# CENG 491

# Senior Design Seminar and Project


***Presents***

**"The Flee From Alcatraz"**

Detailed Design

10.01.2005

**Authored By:**

M.Zahit Özcan 1250588
İbrahim Özbay 1203355
Serhat Solak 1250711

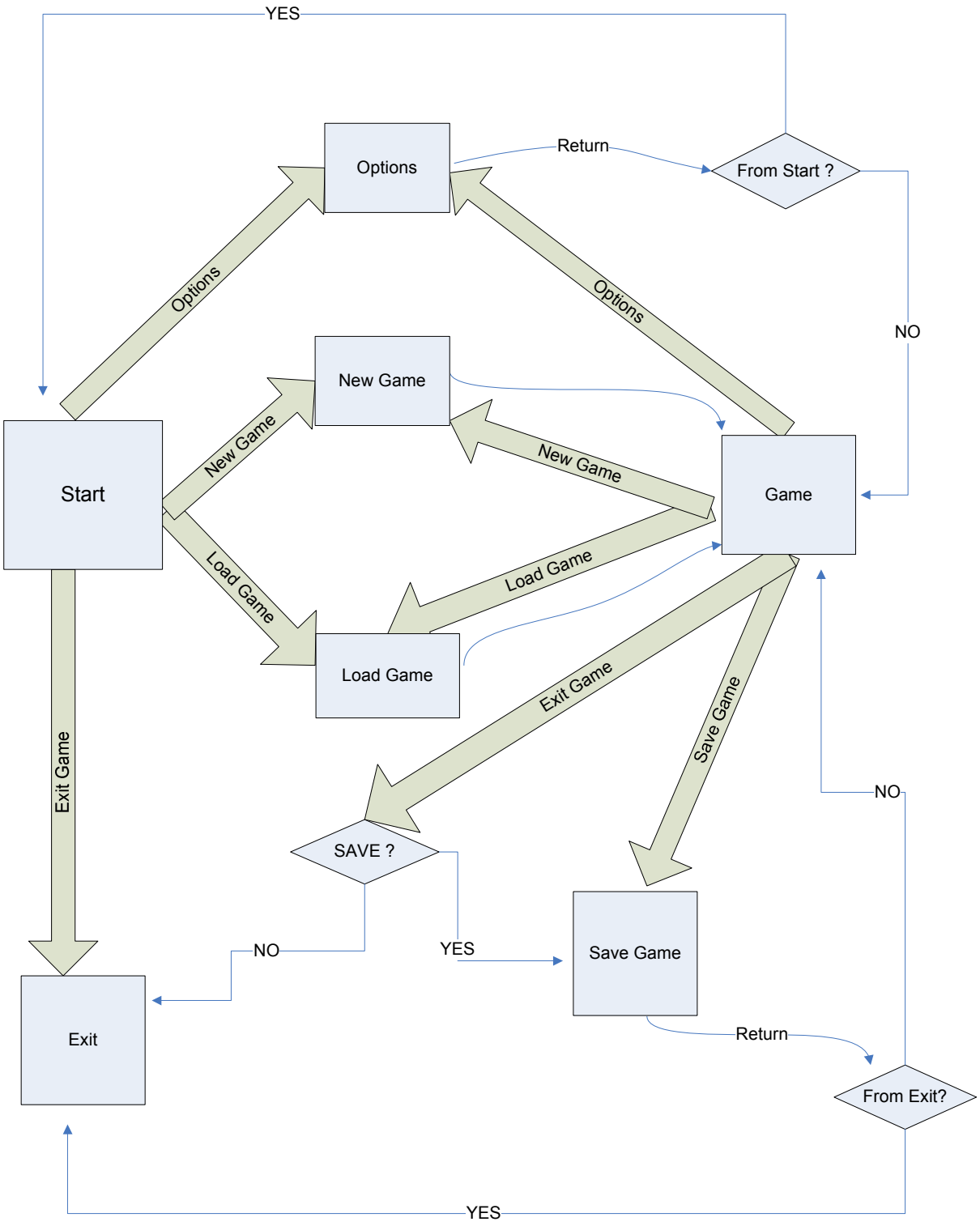
# 1. Introduction

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.

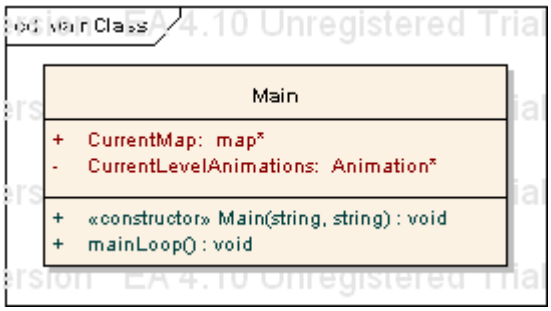
**5.** In the SAVE state the player gives a filename and all the level data is written to a save file. The control can come to this state either when escape buton is pressed or when the user presses save buton in GAME state. Then the control moves back to the state that it came from.

**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. Modules

### 3.1 Main Class




Main Attributes

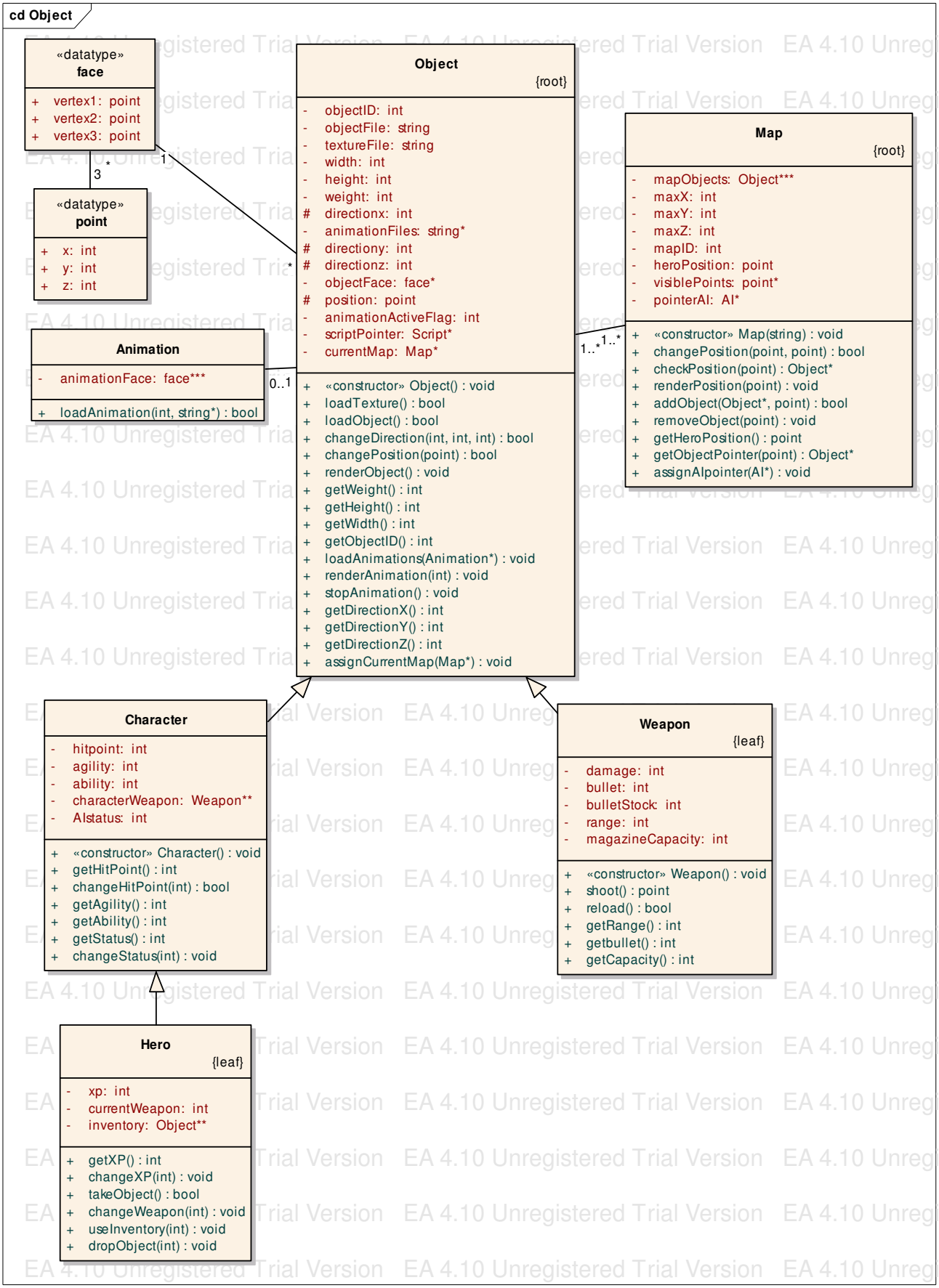
| Attribute | Details |
|---|---|
| *public map** | |

| | |
|---|---|
| *private* *Animation* | |

Main Methods

| **Operation** | **Notes** |
|---|---|
| *public*<br>**Main**(<br>string levelMapFileName,<br>string levelScriptFileName):void | Sequential<br><<constructor>>*Parameters:*<br>in string levelMapFileName<br>in string levelScriptFileName<br><br>it takes the level script file name as parameter, it goes and opens the data driven file to initialize attributes.Also it defines an object of map class with the parameter levelMapFileName and it defines an object of script class with the parameter levelScriptFileName. |
| *public*<br>**mainLoop**():void | Sequential<br>this is the main loop of the game where all the modules are called one after the other. eg. Input->AI->Sound->Render. |

## 3.2 Level Module



## 3.3 Object Module

Object Attributes

| Attribute | Details |
|---|---|
| *private int* **objectID** | If you have two models of the same kind on the map they both have the same object id. therefore it will be used for identifying the type of the objects. eg. the enemy objects will be given objectID's in the range 0-10 but the enemy with object id 0 is different than the one with object id 1 (regarding the look of the model) |
| *private string* **objectFile** | name of the file containing the model vertices. |
| *private string* **textureFile** | the name of the texture file that stores the texture image of the object. |
| *private int* **width** | this is the maximum possible width of the object. |
| *private int* **height** | this is the maximum possible height of the object. |
| *private int* **weight** | |
| *protected int* **directionx** | it specifies the angle of the object from the x axis. |
| *private string* **animationFiles** | contains the filenames of all the keyframes that constitute all possible animations that the character can make. |
| *protected int* **directiony** | it specifies the angle of the object from the y axis |
| *protected int* **directionz** | it specifies the angle of the object from the z axis |
| *private face* **objectFace** | this is the pointer to the planes forming the model. |
| *protected point* **position** | this is the current position of the object on the map object |
| *private int* **animationActiveFlag** | indicates whether any animation is active on some object. |
| *private Script** | contains the address of script class object which is |

| | |
|---|---|
| **scriptPointer** | created in the main package, when the program starts to run. |
| *private* *Map*<br>**currentMap** | |

