, Tom. OpenGL Programming Guide. Fifth Edition. Addison-Wesley Upper Saddle River, NJ. 2006.

4. DeLaura, Mark. Game Programming Gems. Charles River Media, 2000

5. Barret, Sean "Optimizing Path Finding". Game Developer. CMP Media, LLC. January - March 2005

6. Phelps, Andrew Egert, Christopher Bierre, Kevin."Games First Pedagogy: Using Games and Virtual Worlds to Enhance Pedagogy". Journal of Game Development. Charles River Media, 2008.

7. Id Software. Quake 2. Activision Publishing Inc. Santa Monica, CA. 1997

8. Blizzard Entertainment Starcraft. Activision Publishing Inc. Santa Monica, CA. 1998