CENG 491

Senior Design Seminar and Project

NOVART

Presents

"The Flee From Alcatraz"

Initial Design

08.12.2004

Authored By:

M.Zahit Özcan 1250588
İbrahim Özbay 1203355
Serhat Solak 1250711

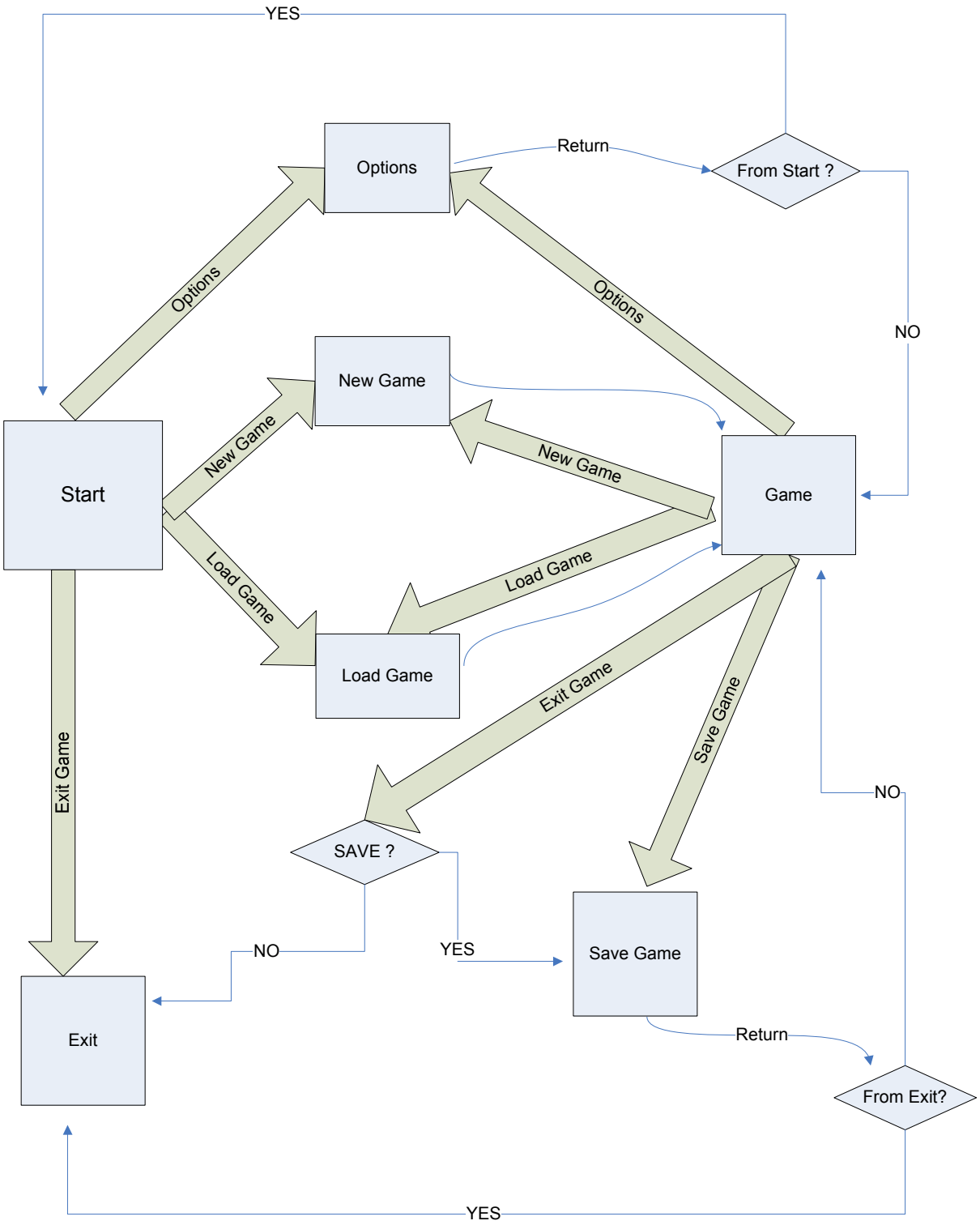
1. Introduction

The Flee from Alcatraz is a 3D first person shooter computer game. The player chooses a character from 3 possible characters and controls this character with keyboard and mouse inputs. The aim of the game is to solve the puzzles given in the levels and eliminate all the enemies that are on the way.

This document's purpose is to build the design of the project roughly, meaning that not every single detail of the design is reported here and also there may be some additions or modifications later.

We have done a bottom up design so that we have decided on most of the data structures and algorithms to be used in the game. We have included the external codes needed, how they will be used so that in the future coding will be easier since most of the things has been structured and only a small amount of additions and some modifications will need to be done.

2. Flowchart Diagram



1. The application starts executing after the user double clicks the game. Now the control is in START state. The main interface will be shown on the screen and user input will be taken. From this state the user may choose to load a previously saved game, start a new game, change configuration settings or exit the game.

2. In the OPTIONS state the player may choose to change the configuration settings in the option menu such as game volume, music volume etc. The control can come to this state from either START or GAME state, when exiting from the options the control is returned to the state that it came from.

3. In the NEW GAME state the player chooses a character from 3 possible ones and the difficulty of the game. Then the level 1 map, the objects on the map, textures are loaded. After exiting this state the control enters the GAME state.

4. In the LOAD state the player chooses one of the saved game files from a list on the load window. The data from this file contains the level data including map, objects and texture. This file is loaded into game data structures. Then the control moves to GAME state.

5. In the SAVE state the player gives a filename and all the level data is written to a save file. The control can come to this state either when escape buton is pressed or when the user presses save buton in GAME state. Then the control moves back to the state that it came from.

6. In the GAME state the actual game playing occurs in a main loop. In this state the player can change options, open a new game, save a game, load a game or exit from the game. The control return back to this state except from EXIT state.

7. In the EXIT state all the memory allocated will be freed and control will return to OS.

3. Game Object Data

3.1 CHARACTER

Attributes

Name: It is a string unique to each character which identify the character .

Position: It is structure with two integers x_coordinate and y_coordinate. It keeps x and y component of the character's position on the map. The **move()** function changes the position of the character with the user input. This information is used while rendering.

Status: It is an integer indicating the status of the character such as 'Sleep', 'Walk', 'Watch', 'Hit'.

Crouch: It is an integer indicating whether the character is crouching or not. So that in raycasting the camera position is lowered.

Direction: Its data type is float and it is the angle between character's current direction and x axis. The **rotate()** function changes the direction of the character . This information is used while rendering.

Speed: It is an integer and determines the speed of character when he is not carrying any object. The **move()** function uses it to determine the speed of move.

Objects: It is a pointer to the first object carried by the character . The objects on the character are kept in a linked list.

Current_Object: Its data type is object and shows the object currently in use.

Weapons: It is a pointer to the first weapon having by the character. As objects weapons are also kept in a linked list.

Current_Weapon: Its data type is a weapon structure. It shows the weapon which is currently in use.

Hit_Point: It is an integer and shows the current hit point of character. It can be changed by the use of some objects such as a potion.

Special_Ability: It is an integer which determines how good a character use weapons. If ability value is high the possibility of shooting the target is high. It changes during game.

Model: It is a string which is equal to an obj. format file name. The obj. file includes some information about character such as vertices , normals , faces, skeleton vertices(for animation and movement of the enemies).

Functions

Move(): It gets an input from user and according to that input changes the **Position** of character if it is possible(i.e. There is not any object in front of character.) To change the **Position** it uses some information of the character. To determine the direction of move it uses the **Direction** of the character. The **Speed** and the weights of **Weapons and Objects** are used for the determination of the speed. It also looks at the **Map** to see if there is an object in the direction of move.

Rotate(): This function changes the **Direction** of the character with the user input.

Shoot(): This function decreases the number of bullet of the **Current_Weapon.** According to the **Special_Ability** of character and attributes of **Current_Weapon** it reduces the hit point of the enemy if there is any in front of the character. **Shoot()** calculates the path of the bullet by using **Position , Direction and Current_Weapon** of the character and the position of the aiming point. If there is bullet in the weapon, the name of the sound of the **Current_Weapon** produced by **Shoot()** is pushed into a

structure which will be passed to the SOUND state for playing. If there is no bullet in the **Current_Weapon** this function has not any effect. Returns a structure indicating the amount of damage made to the character and the x and y positions of the character on the map. There is a global data indicating whether attack button is pressed. It is made 1 if attack button is pressed. See the algorithms section for details.

turnToHero(): When the character(enemy) is attacked from a direction where the enemy can not see the hero, this function changes the direction of the enemy towards the hero.

Move_Toward_Enemy(): If enemy is able to see the hero but the hero is not in range, the AI calls this function, so the enemy tries to come into the range of the hero. See the algorithms section for details.

Take_Weapon(): If there is any weapon in front of the character **Take_Weapon()** takes the weapon out of the map and assign this weapon to the **Current_Weapon.**
It also adds weapon to the **Weapons** list.

Drop_Weapon(): Drop_Weapon() adds the **Current_Weapon** to the map and also remove it from **Weapons** list. Then it assign the previously used weapon to the **Current_Weapon.**

Change_Weapon(): It gets the id of the weapon and makes it the **Current_Weapon** if this weapon is in the **Weapons** list of the character. If it is not in the list it has not any effect.

Take_Object(): If there is any movable object in front of the character this function takes the object out of the **Map** and adds it to the **Objects** of the character. It also changes the **Current_Object** with the taken object.

Drop_Object(): This function adds the **Current_Object** to the map at the **Position** of the character and takes the **Current_Object** out of the **Objects** list.
It also assign the previously used object to the **Current_Object**.

Change_Object(): This function gets an id of a object from the **Objects** list and assign it to the **Current_Object.**

Use_Object():

3.2 MAP

Attributes

Name: It is a string unique to each map which identify the map.

Map_Size: It is a structure which includes two integer (map_height and map_width) to determine height and width of the map.

Map_info: Map_info keeps the information of the map. It is an m*n structure Coordinate array.

