Table of Contents

1. Introduction 2
2. Goals & Objectives 2
3. Architectural & Component Level Design 2
3.1 Graphical User Interface 3
3.2 Game Engine 5
3.3 Input Module 6
3.4 Loader Module 6
3.5 Artificial Intelligence Engine 7
3.7 Multimedia Module 10
3.8 Physics Module 12
4. Scene Management 12
5. Levels & Puzzles 15
6. CLASS DEFINITIONS 19
6.1 Character 19
6.2 Hero 20
6.3 Enemy 20
6.4 Citizen 21
6.5 Node 21
6.6 Object 22
6.7 Weapon 23
6.8. Sword 23
6.9. Pistol 23
6.10 Box 24
6.11. Key 24
6.12. Door 25
6.13 Magazine 25
6.16. Level 25
6.17. MultiMedia Class 26
6.18. PhysicsEngine Class 26
6.19. GameEngine Class 26
6.20. ScriptingEngine Class 27
6.21. Input Class 28
6.23. AIEngine Class 29
6.24. GUI 30
6.25. GraphicsEngine class 31
6.26. Puzzle class 31
7. TESTING ISSUES 31
7.1. Test Design 31
7.2. Test Cases 31
7.2.1. Unit Testing 31
7.2.2. Integration Testing 32
7.2.3 Validation Testing 32
7.2.4 Performance Testing 32
8. DIAGRAMS 33
9. CLASS DIAGRAM 35
10. USE CASE DIAGRAM 37
11. STATE TRANSITION DIAGRAM 41
12. ACTIVITY DIAGRAM 43
13. SEQUENCE DIAGRAM 44
14. COLLABORATION DIAGRAM 46
15. Gantt Chart 47
16. Appendix 49

1. Introduction

PC gaming has been changed a lot since the golden age of multimedia in the early 90's. Since those days of text based adventure games, lots of the elements of adventures moved to other genres. However there is always the desire for treasure hunt & accomplishing missions, we hope to satisfy the needs of the end users by presenting a 3D Adventure Game that is based on the story of the movie "Kill Bill". Bearing the 3D Graphics into heart, the team hopes to present "The Bride", a game under action/adventure genre and support the game by different puzzles for each level, thus increasing the number of amusing factors.

2. Goals & Objectives

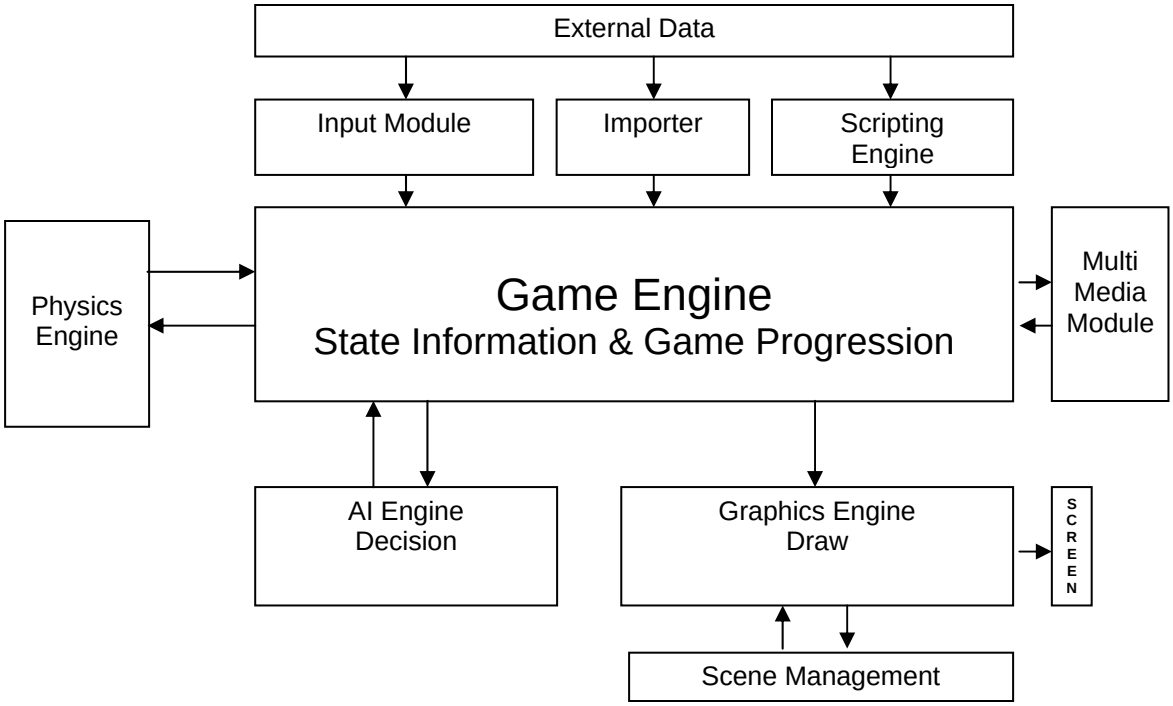
In this paper, our aim is to present the design made by the team in order to materialize the project. The Bride is going to be a game that works with a modern game engine that supports 3D Graphics Rendering, Multimedia (sound & video), Game AI and Collision Detection between game objects. In order to increase possibilities in the development phase, during prototyping, the team decided to use a high level scripting language which will be the basis for a scripting engine.

