

Thelema

?*#! - The Team Formerly Known as Team

**Technical Design Document
DigiPen Institute of Technology
GAM 300B
2003/ 2004**

Document compiled by Pete Dunshee

With contributions, where noted, by:

**Joe Bourrie
Peter Young
Alex Yopp
Sang Park**

Table of Contents

Table of Contents	i
Summary	
Introduction	1
Target Platform Overview	1
Schedule Overview	1
Technical Cost Overview	4
Core Technology	
Target Platforms	5
Development Platforms	5
External Code	6
Version Control and Bug Reporting	7
Utility Modules	
Time - The Timing System	8
StdIncludes.h	8
The Manager Class	9
Game	
The Main Game Object	10
Graphics	
The Graphics Object	11
Color, Images and the Image Manager	11
Models and Animation	
File Formats	14
Internal Format	15
Displaying and Animating	16
The World	
Global Coordinates	18
The Tower Map	18
Level Lookup	18
The Automap	18
Teleporters	18
Levels	19
Backdrops	20
Tiles and Tilesets	21
WPN - The Weapons System	
Description	23
Definitions	23
Weapon Parameters	23
The Weapon List	24

Parameters for Specific Bullet Types	24
Collision with Bullets	25
Special Effects	
Description	25
Common Interface	25
Particle Systems	25
Particle Physics	26
Filters	26
Attached Effects	26
Additional Effects	27
AWARD - The Award System	
Description	28
In-Game Awards	28
endgame Awards	28
Award Triggers	28
Level Editor	
Description	28
Input	29
Tile/Background Selection Menu	29
Setting Award Triggers	29
Entities and Artificial Intelligence	
AI_EntityManager	30
AI_Msg	31
AI_EntityInfo	32
Entity	32
AI Generators	33
AI Functions	34
Input	
Description	35
Terms	35
Public Interface	35
Private Implementation	37
Sound	
Description	38
Public Interface	39
Private Implementation	39
User Interface	
Game Shell	41
Widget	41
Screen	42
UI Messages	42
The UI Object	43

Game Shell Screens	44
Title Screen	44
Game Select Screen	44
Single Player Options Screen	44
Multi Player Options Screen	44
General Options Screen	45
Paused Screen	45
Multi Player Finish Screen	45
Network	
Communication Protocol	47
Synchronization	47
Message System	47
Game Packets	49
Host Management	50
Player Management	51
Socket Object Class	51
Appendices	
Appendix A - Coding Standards	I
Appendix B - Related Topics	II

Summary

Introduction¹

Every fantasy story begins the same. A dark wizard is trying to take over the world, and a group of misfit heroes are sent on a quest to stop him. What if the heroes failed? What if the dark wizard took over? And what would the world look like after 800 years under the rule of black magic?

Thelema is a futuristic fantasy thriller where you play as a member of a rebel faction (called Thelema) who is charged with reclaiming the world from the evil wizard Aleister, who has used fear and black magic to control the world for the past 800 years. At your disposal are the abilities to fly, use six forms of magic, and for a select few, a magical cyber-sword.

With fast action, intelligent enemies, and a huge tower to explore, you will want to replay Thelema over and over again, and if you can't handle it alone, you can network with up to 7 other players in various multiplayer modes. Can you handle it?

Target Platform Overview

Thelema is targeted for a simultaneous release on Windows, Linux, and Macintosh. Delivery medium will be on CD-ROM. Minimum system requirements for Windows will be 500 MHz Pentium II or equivalent, 128 MB of RAM, 8 MB 3D video card, and 100 MB of disk space. Minimum requirements for Linux will be 500 MHz Pentium II or equivalent, 128 MB of RAM, XFree86 4.3 or later, 8 MB DRI compatible 3D video card, and 100 MB of disk space. Minimum requirements for Macintosh will be 500 MHz G3, 128 MB of RAM, Mac OS 10.2 or later, 16 MB ATI or nVidia 3D video card, and 100 MB of disk space.

Schedule Overview²

- ◆ **Week 5**
 - ◇ Game Framework – Joe
 - ◇ Networking Research - Sang

- ◆ **Week 6**
 - ◇ Model Display - Peter
 - ◇ Tile Display – Joe

- ◆ **Week 7**
 - ◇ Input Object - Alex

- ◆ **Week 8**
 - ◇ Level File Format - Joe

¹ Joe Bourrie, GDD p3

² Joe Bourrie, GDD p39

◆ **Week 9**

- ◇ Level Editor - Joe
- ◇ Animation - Peter

◆ **Week 10**

- ◇ Python Interpreter/AI Controller – Pete
- ◇ Network Engine - Sang

◆ **Week 11**

- ◇ 3D Sound Effect Engine (No positional effects) – Alex

◆ **Week 12**

- ◇ Weapon Systems - Joe

◆ **Week 15**

- ◇ AI Scripts (for non-pathfinding enemies) - Pete
- ◇ Level Design/Enemy Placement (Preliminary) – Joe
- ◇ User Interface - Peter
- ◇ Testing - All
- ◇ Pre-Alpha Milestone

◆ **Week 17**

- ◇ 3D Sound Effect engine (With positional effects) – Alex
- ◇ Special Effects (Weapons) – Joe

◆ **Week 18**

- ◇ Background Graphics System - Joe

◆ **Week 20**

- ◇ Special Effects - Peter
- ◇ Level Design/Enemy Placement - Joe
- ◇ Pathfinding - Pete
- ◇ Sound Effects – Alex
- ◇ Music Engine – Alex
- ◇ Multiplayer Code – Sang
- ◇ Particle Systems - Joe
- ◇ Testing - All
- ◇ Alpha Milestone

◆ **Week 25**

- ◇ The Award System – Joe
- ◇ Music Composition - Alex
- ◇ AI Scripts - Pete
- ◇ 3D Art and Textures – Sang
- ◇ Special Effects - Peter
- ◇ Testing – All
- ◇ Beta Milestone

◆ **Week 29**

- ◇ Packaging and marketing materials - Alex
- ◇ Manual – Alex
- ◇ Testing – All (Peter focus on Single Player, Sang focus on multiplayer)
- ◇ Final Product

Technical Cost Overview³

Resources:

Producer – Peter Young	\$70,000
Technical Director – Pete Dunshee	\$70,000
Designer – Joe Bourrie	\$70,000
Art Director – Sang Park	\$65,000
Product Manager – Alex Yopp	\$65,000

<i>Total Salary</i>	<i>\$340,000</i>
---------------------	------------------

Computer Costs:

Five PC Computers, \$1200/ea.	\$6,000
One Macintosh G5	\$2,000
3D Studio Max	\$3,500

<i>Total Computer Costs</i>	<i>\$11,500</i>
-----------------------------	-----------------

Total Cost:	\$351,500
--------------------	------------------

Monthly Burn Rate

Salary	\$42,500
Computers	\$1,438
Expenses	\$5,000

<i>Total Monthly Cost</i>	<i>\$48,938</i>
---------------------------	-----------------

³ Joe Bourrie, GDD p46

Core Technology

Target Platforms

As stated earlier, Thelema is targeted for a simultaneous release on Windows, Linux, and Macintosh. Delivery medium will be on CD-ROM. Minimum system requirements for Windows will be 500 MHz Pentium II or equivalent, 128 MB of RAM, 8 MB 3D video card, and 100 MB of disk space. Minimum requirements for Linux will be 500 MHz Pentium II or equivalent, 128 MB of RAM, XFree86 4.3 or later, 8 MB DRI compatible 3D video card, and 100 MB of disk space. Minimum requirements for Macintosh will be 500 MHz G3, 128 MB of RAM, Mac OS 10.2 or later, 16 MB ATI or nVidia 3D video card, and 100 MB of disk space. Specifications are enumerated as follows:

Executable Game:		0.7 MB
Models:	10 character models @ approx. 1 MB apiece:	10 MB
	7 pickup models @ approx. 150 kb apiece:	1.05 MB
Textures:	10 character textures @ approx. 300 kb apiece:	3 MB
	7 pickup textures @ approx. 150 kb apiece:	1.05 MB
Backgrounds:	10 backgrounds @ approx. 300 kb apiece:	3 MB
Tilesets:	10 tilesets @ approx. 1 MB apiece:	10 MB
Music:	11 songs @ approx. 2.5 MB apiece:	27.5 MB
Sounds:	22 sound effects @ approx. 100 kb apiece:	2.2 MB
Libraries:	8 multimedia libraries @ approx. 700 kb apiece:	5.772 MB
	Python runtimes:	25 MB

SubTotal:		89.272 MB
<i>Margin of Error:</i>		<i>10.728 MB</i>

Total:		100 MB
---------------	--	---------------

Development Platforms

Thelema will be developed on multiple platforms due to its cross platform targeting. On Windows, the development platform will be Dev-C++, using GCC 3.2; on Linux, the development platform will be Anjuta IDE using GCC 3.2; and on Macintosh, the development platform will be Apple Project Builder using GCC 3.2. All art and sound assets will be developed on Windows. Models will be created using 3D Studio Max 5.1, and textures for those models will be created using GIMP. Sounds will be created using Audacity. Music will be recorded from a live performance by Queen Floyd Scorpions from Boston Who Journey in a Zeppelin on the Sabbath, and converted to Ogg Vorbis format using Audacity.

External Code

Thelema will utilize multiple multimedia and scripting libraries within its code design, as well as various OS's and tools. These are as follows:

Windows 2000

This operating system will be used for development and testing of the Windows version of the game.

Slackware Linux 9.1

This operating system will be used for development and testing of the Linux version of the game.

MacOS X Jaguar

This operating system will be used for development and testing of the Macintosh version of the game.

OpenGL

This is the primary graphics API used by Thelema, and will be utilized for the display and manipulation of all graphics in the game.

SDL

This is a cross platform multimedia library that will be used as a wrapper for OpenGL, eliminating the need for platform specific glue code.

SDL_Image

This is a cross platform image utility library that will be used to load images and textures in various formats for the game.

SDL_ttf

This is a cross platform wrapper for the Freetype font library that will be used to display text in the game.

SDL_net

This is a cross platform networking library that will be used in the multiplayer portion of the game.