Object Methods

| Operation | Notes |
|---|---|
| *public*<br>**Object**():void | Sequential<br><<constructor>><br>it doesn't take any parameter but using object id, it goes and opens the data driven file to initialize attributes. |
| *public*<br>**loadTexture**():bool | Sequential<br>loads the texture of the object after reading the string textureFile. |
| *public*<br>**loadObject**():bool | Sequential<br>loads the object model after reading the string objectFile and allocates necessary storage for the objectFace. |
| *public*<br>**changeDirection**(<br>int anglez,<br>int angley,<br>int anglex):bool | Sequential*Parameters:*<br>in int anglez<br>in int angley<br>in int anglex<br><br>takes three integers and rotates the object using these with respect to the x, y and z axes. |
| *public*<br>**changePosition**(<br>point newposition):bool | Sequential*Parameters:*<br>in point newposition<br><br>changes the position attribute of the object by the input newposition amount. |
| *public*<br>**renderObject**():void | Sequential<br>draws the object on the screen by using the objectFace. First of all it checks whether the animationActiveFlag is false. If it is false it works as normal, and renders the objectFace. If it is true it doesn't render the objectFace so that only the animation of that object is rendered. |

| | |
|---|---|
| *public*<br>**getWeight**():int | Sequential<br>returns the weight of the object. |
| *public*<br>**getHeight**():int | Sequential<br>returns the height of the object. |
| *public*<br>**getWidth**():int | Sequential<br>returns the width of the object. |
| *public*<br>**getObjectID**():int | Sequential<br>returns the objectID of the object. |
| *public*<br>**loadAnimations**(<br>Animation*<br>currentAnimationClass):void | Sequential*Parameters:*<br>in Animation* currentAnimationClass<br><br>this function calls the |
| *public*<br>**renderAnimation**(<br>int animationID):void | Sequential*Parameters:*<br>in int animationID<br><br>Gets the id of the animation as input and plays the animation keyframes of that object. e.g Animation MainAnimation.animationFace[objectID][animationID] : plays the walking keyframes of the character by some interpolation. also changes the position of the object in the current class and also on the map. Makes the animationActiveFlag true. |
| *public*<br>**stopAnimation**():void | Sequential<br>stops the current animation if there is any. animationActive flag is reset to zero. |
| *public*<br>**getDirectionX**():int | Sequential<br>returns the direction of the object with respect to x axis. |
| *public*<br>**getDirectionY**():int | Sequential<br>returns the direction of the object with respect to y axis. |
| *public*<br>**getDirectionZ**():int | Sequential |

| | |
|---|---|
| | returns the direction of the object with respect to z axis. |
| *public*<br>**assignCurrentMap**(<br>Map* pointerMapObject):void | Sequential*Parameters:*<br>in Map* pointerMapObject |

## 3.4 Weapon Module

Weapon Attributes

| Attribute | Details |
|---|---|
| *private int*<br>**damage** | |
| *private int*<br>**bullet** | |
| *private int*<br>**bulletStock** | |
| *private int*<br>**range** | |
| *private int*<br>**magazineCapacity** | the maximum number of bullets the weapon can hold. |

Weapon Methods

| Operation | Notes |
|---|---|
| *public*<br>**Weapon**():void | Sequential<br><<constructor>><br>it doesn't take any parameter but using object id, it goes and opens the data driven file to initialize attributes. |
| *public*<br>**shoot**():point | Sequential<br>returns the point on the map that is hit. |
| *public*<br>**reload**():bool | Sequential<br>returns whether the hero was able to reload the weapon, |

| | |
|---|---|
| *public*<br>**getRange**():int | Sequential<br>returns the range of the weapon. |
| *public*<br>**getbullet**():int | Sequential<br>returns how many bullets are left in the magazine. although it may seem unnecessary, we will use it to display it on the screen for the user to see how many bullets are left. |
| *public*<br>**getCapacity**():int | Sequential<br>returns the capacity of the magazine (for rendering |

## 3.5 Character Module

Character Attributes

| Attribute | Details |
|---|---|
| *private int*<br>**hitpoint** | the current amount of health the character has. |
| *private int*<br>**agility** | determines how fast the character can move. |
| *private int*<br>**ability** | specifies the specific ability a character has. |
| *private Weapon**<br>**characterWeapon** | stores the address of the weapon that the character has. |
| *private int*<br>**AIstatus** | sleep, walk, run, hit, etc. |

Character Methods

| Operation | Notes |
|---|---|
| *public*<br>**Character**():void | Sequential<br><<constructor>><br>it doesn't take any parameter but using object id, it goes and opens the data driven file to initialize attributes. |

| | |
|---|---|
| *public*<br>**getHitPoint**():int | Sequential<br>returns the current hitpoint of the character. |
| *public*<br>**changeHitPoint**(<br>int difference):bool | Sequential*Parameters:*<br>in int difference<br><br>First of all changes the hitpoint according to the difference parameter. Returns true if hitpoint is smaller than zero, which means the character dies, also if the character gets some health packs, it makes sure that the health doesn't pass over 100%. |
| *public*<br>**getAgility**():int | Sequential<br>returns the agility of the character. |
| *public*<br>**getAbility**():int | Sequential<br>returns the ability of the character. |
| *public*<br>**getStatus**():int | Sequential<br>returns the status of the character. |
| *public*<br>**changeStatus**(<br>int newStatus):void | Sequential*Parameters:*<br>in int newStatus<br><br>this function will be called by the interface between AI and Level modules. It takes the new status of the character and changes it simultaneously by calling the renderAnimation function so that the animation reflects the status of the character. |

## 3.6 Hero Module

Hero Attributes

| **Attribute** | **Details** |
|---|---|
| *private int* | the experience point determining the amount of |

| | |
|---|---|
| *private int*<br>**currentWeapon** | it is used to determine the weapon the hero is currently using. In the base class of hero which is character class the currentWeapon attribute won't be used because all enemies will have one weapon in their hands. |
| *private Object** | includes the objects that hero can carry other than |

Hero Methods

| **Operation** | **Notes** |
|---|---|
| *public*<br>**getXP**():int | Sequential<br>returns the xp of the hero. |
| *public*<br>**changeXP**(<br>int difference):void | Sequential*Parameters:*<br>in int difference<br><br>takes difference parameter as input and changes the xp accordingly. |
| *public*<br>**takeObject**():bool | Sequential<br>when the hero is closer than a predefined distance to an object which can be collected, the image of a hand will be displayed instantly. so that the player will be aware that there is some collectable object infront of him. from the position on the map this function will reach the object and get its ID. so that it will know where to place this object (characterWeapon or inventory). eg. the weapon ids are in some interval other than inventory ids. after pressing the 'E' button this function will be called for this hero. This function changes the position of the collected object on the map. return type is bool, because the hero has a limited storage and if he cannot collect the object this function returns false. |
| *public*<br>**changeWeapon**(<br>int newweapon):void | Sequential*Parameters:*<br>in int newweapon<br><br>currentWeapon is changed to the new weapon. |
| *public*<br>**useInventory**(<br>int inv):void | Sequential*Parameters:*<br>in int inv |

| | |
|---|---|
| | uses the inventory specified with inv on some object infront of the hero. |
| *public*<br>**dropObject**(<br>int objectID):void | Sequential*Parameters:*<br>in int objectID<br><br>First of all it checks if the object to be dropped is an inventory object. If so it is removed from inventory and added to the map by using the Map's addObject(Object* point) after checking if it can be dropped to that point with Map's checkPosition(point). If the object to be dropped is a weapon the same things above apply just that it will be deleted from characterWeapon list and the next weapon is assigned as the current weapon. |

## 3.7 Map Module

Map Attributes

| Attribute | Details |
|---|---|
| *private Object***<br>**mapObjects** | holds the pointers to the objects on the x, y, z point. |
| *private int*<br>**maxX** | |
| *private int*<br>**maxY** | |
| *private int*<br>**maxZ** | |
| *private int*<br>**mapID** | every map is given a unique id. a level may consist of a few maps if the level should have a big size. |
| *private point*<br>**heroPosition** | holds the position of the hero. |
| *private point*<br>**visiblePoints** | |
| *private AI*<br>**pointerAI** | |

Map Methods

| Operation | Notes |
|---|---|
| *public*<br>**Map**(<br>string levelMapFileName):void | Sequential<br><<constructor>>*Parameters:*<br>in string levelMapFileName<br><br>it takes map script file name as parameter, it goes and opens the data driven file to initialize attributes and allocate necessary space.Also it calls the constructors of each object that is on the map. |
| *public*<br>**changePosition**(<br>point newpoint,<br>point oldpoint):bool | Sequential*Parameters:*<br>in point newpoint<br>in point oldpoint |
| *public*<br>**checkPosition**(<br>point checkPoint):Object* | Sequential*Parameters:*<br>in point checkPoint<br><br>returns the pointer to the object on that position we are checking with the checkPoint parameter. |
| *public*<br>**renderPosition**(<br>point renderPoint):void | Sequential*Parameters:*<br>in point renderPoint<br><br>renders the object on that point. it first gets the object on that point and calls its Object::renderObject |
| *public*<br>**addObject**(<br>Object* newObject,<br>point objectPoint):bool | Sequential*Parameters:*<br>in Object* newObject<br>in point objectPoint<br><br>adds the object to the position given by the point parameter to the map. if there is already an object on that position it returns false. |
| *public*<br>**removeObject**(<br>point position):void | Sequential*Parameters:*<br>in point position<br><br>deletes the object from the position given by the position parameter. |
| *public* | |

| | |
|---|---|
| **getHeroPosition**():point | Sequential<br>returns the position of the hero on the map. |
| *public*<br>**getObjectPointer**(<br>point position):Object* | Sequential*Parameters:*<br>in point position<br><br>returns the pointer of the object at the position. |
| *public*<br>**assignAIpointer**(<br>AI* pointerAIobject):void | Sequential*Parameters:*<br>in AI* pointerAIobject |

point Attributes

| Attribute | Details |
|---|---|
| *public int*<br>**x** | |
| *public int*<br>**y** | |
| *public int*<br>**z** | |

face Attributes

| Attribute | Details |
|---|---|
| *public point*<br>**vertex1** | |
| *public point*<br>**vertex2** | |
| *public point*<br>**vertex3** | |

## 3.8 Animation Module

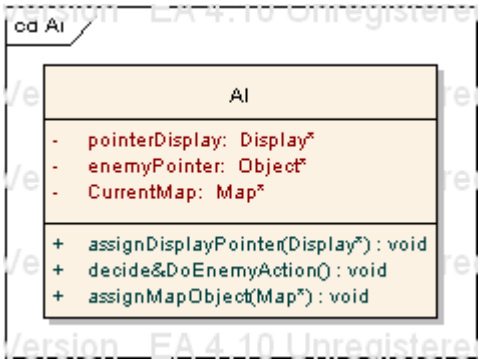
Animation Attributes

| Attribute | Details |
|---|---|
| *private face*** **animationFace** | |

Animation Methods

| Operation | Notes |
|---|---|
| *public* **loadAnimation**( int objectID, string* animationFiles):bool | SequentialParameters: in int objectID in string* animationFiles<br><br>First it looks at the AnimationFace and checks if the animation of the caller object's type (objectID), is loaded or not. If it is not loaded, it loads the animations required for that object. e.g Animation |