On the other hand, we agreed that speed is another important design goal. Thus in order to increase performance we use BSP trees for the culling system and mipmaps in order to render the environments faster, using textures with increasing level of detail.

Within this document we present the modules that are needed to implement the features described. We first present the use case diagrams accompanying graphical user interface designs to achieve user satisfaction. Then we present state diagrams for showing dynamic aspects of software, flow charts and class hierarchies to understand the problem better and to test the ideas in the creation phase.

3. Architectural & Component Level Design

Overall Architecture



Detailed Design of the Modules

3.1 Graphical User Interface

The Graphical User Interface is the entry point either to start a new game or to load an old one. GUI also enables the user exiting the program and configuring settings of the game environment via options menu.

The team is going to code GUI as a distinct module, which is at the top level that triggers the game engine and so starts or ends a game. The main window for the graphical user interface has five buttons that are identified as:

1 -) New Game: This button enables the user to start a new game with the beginning of the first level. When the user starts a new game he/she is able to watch a movie that describes, how the story begins. The user is able to skip this movie pressing the 'Esc' key.

The “New Game” button is the entry point to the game and GUI enters to the loading phase. In this phase a new instance of the game engine is created with the modified or default configuration data. The game engine first calls for importer functions and loads the model, level data and loads scripting engine modules and scripts to the corresponding data structures. When all the information required for the initialization of the game for the first level is imported to the data structures the game engine creates

instances for other modules and ready to realize user commands.

2 -) Load Game: This button enables the user to load a game that is saved before. When this button is clicked the user meets a new window, namely 'Load Menu', displaying the names of all the games that are saved previously.

Load Menu is a list containing the names of the games that are saved before. When the user chooses a name and clicks on the 'OK' button that is defined within the same menu, all the information that is saved for the game chosen is loaded as the current environment data and the user is able to continue to the game under the environment conditions saved. When a name is chosen GUI creates an instance of the game engine and the game engine starts the game using the level and model information stored within the saved file.

3 -) Exit Game: This button is defined as the exiting point from the program. When it is clicked the program will terminate. Exiting the game without saving will cause all the information hold to be released and thus free the memory for the game engine and the modules except GUI.

4 -) Options: This button enables the user to configure the settings of the game. When the user clicks on this button a new windows shall appear that displays the current sound, video control configuration including the difficulty level of the game.

The user is able to modify the settings identified within this window using the buttons defined for each option those of which will be used during the game engine initialization.

5 -) Credits: This button is designed to display Company & Game information. When it is clicked a new window displays the context defined for 'Credits Window'.

6 -) Save Game: During a game when the user attempts to quit the game a new menu that is identified as 'Save Menu' appears that asks for confirmation to save the current game data. When the user confirms this question by clicking on the ‘Yes’ button a menu appears that contains the names of the currently saved games and asks the user to specify a name for the game to be saved as.

If the user selects a name that is already on the list, its contents will be overwritten without asking for more confirmation. If the user specifies a new name all the data needed to begin the saved game from this entry point will be saved under the name specified. This data includes the model data (place, position), hero’s attributes (experience, strength, health) which are going to be saved as python script file, inventory data and current level data including puzzles.