OpenAL

This is the cross platform 3D audio API that will be used to play music and sound effects in the game.

libvorbis

This is a cross platform implementation of the Vorbis audio standard. It will be used to load and translate Ogg Vorbis files for use in the game.

libogg

This is a cross platform implementation of the Ogg bitstream standard. It is used by libvorbis as part of the audio compression specification.

Python

This is a flexible cross platform embeddable scripting language that will be utilized by the Entity and AI subsystems for modular AI scripting.

Version Control and Bug Reporting

We will be utilizing CVS for version control during the development of Thelema. The CVS server is located offsite in Minnesota, giving us a safe backup system in addition to being our version control system. It is accessed through the encrypted SSH protocol using 1024 bit encryption keys for authentication. Normal password access is disabled on the server for security reasons.

In addition to the CVS server, the web based CVS View program has been installed on the server. This enables us to easily view graphs of revision trees and differences between revisions. This is secured behind a digest password authenticated directory on the web server.

For bug tracking we are using phpBugTracker. It is a simple but powerful web based bug reporting utility with a searchable database and statistic graphs for easy access to information. Also, references to files in the CVS repository are automatically translated into links to the CVS web viewer. It too is located on the offsite server, and is accessed with password protected accounts.

Utility Modules⁴

TIME - The Timing System

Description

The timing system will allow us to use real-time to calculate movement, firing, and animation, instead of having to rely on an unstable frame rate. Basically, all TIME does is calculate the current frames per second, and do calculations to scale vectors.

Functions

VECTOR ScaleVector(VECTOR) -

The vector passed in is in the form of “distance per second”, and the vector returned is in the form “distance this frame”. For example, if the vector passed in is [10, 10] and the game is running at 20 frames per second, the vector returned is [0.5, 0.5].

double GetTime(double) -

The double passed in is the time that something began, and the double passed back is the number of seconds elapsed since then. If the parameter is 0, then it returns the current time. For example, consider an animation with a 1 second frame delay. When the animation begins, it would record the current time (pass 0), and for each update, the animation would check the timer to see if 1 second has elapsed (pass time, check return). If so, it will move on to the next frame.

StdIncludes.h

Description

This file will be included in all .cpp files. It includes only files that are included everywhere in the game. The following files will be included:

SDL_Types.h

The type definitions we will be using. We will not be using the predefined types, because with SDL_Types we can maintain portability.

StdOutput.h

A useful debugging tool, because when we compile a release build, these calls disappear, eliminating any possible performance hit. It contains one function, StdOutput(), which outputs whatever is passed to it to stdout.txt. It also defines STDOUTPUT_BEGIN, which deletes the old stdout.txt file and time/date stamps the new one.

StdConstants.h

Any constant that is not part of any particular module goes here. These are usually debugging constants, but it will also contain the enumeration for game state.

⁴ Joe Bourrie

The Manager Class

Description

A templated manager for keeping track of large numbers of objects that will be used by multiple objects within the game, but should only be loaded into memory once. For example, there may be ten of a particular enemy in a level, but we only want to load its mesh once. We use the manager to store the mesh, and each enemy that uses that mesh will hold a key that lets them retrieve the mesh when drawing. When no more enemies are using that mesh, the manager will delete it so that it is no longer taking up memory.

In order to be contained in a Manager, an object must contain a public variable called ref. This is used to keep track of the reference count of the object so that it can properly delete itself after usage.

The Manager class is a preexisting resource, so it will not be factored into our development time or costs.

Public Interface

AddItem(key, item) - Adds an item to the manager if it does not yet exist. If it does exist, the Reference count for the item is incremented.

RemoveItem(key, item) - Decrements the reference count for the item specified by key, and deletes it from the manager if the reference count is now 0.

Item GetItem(key) - Returns the item specified by key.

int GetNumItems() - Returns the number of items in the Manager.

int GetRefCount() - Returns the reference count of the item specified by key.

GAME⁵

The Main Game Object

Description

The game object is a single global where all game-related data is stored. This includes level data, the current game state, and the game initialization and shutdown code. The game object also acts as an organizer between the other modules, determining the order in which each other module is updated.

External Interface

GameLoop() - GameLoop will be called each frame from the main loop. It will in turn call the update function for each module, including GAME::Update().

Reset() - This will be called whenever the game resets to the title screen. This will clear out any level data, game specific settings, and reset everything to default. This will NOT call reset for the other modules.

GetState() - This will return the current game state, allowing other modules to check and modify their actions depending on the current state.

ChangeState(STATE) -

Another module can request a change of game state. This is how the game object knows when to change states.

Init() - Should only be called before the first main loop. This initializes the game object and in turn calls Init() for each other module.

Quit() - Should only be called after the last main loop. This uninitialized everything in the game object and in turn calls Quit() for each other module.

Game Organization

```
enum STATE { GS_NONE = 0, GS_SINGLEPLAYER, GS_MULTIPLAYER,  
             GS_PAUSE, GS_UI };
```

The game object contains an instance of this enum as its state variable. This variable is returned by the public GameState() function.

Update() - Updates the game object, including checking if a new level needs loaded and checking for events and state changes.

⁵ Joe Bourrie

GRAPHICS⁶

The Graphics Object

Description

The graphics object will be the only place where graphics code exists. It will be a global object with only one interface used every frame - Update(). Internally, when update is called, it will poll the current GameState to decide what it should draw. Then it will read all needed data to draw and update the screen.

OpenGL

The main graphics API used throughout Thelema will be OpenGL. It will be treated as its own subsystem of the GRAPHICS object and will be used for rendering all image and mesh data.

SDL

The initialization and utility API that we will be using. SDL will give us cross-platform window initialization, standard types, cross-platform input, and a image loading/font rendering engine.

Camera

The camera for Thelema will use an Orthogonal projection to simulate 2D graphics using a 3D API. The playing field will be the plane $z = 0$. The camera will always be at a position $z > 0$, will have a viewing direction perpendicular to the plane $z = 0$, and will be facing toward the player.

Color, Images and the Image Manager

Color

```
struct Color
{
    float r,g,b,a;
};
```

A simple color struct that stores the four color values needed for a 32bit color.

Image

A simple 32 bit image struct. This will store simply a width, height, and an array of Color values of size (width x height).

The Image Manager

The image manager will be an instantiation of the Manager class with an ImageKey key and an Image for data. Anything that references an image does so through the image manager. ImageKey will be a std::string object and double as the file name for the image (minus the extension). If the image is part of an animation, the key will be the file name + -#, where # is the frame number (counting from 0).

⁶ Joe Bourrie

Loading Images

Images will be loaded using SDL using SDL's provided image loaders. Afterwards, a `memcpy()` will be used to transfer the image into an Image struct.

Loading Fonts

Fonts will be loaded similarly to images, by creating an SDL surface and drawing fonts to it using `SDL_ttf`. Then the surface will be loaded into an image struct.

Texture Mapping

All meshes, tiles, and backgrounds will have at least one texture associated with it. These textures will be stored as Images in the image manager. Any object that references an Image will contain an ImageKey, and when it is drawn it will request the Image from the Image Manger.

To remain compatible with the most video cards possible, we will limit the size of textures to 256 x 256. Anything that uses a larger texture will need to store multiple ImageKeys and parse the texture to multiple images of an appropriate size. This should not pose a problem because most on-screen items will be not be too large, but some items such as backdrops may need to use multiple images.

Animations

All animated 2D images, including animated textures and tiles, will contain an Animation in place of an ImageKey. When drawn, it will poll the Animation for its current frame, and use that as the image to draw.

```
struct Frame
{
    ImageKey    graphic;
    int         delay;
};

class Animation
{
private:
    list<Frame>    frames;           //The list of frames in this animation
    list<Frame>::iterator currentFrame;
    double        timeElapsed;
public:
    Update();           //Updates the animation by checking if the frame delay has
                        //passed, and
                        //if so it moves on to the next frame.
    GetImageKey();     //Returns the image key for the current frame.
};
```

Storage of Animated Images

Although a series of multiple Images when loaded, an Animation is stored in a single image file. This file will be $p \times np$ pixels, where p is the size of one frame and n is the number of frames. This file will be parsed through and each image in the file will be stored in the image

manager. When an animation is deleted, it will call `Manager::RemoveItem()` for each of its frames.

Models And Animation⁷

File Formats

The models expected at the outset will be in the 3ds file format, it is assumed that 3D Studio Max will be used for the creation of 3D models. Also, each key frame of an animation for a given model will need to be in separate 3ds files. These files will then be compiled into one ascii based file that will represent a complete model and its animation. There are three main reasons for handling models and animation in this way:

1. The way the 3ds file format supports animation is far more complex than what we need.
2. 3ds files will likely contain additional rendering information that will not be used in our game, so it will be useful to do away with the unneeded parts,
3. and having the models ascii based prevents any problems that might occur from the game being multi-platform (ie. endianness issues)

As such, a tool needs to be written to accomplish this. The following is the expected format for the ascii model file (expected tags are in <> brackets):

[modelname].mdl

```

<MODEL> //indicates that this file contains model information
<VERSION> N //gives the version number of this file (N should be a whole
number)
<NUMFRAMES> A //A number of frames of animation this model has
<NUMVERTICES> B //B number of vertices contained in this file (used for error
checking)
<NUMTEXCOORDS> C //C number of texture coordinates given (again used for error
checking)
<NUMTRIFACES> D //D number of triangle faces given (once again just for error
checking)
<VERTICES> //indicates that the following values are vertices
0.0 0.0 0.0 //three floating point values corresponding to x, y, and z
.
.
.
<TEXCOORDS> //indicates that the following values are texture coordinates
0.0 0.0 //two floating point values corresponding to u and v
.
.
.
<TRIFACES> //indicates that what follows are values that make up a triangle
face
1 2 3 4 5 6 //the first set of 3 values are indices into the list of vertices, the
//second set of 3 values are indices into the list of texture
coords
.
.
.
//Note: the only accepted polygon is the triangle