Coordinate: The coordinate structure will be as follows.

```
struct Coordinate{
        int height;
        int ObjectId;
        Character * character;
}
```

If there is only an object (not a character like the enemies or the player), Coordinate[x][y] only holds the height and the ObjectId and the character = NULL. If there is an enemy on that location on the map, we have a pointer to the character object and ObjectId=0. If ObjectId = 0 and character = NULL then there is nothing on that location other than terrain.

Functions

Initialize_Map(): This function allocate the necessary memory for the map info and filled the Array with the corresponding values . To get the values it opens an external file by the use of unique name and reads the necessary values from that file by scripting. (For an m*n map the necessary memory is an m*n*2 integer array)

Add_Object(): This gets an object and a coordinate point as input and add the object to the specified point of the map.

Remove_Object(): This function takes the object out at the coordinate point which is taken as input.

3.3 OBJECT

Attributes

Name: It is a string unique to each object which identify the object.

Weight: It is an integer which determines the weight of the object.

Texture: It is a string which is equal to a texture file name.

Model: It is a string which is equal to an obj. format file name. The obj. file includes some information about object such as vertices , normals , faces etc.

ModelStructure: This will hold the position of the vertices, vertex textures, face normals and faces.

```
Struct{
        vertex[numberofvertex][3];
        face[numberofface][3][2][3];
}
```

vertex[numberofvertex][3: x,y,z components of the vertex]

face[numberofface][3: vertex index of the 3 vertices corresponding to the face][2: texture coordinates that map to a given pixel on a bmp image][3: vertex normal x, y, z coordinates]

Functions

Initialize_Object(): This function opens an external file by the use of unique name , reads the values in the file by scripting and assign these values to the attributes of object.

3.4 WEAPON

Attributes

Name: It is a string unique to each weapon which identify the weapon.

Range: It is an integer which determines the range of the weapon.

Type of weapon: It is a string which determines the type of the weapon (laser, shotgun, rifle)

Bullet Amount: It is an integer indicating the amount of bullet in the weapon.

Damage: It is an integer which corresponds to the amount of damage that the weapon gives.

Splash Damage: It is an integer which corresponds to the damage that the weapon gives in a radius around the point it hits. For example a grenade will damage the player if he is near enough to it even though if he is not directly on the grenade.

Weight: It is an integer which determines the weight of the weapon

Texture: It is a string which is equal to a texture file name.

Model: It is a string which is equal to an obj. format file name. The obj. file includes some information about weapon such as vertices , normals , faces etc.

Magazine_Capacity: It is an integer which keeps the maximum number of bullet on a magazine

Functions

Initialize_Weapon(): This function opens an external file by the use of unique name , reads the values in the file by scripting and assign these values to the attributes of weapon.

Weapon, and character are inherited from object class.

4. External Code

For scripting and playing sound in an OpenGl application we need to use some external code. These are SDL, Python, SWIG. There are two fundamental ways in which scripting is used. In the first one you embed a script in your main application written in a compiled language such as C++, so from the code in C++ you call the scripts when it is needed. In the second one, everything is vice versa. So you write modules in C++, the scripting language runs the main application and calls these external modules as needed. We have decided to use Python as our scripting language because it is used in many games and applications and it has many documents. There exists a problem of binding these two different languages, C++ and Python. You have to bridge function in order to forward parameters and return values between the two languages. There are tools to do this. One of them is SWIG, Simplified Wrapper and Interface Generator. SWIG is a tool that connects programs written in C and C++ with a variety of high-level programming languages, such as Perl, Python, Ruby,. Assume you have some functions written in C++. Your main goal is to turn them into modules so that they can be called from the scripting language. For this you first write an interface file in which you name the module, include headers and stuff. Then swig ouputs the module and in the scripting language you directly call the functions by import module1, module1.f(int x, int y).

Our main objective in using scripting is to manage object systems, describe weapon effects, specify events and triggers. We will use scripting for defining weapon attributes, maps, heroes and levels. The main application will run in C++ and we will get the objects from the script files when needed.

We will use SDL (Simple Direct Media Layer) for playing sound in a multithreaded fashion. So while the game play continues the music and sound of the other objects like the weapons will still be heard.

We will be using one of the free 3d modelling programs called MilkShape 3d. It is a low polygon modeler in which you can draw simple models by creating vertices by simply

clicking on a point in one of the views; front face view, side view, top view. After creating the vertices you create faces by selecting the vertices with select option. So we will draw simple objects such as a desk, a switch or a door by using this program. But the process of creating an object to be used in a game such as a weapon is very complicated if you want a weapon to look realistic so we will import the free modesl that we find to be useful on the internet to the MilkShape 3d program. Then we will export these objects with this program as a .obj file. An obj file consists of vertice coordinates (written one after the other and indexed starting from 1), normal vectors, faces. In addition to holding the geometric information in an obj file you can also specify the materials to be used. All the faces are mapped to the texture material nearest to it from above. We will be using code from a program that loads an .obj file into an OpenGl application and then maps a texture in .bmp onto the object .

5. Algorithms

5.1 Finding the Path of the Bullet

Firstly we must find the direction of the bullet. As we explained in ray casting the character has a 60 degrees field of view. If the hero has a weapon in hand there will be an aiming circle on the screen which is used to aim at enemies. We know the x coordinate of the circle. If we subtract the x coordinate of the circle from the x coordinate of the center of the screen and multiply this value with 30 degrees we will find the angle of the bullet with respect to character's current direction. Than adding this angle to the character's current angle gives us the direction of bullet with respect to x=0 on the map.

Now we know the direction of the bullet and the position of the character. We will cast an imaginary ray starting from the position of the chracter which has a direction equal to the direction of the bullet. If the ray hits some object (i.e enemy , wall etc.) on the first square on the direction of bullet then we return the position of hit on the map. Else we continue with tracing the ray until it hits an object. (until it pass the range of the weapon). We can summarize the algorithm as follows.

1.Calculate the direction of bullet path with respect to character's direction.

2.Add the direction of bullet to the direction of character to find the direction of bullet with respect to x=0 on the map.

3.Starting from the position of the character cast a ray

A.Calculate the next square in the direction of bullet on the map.

B.If it hits an object return the position of hit

C.If it is in the range of weapon trace the ray and go to step **3A**

4. Return 0 (It does not hit any object)

5.2 Finding the Shortest Path Between Two Points

If an enemy sees the hero and the hero is out of the range of the weapon the enemy uses , enemy needs to find the shortest path to the hero. To find the shortest path between two points we will use A* algorithm.

We start with the position of enemy and add that square to the open list. (The open list is a list of squares that may need to be checked out.). Than we look the walkable squares adjacent to the starting point and add them to the open list. Every square on the open list has the following three values.

1.The parent square : We need the parent square to find the path when we reach the destination.

2.Cost: The movement cost to move from the starting point (position of enemy) to that square following the path generated to get there.

3.Remaining cost : The estimated movement cost from that square to the destination.

To find the cost we need to add 10 to the cost of the parent if that square is reached with a horizontal or vertical move and add 14 if it is reached with a diagonal move. ($\sqrt{2} \cong 10/14$) . To find the remainig cost calculate the number total of squares moved horizantally and vertically to reach the target square from the current square.

After looking at the adjacent squares we drop that square from open list and add it to close list(not look again). Than pass to the square with minimum (cost + remaining cost) value. Look at walkable adjacent squares , add them to open list, calculates the three values for those squares and remove the current square from the open list and add it to close list. If an adjacent square is already in the open list and the cost from the current square is smaller we have to change the cost and parent of that square.

We have to do these operations until we find the destination square. When we find it we can easily find the path between two points because we know the parent of each square. We can summarize the algorithm as follows.

1.Add the starting square to the open list.

2.Repeat the following

A.Look at the lowest (cost + remaining cost) square on the open list.

B.Switch it to the close list

C.For every adjacent square to this square

a.If it is not walkable or it is on the close list ignore it.

b.If it is not on the open list make the three assignments and add it to open list.

c.If it is in the open list calculate the new cost. If the new cost is smaller change the cost and parent values.