5 -) Play Game: Play mode which is going to be implemented as the game engine shall display all the game information during the game. The game engine shall update the entire scene in the current window and the values in the fields that correspond to the health, strength, experience & attack damage. Game window also includes buttons that displays the information about inventory of the hero, the map of the current level and a label as 'Exit Game' that able the user to jump to the 'Save Menu'. During the game the 'Esc' like keystrokes will also be detected by the GUI and will be accepted as defined.

3.2 Game Engine

The team is going to implement the game engine as a module in its own, that handles the progression of the game independent of other modules, graphics, physics, sound and scripting engines. By this way, it will be easier to handle items in a modular way that makes our game engine extensible, maintainable and portable. The general flow of a program shall be through these states;

1 -) Pre-Game Setup: When the user interface triggers an entry point for a game, an instance of the game engine is created with the selected options from the user are passed directly to the game engine, through the constructor defined for the engine. So this step is to format some a piece of data that is passed to the game engine in construction.

2 -) Game Initialization: In this step, the level information will be parsed from the files that contain the game data with the help of loading modules which will be explained deeply in the levels and puzzles section. In order to load the game data to the corresponding data structures game engine calls the initialization methods defined for each module.

3 -) Game State Rendering/Display: This step is where the game engine begins rendering the state to the screen so the user can see the game in progress. In order to render the current state to the screen the game engine uses the graphics engine and scene management module, these are detailed in the scene management section.

Game engine calls an update method that calls update methods of all the objects where it is necessary to redraw the object on the screen.

4 -) Game Play/Inputs: This step is the process where the user plays the game by making moves or interacting with the game through the inputs from keyboard, mouse, command-line or scripts that are piped into the game from pre-saved input files. The game engine handles all the input data through an input module. Owing to this module,

input that is not an actual part of the game engine will be separated from the game engine module. Input module maps all input to actual method calls or formats into action objects that are passed to the game engine for processing.

5 -) Game Progression: This is the place where the user input/actions modify the current game state. The engine handles this item entirely, which means game progression is done via the methods defined for game engine that uses current state & level data plus interpretation of user inputs. In order to progress within the game the engine uses the state information conveyed by the story that are saved as scripts. When the game state points out an end point for the level, the game engine jumps to one level higher and loads the data needed for the corresponding level. If the state points out the end for the last level the engine exits to the game menu.

When an instance of the game engine is created the engine loads the data corresponding to the level and creates instances of other modules in order to progress in the game. In order make a real time application our system will work with a constant frame rate that is more than 25 fps. In order to support this rate we will manage the rendering and calculating the information of the data rendered effectively.

3.3 Input Module

In order to separate the input from the game engine totally, an input module is going to be implemented that will handle all the input data through the use of mouse, keyboard and console. The module uses buffers effectively, in order to handle input data in an efficient way.

We will integrate DirectX 8.0's DirectInput library to out application which can query the system for all available input devices, determine whether they are connected, and return information about them using the process called enumeration. We will create a single Microsoft DirectInput object and not release it until the game is terminated by the GUI due to a 'Quit' or 'Save' action is approved.

3.4 Loader Module

This module handles the interpretation of the animating model & environment data that stays within the specific files and parses these formatted data to the identified data structures. In order to parse the model data that is in md3 format we are going to use a parser and we will convert the data structure of the parser to our data structure. Our structure for an animated object includes the vertices, normals and the triangles made up

of these vertices as the faces and the texture coordinates for each face. On the other side, the structure for the animated model includes the animated objects, textures and colors. This information also include the data of all frames of the animation.

3.5 Artificial Intelligence Engine

The main duty of the AI engine is an inference mechanism which applies knowledge from the knowledge base to the current situation to decide on internal and external actions, that the game engine is going to be fed.

Current situation of the agents are represented by data structures representing the results of simulated sensors implemented in the interface and contextual information stored in the AI engine's internal memory. The inference mechanism selects and executes the knowledge relevant to the current situation. This knowledge specifies external actions, the agent’s moves in the game, and internal actions, changes to the AI engine's internal memory, for the inference mechanism to perform. So the engine which acts like an inference machine constantly cycles through a perceive, think, act loop, that is called the decision cycle.

In the bride, the hero struggles with other human characters. These enemies mimic the behavior of individuals in the real world. They try to survive in the sense of running away when wounded seriously. They also follow hero if he tries to escape when they are healthy enough. Basically there are predefined states which identify the behavior of these characters. The actions of the hero and changes in the environment caused by these actions are all plays role in the state transitions. That is the sound that a box hero opens may cause an enemy to get in fetch and then attack state if the hero is in the range.