```

⁷ Peter Young

Although it isn't necessary for the VERTICES, TEXCOORDS, and TRIFACES sections to be in this particular arrangement or be contiguous for that matter, it is advised that this form be kept in order to avoid confusion and aid in future debugging. For purposes of animation, each frame is expected to contain the same number of vertices. See the Animation section for more information. Notably missing from this format is the texture to be used for this model. This has been done intentionally. Several of the enemies may share the same model, but require different textures to distinguish them. This may also be applicable to main players selecting their color. Entities within the game will specify what model they use and what texture to use for that model. It is also important to note the order in which the vertex indices are given in the TRIFACES section. Since the facing of the triangles may be important for rendering, use the following guidelines for calculating the normals:

1. vertices are given in the following form: p1 p2 p3
2. obtain $v1 = p2 - p1$ and $v2 = p3 - p1$ ($v1$ and $v2$ vectors)
3. find $N = |v2 \times v1|$ (N is a unit normal vector in the front facing direction)

Internal Format

The Model Struct and Supporting Structs

```
struct Vertex
```

```
{
    float vertex[3];
};
```

Defines a 3 value vertex.

```
struct TexCoord
```

```
{
    float u, v;
};
```

Defines a texture coordinate.

```
struct TriFace
```

```
{
    ushort vertIndex[3];           //3 indices into a vertex array defining this triangle
    ushort tcIndex[3];            //3 indices into a tex coord array, tells which texture
                                   //coordinate to use with which vertex
};
```

Defines a triangle face.

```
struct Model
```

```
{
    uint numFrames;                //number of frames of animation for this model
    uint numVertsPerFrame;         //number of vertices per frame
    uint numTriangles;            //number of triangles composing the model
    uint numTexCoords;            //number of texture coords this model uses
    vector<Vertex> vertices;        //the array of vertices
    vector<TexCoord> texCoords;    //the array of texture coordinates
    vector<TriangleFace> triFaces; //the array of triangle faces
};
```

};

All 3D models in the game will use this struct. Static models, if any, will simply just have one frame of animation.

The Model Manager

The model manager will be an instance of the Manager class in order to handle all models. The data it will store are of type Model. The key will be a ModelKey that is simply a string that corresponds to the model filename.

Displaying And Animating

Animation will be based on simple key frame animation. Each key frame will contain the same number of vertices. Which means that the partial time in-between key frames will be calculated through interpolation of the corresponding vertices. Within the Graphics object will be two methods that are called every frame to draw the entities:

1) *void RenderEntities(list<Entity>)*

Renders the entities contained within the list it is passed. Looping through the list, it will send each to the RenderEntity() method.

2) *void RenderEntity(Entity)*

Will render the entity passed to it. The entity will be queried to find out several pieces of information necessary for rendering: its position, the key of the model it uses, the key of the texture to use, direction it is facing (left or right), angle of rotation, the animation state, and the frame within that animation to use. See the section on Entities for a more detailed description of all these values.

There are two values that need further elaboration. That being the animation state and the frame number.

Animation States

All animated objects (which at this point are only enemies and player characters) will require that their models contain the following form for animation states:

Frames	State
0 - 19	IDLE
20 - 59	MOVING
60 - 79	ATTACKING
80 - 89	STUNNED
90 - 99	DEAD

These values are subject to change as need be; however, only the entity rendering function needs to know the mapping of the animation states to the frames.

Frame

This is NOT the same as the frames within the model. The frame number obtained from the entity is in the range of 0 to 1000 and corresponds to the current frame to draw within the

current Animation State. This value will be scaled down to match the appropriate model frame to draw. For example, if MOVING is the animation state and 560 is the frame number the resulting model frame to render should be 42.4 (ie. key frame 42 with 40 percent interpolation towards key frame 43).

The World⁸

Global Coordinates

All positions will be stored in a global coordinate system that encompasses the entire tower. This allows us to work with a single coordinate system throughout the entire game. Every level will have a position in global space (and none of them will overlap), and any coordinate in global space can be found in local space by subtracting this position, which will be useful for such things as tile collision.

The Tower Map

The tower map will be a large grid of levels (45 in all), 11 wide by 7 high. This will be stored as an array of MapCell objects in the GAME object.

```
struct MapCell
{
    string  areaFileName; //The name of the .lvl file that corresponds to this cell of the map.
    bool    visited;      //True if the player has visited that level before.
    string  areaName;     //The name of the area that this belongs to.
    string  levelName;    //The name of this level.
};
```

Level Lookup

Each turn, the player's global coordinates are checked to see which cell of the map the player lies in. If this cell is not the one that is currently loaded into memory, the game will pause for a moment to load in the new area.

This eliminates the need to set up level-to-level triggers for every level transfer point, which will make level creation and modification much smoother, as well as smoothing out level transitions.

The AutoMap

The automap appears when the game is paused. Once a MapCell has been visited it will appear on the automap. As the player progresses in the game, more cells will be visible on the automap, along with all teleporters marked on the map. These cells will be drawn as filled squares, and there will be a legend below the map that shows the correspondence between colors and areas.

Teleporters

Teleporters will be drawn as a swirling, wavy light, using a particle system. Every time a teleporter has been activated, it is added to the automap. Any time a teleporter is touched, the screen changes to the automap, and the player can choose, using the mouse or joystick, which teleporter they want to reappear at. This will be a simple implementation, with a list of Teleporter objects that define what teleporters have been found.

```
struct Teleporter
{
    float globalPos; //The position the player will warp to when going to this teleporter.
```

⁸ Joe Bourrie

```

    int mapX, mapY; //The position of this teleporter on the automap, in cell coordinates.
};

```

Levels

Description

A level consists of a Tileset, Backgrounds, and a Layout. The Tileset determines what tiles that will be indexed by the numbers in Layout. There can be one or more Background tags that will create decorative backdrops for the game. Finally, the Layout of the level will be an array of size $n \times n$, where n is the number of tiles on a side of the level. The data in Layout are indices into the Tileset.

```

class Level
{
private:
POINT globalPos;           //The position of the bottom left corner of the level in global
                           //coordinates.
Tileset tiles;             //The list of tiles used in this level.
list<Backdrop> background; //The list of backgrounds for this level.
vector<int> layout;        //The actual tile layout of the level.
public:
TileType GetTileAt(POINT); //Returns the tile at a given point, use for drawing,
                           //pathfinding, and collision detection.

POINT FindCollisionPoint(POINT start, POINT dest) //Finds the first collision point along
                                                    //the line segment with a tile.

Award CheckAwards(POINT); //Checks if the current point is within an award rect. If so,
                           //return that award.
};

```

Level File Format - .lvl

```

<LEVEL>           //Must be at the top of the file to confirm that this is indeed a valid level
                  //format.
<VERSION> X       //The version of the level file format that this uses. If we have to
                  //update the format, we want to keep backward compatibility. Only
                  //one tag.
<TILESET> Name    //The tileset to use along with this level, there should be only one
                  //tileset tag
<LAYOUT>          //There should only be one layout tag n x n values which specify the
                  //layout of the tiles in the level.
<BACKGROUND>     //There can be multiple background tags
<AWARD>          //There can be multiple award tags
float[4]          //Defines the rectangle that will trigger the award
Name              //The name of the award
Description       //The description for this award
pointValue        //The number of points granted for having this award.

```

```
<TELEPORTER> //There should be zero or one teleporter tag.
Point          //The position of the teleporter.
```

Backdrops

Description

A backdrop is used as a decorative background for the game. They are non-interactive, but they add a lot to the overall look and feel. Each backdrop is stored as a planar mesh that is (usually) the same size as the tileset for the level.

```
struct Backdrop
{
POINT position; //The position of the backdrop, this also determines the backdrop's Z
                coordinate.

int width, height;
float scrollRatio; //The speed of parallax scrolling
vector<ImageKey> imageSegs; //The set of images used to draw the
                             background.
};
```

Parallax Scrolling

Each background will have an associated scrolling ratio. The speed of the tileset scrolling is considered 1. Any ratio r where $0 < r < 1$ scrolls more slowly than the tileset, and should be used for any Backdrops behind the player. Any ratio $r > 1$ will scroll more quickly, and should be used for the foreground.

Backdrop Images

Every backdrop will be textured with an image. These images will have sides multiples of 64. When the backdrop is created, the image will be parsed into images of size 64 x 64. Each of these will texture one square on the backdrop mesh. The texture will repeat at the edges.

The reason for handling backdrops this way instead of using Blit routines is that alpha blending, resolution scaling, and rasterization will all be hardware accelerated using the machine's 3D hardware. Since we plan to have a large number of polygons on-screen (in the form of enemies and particle systems) we do not want our tile rendering to slow the game down.

Also note that the Z coordinate decides where the backdrop will be displayed. The playing surface and tile display is at $Z = 0$. Any backdrop with $Z < 0$ is behind the playing surface, and any backdrop with $Z > 0$ is in front of the playing surface. Note that we will be using an Orthogonal projection, so the backdrops with a smaller Z will not appear smaller on the screen.

The Backdrop List

Backgrounds will be stored by the Level object as a linked list of Backdrops, sorted by Z order. Each backdrop in the list will be displayed in order from back to front.

Tiles and Tilesets

Tile

The main building block of a level is the tile. Tiles make up the walls, floors, and ceilings of the tower. There are also special tiles such as the WEB, which is used by the Spider AI to decide where to go, and the special walls, which can only be destroyed by a particular weapon.

In the game's coordinate system, one tile is exactly 1 unit x 1 unit square. Each tile will contain an Animation object which specifies one or more frames of 64 pixels X 64 pixels which will be used to texture the tiles.

```
enum TileType { TT_EMPTY = 0, TT_SOLID, TT_BREAKABLE, WEB, FLAMES,
AUTO, PLASMA, TRAPDOOR };
```

```
struct Tile
{
    TileType    type;
    Animation   anim;
};
```

Tileset

Every area has a tileset associated with it. This is a list of each tile the area will use. The actual level data will be an array of indices into this tileset. This cuts memory consumption by huge amounts, and will simplify the level file format greatly.

```
class Tileset
{
    private:
        vector<Tile> tiles;
    public:
        Tile * GetTile(int index);    //Gets a pointer to the tile specified by index
        Clear();                      //Erases the tiles from the set
        Load(string filename);        //Loads a tileset from a file
};
```

There will always be only one Tileset in memory at any given time. The first tile in the set will always be the empty tile. All other tiles will be read from a .til file.