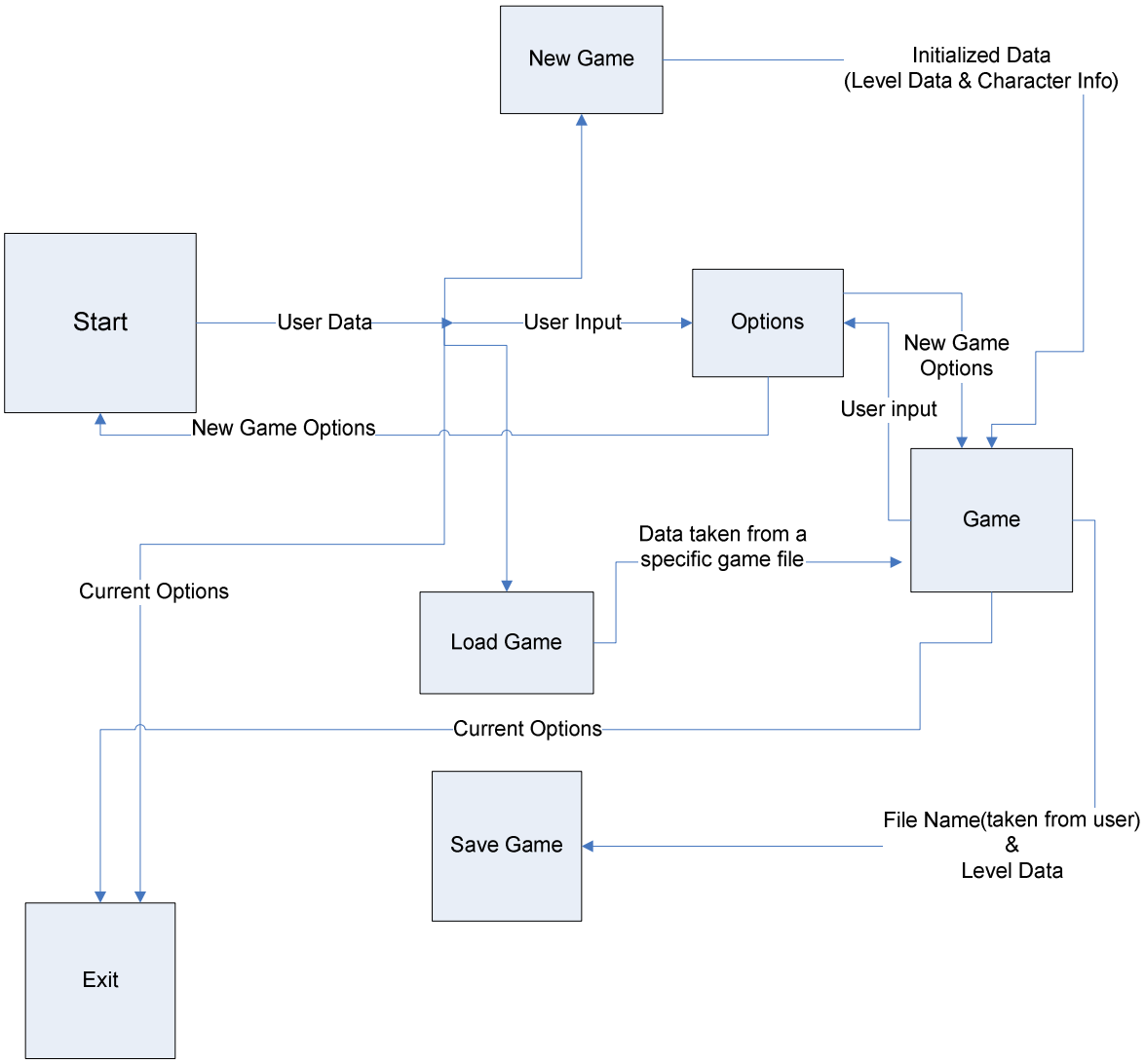
D.If you reach the target square or the open list is empty (no path) than stop.

3.Save the path.Working backwards from the target square , go from each square to its parent until you reach the starting square.

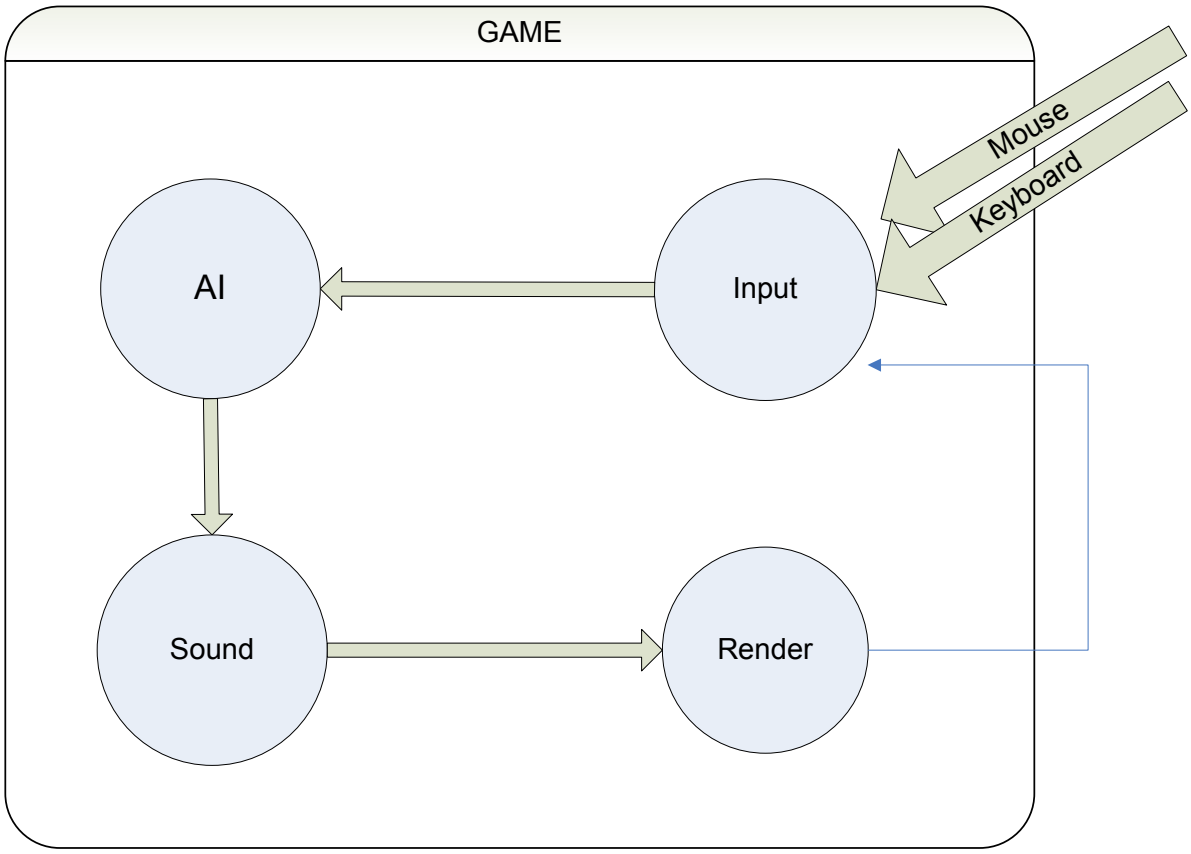
6. Data Flow

Our game data will be stored in script files written with Python scripting language. These will include level scripts, object scripts, map scripts, weapon scripts, hero scripts. Each level will point to a map script and several object scripts that will take place in the course of the level. When the game starts first of all the player will choose a new game or load a previously played game. As soon as he chooses a new game the information about the level, the map, the hero and the objects residing on the map will be loaded from the script files into the required data structures allocated in main memory. If this wasn't a new game then when the user quitted previously a new script file would have been created including his level, his heroes status, the status of the objects on the map, the weapons he is carrying currently. The transfer of data will be carried on between Python and our main application C++ via the bridge function created by SWIG. So the data will flow in both directions between them. When loading it will flow towards C++, when saving it will be towards Python.

6.1 DFD Level 0

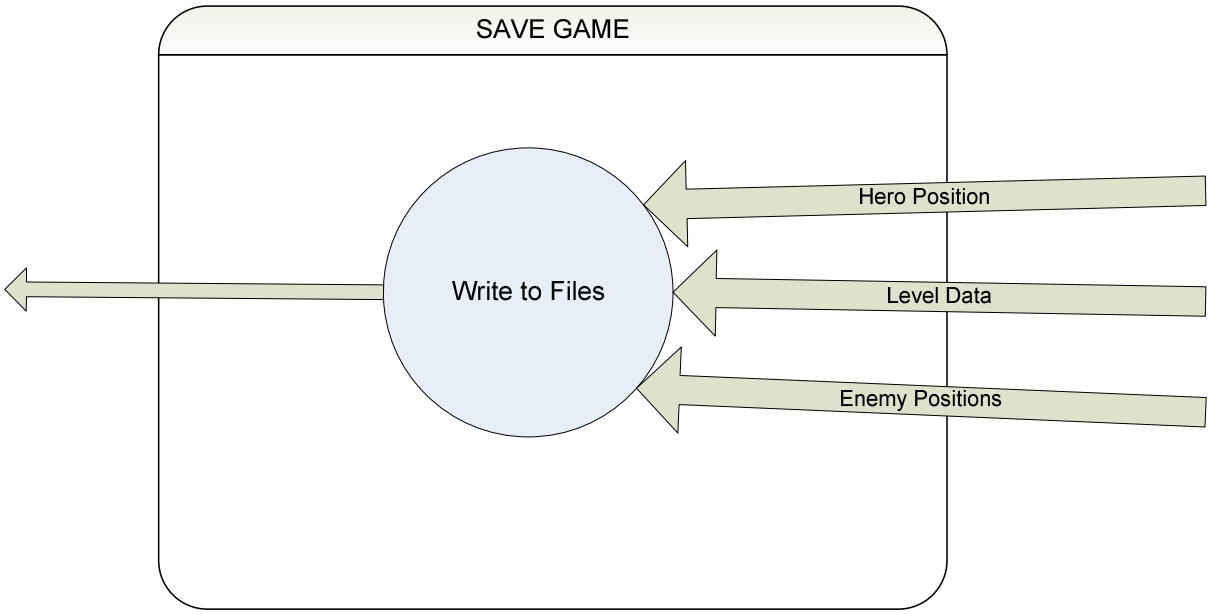


6.2 Game DFD Level 1



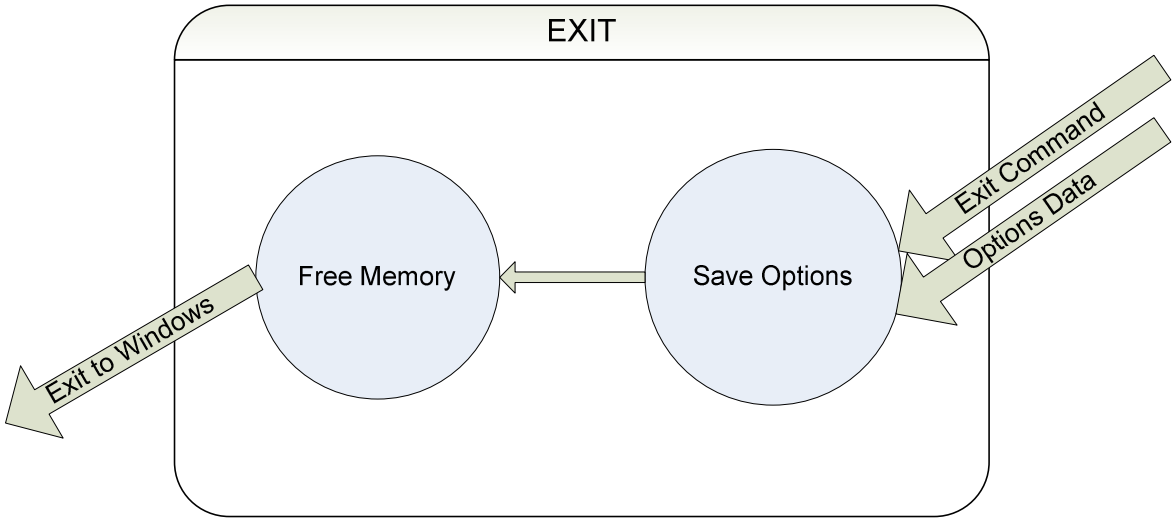
The GAME state is actually the main loop of the game. The user's mouse and keyboard inputs are fed into the INPUT state. In this state the character position and direction are affected by the mouse and keyboard inputs. There are two calculations in this state. First one checks whether the hero can actually move in the direction he is trying i.e if he is not trying to go through a wall. If he can move then the heroes position attributes are updated. The second calculation occurs when the mouse input is attack. When the user attacks character.shoot() function of the hero is called (go to character class for function's properties). Then the same input which entered INPUT state and the structure returned from character.shoot() is fed into the AI state.