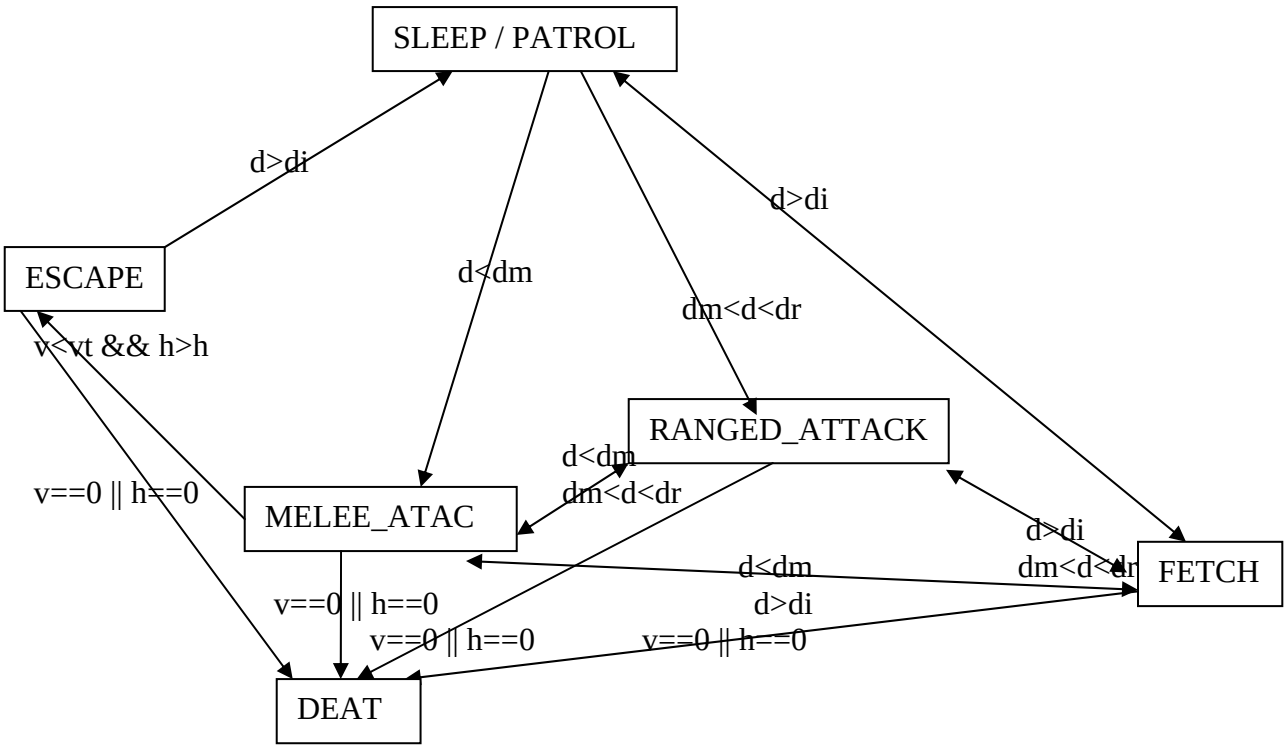
AI module is responsible from imitating these behaviors of agents. It contains necessary routines for an enemy to make decision given some inputs from the game world. Implementing such a decision mechanism for the bots in the game can be achieved by using various searching and planning methods. Finite state machines, decision trees, neural networks are the most common one of these methods. In our case the power of a simple finite machine is enough. Thus the behavior of the characters in the game will be modeled with finite state machines. A finite state machine basically consists of a start state, one or many final states and transitions between these states that occur with respect to given inputs. Finite-state machines are a good way of create a quick, simple, and sufficient AI models for the games.

The inputs of the finite state machine may be the actions of the hero and the effects of them in the environment as mentioned before. Final state corresponds to death of the character himself or death of the hero, whereas start state is the initial state of the character which may be patrolling or sleeping (standing with no action). In addition to these states there are intermediate states where enemy tries to fetch hero or escapes from hero or makes melee attack or make ranged attack to hero. Consequently there are a total of 6 states that enemies can be assigned where each state has its own behavior, and its own trigger.