Note: Since the animation is part of a global tile type, instead of a local part of each tile, all tiles of a particular type will be on the same frame at the same time. This is intentional, because it ensures synchronization of water, lava, and other animated tile types.

Tileset File Format - .til

```
<TILESET> //Must be at the top of the file to confirm that this is indeed a
valid TILESET
<TILE> 1 //The next tile number to fill out
    <T> SOLID //Type of tile
    <G> tileGfx1 //Image used for tile
    <F> 5 2 2 3 4 2 //Number of frames in tile animation, and a list of frame
delays
<TILE> 2
    <T> WEB //Type of tile
    <G> tileGfx2 //Image used for tile
    <F> 1 //Number of frames in tile animation, and a list of frame delays
... //And so on for each tile
```

WPN - The Weapons System⁹

Description

Thelema features a very complex weapons system that will make the game a much greater challenge than if enemies just shot at you randomly. The weapon system is designed to fill the screen with bullets much of the time, causing much grief to the player who does not deftly dodge and avoid enemy fire.

The weapons system will be handled by a global object that stores the list of weapons, the list of on-screen bullets, and the interfaces to the weapons system.

Definitions

Bullet - A general description used for the entity spawned by either a projectile, linear, or radial attack.

Projectile - A (usually) small, round ball that flies at a straight vector until it hits a target.

linear - A line segment that fires at a straight vector until it hits a target.

Radial - An attack that fires from the center of its origin outward.

Origin - The point at which a bullet spawns, also the location of the weapon object.

Blast - The name for a shot or group of shots let off from a weapon at the same time. One weapon can let

off multiple blasts, with a small delay between each, before it is finished.

Directed - Fires each bullet at a given vector.

Targeted - Fires each bullet toward a given point.

Leading - Fires each bullet toward the estimated future location of a point, calculated by being given the

vector direction of the motion of the point.

Fan - Fires multiple shots interpolated evenly between two given vectors, given in a clockwise direction.

Burst - Fires multiple shots interpolated evenly around a circle with the origin as center.

Weapon Parameters

Every enemy can spawn one or more weapons. The enemy AI dictates the parameters that are used when spawning their weapons. For example, some enemies can release fan shots into the air, but different enemies will release different numbers of shots per burst.

BulletType - PROJECTILE, LINEAR or RADIAL

WeaponType - DIRECTED, TARGETED, LEADING, FAN, BURST

Position - The position of the weapon on the screen.

Parent - The character that created the weapon.

Damage - The amount of damage a bullet does.

Radius - The radius that the bullets effect.

Level - The level of the weapon, used for graphical purposes.

Duration - The duration of a single linear bullet.

Targets - The list of vectors/points that are used to aim the weapon.

The magnitude of the vectors correspond to the speed per second of the bullet.

BlastCount - The number of blasts that will be fired before the weapon is destroyed.

BlastDelay - The delay between blasts.

⁹ Joe Bourrie

Rotation - The angle that the weapon's direction vectors rotate after each blast.

BurstCount - The number of shots in a single FAN or BURST blast.

The Weapons List

Each time a new weapon is spawned, it is added to the weapons list. Each time the game is updated, each weapon in the list is updated as well. A weapon is removed from the list only after it has fired its final blast.

A weapon is spawned by calling the member function `SpawnWeapon(WPN_INFO)`, where `WPN_INFO` is a filled out struct containing all of the parameters of the weapon. These weapons will keep a pointer to the character that fired it, so that it can send back messages such as accuracy and number of kills.

Parameters for Specific Bullet Types

Depending on the bullet type of a particular weapon, the parameters may mean slightly different things. These parameters will be part of a struct called `WPN_INFO`.

PROJECTILE

WeaponType - Decides how to interpret the list of vectors

Parent - For record keeping

Position - This is the bullet's Origin and beginning Position

Damage - The amount of damage taken when hit by the bullet

Radius - The radius of collision for the bullet

Level - The level of the attack, used for graphical purposes

Targets - The list of vectors that will be used in a single blast

BlastCount - The number of blasts that will be used before the weapon completes

BlastDelay - The delay between blasts

Rotation - The Targets list will be rotated by this angle between each blast

BurstCount - The number of shots in a single BURST or FAN shot

Linear

WeaponType - Decides how to interpret the list of vectors

Position - This is the bullets' Origin and beginning Position

Parent - For record keeping

Damage - The amount of damage per second of contact with the linear weapon

Radius - The collision distance between the line and the character

Level - The level of the attack, used for graphical purposes

Duration - The amount of time that a single linear will be firing

Targets - The list of vectors that will be used in a single blast attack

BlastCount - The number of blasts that will be used before the weapon completes

BlastDelay - When added to the Duration, the delay between blasts

Rotation - The Targets list will be rotated by this angle between each blast

BurstCount - The number of shots in a single BURST or FAN shot

RADIAL

Position - This is the radials' position

Parent - For record keeping

Damage - The amount of damage per second of contact with the radial
Radius - The radius of the attack
Level - The level of the attack, used for graphical purposes
Duration - The amount of time that the radial attack will be used

Collision with Bullets

Each frame, each bullet is going to check what entities it is colliding with. If the bullet collides, four things will happen:

- 1) If the bullet is a projectile, it is destroyed.
- 2) If it is a linear type, it is cut off at the point where it intersects with the entity.
- 3) It sends a message to its parent telling how much damage it dealt to the enemy.
- 4) It sends a message to the enemy telling how much damage it dealt.

Projectiles will check collision using a radius from the current position of the projectile. It will be a standard bounding-circle collision system. Projectiles will never move too fast, so we will not have to worry about them skipping over a target.

linears will check collision using a point-to-line distance function, emulating the radius of a cylinder with the axis on the linear's line segment.

Radial attacks will check collision using a radius from the attack's origin.

Special Effects

Description

Special effects can be spawned by any object at any time. They are a special subsystem of the graphics interface, and they are non-interactive entities that update every frame. Effects will fall into two categories, particle systems and graphical filters. Particle systems draw new graphics used for smoke, flame, explosion, and motion trails. Filters take the preexisting graphics and modify the values in a specific area for things such as blur and heat-waves.

Common Interface

All effects will be added to the effect list, an STL list structure storing each effect, when `SpawnEffect(EFFECT_INFO)` is called where `EFFECT_INFO` is a struct holding information such as graphics, number of particles, radius of effect, and the type of effect. Every turn, the `Update()` function is called for each effect on the list, and `Update()` will call the correct function for the type of effect it is updating. Every type of effect will have its own function.

Particle Systems

Particles will be mapped onto quads (2 triangles) and drawn directly before player drawing. Each update will move the particles according to their update function. When a condition in the update function is met, such as collision with the map or a time duration (changes with type of system), a particle is destroyed. When all particles in a system are destroyed, the system is removed from the list.

Particle Physics

Many particle systems will use a simple physics model to update. This model uses certain properties of vectors to create a very realistic model of acceleration and gravity without actually using the original physics functions.

When a particle is spawned at a particular point, it is given a vector which it will move every second. Also, each update a certain gravitational vector (about (0,0.5) per second) will be subtracted from the object's movement vector. At some point, it will have decelerated enough that its upward motion is 0, and the movement vector will begin to point downward. It will accelerate downward until its maximum velocity is reached (changes depending on the system), then move at a constant "terminal velocity" until it is destroyed.

Filters

Filters will run for a particular given duration, and each update they will effect the area (radius or bounding polygon) that has already been drawn. Therefore, they will update after all other drawing is complete. Note: Filters are an optional feature and may not be implemented in the final product.

Attached Effects

There are a special type of effect called an Attached Effect, which are attached to an entity, bullet, or other object. These are updated slightly differently than the other effects on the list because they will receive three parameters per update: Current Position, Origin and Radius. The current position is usually the position that the bullet, entity, or other object that spawned it is currently at. The spawning object will be polled for this information each frame.

Projectile

Current Position - The current center of the projectile. This will be the center of the effect.

Origin - The location of the weapon that spawned this projectile. This will be used for motion lines and trails.

Radius - The radius of the projectile's range. This define a circle that the particles are confined within.

linear

Current Position - The endpoint of the linear weapon.

Origin - The beginning point of the linear weapon. This will not begin moving until the duration of the blast has passed.

Radius - The distance from the line that the linear weapon will affect. This will be the thickness of the "line" effect drawn for the linear weapon.

Radial

Current Position - The magnitude of the vector is the current radius of the attack. As the position gets further from the origin, the attack extends outward, until it hits its

maximum radius.

Origin - The point at which the attack originates, and the center of the radial attack.

Radius - The maximum radius of the attack.

Additional Effects:¹⁰

Heat Effect

Can be both a radial filter, or be contained within a bounding polygon and are all likely to be attached to an object. It is meant to distort the scenery to give the illusion of heat. The radial filter version could be used for explosions with the bounded version having several applications. These include heat effects for jet packs, fireballs, and anything else that would have a directed heat source.

Wind Effect

Most likely it will only be a radial filter, although if a use is found for the bounding polygon that may be used as well. Right now it is only envisioned to be used for the Wind spell. In appearance it would most likely look similar to wind effects used within image processing programs.

¹⁰ Peter Young

AWARD¹¹

Award System

Description

The award system will count up all of the players awards at the end of the game. There will be eight lists of award objects, corresponding to the eight players. Whenever a player receives an award, it is added to the award list corresponding to that particular player's number.

```
class AWD
{
    list<Award> awards[8];    //List of player awards
    GiveAward();             //Gives an award to a player
};

struct Award
{
    string Name;             //The name of the award
    string Description;      //A short description for the player of what the award is for.
    int pointValue;         //The point value used to calculate winners in multiplayer.
};
```

In-Game Awards

When certain actions are performed within the game, such as collecting a certain spell, finding a secret, or killing a certain monster will grant an award. These will pass the player number and an award struct into a function called GiveAward(), and the function will add that award to the players award list. Some of these in-game awards will be hard coded, but many of them will be Award Trigger points within the levels themselves.