## 3.9 AI Module




AI Attributes

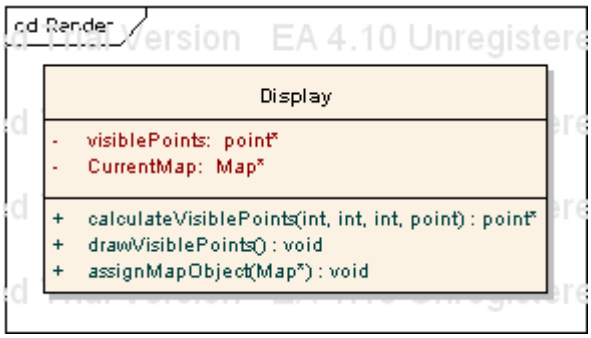
| Attribute | Details |
|---|---|
| *private Display* **pointerDisplay** | |
| *private Object* **enemyPointer** | |

| | |
|---|---|
| *private Map** **CurrentMap** | |

AI Methods

| Operation | Notes |
|---|---|
| *public* **assignDisplayPointer**( Display* pointerDisplayObject):void | Sequential*Parameters:* in Display* pointerDisplayObject |
| *public* **decide&DoEnemyAction**():void | Sequential this is the main function of the AI class.It finds the enemyPositions one by one by using enemyPointer attribute of the current class and calls the calculateVisiblePoints function of Display Class with the positions of enemies.It gets the visible points of enemies from that function as return value.After that it decides the next action of the enemy according to the objects that are on the visible points.Then it changes the status of the enemy according to do decided action. |
| *public* **assignMapObject**( Map* pointerMapObject):void | *Parameters:* in Map* pointerMapObject |

## 3.10 Display Module




Display Attributes

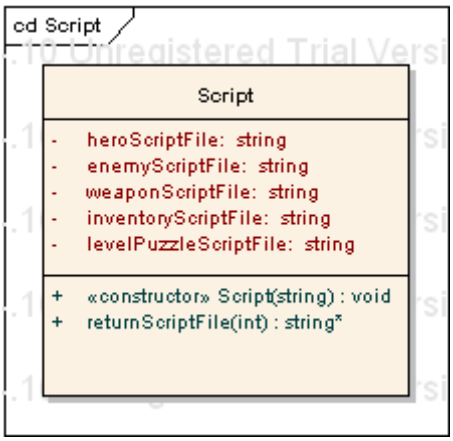
| Attribute | Details |
|---|---|
| *private point** **visiblePoints** | |

| | |
|---|---|
| *private Map** **CurrentMap** | |

Display Methods

| Operation | Notes |
|---|---|
| *public* **calculateVisiblePoints**( int directionZ, int directionY, int directionX, point currentPoint):point* | Sequential*Parameters:* in int directionZ in int directionY in int directionX in point currentPoint<br><br>This function calculates the visible points for the object whose current coordinates and directions are given as input.And returns them while it stores them in the visiblePoints. |
| *public* **drawVisiblePoints**():void | Sequential<br>calls all the objects' render functions in these points.To do this it calls maps renderPosition function for the visiblePoints and the function in the map class calls individual objects own render functions. |
| *public* **assignMapObject**( Map* pointerMapObject):void | Sequential*Parameters:* in Map* pointerMapObject |

## 3.11 Script Module



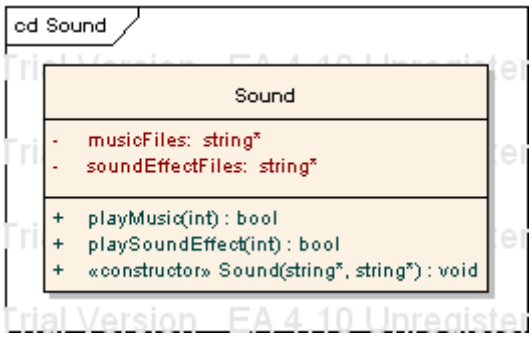
Script Attributes

| Attribute | Details |
|---|---|
| *private string* **heroScriptFile** | |
| *private string* **enemyScriptFile** | |
| *private string* **weaponScriptFile** | |
| *private string* **inventoryScriptFile** | |
| *private string* **levelPuzzleScriptFile** | |

Script Methods

| Operation | Notes |
|---|---|
| *public* **Script**( string scriptFiles):void | Sequential<br><<constructor>>*Parameters:*<br>in string scriptFiles<br><br>it takes the script file which contains names of all the script files.Also according to these file names, it assigns the names of script files one by one to the attributes of this class. |
| *public* **returnScriptFile**( int objectID):string* | Sequential*Parameters:*<br>in int objectID<br><br>the constructor of object calls this and according to the id of the object the corresponding script file name is returned. |

## 3.12 Sound Module



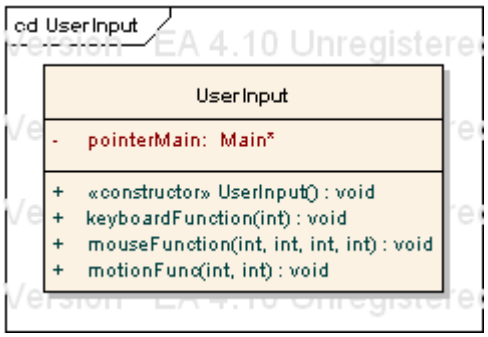

### Sound Attributes

| Attribute | Details |
|---|---|
| *private string*<br>**musicFiles** | contains the filenames of all the music files. |
| *private string**<br>**soundEffectFiles** | contains all the filenames of the soundeffects that will be used in the game. |

### Sound Methods

| Operation | Notes |
|---|---|
| *public*<br>**playMusic**(<br>int musicID):bool | Sequential*Parameters:*<br>in int musicID<br><br>plays the music given with the musicID. returns true if it can play the music. |
| *public*<br>**playSoundEffect**(<br>int soundEffectID):bool | Sequential*Parameters:*<br>in int soundEffectID<br><br>plays the sound effect with the given id. returns true if it can play the sound effect. |
| *public*<br>**Sound**(<br>string* levelSoundEffects,<br>string* levelMusicFiles):void | Sequential<br><<constructor>>*Parameters:*<br>in string* levelSoundEffects<br>in string* levelMusicFiles |

## 3.13 UserInput Module



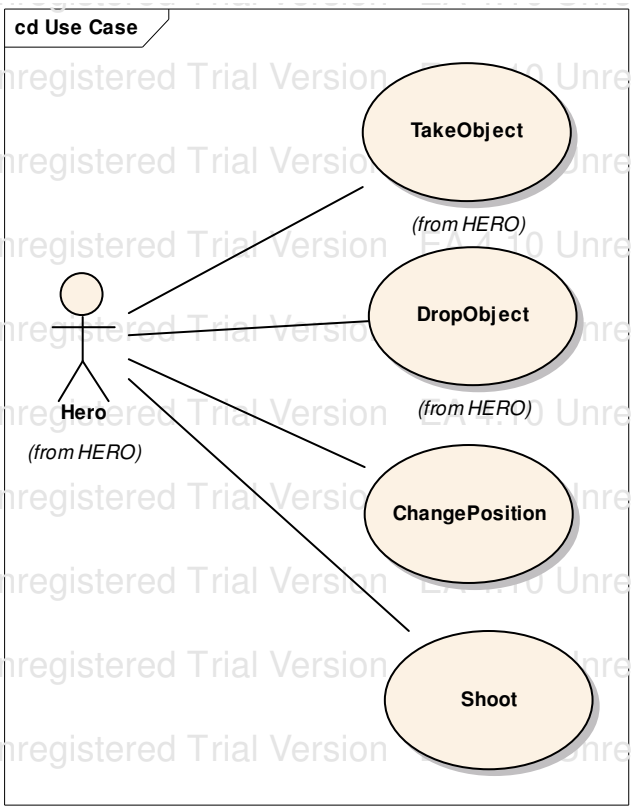

UserInput Attributes

| Attribute | Details |
|---|---|
| *private Main** **pointerMain** | |

UserInput Methods

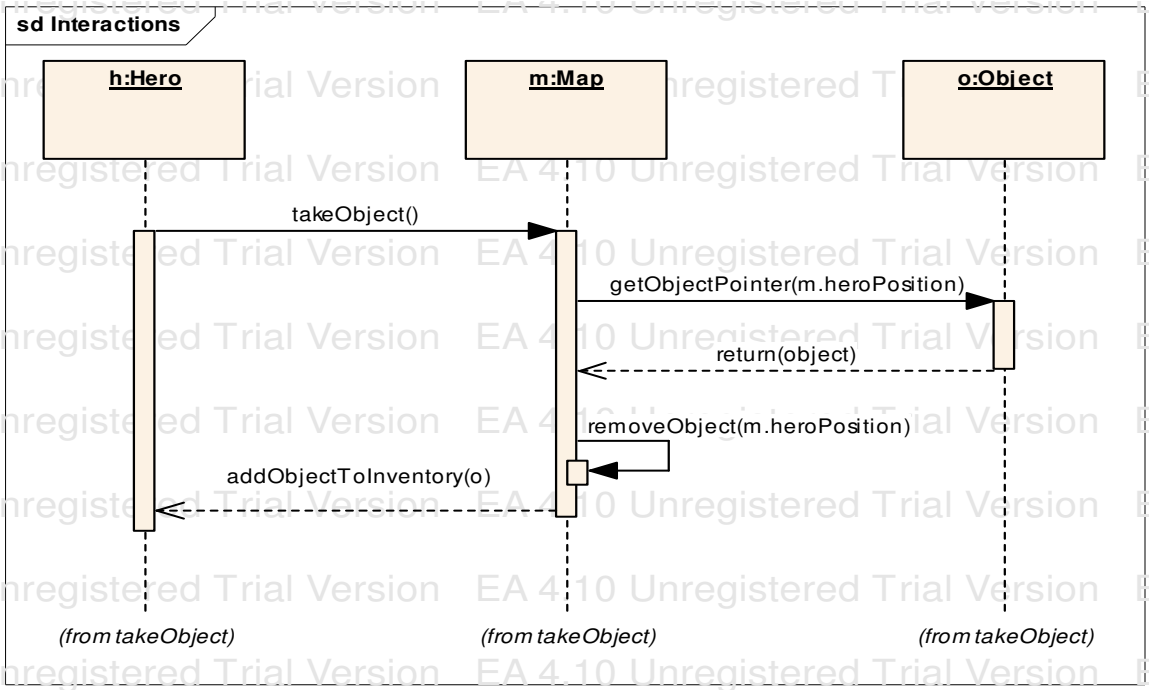
| Operation | Notes |
|---|---|
| *public* **UserInput**():void | Sequential<br><<constructor>><br>address of Main will be passed to the instance of the input. |
| *public* **keyboardFunction**( int button):void | Sequential*Parameters:*<br>in int button |
| *public* **mouseFunction**( int button, int mouseY, int mouseX, int state):void | Sequential*Parameters:*<br>in int button<br>in int mouseY<br>in int mouseX<br>in int state |
| *public* **motionFunc**( int mouseY, int mouseX):void | Sequential*Parameters:*<br>in int mouseY<br>in int mouseX |