6.3 Save Game DFD Level 1



The control comes to this state from GAME state. Hero position, level data, enemy positions which are currently in appropriate classes' objects are passed to the WRITE TO FILE state which in turn writes these into the harddisk in level file format (see file formats section for level file).

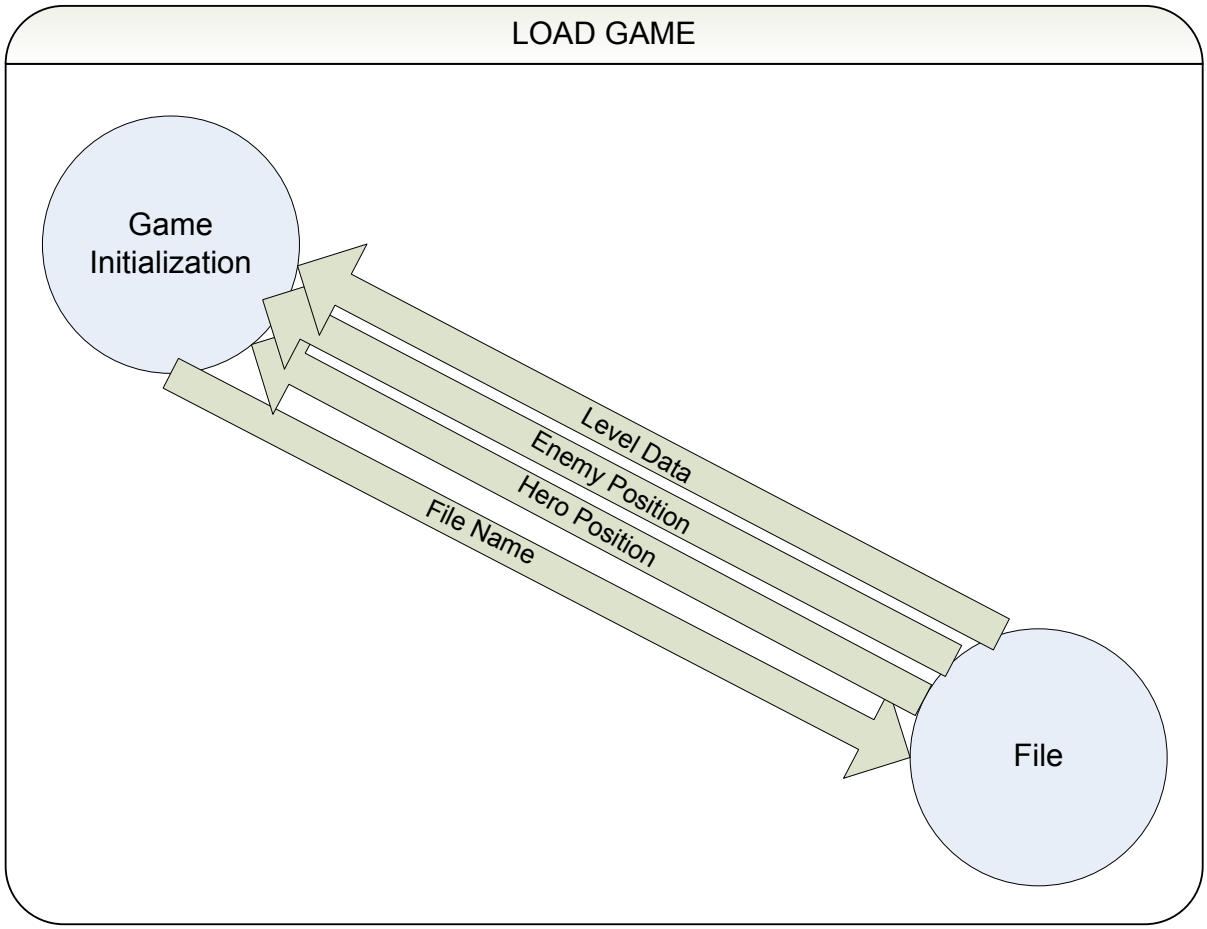
6.4 Exit State DFD Level 1



The 'option flag' is checked to see whether anything has been changed in the options menu. If a change is made user's option data that he has changed is fed into this state as data. So that when he loads the game in future, he will be able to play it with the same

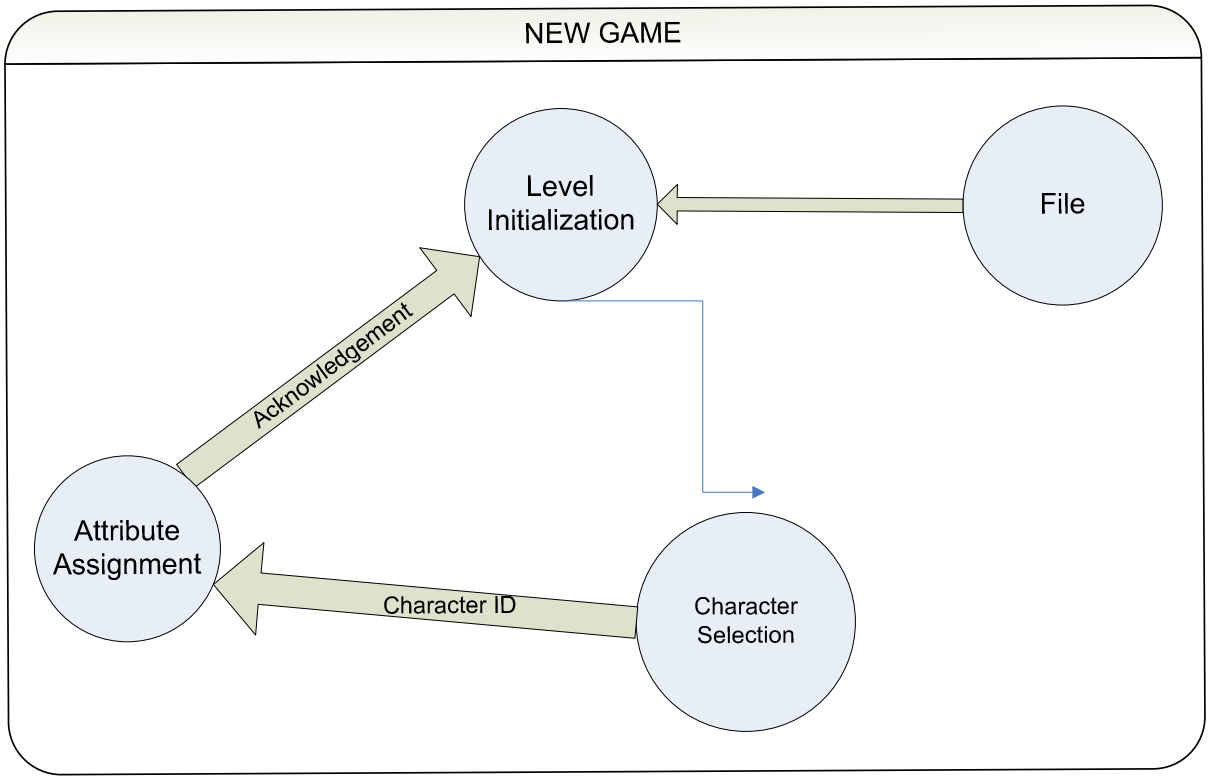
configuration. After saving options all the memory that has been allocated for level, map, characters, objects are deleted and the game exits to windows.