Endgame Awards

At the end of the game, CalculateAwards() is called, which goes through any endgame awards that should be given out. For each award given, GiveAward() will be calculated similarly to In-Game Awards.

Award Triggers

When the player collides with certain rectangles set within the levels, an award will be granted. These are stored within the level objects themselves, and are usually used to confirm that a player found a secret area.

¹¹ Joe Bourrie

Level Editor

Description

The level editor will be a stand-alone tool that is used to set backgrounds, enemies, and draw in the tiles of the level. It will be a bare bones editor, because we do not plan to ship it with the game. The main screen will be the level layout, displaying all tiles that have been set, as well as any backgrounds.

Input

Mouse Movement - Move mouse cursor.

Arrow Keys - Scroll around the level.

Left Click - Set a tile.

Right Click - Erase a tile.

T - Toggle on and off tile display.

B - Toggle on and off backgrounds.

A - Set an Award Trigger.

M - Bring up the tile/background selection menu.

S - Save the current level. - See **Level File Format - .lvl**

L - Load an existing level.

Tile/Background Selection Menu

This will display the tile set in a window on the left, and a list of backgrounds on the right as well as three buttons (Add, Remove, Change Z). Left clicking on a tile in the left window will select it as the current tile. Left clicking on a background on the right will select it. ESC will exit back to the layout screen.

Add - Will prompt for a filename to add as a background for this level. This will default to Z = -1

Remove - Will erase the currently selected background from the list.

Change Z - Will prompt for a floating point number to set as the selected backgrounds Z.

Setting Award Triggers

When the A key is hit, the cursor goes into Award trigger mode. To exit this mode, hit ESC or create an award trigger. To create an award trigger, click and drag to draw a box. A prompt will then appear to give the trigger a name, a description, and a point value.

Entities and Artificial Intelligence¹²

The entity and AI subsystems will be closely intertwined, and will consist, within the main runtime of the game, of a main entity manager class which mainly act as a marshaling agent between the main game program and the python scripts which will do the real work in the entity/AI subsystem. The entity manager class will contain a python dictionary (a “map” in c++) which will contain the entire current active entity list for the game. Keys for the map will be randomly generated unique integers, and the data each key will point to will be an instance of a python Entity class which will contain all the information regarding a specific entity. Each python Entity class will then in turn possess an instance of an AI. This AI instance will in fact be an instance of a python generator, which is a special type of python function which is able to pause itself and resume from where it left off. These AI generators may consist of one or more references to different python AI functions which may or may not be generators themselves, depending on their uses. I will now enumerate the classes and functions of this subsystem; in parenthesis next to each name will be the language in which that module will be written.

AI_EntityManager (C++)

This class will be the main interface between the AI subsystem and the rest of the program.

Initialization:

On creation, the AI_EntityManager will initialize the python interpreter and set up the name space dictionaries for the interpreter.

Destruction:

When destroyed, the AI_EntityManager will destroy its python name space dictionary and finalize the interpreter.

Interface Methods:

void Update(void): This function takes no arguments and returns nothing. It attempts to update the entire entity list, running the next frame of all AI's (in the case of non player entities) or input controllers (in the case of player entities). If it encounters problems of any kind, it will throw an exception.

void Reset(void): This function clears out and deletes all entries in the python name space dictionary. If any problems are encountered, it will throw an exception.

list<unsigned int> GetEntities(void): This function takes no arguments and returns an STL list of unsigned integers corresponding to the entities currently held by the entity manager.

*int GetProperty(unsigned int entityKey, char** propertyString)*: The first argument is a key corresponding to an entity in the dictionary, and the second argument is the c-string name of a property within the entity class. The function returns the value of that property within the entity class.

¹² Pete Dunshee

void SendMsg(unsigned int entityKey, AI_Msg msg): This function merely takes a key to an entity and an instance of an AI_Msg struct. It sends a message to the entity corresponding to the key.

void SpawnNPC(AI_EntityInfo entity): This function adds an entity to the dictionary. Information about the entity to be added will be contained in the AI_EntityInfo struct. If the function is unsuccessful, it will throw an exception.

void SpawnPlayer(AI_EntityInfo entity): This function is identical to the SpawnNPC function, except that it will attach an input controller to the entity and flag the entity as a player controlled entity.

AI_Msg (C++)

This struct will be used to send control messages to the entity subsystem from other subsystems in the game. It is designed to be as abstract and extensible as possible in order to accommodate any sort of message that needs to be sent to an entity.

Data:

AI_MsgType msg: This is an enum describing which message to send.

unsigned char validOps: The first three bits of this character are used to specify which of the secondary operands are used and valid.

int op: The first parameter pertaining to the message will be stored here.

int op2: The second parameter pertaining to the message will be stored here; this is the first secondary operand.

int op3: The third parameter pertaining to the message will be stored here; this is the second secondary operand.

int op4: The fourth parameter pertaining to the message will be stored here; this is the third secondary operand.

Message Manifest:

Messages described here will be illustrated using the syntax "*msg <op>, <op2>, <op3>, <op4>*". Operands in angle braces are mandatory, and operands in square braces are optional. (e.g. *msg <op>, <op2>, [op3]*)

TAKE_DAMAGE <amount>, [x_vector], [y_vector], [force]: Cause an entity to take a specified amount of damage, and to recoil in a specified direction with a specified force.

ADD_ACCURACY <damage>: Inform an entity that it has hit another entity with its projectile so that it can update its accuracy rate and damage rate.

ADD_KILL <spirit>: Inform an entity that it has killed another entity and give it the amount of spirit that was earned.

AI_EntityInfo (C++)

This struct will be used to specify details for loading a new entity into the game.

Data:

char filename*: This is a string containing the filename of the entity description file.

unsigned int x, y: These hold the world coordinates of the placement of the entity.

Entity (Python)

This class will hold all of the data for a single entity, and will possess all the internal functions required for the operation of that entity.

Functions:

Update(): Causes the entity to perform its next frame of animation, which most often involves causing its AI to perform its next action.

SendMsg(msg): Adds a message to the end of an entity's message queue. "msg" will be a dictionary built by the C++ SendMsg function.

Attributes:

x, y: Floats describing the current world position of the entity.

vx, vy: Floats describing the current velocity vector of the entity.

AI: An instance of the entity's AI generator.

msgList: A sequence of messages that have been sent to the entity.

mesh: A string representing the mesh this entity is using.

texture: A string representing the texture this entity is using.

direction: A character corresponding to the direction the entity is facing: 'l' for left, 'r' for right.

angle: An integer from 0-360 corresponding to the current angle of rotation on the entity, 0 being no rotation.

animation: An integer corresponding to the current animation this entity is using.

frame: An integer from 0-1000 corresponding to the current frame or frame of interpolation this entity is on.

focus: The entity that this entity is currently focused on, as applies to the AI. It will simply

be a reference to the respective entity object.

health: An integer corresponding to the amount of health the entity has.

mass: A float corresponding to the mass of the entity, used in physics calculations.

Loading from disk

Entity descriptions on disk will be stored as Python modules which will consist of a class that will derive from the Entity class and which will override the `__init__` function of the Entity class. That `__init__` function will then set the entity's attributes according to how it needs to describe itself. Entity's will then be imported as Python modules to the game, and when the game requests an entity whose type is already imported, the manager will be able to tell that it does not need to reimport it by virtue of polling its name space dictionary.

AI Generators (Python)

At the base level, there will be a set of simple generators which will be specific to most entities. These main generators may then contain instances of other generators which will be used for different behaviors that AI generator will utilize. All AI's will also utilize various AI utility functions to perform operations that are shared among various entity behaviors and AI types. Therefore, at a glance there will be two types of generators, general personality generators, and behavior generators.

Behaviors:

Head for player: Entity will move towards the player in the most expedient way possible.

Head away from player: Entity will move away from the player in the most expedient way possible.

Head for player at distance: Entity will move towards a specified distance from the player. Utilizes the "Head for player" behavior.

Head for player at distance and orientation: Entity will move towards a specified distance from the player and at a specified orientation. Utilizes the "Head for player at distance" behavior.

Attack player: Entity will move to attack the player, and will attack when attacking is deemed possible. May utilize any of the other movement behaviors, depending on the type of attack.

Patrol: Entity will walk in a specified direction on the ground.

Targeted patrol: Entity will utilize "Patrol" behavior to get beneath the player.

Attacking patrol: Utilizes the "Patrol" behavior, changes direction when it reaches a wall or cliff, and utilizes the "Attack player" behavior.

Stationary: Entity remains still.

Teleport: Entity attempts to find an open space nearby, and instantly moves there.

Personalities:

A personality is a collection of the aforementioned behaviors. It decides which behavior to act on based on its individual script.

AI Functions (Python)

These are utility functions utilized by the various AI behaviors.

Function list:

Find Path: Returns a list of points for a path between two points. This algorithm will possibly eventually be based on A* if its speed is satisfactory, and if the time is available. Otherwise it will most likely be based on a right turn rule edge following algorithm. This function may in fact be a generator function which yields singular movement vectors until the ultimate path is discovered and returned as a sequence.

Normal shot: Shoots weapon in a specified direction vector.

Targeted shot: Shoots directly at focused entity.

Leading shot: Shoots at focused entity with respect to the entity's position and velocity.

Multidirectional shot: Shoots along a list of vectors.

Fan shot: Fires n shots between two vectors in a specified rotational direction.

Burst shot: Fires n shots equally spaced around a circle.

Torch: Initiates torch in a specified direction, to live a specified time.

Linear weapon: Initiates linear weapon in a specified direction, to live a specified time.

Python Extensions for Entities and AI (C++)

These are extensions to the Python language that will be written in C++ and will give Python scripts access to various subsystems of the game.

Map Extension

This extension will consist basically of a few functions that return map collision information. The Python scripts will be able to use this information to do path finding.

Input Extension

This extension will be a complete wrapper of the Input subsystem. Basically there will be a one to one ratio of wrapper functions to public input system functions.

Sound Extension

This extension will most likely consist of a single function which can cause the sound system to play a sound effect.