## 4. Use Case Diagrams



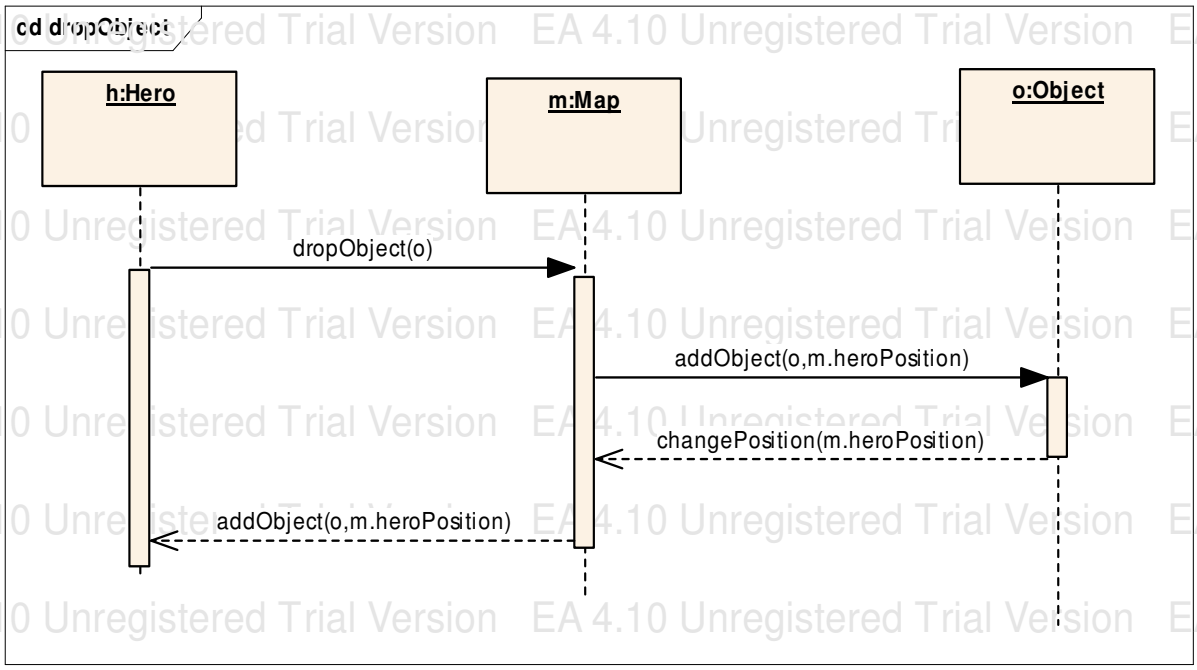

The use case diagram above explains the use cases of our “hero”. The “hero” can takeObject from a position which is close to the hero.It can Drop any Object that it carries.Also It can shoot by the input given by the user and change it’s position according to the user input. We didn’t show the use case diagrams of the user.Because they are explained in the following parts as Data Flow Diagrams(Start New Game,Load Game,Save Game,Play Game,Exit Game).

### 4.1 takeObject() Collaboration Diagram



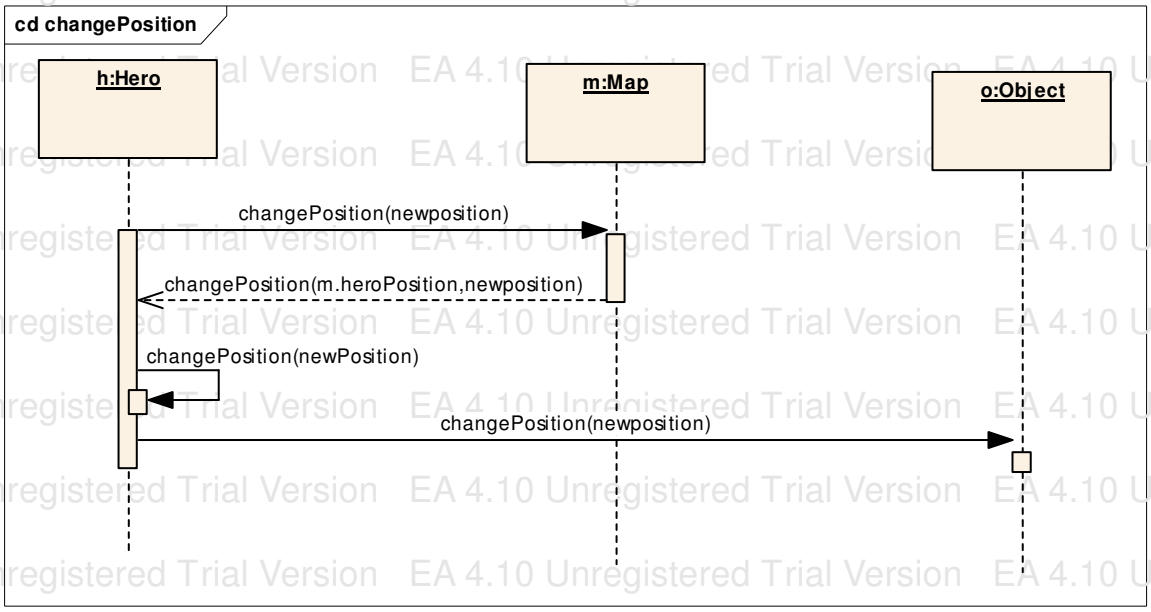

When hero takes an object it sends a message to map. Map finds the object at the position of hero then hero adds the object at the position of hero to the inventory.

### 4.2 dropObject() Collaboration Diagram



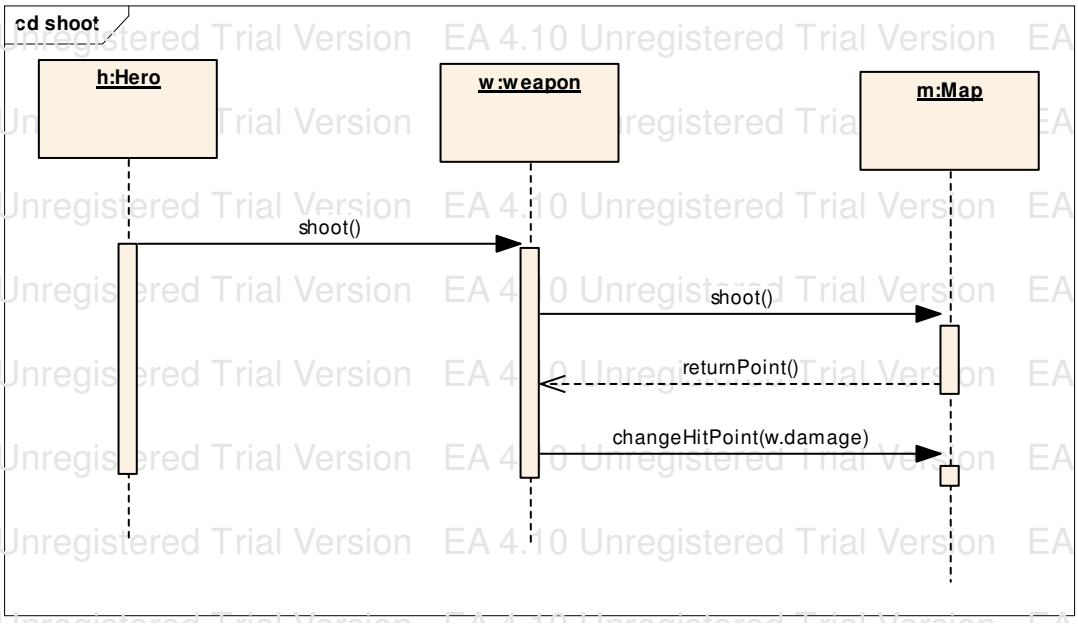
When hero drops an object it sends a message to map. Map determines if it is possible to add the object to the position of hero. If it is possible adds the object to map and returns true to hero.

### 4.3 changePosition() Collaboration Diagram




When hero tries to change his position firstly it sends a message to map to identify whether it is possible to change its position to new position. If it is possible firstly he changes his position to new position then he change all the objects positions in the inventory.

### 4.4 shoot() Collaboration Diagram



When a hero shoot it send a message to the current weapon of the hero. Then weapon sends to map the direction and position of the weapon to calculate the point hit. Then there is an enemy at the point the hitpoint of the enemy will decrease according to the weapon damage.

## 5. 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 .

## 6. Algorithms

### 6.1 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)

### 6.2 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.

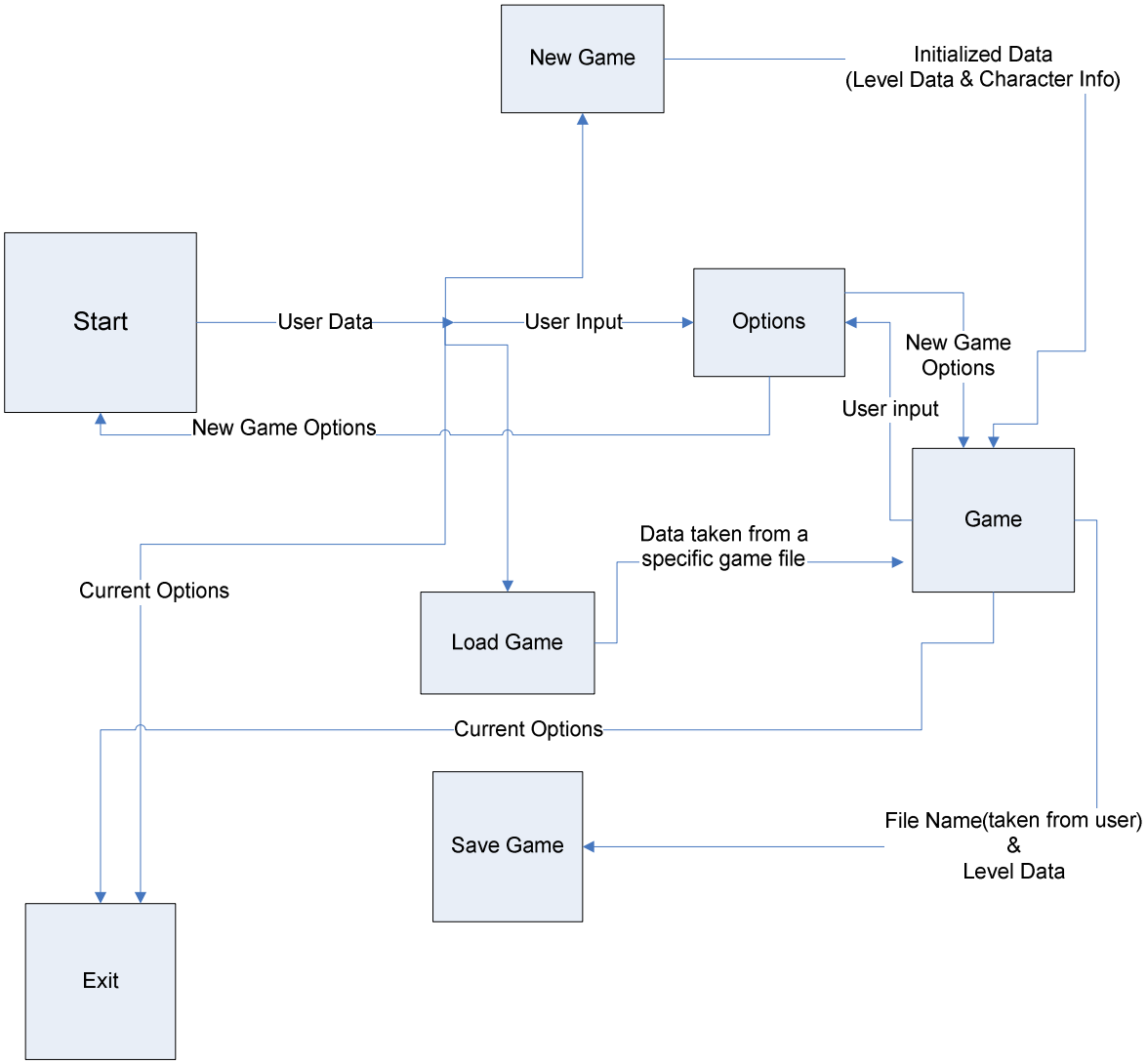
**D.**If you reach the target square or the open list is empty (no path) than stop.