6.5 Load Game DFD Level 1



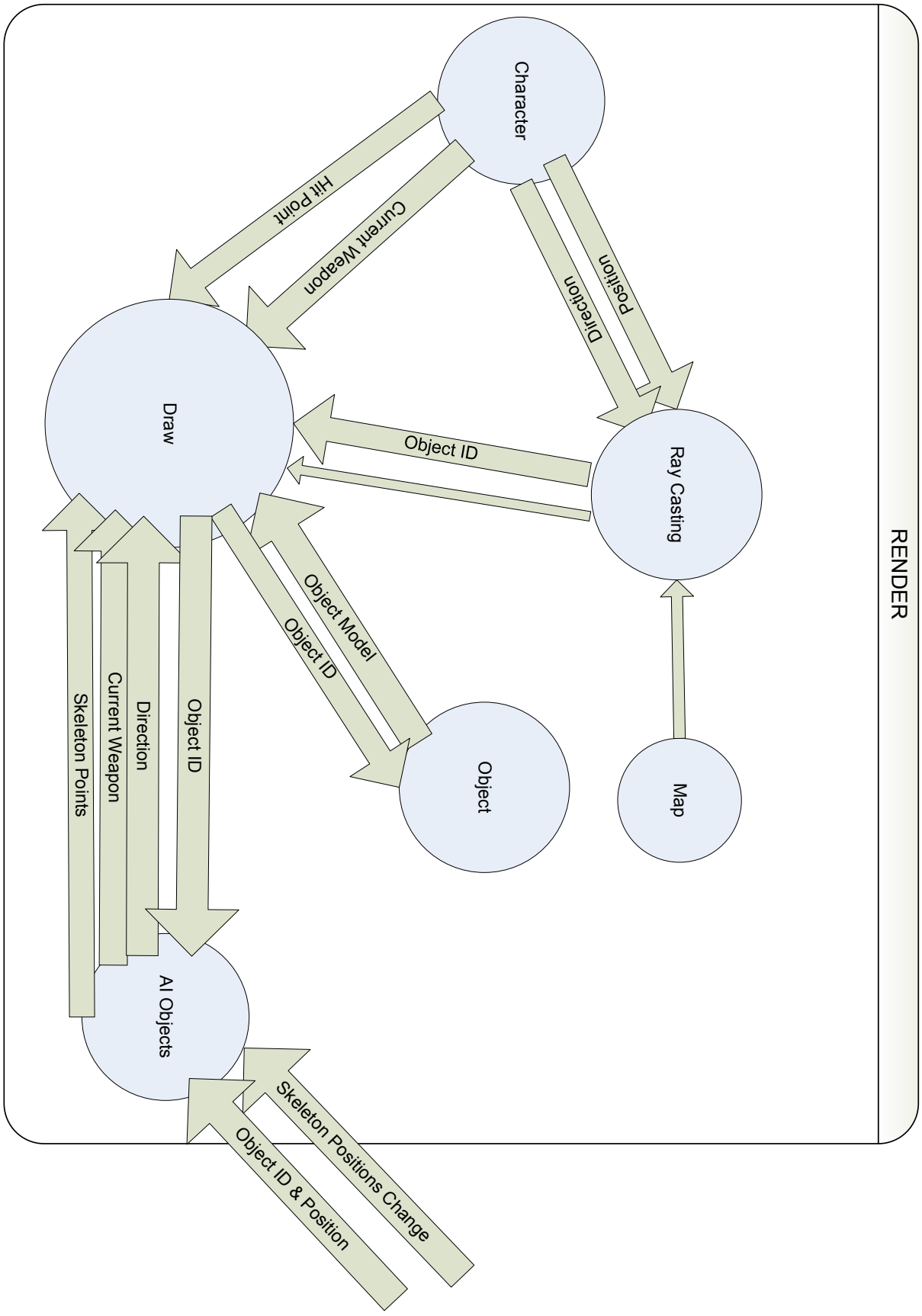
When the user is in START state, if he chooses to load a game he clicks on a saved game and then presses the load button. Therefore the filename is fed into the LOAD state. FILE state takes this filename input, and reads in the level file (see file formats for level file) which holds all the information about the level script files, enemy AI scripts, map, places of objects and characters (which can be both the hero or enemies) on the map and the characters attributes. Then passes these data to GAME INITIALIZATION state, which in turn stores them in the appropriate classes' instances for use in the game.

6.6 New Game DFD Level 1



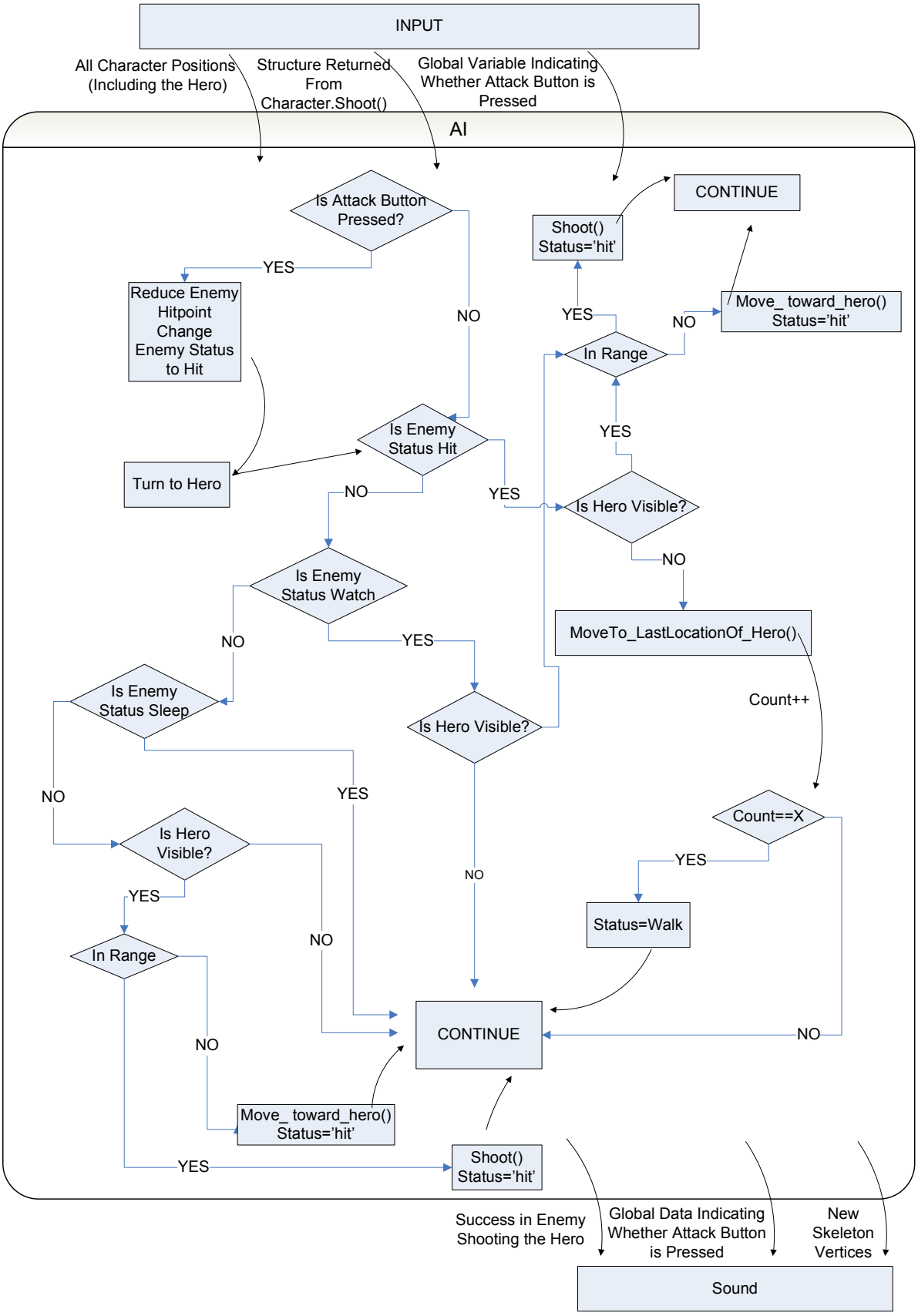
When the user is in START state, if he chooses to start a new game we are in LEVEL INITIALIZATION state. The user is prompted for character that he wants to play with in CHARACTER SELECTION state and then this state feeds the id of the character into the ATTRIBUTE ASSIGNMENT state which in turn creates a hero class object and puts the character file into it. Then FILE state reads in the 1st level file from the level script files and again creates the level class object and puts these values into the object.

6.7 Render DFD Level 2



There are 2 main states included in RENDER state. One of them is RAY CASTING and the other one is DRAW. In raycasting state only the map is drawn according to the input coming from the hero's position and direction, and additionally map data is coming from the level object. So in this state we have enough information for ray casting. In the level file map we have all the object id's so after this state finishes the object ids existing in this level are passed to the draw state in order. DRAW also gets the current_weapon and hitpoint. These two are needed because there will be a health gauge horizontally placed on the top left hand corner of the screen, which increases or decreases with respect to hitpoint and ofcourse the current weapon is the weapon in the hero's hand and it has to be displayed on the screen with it's model information (vertices, texture, faces etc.) as we are seeing from the hero's eye. All the objects that aren't connected to AI such as a desk, a door, a switch will be held in object class objects. DRAW state will give object id to OBJECT state and the model data of the one with that id will be passed to DRAW state. The other objects are related to AI for example enemies. The object id is passed to AI OBJECTS state and direction of the object, current weapon, model data, texture, and skeleton points are retrieved.

6.8 AI DFD Level 2



The input values passed from INPUT state to AI state are all the character's positions, a global variable indicating whether attack button is pressed or not, and the structure returned from character.shoot() function of the character class.

```
Struct shoot_output {
        int damage;
        char * sound_file_name;
        int position_x;
        int position_y;
}
```

All the characters have a status attribute which are 'Hit', 'Sleep', 'Walk', 'Watch'. The AI module makes the characters act in different fashions according to these attributes.

In the AI state the structure returned from character.shoot() contains amount of damage and the position of the enemy affected by the damage. We go to that position in the map and take the character* character which points to the enemy on that location then the amount of damage is deducted from the enemy's hitpoint and enemies status is changed to 'Hit'. Also if the enemy is not looking toward the hero then after character.shoot() is called(by the hero) the enemy's direction will be changed to hero's direction by character.turnToHero() (by the enemy) function. Then we check for enemy status. If it is 'Hit' we check for whether hero is in range or not. If he is not in range the character moves to the last seen position of the hero on the map and increment a movedtolastlocationcounter. When this character that was hit enters this AI module the second time his status will still be 'Hit' so AI will again check if hero is visible. If again the hero is not visible then the character will stop trying to go to the last position the hero was seen. After a predefined number of movedtolastlocationcounter is reached the AI will decide that the hero is not around so the status of the character (that was hit some time ago) will be changed to 'Walk'.

If the character is sleeping then AI module doesn't change his position on the map but ofcourse if the hero shoots at the sleeping character the status is changed to 'Hit' and the routine for characters with status attribute 'Hit' is executed.

If the character status is 'Walk' then AI makes the character change position in the predefined manner such as going back and forth between two points in the map. which will update the new value of the position of the enemy. Also whether the hero is visible or not will still be checked by the AI. If the hero is visible then range will be checked. If hero is near enough character.shoot() will be called and character status will be changed to 'Hit' even though this character wasn't hit. If hero is not in range character.movetowardhero() will be

called and the character status will be changed to ‘Hit’. We do this so that the character tries to search for the hero.