Weapons Extension

This extension will consist of wrappers for the weapon spawning functions in the weapon subsystem.

Network Extension

This extension will consist of functions which can be used to poll the network subsystem for the positions and actions of units controlled by the server so that units on the client can mimic and duplicate those actions and movements.

Input¹³

Input.h Input.cpp

Description

This file will contain the definition of the InputObject class. Input will be handled using SDL and SDL's event-handling system. The InputObject will implement a polling system and will keep track of the current state of the keyboard, mouse position, and/or any other controllers. The InputObject, when queried, will return currently activated buttons and current positions of the mouse and/or any game controller axes that may have changed from the previous state. All key constants used are defined by SDL.

Terms

Movement Axis - the axis of control that is the player's movement around the game world controlled by an analog stick or the keyboard. An input of up will result in moving the character up on the screen. Related directly to the Movement Vector.

Firing Axis - the axis of control that is the player's firing direction, controlled by an analog stick or the mouse. An input of up will result in the player firing the current weapon upwards. Related directly to the Firing Vector.

Public Interface

`bool FirePressed()` - Checks if the Fire Spell button was pressed since the last update. Returns true if it was and false if it was not.

`bool IcePressed()` - Checks if the Ice Spell button was pressed since the last update. Returns true if it was and false if it was not.

`bool StormPressed()` - Checks if the Storm Spell button was pressed since the last update. Returns true if it was and false if it was not.

`bool IsModPressed()` - Checks if the Modifier button is being pressed. Returns true if so and false if not.

`VECTOR GetFiringVector()` - Returns the current firing vector, set by checking the user's activation of an analog stick or the mouse's current position in relation to the character's current position. In the case of the keyboard/mouse controller, if the left mouse button was not clicked (indicating the firing of the weapon), the vector is set to the 0 vector.

`VECTOR GetMovementVector()` - Returns the current movement vector, set by checking the user's activation of an analog stick or combination of the set keyboard movement keys.

`int GetFireKey()` - Returns the key that selects the Fire Spell. In the case of a gamepad, returns the button number. For use by the User Interface system to display the current controller configuration.

¹³ Alex Yopp

int GetIceKey() - Returns the key that selects the Ice Spell. In the case of a gamepad, returns the button number. For use by the User Interface system to display the current controller configuration.

int GetStormKey() - Returns the key that selects the Storm Spell. In the case of a gamepad, returns the button number. For use by the User Interface system to display the current controller configuration.

int GetModKey() - Returns the key that activates the secondary versions of the various spells. In the case of a gamepad, returns the button number. For use by the User Interface system to display the current controller configuration.

int GetUpKey() - Returns the key that moves the player up. This is only in the case of the keyboard, as movement on a gamepad will be controlled by an axis. For use by the User Interface system to display the current controller configuration.

int GetDownKey() - Returns the key that moves the player down. This is only in the case of the keyboard, as movement on a gamepad will be controlled by an axis. For use by the User Interface system to display the current controller configuration.

int GetRightKey() - Returns the key that moves the player right. This is only in the case of the keyboard, as movement on a gamepad will be controlled by an axis. For use by the User Interface system to display the current controller configuration.

int GetLeftKey() - Returns the key that moves the player left. This is only in the case of the keyboard, as movement on a gamepad will be controlled by an axis. For use by the User Interface system to display the current controller configuration.

SetFireKey(int key) - Sets the key that selects the Fire Spell. In the case of a gamepad, sets the button number. For use by the User Interface system if the user changes the current controller configuration.

SetIceKey(int key) - Sets the key that selects the Ice Spell. In the case of a gamepad, sets the button number. For use by the User Interface system if the user changes the current controller configuration.

SetStormKey(int key) - Sets the key that selects the Storm Spell. In the case of a gamepad, sets the button number. For use by the User Interface system if the user changes the current controller configuration.

SetModKey(int key) - Sets the key that activates the secondary versions of the various spells. In the case of a gamepad, sets the button number. For use by the User Interface system if the user changes the current controller configuration.

SetUpKey(int key) - Sets the key that moves the player up. This is only in the case of the keyboard, as movement on a gamepad will be controlled by an axis. For use by the User Interface system if the user changes the current controller configuration.

SetDownKey(int key) - Sets the key that moves the player down. This is only in the case of the keyboard, as movement on a gamepad will be controlled by an axis. For use by the User Interface system if the user changes the current controller configuration.

SetRightKey(int key) - Sets the key that moves the player right. This is only in the case of the keyboard, as movement on a gamepad will be controlled by an axis. For use by the User Interface system if the user changes the current controller configuration.

SetLeftKey(int key) - Sets the key that moves the player left. This is only in the case of the keyboard, as movement on a gamepad will be controlled by an axis. For use by the User Interface system if the user changes the current controller configuration.

SetDefaultControls() - Resets the control configuration to the default configuration.

Update() - Backs up the current controller state and saves the new state of the game controller to determine if any keys have been pressed, or vectors changed.

Reset() - Resets all member values to default values.

Init() - Initializes SDL Input and the InputObject itself.

Shutdown() - Uninitializes SDL Input and destroys the InputObject.

Private Implementation

Certain actions must be performed by the private implementation. Mostly those actions will consist of detecting the current state of the game controllers and its buttons and axes. However, in the case of using the combination of mouse and keyboard as a controller, the firing vector must be determined by calculating the vector between the current mouse position and the current player position. Such is not the case with an analog stick as the position of the stick is already a vector initiated at the origin of the Firing axis.

Also, the private implementation will be responsible for creating or modifying a new configuration file.

Sound¹⁴

Sound.h Sound.cpp

Description

The SoundObject will be implemented using OpenAL, specifically intended for 3D positional audio. However, a 2D setup will be available to the user. There will be a total of four different sound modes:

High Quality Sounds, 2D, .ogg format

kBits per Sample: 128
Sampling Rate: 48.0 kHz
Sample Type: Stereo

Low Quality Sounds, 2D, .ogg format

kBits per Sample: 16
Sampling Rate: 22.05 kHz
Sample Type: Stereo

High Quality Sounds, 3D, .ogg format

kBits per Sample: 128
Sampling Rate: 48.0 kHz
Sample Type: Mono

Low Quality Sounds, 3D, .ogg format

kBits per Sample: 16
Sampling Rate: 22.05 kHz
Sample Type: Mono

Should it be deemed necessary, a mid-range setting for both 2D and 3D could be made available.

Dependent on the user's sound card and its properties, which will be queried upon initialization, the SoundObject will make use of hardware voices, or software voices if there are no hardware voices (or not enough) available. Only one type, hardware or software will be used, due to the inconsistent sounds produced by the two. The user will be able to select the number of voices they wish to have available. Should they set the number to a higher value than their hardware supports, they will be informed that the voices will have to be played in software. This is not anticipated as a real problem, as most contemporary sound cards have enough voices for the purposes of *Thelema*.

The .ogg file format, heretofore unmentioned, is similar to an .mp3 file, in that it is a high quality, compressed audio file. However, .ogg is a free file format and, while OpenAL extensions for .ogg are not available on all of the target platforms, the extra overhead needed to make them work on each platform is worth the effort. And the extra overhead is minimal.

¹⁴ Alex Yopp

Public Interface

bool IsSoundOn() - Returns a boolean designating the on/off state of the Sound Effects. True for on, false for off.

bool IsMusicOn() - Returns a boolean designating the on/off state of the Music. True for on, false for off.

SetSoundState(bool SoundOn) - Sets the on/off state of the Sound Effects. True for on, false for off.

SetMusicState(bool MusicOn) - Sets the on/off state of the Music. True for on, false for off.

SetSoundVolume(float Volume) - Sets the volume of the sound effects. The value passed in, a float, should be between the values of 0 and 1; 0 designating off and 1 designating full volume.

SetMusicVolume(float Volume) - Sets the volume of the music. The value passed in, a float, should be between the values of 0 and 1; 0 designating off and 1 designating full volume.

float GetSoundVolume() - Returns a float designating the current volume setting of the sound effects. The value should be between 0 and 1: 0 designating off and 1 designating full volume.

float GetMusicVolume() - Returns a float designating the current volume setting of the music. The value should be between 0 and 1: 0 designating off and 1 designating full volume.

PlaySound() - Creates a buffer and loads in the appropriate sound depending on the event responsible for calling the function.

PlayMusic() - Streams in and plays a music file.

SetDefaultSettings() - Resets the sound settings to the default configuration.

Update() - Continues playing any sounds (or songs) that are currently playing, cleans up any sounds that have finished playing, and starts playing sounds that need to be played.

Reset() - Resets all member values to default values.

Init() - Initializes OpenAL and the SoundObject itself.

Shutdown() - Uninitializes OpenAL and destroys the SoundObject.

Private Implementation

The private implementation will be responsible for managing the sound effects and music

for *Thelema*. The many buffers need to be loaded based on a priority system (in case too many sound effects occur simultaneously) and then the buffers must be mixed together for multiple sound effects to play concurrently. Unless turned off, one buffer will be constant as a resource for the music. Similarly, should the sound effects be turned off in the settings, much of the SoundObject's functionality will be skipped over.

Each sound effect needs to have a world or screen position so that the sound can be placed in a 3D world to provide the effect of 3D sound. Each sound also needs to have a velocity setting; however, this value will be constant among all sounds, as the medium through which the sound is playing will always be constant. For example, there are no underwater areas. Additionally, each sound will have a number relating to its priority; the lower the number, the higher the priority, with 0 being the highest priority. Sounds are played and loaded according to their priority member. Important sounds, such as sounds created by the player and explosions have the higher priorities, whereas sounds created by enemies, including their weapons fire, have lower priorities. As mentioned earlier, excluding the case where it is turned off, the music will be constant, and therefore, technically has the highest priority. However, since it is rare that the music buffer will be deleted and recreated, this should not be an issue.

User Interface¹⁵

Game Shell

The game shell will be managed by a UI class and through the use of a Widget class. Everything seen on screen will consist of Widgets, or more precisely it will consist of derivatives of a Widget.