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

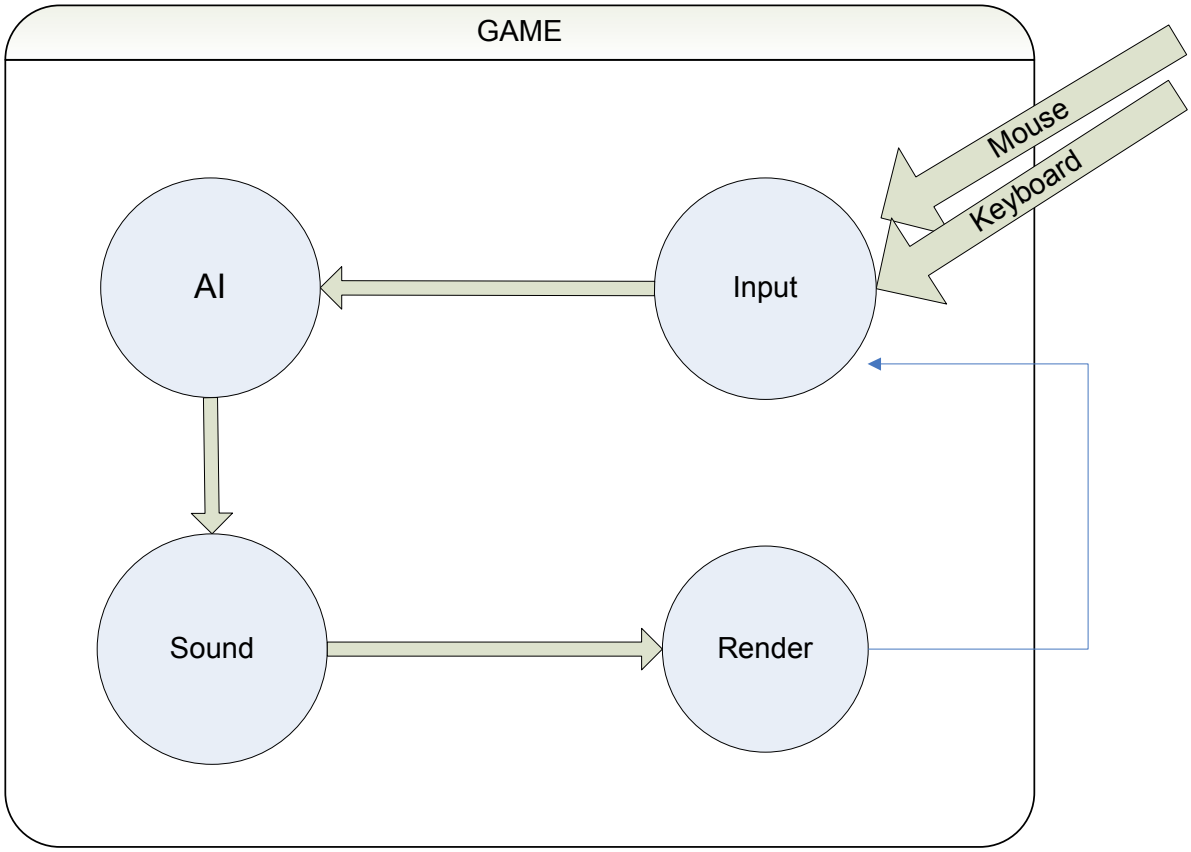
## 7. Data Flow

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.

## 7.1 DFD Level 0

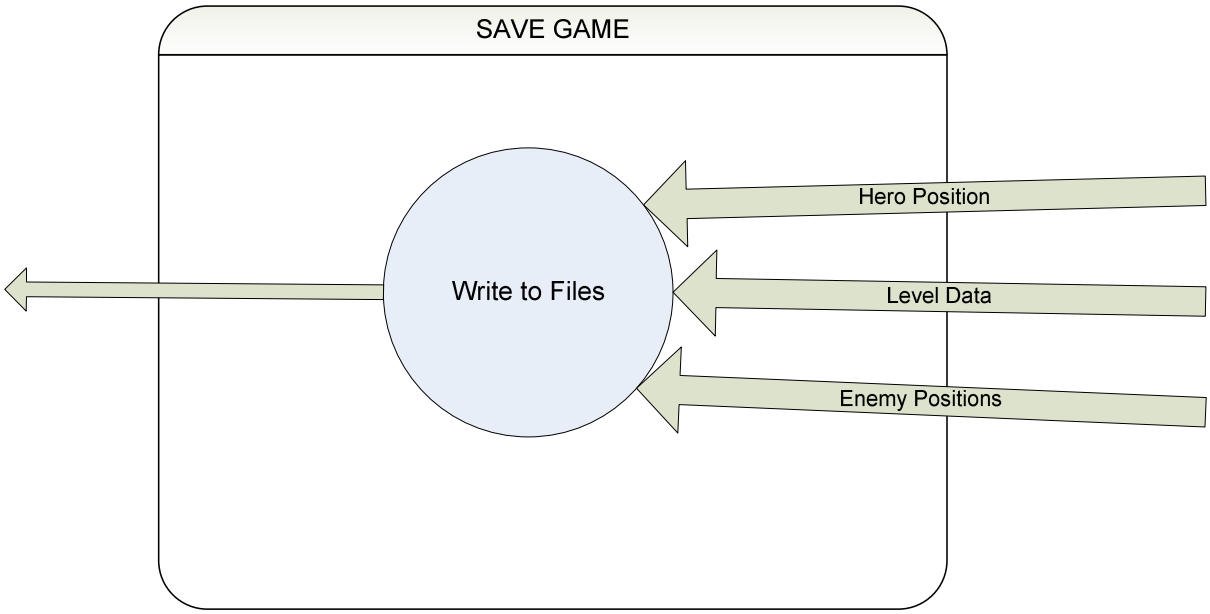


### 7.2 Game DFD Level 1



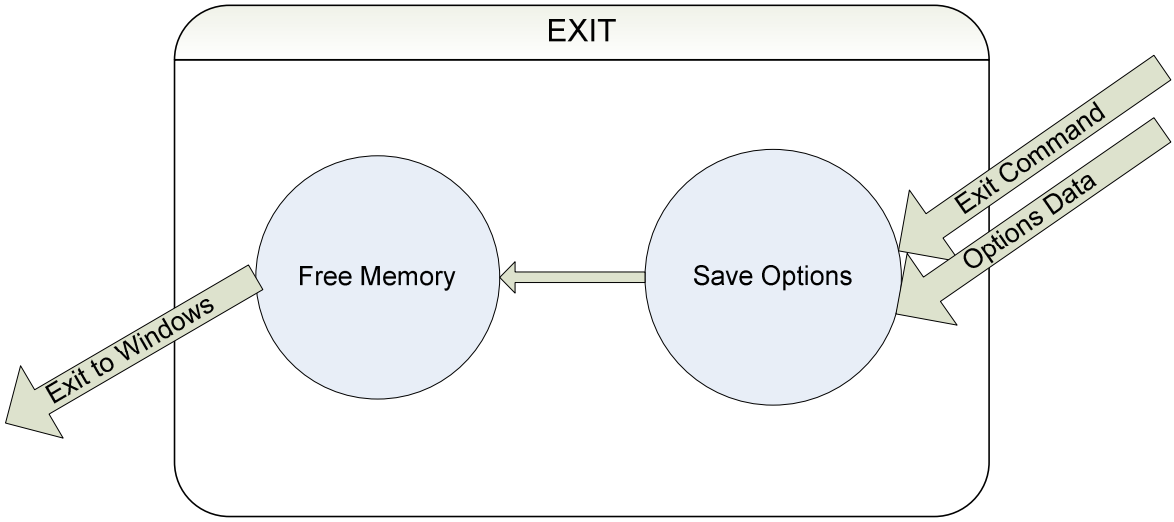

The GAME state is actually the main loop of the game. The user's mouse and keyboard inputs are fed into the INPUT state. In this state the character position and direction are affected by the mouse and keyboard inputs. There are two calculations in this state. First one checks whether the hero can actually 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.