If the character status is ‘Watch’ the character stays in one position and the position is not changed as long as the hero is not seen or the status is not changed by AI because AI will change the status of the characters randomly to make the characters act like in a random manner. After some time which is passed to AI by a timer, AI will interchange the status other than ‘Hit’. For example after 5 minutes a character’s attribute will be changed from ‘sleep’ to ‘Watch’.

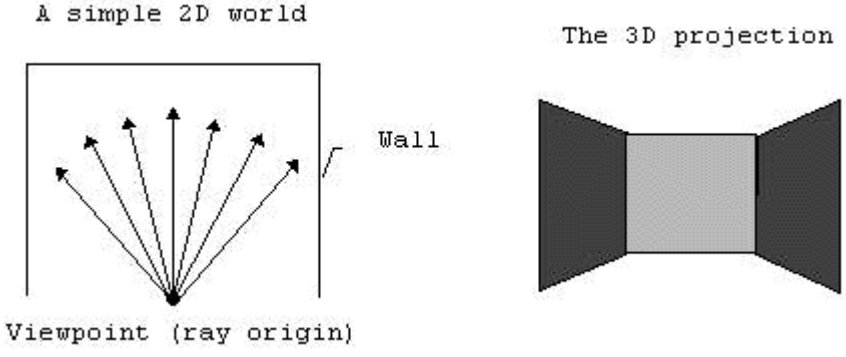
So the output of the AI state will be the ‘success in enemy shooting the hero’, ‘success in hero shooting the enemy’, ‘global data indicating whether attack button was pressed’, ‘the hero has moved’ and ‘new skeleton vertices of the visible enemies’. These values will be passed to the SOUND state. So if ‘success in hero shooting the enemy’ is passed then the ‘hit sound’ which was previously loaded is played. If ‘the hero has moved’ is output, then ‘walking sound’ will be played also in another thread. Then these values which were output from the AI state to SOUND state will also be passed to RENDER state from SOUND state.

7. Game Techniques

7.1 Ray Casting

We are planning to use **Ray Casting** method **in Flee from Alcatraz** as a rendering method because of its efficiency.

Ray Casting determines the visibility of of surfaces by tracing imaginary rays of light from viewer's eye. To reduces the number of rays it utilizes some “geometric constraints” such as the walls must be perpendicular with floors. So we only need the map of the environment for rendering because we know that if there is an object in the map we can render it by using geometric constraints . Figure shows the 3d projection of a simple map.



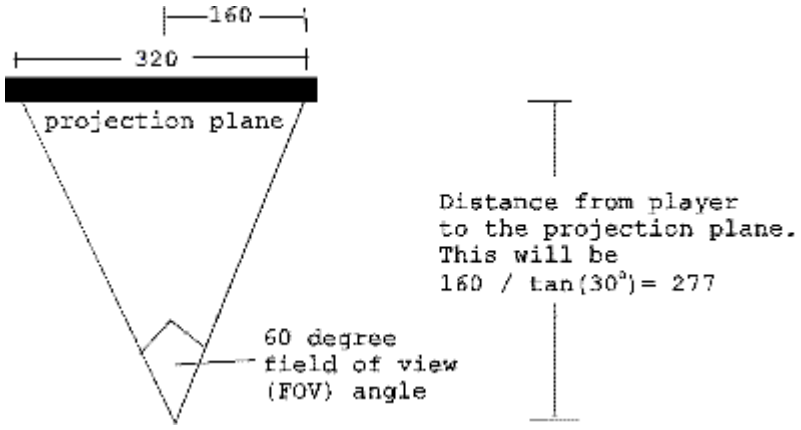
For an m*n display resolution , ray casting is n times faster because it sends only one ray for every vertical column and use geometric constraints to render the scene.

We must define the field of view (FOV) angle to now which part of the world in front of the character must be rendered. 60 degrees is the best FOV because the eye of a human can see with a 60 degrees FOV.

If we know the dimension of projection plane and FOV we can calculate distance between character and projection plane and the angle between subsequent rays from the below figure.

Angle between subsequent rays= 60/320 degrees

Distance from character to projection plane= 160/tan(30)=277 unit



Now we know FOV , angle between subsequent rays and dimension of projection plane so we can write our algorithm to render the scene.

Subtract 30 degrees from the direction of character.

Starting from column 0

A. Cast a ray.(An imaginary line extending from character

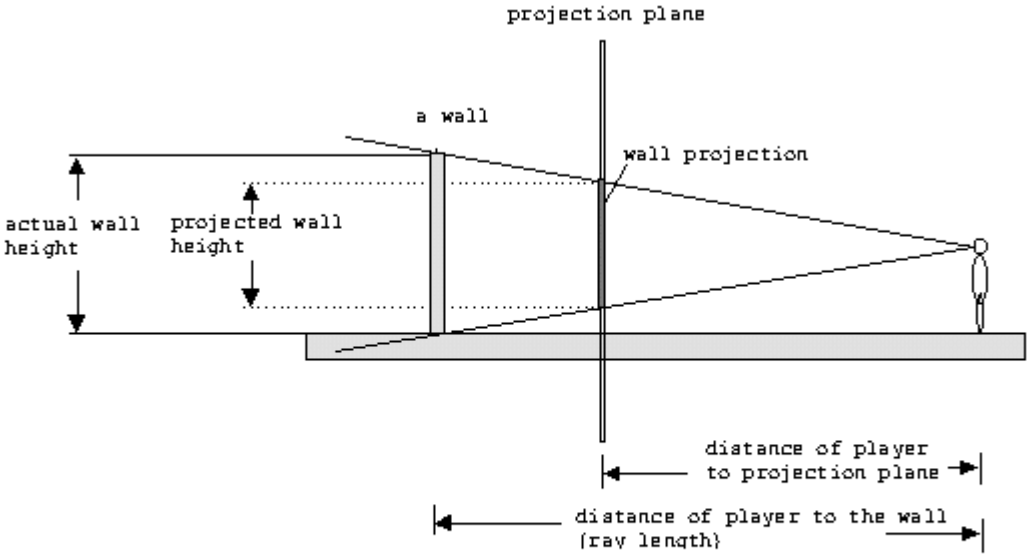
B. Trace the ray until it hits an object.

C. Record the distance to the object.

3. Add the angle increment (60 / 320) to get the next ray.
4. Repeat 2 and 3 until all rays are cast.

In 2.C to find the distance we can use sin cos and tan values. To be more efficient we can pre-compute them and put into tables.

Now we know the distance between objects and character for each 320 rays , we can project the object slice to the projection plane.

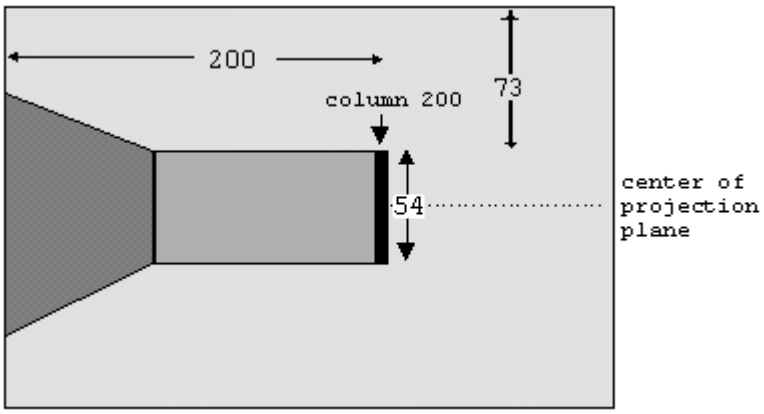


From similar triangle equation,

$$\frac{\text{projected wall height}}{\text{distance of player to projection plane}} = \frac{\text{actual wall height}}{\text{distance to the wall}}$$

If the height of the wall is 64 unit and the ray at column 200 hits a wall at distance of 300 units then the projected wall height will be

64/330*277=54 units. So the projection of the slice will look something like in the below figure.



7.2 Data Driven Game Design

Games are very complex applications and must be subdivided so that it would be easier to implement. They are usually divided into two main parts , Data and Logic. Otherwise , by programming all of them in code , any small changes to the data means a complete recompile. This means that staff other than programmers can not make changes to the game design. To separate game logic and game data , game logic should be an executable application and game data must be separated from game logic into external data files. This process is called **Data Driven Game Design.**

To make the separation of game data from game logic we need to a parser that can be read these external data. In flee from alcatraz we will use phyton as scripting language to read external files.

In our game we will keep constants and some AI behaviors in external files. Properties of weapons and characters , and map information can be some examples to what an external file contains. By keeping these data in externals file we can easily make changes in our data and we can see the results of our changes without recompiling the code.

8. Sound and Music

Music and Sound effects are integrals part of any 3D adventure game. These effects help the game player feel more in the action. SDL (simple direct media layer) will be used for this purpose. “SDL is a cross-platform multimedia library designed to provide low level access to audio, keyboard, mouse, joystick, 3D hardware via OpenGL, and 2D video framebuffer.”[http://www.libsdl.org/index.php]. It is used by MPEG playback software, emulators, and many popular games, including the award winning Linux port of "Civilization: Call To Power". SDL is a library which normally requires installation of it and any of its extension libraries on the system which the game will run. Since our development and testing platforms are “windows” machines, it is possible to perform these installations.

Audio for the game will be mixed through the use of SDL Mixer. There will be multiple channels for sound effects and a channel for background music.

SDL_mixer: "SDL_mixer is a simple multi-channel audio mixer library. It supports any number of simultaneously playing channels of 16 bit stereo audio, plus a single channel of music" [http://www.libsdl.org/projects/SDL_mixer/index.html]

SDL Mixer supports many formats for playing music and sound samples including the formats that we will use which are ogg, mp3 and wav.

Here are the steps in using SDL_Mixer in a program

Open up the audio device

Load samples into memory

Play them when necessary

Clean up

To use SDL_mixer functions in a C/C++ source code file,SDL_mixer.h has to be included with the main library of SDL which is “SDL.h”. Required library files (ex:SDL.lib) should be added to the project.