Widget

```
class Widget
{
    private:
        Point      location;           //location of the widget (local coordinate)
        int        width, height;     //width and height of the widget
        WidgetID  ID;                 //widget's own ID
        WidgetID  parentID;          //ID of this widget's parent widget

    public:
        void Update();               //checks to see if this widget should act in any way
        void Draw();                 //widgets will draw themselves
};
```

Defines at the most basic level what a Widget should be. All other widgets will be derived from this base. As such, the functions within Widget should be overridden. The following are derivatives that will be useful:

- Text - a simple widget that displays text; there should be a way to get and set the text
- it should not respond to user input
- TextInput - a variation on the Text widget, this will respond to keyboard input and should update the text it displays
- it would be useful for such things as entering names
- there should be a way to retrieve the text data
- SlideBar - a widget that would allow for selecting a value in the range of 0 to 100
- this would be useful for setting sound levels and such
- CheckBox - a simple box that can be checked or not, useful for boolean settings
Button - a widget that can be clicked on resulting in some action resulting
- may contain graphics or text
- Graphic - a widget that simply consists of an image to draw

Widgets will get input information by querying the Input object. Whenever the status of a widget changes, it will post a message to the UI object. Since the UI object will be a global instance, widgets can do this through using the UI's PostMessage() function.

¹⁵ Peter Young

Screen

```

class Screen : public Widget
{
    private:
        list<Widget> widgetList;    //list of all the widgets that make up the screen

    public:
        void Activate();            //will load up whatever widgets and such it needs
        void Deactivate();         //removes from memory what is no longer needed
        void Update();              //calls update on all its widgets
        void Draw();                //calls draw on all its widgets
        void AddWidget(Widget);     //adds a widget to the screen
        Widget GetWidget(WidgetID); //returns the widget of the specified ID
};

```

Each game shell screen will be a derivative of Widget. This will allow for more control over the layout of the user interface. For instance, the pause screen will not have to take up the entire screen. The paused game can still be seen behind the pause screen. Widgets will be drawn in the same order that they occur in the list. A screen should have no parent. The screens ID should be the name of the screen it is representing, see the Game Shell Screens for a list of the screens that will be needed. The GetWidget() function will be necessary for UI in order to get or set data for any given widget.

UI Messages

```

struct UIMsg
{
    uint        msgID;            //ID that identifies the type of message
    WidgetID    screenID;        //the screen that generated this message
    WidgetID    widgetID;        //the specific widget that generated this message
    int         param1;          //the last two values are determined by the type of message
    int         param2;
};

```

This message struct will be used by widgets to send any info necessary to the UI class. The UI class will process the messages and determine if any action needs to be taken.

All message IDs should be prefaced with **MSG_** and will be contained within an enumeration of type **UIMessages**. The following are categories of messages UI will need to process:

- User Interface Messages:
 - ID will be prefaced by **MSG_UI_**
 - this will be used primarily to indicate to UI that the current screen should be changed
- Graphics Messages:
 - ID will be prefaced by **MSG_GFX_**
 - used to indicate that UI needs to alter the graphics settings which entails calling the interface methods on the GRAPHICS object

- Sound Messages:
 - ID will be prefaced by **MSG_SND_**
 - used to indicate that UI needs to alter the sound settings, achieved through method calls on the SOUND object
- Input Messages:
 - ID will be prefaced by **MSG_INP_**
 - used to indicate that UI needs to alter Input settings, key mappings for instance
 - the INPUT object will be accessed for these settings
- Game Settings Messages:
 - ID will be prefaced by **MSG_GST_**
 - used to indicate that UI needs to alter the game settings, such as single player or multiplayer settings
 - the GAME object will be used for these messages

The UI Object

```
class UI
{
    private:
        list<UIMsg>      msgQueue; //messages needing to be processed
        list<Screen>    screensList; //list of all screens
        WidgetID        activeID; //ID of the currently active screen
        ImageKey        cursorKey; //key to the cursor image

    public:
        void Init(); //Initializes all the screens, activeID should be the title screen
        void Reset(); //should basically just set the active screen to the title screen
        void Shutdown(); //deallocates anything that needs to be
        void Update(); //will process all messages, calls update on the active screen
        void Draw(); //the active screen will draw
        void PostMessage(UIMsg); //posts a message to the message queue
};
```

The UI object will handle all user interaction within the game shell. When Init() is called, all game shell screens within the game will be created. UI will keep track of the currently active screen with an ID. Widgets will post messages to UI using the PostMessage() function. Within the Update() function, UI should first check to see what the current game state is in order to see what, if any, updating needs to occur. The Game object's GetState() function will be used to find this information. The only states of particular interest to UI will be the GS_PAUSE and GS_UI game states. UI will then need to process the messages within its queue and determine if any actions need to be taken. Update() for the currently active screen will then need to be called. Also, whenever UI determines that a screen will be active, Activate() will be called on that screen and the previous screen will have Deactivate() called. Draw() will simply call Draw() on the currently active screen, and then display the cursor.

Game Shell Screens

As a note, for purposes of uniformity all screens should have the same background graphic. As well as each screen will be using UI's PostMessage() function and querying the Input object for input from the cursor and key presses.

Title Screen

Consists of text and graphics to display the team logo and the title of the game. Either by the user pressing a key or from the passage of time, the screen will post a message to UI indicating that it should transition to the Game Select Screen. Only User Interface category messages will be sent. No data will need to be accessed or set within this screen.

Game Select Screen

This screen will consist of four buttons to indicate four actions that can take place: Single Player, Multi Player, Options, and Quit Game. Clicking on Single Player should transition to the Single Player Options Screen, for Multi Player transition to the Multi Player Options Screen, for Options transition to the General Options Screen, and for Quit Game simply quit the game. Only User Interface messages will be sent to indicate which screen to change to. No data will need to be accessed or set within this screen.

Single Player Options Screen

Displaying in this screen will be four widgets. One for setting the difficulty, second for selecting the player's color, third for starting the single player game, and fourth a button to back up to the Game Select Screen. Pressing ESC should also back up to the Game Select Screen. When first entered the game settings will need to be queried to get the current difficulty level and the player's color. Changing these settings will result in a Game Settings message being posted. User Interface messages will be sent to result in backing up a screen or to transition to the Single Player game.

Multi Player Options Screen

The Multi Player Options screen will be divided up into four main areas:

1. Player Name - an InputText widget allowing the player to set their name. It should default to "PlayerN" where N is the player's number.
2. Team Selection - will have at most 8 boxes representing the available teams the player can choose to be on.
3. Team Options - displays the current team name, and the teams color.
4. Host Options - the options that the host player has set for this game. Those options being: force even teams (ie. teams can not differ in size by more than 1), the number of teams that can exist, and the difficulty setting.

In addition to these four sections, two buttons will be needed. One for backing up to the Game Select Screen, and one for starting the multiplayer game. When first entered, the game settings will need to be queried to get the correct settings with which to start. Any changes to these settings will result in the posting of Game Settings messages. User Interface messages are sent for starting the multi player game or backing up a screen.

General Options Screen

Consists of three distinct areas of settings and a back button. These settings are for graphics, sound, and control options. As such, the graphics, sound and control settings will

need to be queried for the current settings. Any changes to these settings will result in their respective messages being posted (Graphics Messages, Sound Messages, and Input Messages). Only one User Interface message will be needed for backing up to the Game Select Screen.

Graphics Settings:

Possibly set gamma, resolution, etc.

Sound Settings:

Such things as music volume and sound effects volume.

Control Settings:

Most likely just key mappings.

Paused Screen

The Paused screen will consist of the following:

- 1) text indicating that the game is in a state of pause
- 2) a button to continue playing the game
- 3) a button to quit playing the game, the UI should then transfer to the Title Screen
- 4) a list of all the players' stats (multiplayer is also included in this)

User Interface messages will be sent for continue game play, or quitting. The players' stats will need to be accessed from the GAME object. It should be noted that while in single or multi player games the Paused Screen should remain loaded even when not being displayed. UI will make sure that it is not drawn or updated until need be. In addition to these requirements, a portion of the Paused Screen will display the Automap. The drawing of the Automap will take place outside of the UI object, so a portion of the screen should not be drawn to in order for there to be space for the Automap.

Multi Player Finish Screen

The screen will contain a section detailing each of the players' awards won during the course of play (see the Awards section for details on what awards may exist). The other section will list the teams and their respective rankings (starting with the first place team and going through the Nth place, where N is the total number of teams). Of course, it will be necessary to query the GAME object for retrieving the correct information to display. Either a key press or cursor click will send a User Interface message to change to the Title Screen.

Network¹⁶

Thelema network engine is a flexible cross-platform engine. By using SDL_net network functionality to send and receive data, programmers will not have to code different things over different platforms. This method helps computers with different operating system communicate with each other. The network functionality is based on peer-to-peer architecture. Each machine is responsible for updating the game state. The machine then stores the changes in players current state into a packet, which will be sent to the other machines. The other machines in the game use this data to update their game-state, and send out information about their players. Since Thelema consist of multiple rooms, data transmission allows each client talk to the other clients that are in the same room.

For example, 4 players client A, B, C, and D are in the game. Client B, C, and D are in the same room of the game. When client-A enters this room, client-A will send a change-room notification packet to all other clients that are in the current game. When other clients (B, C, and D) receive the change-room notification packet from client-A, client (B, c, and D) will first identify if all clients are in the same room or not, and if everyone is in the same room client A, B, C, and D will start communicating each other.

Communication Protocol

Both UDP and TCP will be used for this game. Since TCP cant send broadcast message through the network UDP will be used to send broadcast message to local network. But mostly, TCP will be used to send data packet that is vital to game.

Synchronization

In multiplayer game, synchronize communication must be synchronous to all the clients. The synchronization is done so that each client keeps track of what local player is doing, and then sends the game-packet from time to time. The game will be updated every 10 times per second to limit the amount of bandwidth for gaming. To accomplish this task, the main game-loop will call the update function, and will constantly update the game-state from time to time.

Message System

Description:

Message for this game will be held in a message structure which holds message priority, the type of message that module is sending, and the actual data that will be sent to client.