### 7.3 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).

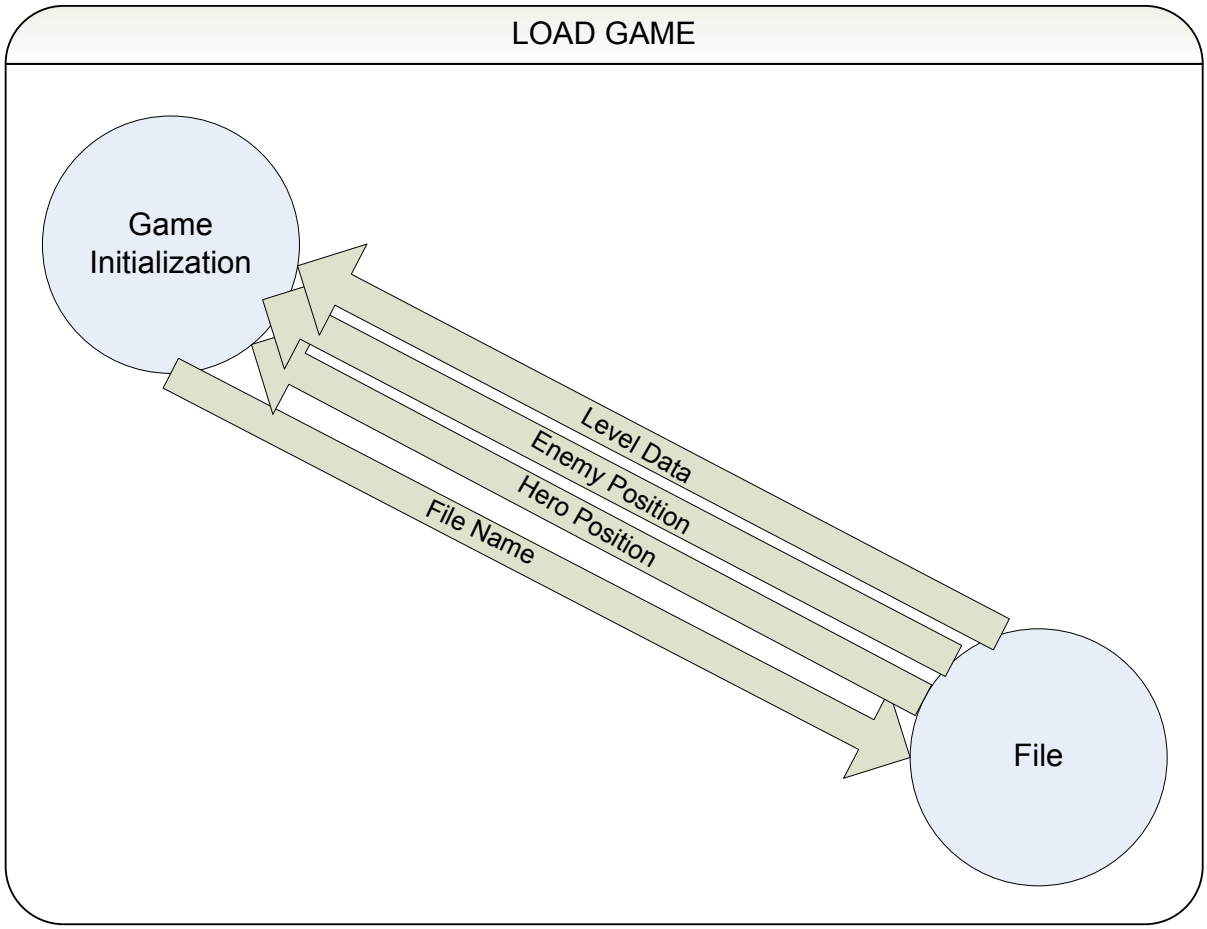
### 7.4 Exit State DFD Level 1




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

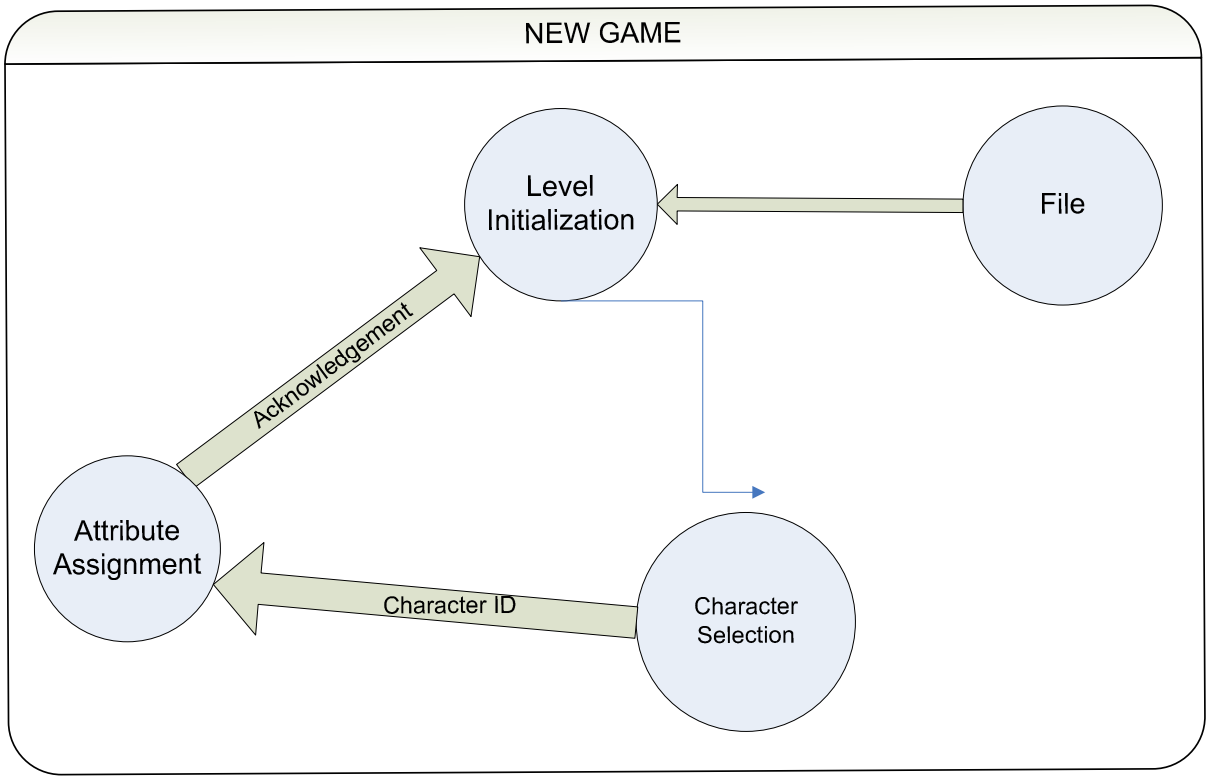
configuration. After saving options all the memory that has been allocated for level, map, characters, objects are deleted and the game exits to windows.

### 7.5 Load Game DFD Level 1



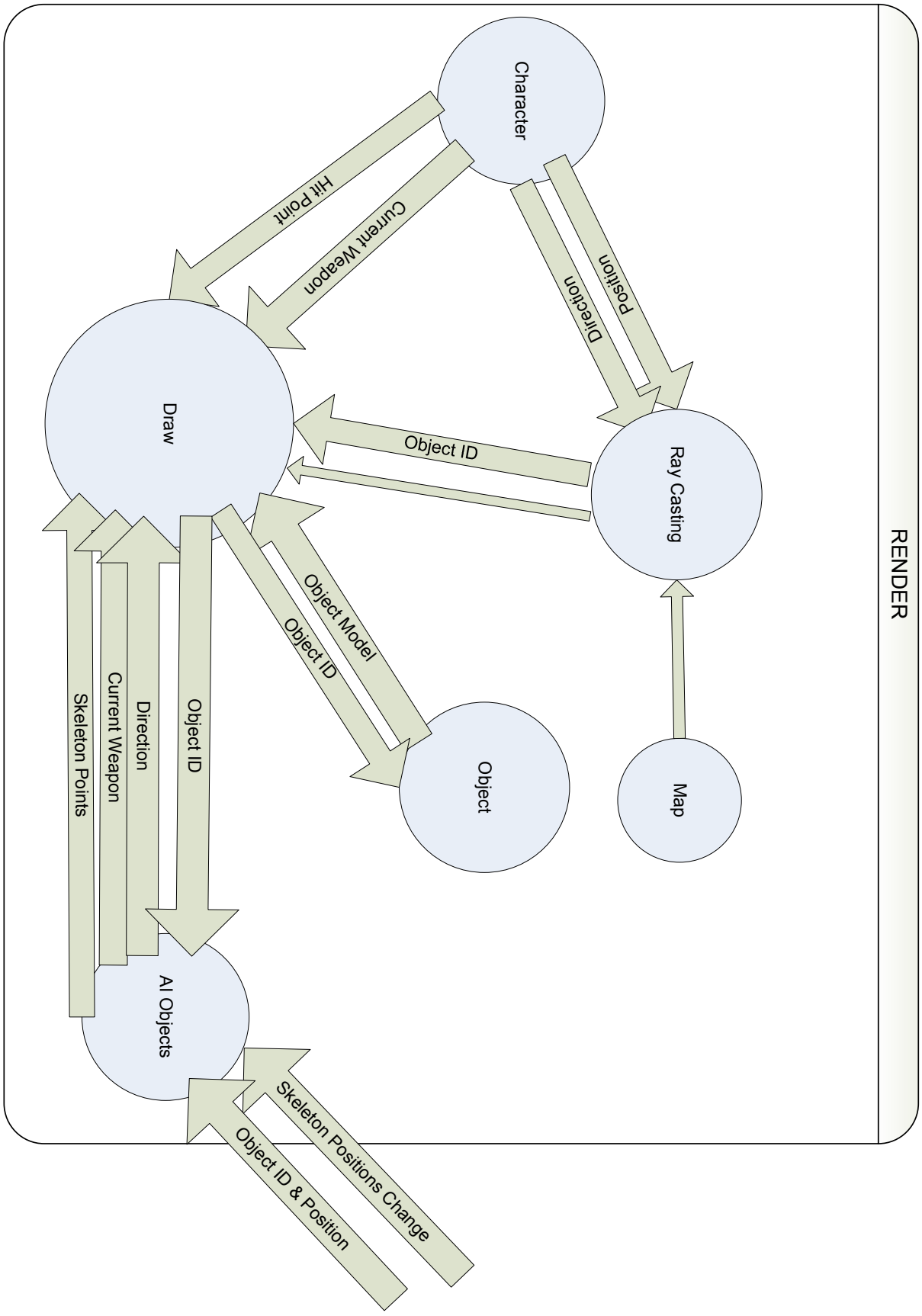

When the user is in START state, if he chooses to load a game he clicks on a saved game and then presses the load button. Therefore the filename is fed into the LOAD state. FILE state takes this filename input, and reads in the level file (see file formats for level file) which holds all the information 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.