#include <SDL_mixer.h>

#include "SDL.h"

Starting from here we will try to write some information about the main functions of SDL_Mixer.

When using SDL mixer functions, the use of some SDL functions has to be avoided. (Ex:SDL_OpenAudio, SDL_LockAudio).

Before using SDL_mixer functions , SDL must be initialized with SDL_INIT_AUDIO. After that the most important function of SDL_mixer library which is Mix_OpenAudio should be called with the required frequency.

int Mix_OpenAudio(int frequency,Uint16 format,int channels,int chunksize);

Most games use 22050Hz and some of them use 44100Hz for frequency. The frequency in our game will be decided later. Chunksize is the size of each mixed sample. If this number is small on a slow system sound may be skip. If it is too large sound effects will lag behind the action more. So that a medium
value will be chosen for this integer.

Mix_chunk *Mix_LoadWav(char *file)

*This function loads the sample file for example “sample.wav” .Then Video Mode should be chosen by SDL_SetVideoMode function.

Int Mix_PlayChannel (int channel,Mix_Chunk *chunk,int loops)

Channel--> Channel to play on. For the first free unreserved channel “-1” is chosen.

Chunk --> sample to play.

Loops --> It is the number of loops. -1 means infinite loop. 1 means the sample will be played with one loop that is two times.

Int Mix_HaltChannel(int channel)

channel--> channel to stop playing , or -1 for all channels

Int Mix_PlayMusic (Mix_Music *music ,int loops)

music--> pointer to Mix_Music to play.

Loops-->Number of times to play through the music.

To shutdown and cleanup the mixer Mix_CloseAudio should be called.

void Mix_CloseAudio();

After calling this all audio, is stopped and the device is closed. However it has to be called the same number of times that Mix_OpenAudio called.

Sound Effects:

Mostly, We will use preconstructed sound effects in our game that can be taken from following websites.

http://www.therecordist.com/pages/game_sfx.html

http://soundmatter.thegamecreators.com/?f=pack3

9. FILE FORMATS

In this part of the report the file formats used by the game will be described. There is an example after each attributes in parenthesis for clearance . At the end of each attributes there will be an end line character.

9.1 Character File Format

Name	(novart1)
Position	(35 21)
Direction	(53 degree)
Speed	(15)
Model	(novart1.obj)
Objects	(112 53 35)
Current Object	(53)
Weapons	(10 15 33)
Current Weapon	(33)
Hit Point	(35)
Special Ability	(43)

9.2 Weapon File Format

Name	(rifle)
Range	(20)
Weight	(3)
Texture	(rifle.bmp)
Model	(rifle.obj)
Damage	(15)
Splash Damage	(0)
Magazine Capacity	(12)

9.3 Map File Format

Name	(map1)
Map size	(50 30)
Map information	(5 3 0 , 5 0 character information ... *)

*From map information example we understand that in the coordinate position (0,0) the height = 5 , there is an object with id = 3 and there in no character. In the position (0,1) the height =5 there is not any object but since the third value is not 0 there is information about a character until another ','.

9.4 Object File Format

Name	(box)
Weight	(10)
Texture	(box.bmp)
Model	(box.obj)

9.5 obj File Format

The file format to make 3D modeling in our game is obj. format. Blank space and blank lines can be added to obj. files for readability .The obj. file format includes the followings.

 # some text

means that it is a comment line

v float float float

The coordinates of vertex's geometric position in space. The first vertex has index 1 and subsequent vertices are numbered sequentially.

vn float float float

It represents the coordinates of a normal. The first vertex has index 1 and subsequent vertices are numbered sequentially.

vt float float

It represents a texture coordinate. The first texture coordinate has index 1 and subsequent texture coordinates are numbered sequentially.

f int int int or

f int/int int/int int/int or

f int/int/int int/int/int int/int/int

It represents a polygonal face. The numbers are indexes into the arrays of vertex positions , texture coordinates and normals respectively. A number may be omitted if , for example , a texture coordinates are not being defined in the model.

g group name

It represents a group. When a group is defined it remains the current group until another group is defined. If no group is specified in the file, everything in the file is in the current group.

usemtl material name

It represents a material. Like groups materials are applied to all faces following the material declaration until another material declared.

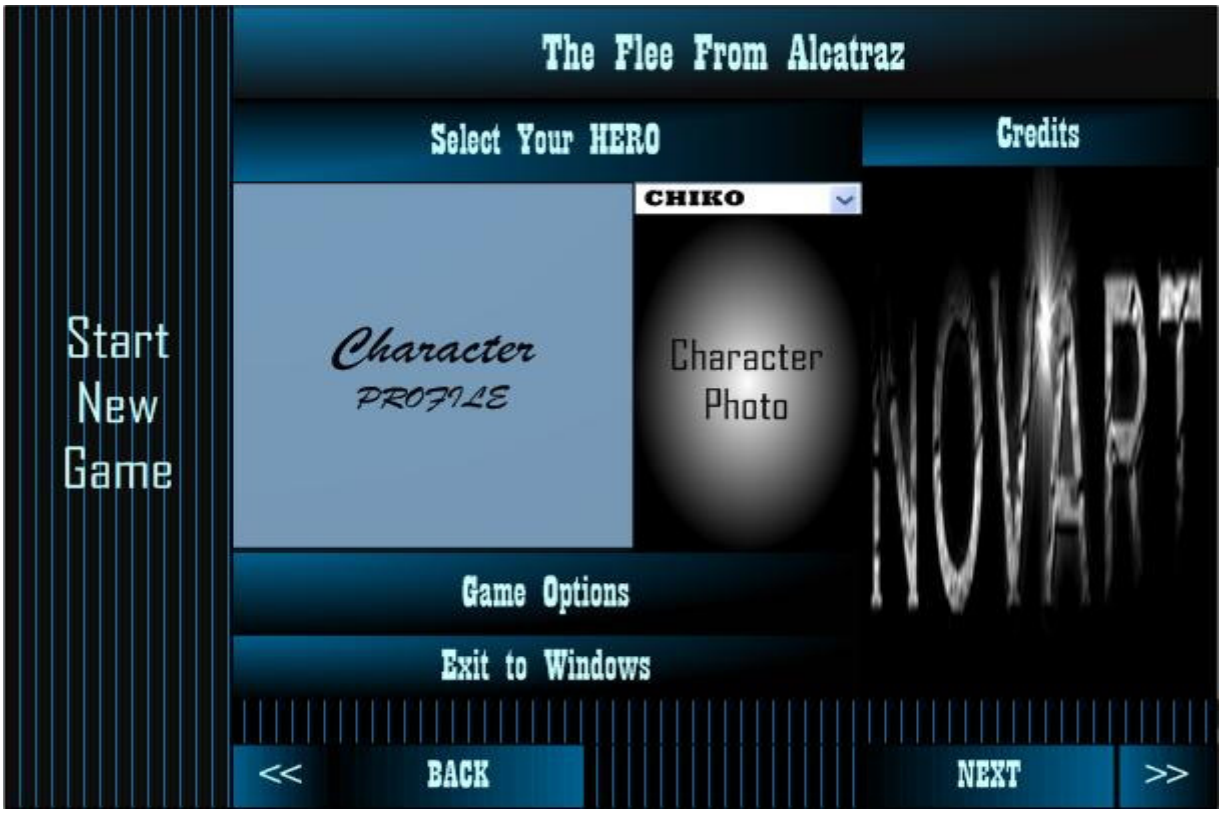
A valid obj. file can be made using only “v” ”f” flags. The use of other flags are optional.