Message Priority State:

Message priority state will be enumerated, and will hold all the priority type that is needed for this game. Enum type will be used by message structure to determine the priority of the

¹⁶ Sang Park

message, and when it determines, a data will be sent depends on the priority type. Example of message priority is shown below.

```
enum NET_MSGPRIORITY
{
    NET_MSGPR_REGULAR, // Sends message to all clients in same room.
    NET_MSGPR_ALL      // Sends message to everyone.
    NET_MSGPR_SENDTEAM, // Sends message to team.
    NET_MSGPR_SENDDONE, // Sends message to the specific one you want.
    NET_MSGPR_BROADCAST, // Sends broadcast message.
    NET_MSGPR_NUMPRIORITY, // Max number of message priority.
};
```

Message Type:

Enum type will define our networking package event, and will be used by message structure to determine type of message that server or client is sending. This will also be used in message structure to determine the type of message. Example of message type is shown below.

```
enum NET_MSGTYPE
{
    NET_MSGTYPE_ENTITYPOSITION, // Position of entity.
    NET_MSGTYPE_ENTITYVECTORS // Vector direction of entity.
    NET_MSGTYPE_ENTITYHEALTH // Health of entity.
    NET_MSGTYPE_WEAPON // Weapon.
    NET_MSGTYPE_ROOMNOTIFY // Room notification packet.
    NET_MSGTYPE_ROOMRESPONSE // Room response packet
    NET_MSGTYPE_JOIN, // When someone joins the game.
    NET_MSGTYPE_ALLOW // Allows player to join game.
    NET_MSGTYPE_LEAVE // When player leaves the game.
    NET_MSGTYPE_REMOVEVPLAYER // Remove player.
    NET_MSGTYPE_CHATMSG, // Chat message.
    NET_MSGTYPE_ACCEPT, // Accept the connection from client.
    NET_MSGTYPE_NUMMSG // Total numbers of message type.
    // etc.
};
```

Message structure:

Message structure will be used to send a message of any data from client to client. This message structure will contain message priority to identify the priority of the message, message type to identify the each message, destination IP address, source IP address, size of data, and data buffer where you can stick any information. Example of message structure is shown below.

```
typedef struct
{
    uint msgpriority // Priority of the message.
    uint msgtype; // Type of message.
    IPaddress fromIP; // Source IP address.
```

```

        IPAddress toIP           // Destination IP address.
        int size;                // Size of data.
        int data[NET_DATASIZE]  // The message data itself.
    } NET_MSG;

```

Game Packets

Description:

Game packets will be used to send all the information that is needed for the game. Packets will consist of standard packet, change room notification packet, and change room response packet.

Standard Packet:

Standard packet will be used to send information of the entity. This packet struct will contain entity information and weapon information. Although weapon information is sent to other clients, the calculation of weapon will be handled by each client. Example of structure is shown below.

```

typedef struct
{
    uint msgpriority           // priority of the message
    uint msgtype               // identifies the type of message
    float entity_x,entity_y;   // position of entity
    int u,v;                   // entity vector
    uint weapon;               // type of weapon
    float weapon_x,weapon_y;   // position of the weapon
} NET_STANDARD;

```

Change Room Notification Packet:

Change Room Notification packet is sent when a player enters a new room. Coordinate of the room will be sent to all the clients that are in the current game to identify if players are in the same room or not.

```

typedef struct
{
    uint msgpriority           // Priority of the message
    uint msgtype;              // Type of message
    int x, y;                  // Coordinate of the room
} NET_ROOMNOTIFY;

```

Change Room Response Packet:

This packet is sent when a player receives Change Room Notification packet from other client. When a player receives Change Room Notification packet from another player, player will first check their room coordinate and send true if they are in the same room and send false if they are not in the same room. If they are in the same room,

```
typedef struct
{
    uint msgpriority           // Priority of the message
    uint msgtype;             // Type of message
    bool response;            // If clients are in same room this will be set to true.
    // These will be sent if response was true
    float player_x, player_y  // Player position
    int player_x, player_y    // Player vectors
    int health                 // Player health
    // AI information.
    // Room information.
} NET_ROOMRESPONSE;
```

Host Management

Host List Manager Class

Host list manager class is responsible to find and keep track of all the games that are currently running. To find the host this class will first send out a broadcast message, and then store the games that are currently in session.

Private Interfaces:

*list<NET_HostListManager *> m_HostList*: This will keep track of all the games that are currently running

Public Interfaces:

FindHost(): This function will send out broadcast message to find the host on LAN.

AddHostToList(IPAddress): This function will store the games that are currently in session.

RemoveHost(IPAddress): This will remove host from the host list.

Host Structure:

Host structure will be used to store the host information. Example is shown below.

```
struct NET_HostInfo
{
    // IP address of the host
    IPAddress ip;
    // Name of the game
    char gamename[GAMENAME_SIZE];
    // Number of player that are currently in the game
    unsigned int num_player;
}
```

Player Management

Player Structure:

Player structure will be used to store player information. Example is shown below.

```
typedef struct
{
    // Player number
    int player_number
    // Players Team
    int team_number;
    // IP address of player
    IPaddress player_ip;
} NET_PlayerInfo;
```

Player List Manager:

Player list manager class is responsible for maintaining all the players that are currently in the game.

Private members:

list<NET_PlayerListManager *> m_PlayerList : This will keep track of all the players that are in a current game.

Public Interfaces:

AddToList(IPaddress) : This function will add a player to the list. If it encounters any problem, it will throw an exception.

RemoveFromList(IPaddress) : This function will remove the player from the list. If it encounters any problem, it will throw an exception.

Socket Object Class

Description:

Socket object class is where all the functionality that needs to do with the networking stuff. It basically encapsulates the functionality that SDL_net UDP/TCP socket has. This class serves as a base class for game socket class.

Protected Members:

HostListManager m_HostList; This will contain the list of all games that is being played.

TCPsocket m_TCPSock; TCP socket that will be used to send any game data.

UDPsocket m_UDPSock; UDP socket that will be used for broadcasting.

Public Interfaces:

void UDPIInit(void); Initializes UDP socket.

void UDPClose(void); Closes UDP socket.

void TCPOpen(void); Opens TCP socket.

void TCPClose(void); Closes TCP socket.

void Send(*NET_MSG*) This function will first check what priority the message is and then it will send the message depends on message priority.

void Recv(*NET_MSG*) This function receives message and determines the type of message and it will do different stuff depend on the type of message received.

In Game Communication:**Description:**

Since player management class keeps track of all the players lists that are in the current game, IP address of each players, and what team they are on, chatting function will be implemented based on player management class.

Chat Message Structure:

```
typedef struct
{
    uint msgpriority           // Priority of message
    uint msgtype;             // Type of message
    uint playerNumber;        // Player number
    uint teamNumber;          // Team number
    char messageText[128];    // Chat message
} NET_CHATMSG;
```

Game Socket Class**Description:**

Derived from Socket Object Class. This class is responsible for any in game data transmission.

Private Interfaces:

PlayerListManager m_PlayerList; List of all players that client is connected to.

bool HostInRoom; When this is set to true, this socket will become host in the room.

Public Interfaces:

void Update(void); This function will be called by the main game loop every frame. It first checks its time limit (10 times per second = .1 second) and when its time to update it will send game packets such as entity positions, entity vectors, AI calculations, player weapons, and bullet positions.

void SendChatMessage(NET_CHATMSG); This function will be used to send chat messages to other players.

bool IsHostInRoom(void); Checks if this socket is the host in the room or not.

Appendix A - Coding Standards

- ◇ No #including in .h files
- ◇ Avoid defining functions in .h files
- ◇ No globals (If you must use one, ask me, Pete, how to not use it)
- ◇ Use SDL_types and GL_? types only. (GL_Uint, etc.)
- ◇ StdIncludes.h, this is your utility header, there are many like it but this one is yours
- ◇ DO NOT USE NULL, initialize pointers to 0
- ◇ Use explicit casting, ie. static_cast, reinterpret_cast, etc.
- ◇ Rather than returning 0, -1, false, or any other random error behavior, throw instances of exception classes
- ◇ To ensure full cross-compatibility, code must ultimately compile with no warnings of any kind
- ◇ At the top of every file you create, include a comment header with your name, date, and a brief description of the file's purpose
- ◇ At the top of every function you define, include a comment header with your name, date, and a brief description of what the function does
- ◇ If you change the purpose of a function you define, change the function header!
- ◆ Variable naming conventions:
 - ◇ No underscores
 - ◇ Capitalize the first letter of every word: NumSprites
 - ◆ Use consistent one character looping variables:
 - ◇ Start most loops with 'i', then 'j', etc.
 - ◇ For loops dealing directly with positions, use 'x', 'y', and 'z', etc.
 - ◇ **Exception:** in loops where the operation is not necessarily that obvious, use more descriptive looping variables: eg. BitmapIndex
 - ◇ Prefix pointers with "p": pVariableInstance
 - ◇ Prefix member variables with "m_": m_PlayerPosition
 - ◇ Prefix function parameters with "p_": p_Index
 - ◇ Use combinations where applicable: m_pSpriteTexture, p_pBitmap
- ◆ Function naming conventions:
 - ◇ No underscores
 - ◇ Capitalize the first letter of every word
 - ◇ For example: BlitSpriteToScreen()
- ◆ File naming conventions:
 - ◇ Prefix all filenames in a module with the same prefix
 - ◇ eg. AI_Main, AI_Manager, SFX_Manager, SFX_Loader, etc.
- ◆ ANSI style indenting will be used:
 - ◇ Four space indents
 - ◇ Curly braces on their own line, unindented
- ◇ Comment! Rule of thumb: if you have to justify an explanation for an algorithm to yourself, write it out in a comment

Appendix B - Related Topics

- ◇ Commit updated code and added files to CVS often
- ◇ Do a CVS update on the whole project before committing to get changes to other portions that may have happened while you were editing
- ◇ Do not commit to CVS unless your code compiles, ie. the project can be built successfully
- ◇ Check the bug tracker at least once a day for bugs you are responsible for (if you don't have it set up to email you updates)
- ◇ Submit bug reports to the bug tracker for bugs in other people's modules
- ◇ Be sure to close bugs that you have fixed