### 7.6 New Game DFD Level 1



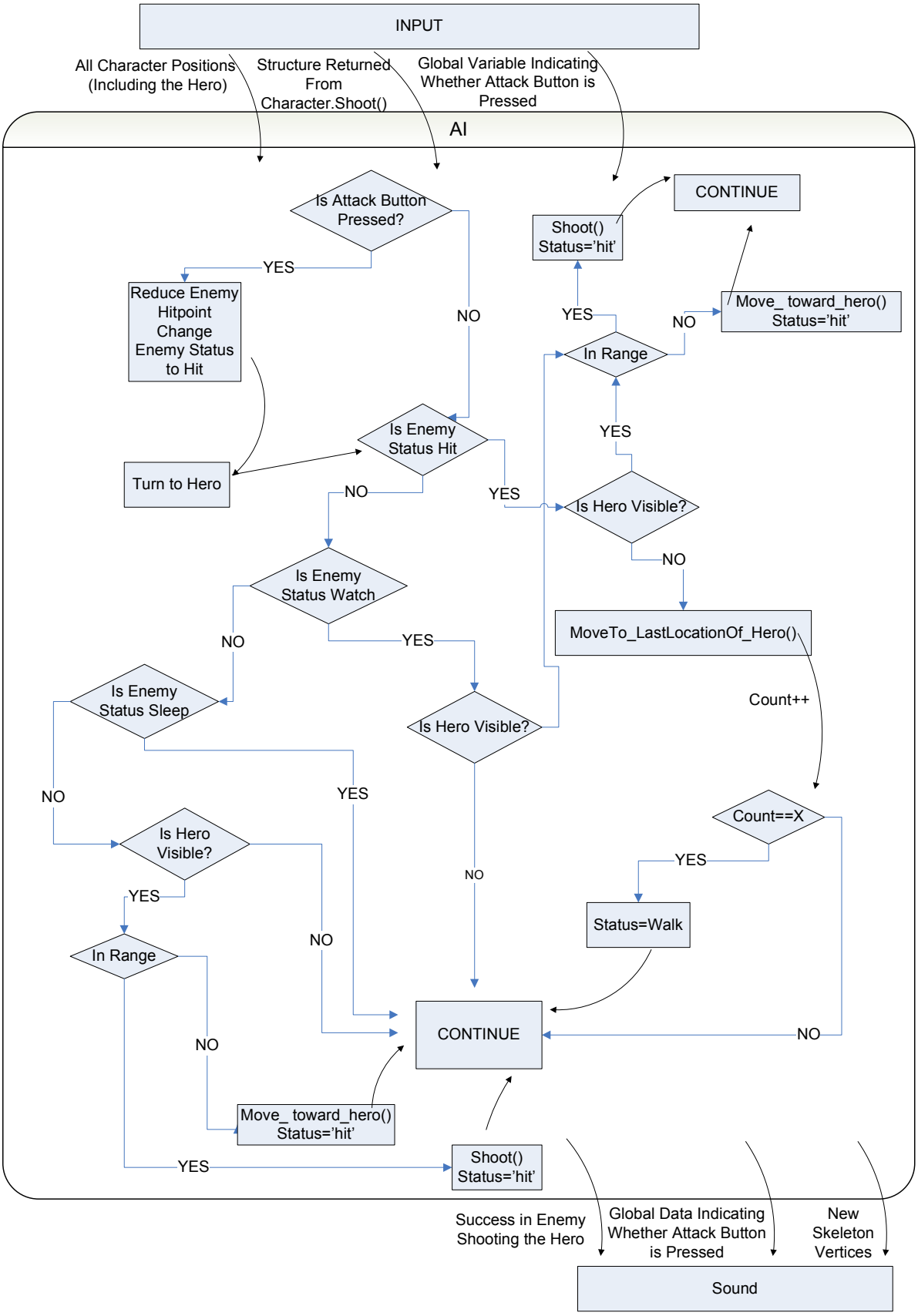

When the user is in START state, if he chooses to start a new game we are in LEVEL INITIALIZATION state. The user is prompted for character that he wants to play with in CHARACTER SELECTION state and then this state feeds the id of the character into the ATTRIBUTE ASSIGNMENT state which in turn creates 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.

### 7.7 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.

## 7.8 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.

# 8. Game Techniques

## 8.1 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.

## 9. 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

## 10. 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.

### 10.1 Character File Format

| | |
|---|---|
| CharacterID | (103) |
| 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) |

### 10.2 Weapon File Format

| | |
|---|---|
| ObjectID | (68) |
| Range | (20) |
| Weight | (3) |
| Texture | (rifle.bmp) |
| Model | (rifle.obj) |
| Damage | (15) |
| Magazine Capacity | (12) |
| AnimationFileName | (“shoot.anm”) |
| MaxWidth | (10) |
| MaxHeight | (30) |

**10.3 Map File Format**

| | |
|---|---|
| MapID | (map no\n) |
| Map size | (width*height*depth\n) |
| Map information | (objectID x_coordinate y_coordinate z_coordinate\n) |
| | (…….....\n) |

e.g. Assume that we have 3 objects on a map. The map id is 2 and its size is (20*30*10). The id's of the objects are 1, 7, 22, and they are on the positions [1][10][15], [2][16][23], [19][40][78]. Then our map file will be as follows.

2
20*30*10
1 1 10 15
7 2 16 23
22 19 40 78

**10.4 Object File Format**

| | |
|---|---|
| ObjectID | (18) |
| Weight | (10) |
| Texture | (box.bmp) |
| Model | (box.obj) |

**10.5 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**

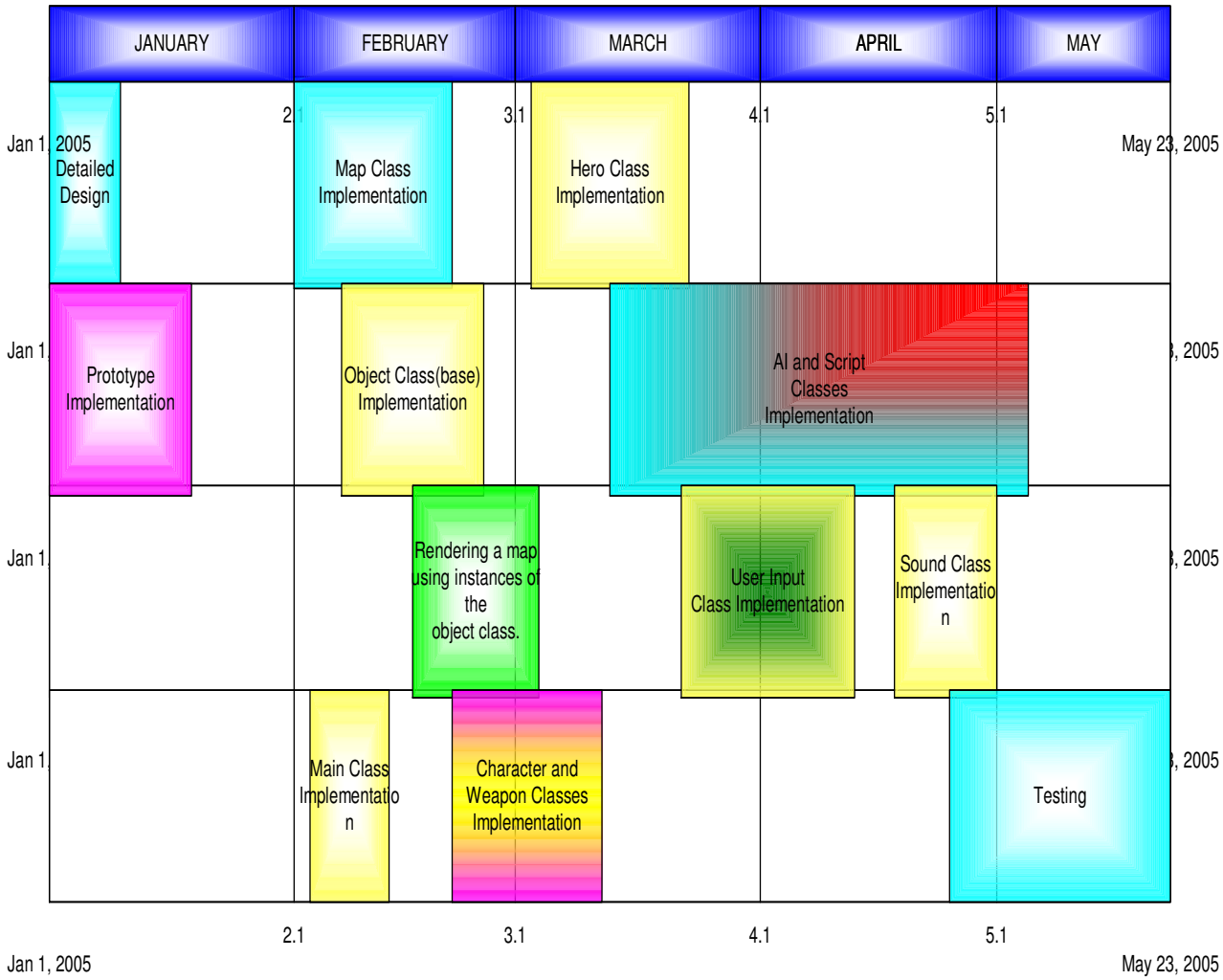
It represents a group. When a group is defined it remains the current group until another group is defined. If no group is specified in the file, everything in the file is in the current group.