10. User Interfaces

10.1 Main Menu

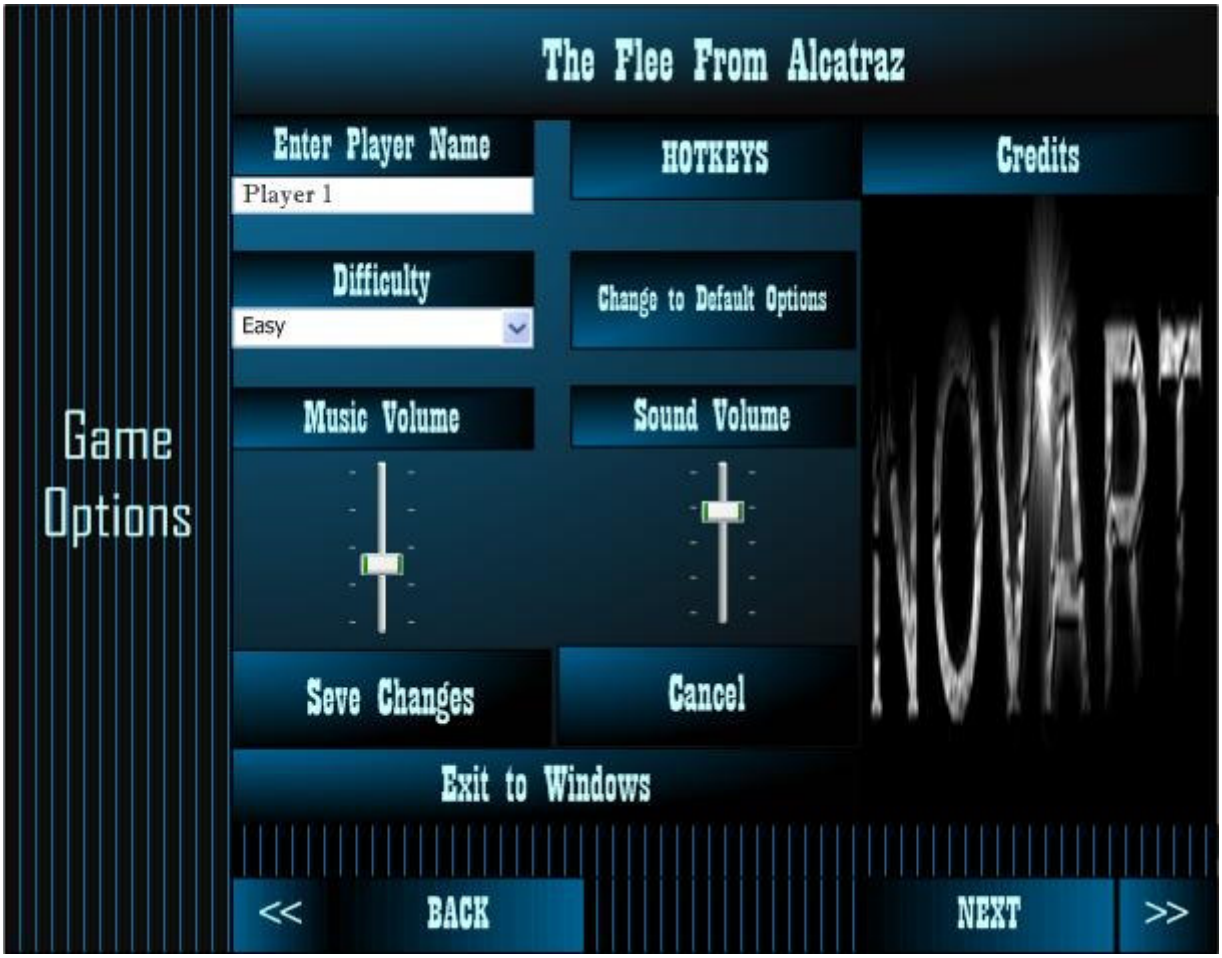
10.2 Load Game

10.3 Start New Game

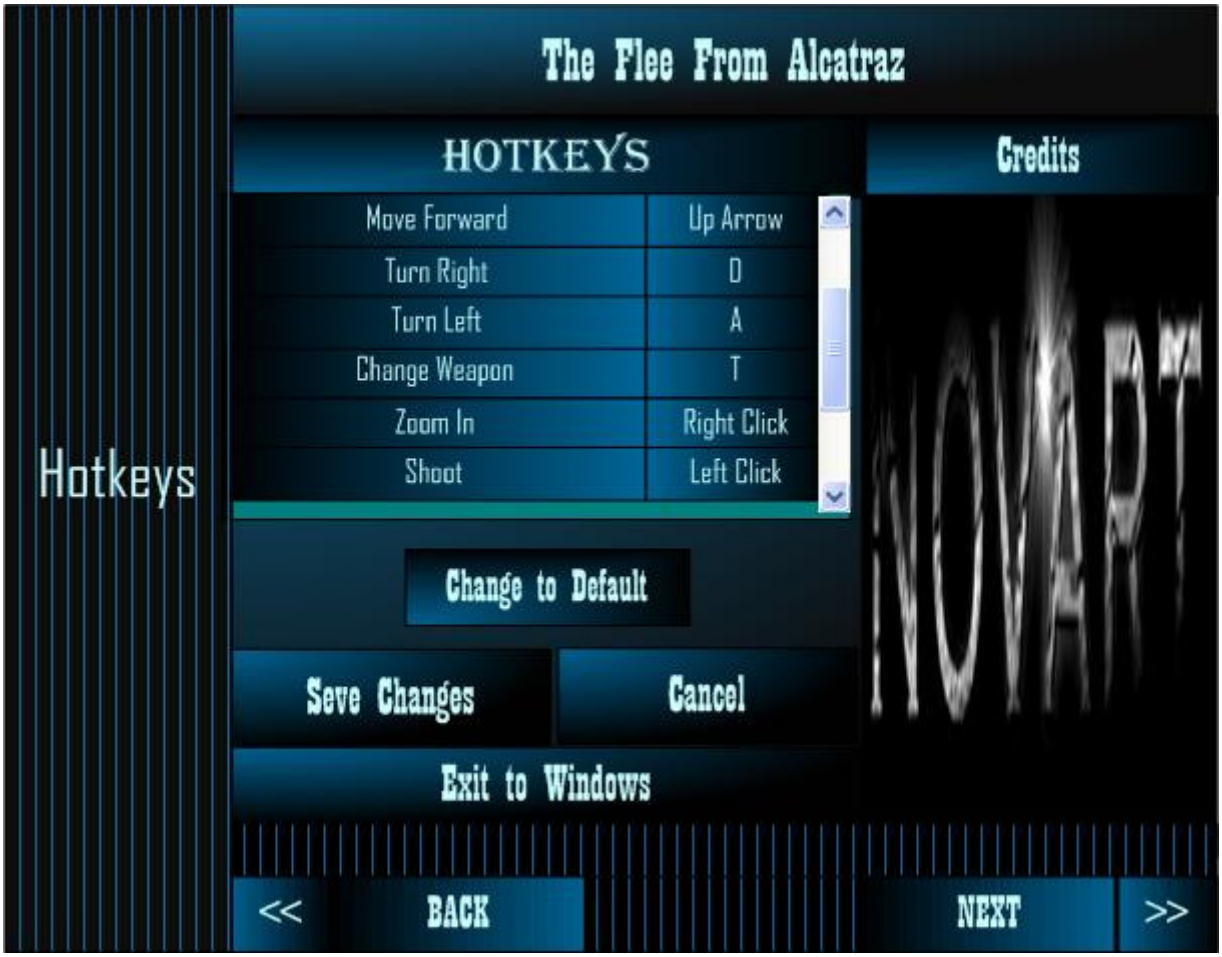


10.4 Main Menu when game is paused

10.5 Game Options

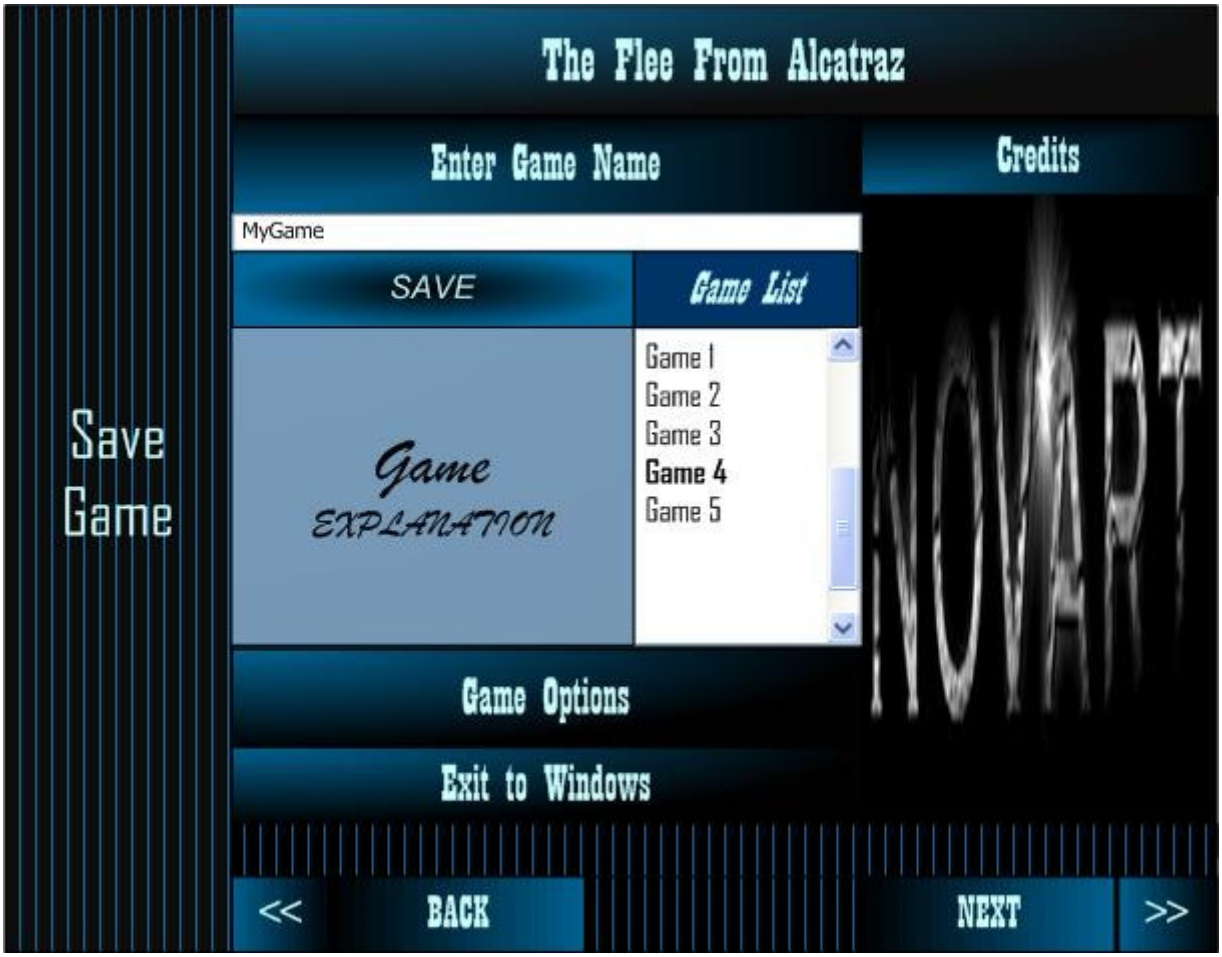


10.6 Hotkeys


The Flee From Alcatraz
HOTKEYS
Credits
Move Forward
Up Arrow
Turn Right
D
Turn Left
A
Change Weapon
T
Zoom In
Right Click
Shoot
Left Click
Hotkeys
Change to Default
Seve Changes
Cancel
Exit to Windows
<<
BACK
NEXT
>>


Exit to Windows?
OK
CANCEL

10.7 Save Game



11. Conclusion

In this document we have outlined the technical aspects of our game development project. Upto this time we had been very concerned about the implementation phase. Now we have created the foundation for our game and the implementation will build on top of it. The classes, data structures, modules have been decided which help us to see the whole picture rather than the details. The flow chart have been created which help us visualize the main flow of control. The data flow diagrams have been created which help us to visualize the flow of data, what module needs what kind of data, which module has to be connected to which one.

To conclude the making of this initial design document helped us create a picture of what is going on in the game control and architecture.