**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.

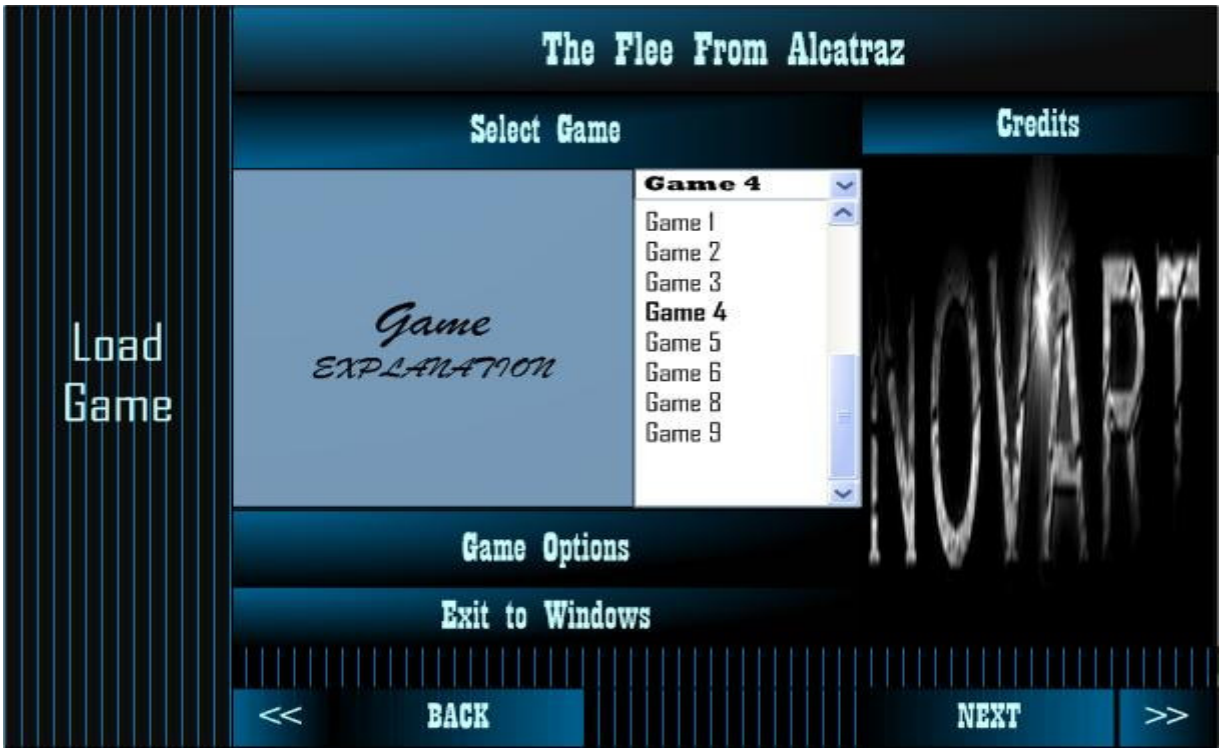
# 11 Time Chart



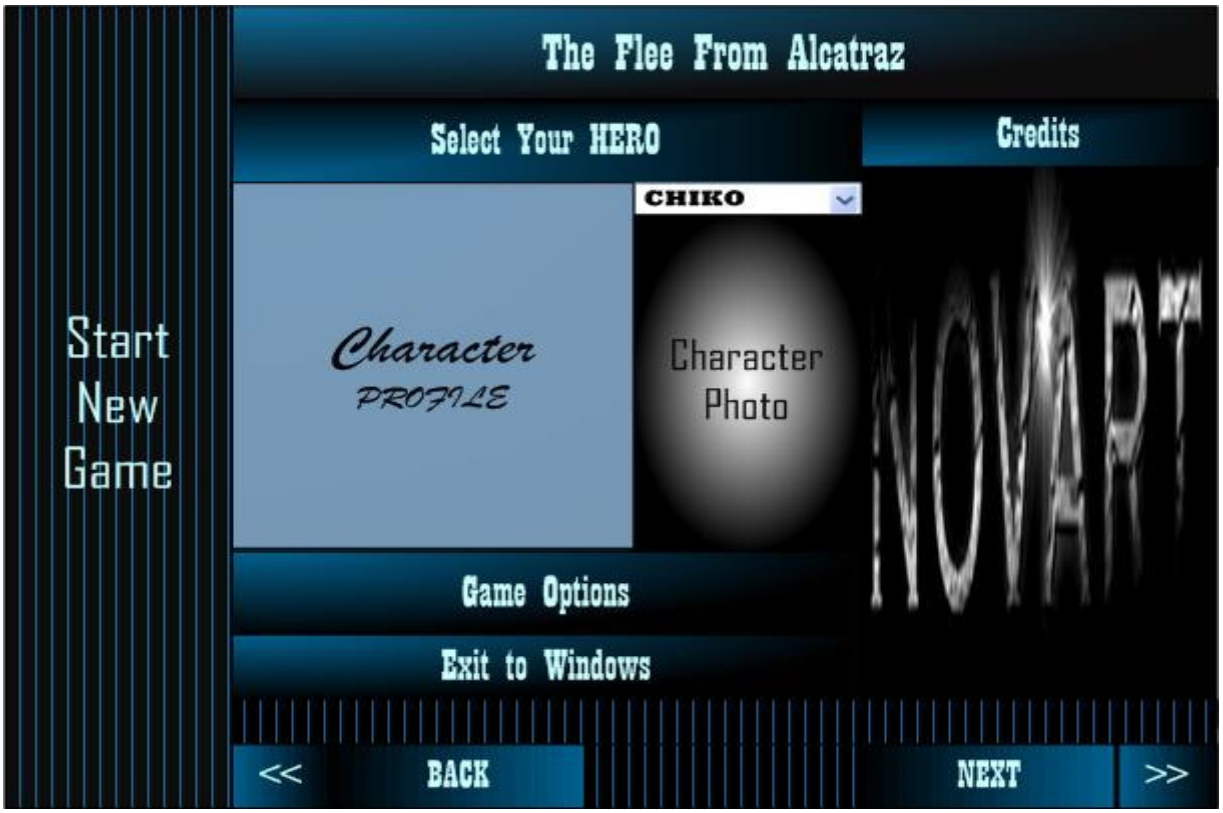
# 12 User Interfaces

## 12.1 Main Menu


## 12.2 Load Game



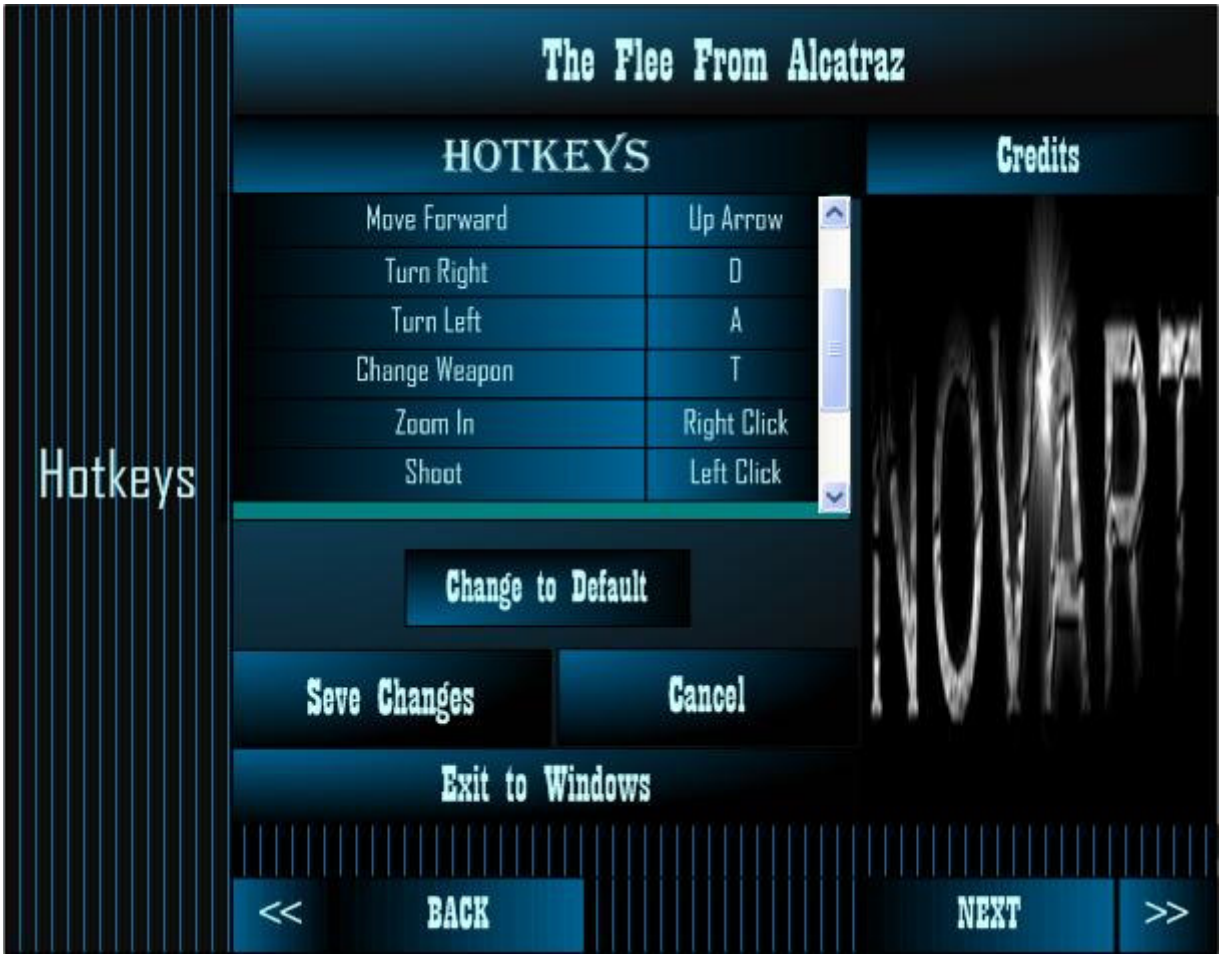
**12.3 Start New Game**




**12.4 Main Menu when game is paused**

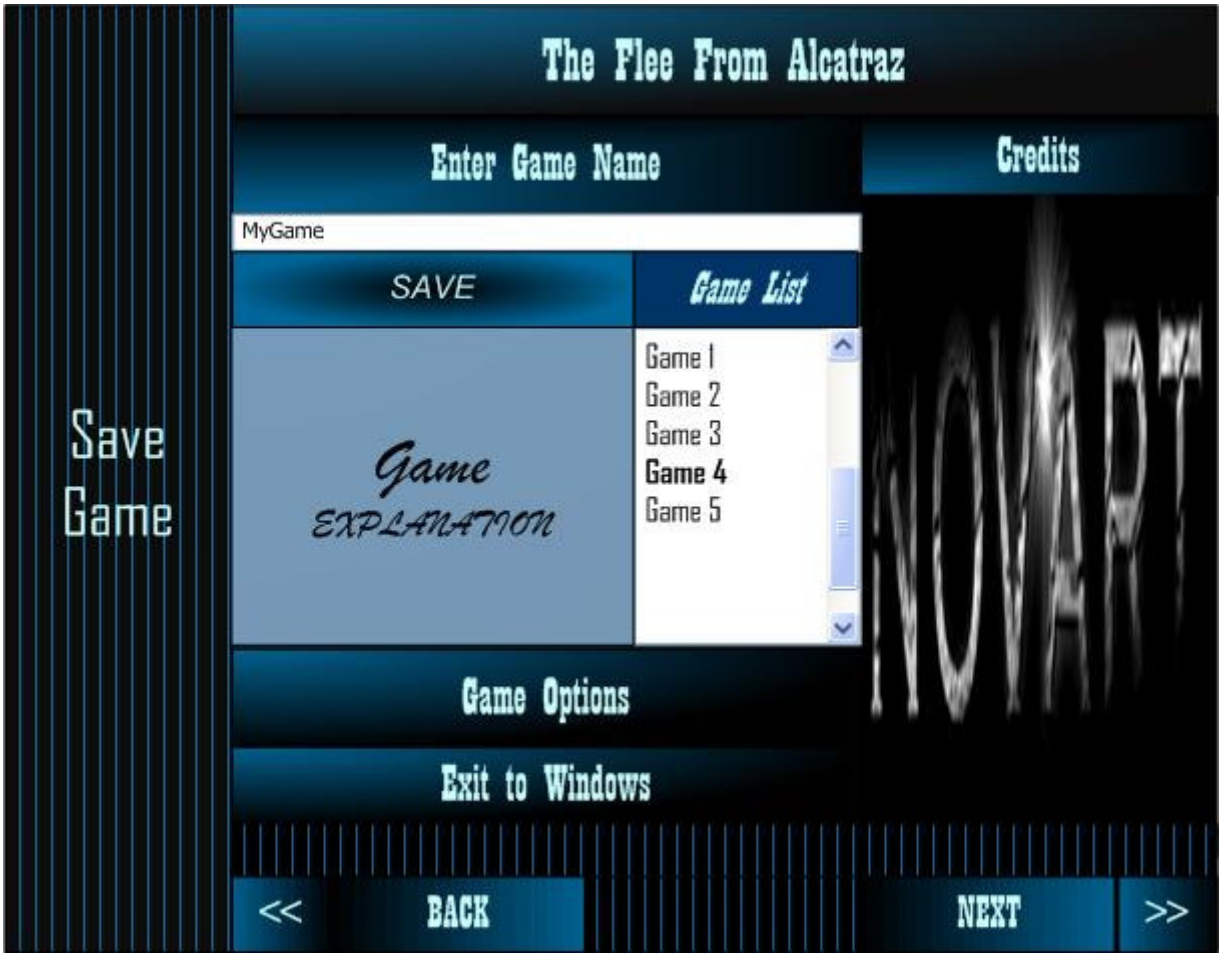

**12.5 Game Options**

**12.6 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

### 12.7 Save Game




## 13. 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 detailed design document helped us create a picture of what is going on in the game control and architecture.