The sates can be implemented as constants and functions taking certain parameters such as health of the hero, enemy itself, distance between hero and enemy, current state to produce the next state. The implementation can make use of some artificial intelligence techniques such as A* search with a predefined heuristic function and cost function to generate more complex decisions such as instead of escaping from hero fighting with him if the enemy has obstacles on his escape path or while escaping finding the rooms that have more enemies inside with respect to others. The code of these functions would be similar to the one below.

```
void check_enemy_state(enemy)
{
        switch(enemy.getState())
        {
        case FETCH:
        if(Hero.health > enemy.health &&
                        distance(Hero, enemy) < MIN_SECURE_DIST)
                enemy.setState(ESCAPE);
        case SLEEP:
                if(distance(Hero, enemy) < MIN_SECURE_DIST    &&
                                distance(Hero, enemy)<=RANGED_ATTACK_DIST)
                        enemy.setState(RANGED_ATTACK);
        ...
        }
        ...
}
```

Possible transitions between the states of the enemies in the project are basically demonstrated as follows:



At each transition the next state is decided according to the distance between hero (d) and enemy and, vitality ratio (v) of the enemy and the health (h) of the hero. The exact conditions that defines state transitions is shown on the figure where dr is the min ranged attack distance, dm is the max melee attack distance, di enemies sight range, vt is the threshold vitality ratio, ht is threshold health value for the hero. These conditions may be improved considering other elements in the environment (obstacles, sound made by an object or hero, etc...).

AI engine is in a continuous interaction with the game engine. Game engine sends necessary inputs to AI engine and AI engine sends back the decision it has made. This is how agents respond quickly when events occur in the environment.

3.6 Scripting Engine

In order to handle the code written for the project in a more effective way a scripting engine is going to be implemented that will allow the developers to write the functionalities through scripts thus increasing the modifiability of the code during development phase.

By the use of scripts, it will be easier to add, modify, test & debug code for new game play features and functionalities that are specified to be implemented in the design.

We will use python as the scripting language. We are going to have a library and also use the APIs of the python to embed it into C++. We will implement this by using ".py" script files written in python and using these as modules for any behavior. After loading all of the modules by using python APIs and "pyembed" library on the loading phase of the game, we are going to call the related modules and their methods on the specific conditions that need decisions and behaviors in the AI engine module.

3.7 Multimedia Module

A computer game constructs its own world and takes users at the center of it. Therefore the success of the game highly depends on the realism of the atmosphere and the flow of story. Sound effects are one of the major components that affect the realism of the atmosphere. Besides, playing music during the game increases the entertaining factor and therefore emphasizes attraction of the product. Handling these tasks requires the software being able to play audio files. Similarly, playing videos during the game not only utilizes realism but also helps switching between the parts of the story. Handling these tasks requires the software being able to play video files.

Multimedia module includes objects to handle playing audio and video files. Instances of this module are widely used during the program. The background music played during game, sound of the characters, environment objects and weapons, videos related with the story in between levels are all constructed by this module. Game engine triggers the multimedia module according to the state of the game user is playing.

Multimedia module provides routines to load audio or video files, to invoke audio (sound, music) or video, to stop it or pause it. It supports playing more than one music or video file simultaneously.

As an implementation issue it is wiser to use an already available library to control audio and video streams and utilize peripherals such as sound and graphics card.

There are many APIs providing methods for playing sound and video but the main question arises here is which of these APIs is the most appropriate for the project.

OpenAL (Open Audio Library) is just one of the APIs mentioned above. It is basically an audio library that contains functions for playing back 3D sounds and music in a game environment. It allows a programmer to load whatever sounds he likes and control certain characteristics such as position, velocity, direction and angles of the sound. All sounds are positioned relative to the defined listeners which represents the current place of the user in the game universe. This way as the user gets closer to a sound, he hears it louder. The main advantage of OpenAL is its being open source and designed to be cross platform API. Moreover its syntax and usage is closer to OpenGL providing an ease of use for the programmer like us who are familiar to OpenGL.

Another API that is heavily used in games is SDL (Simple DirectMedia Layer), a free cross-platform multi-media development API. SDL does provide methods for not only playing audio but also playing video files. In fact, it also handles events, CD-ROM audio, threads, timers, endian independence.

One may also consider using DevLib as a solution for implementing audio and video playback. It provides user friendly abstraction of heavily used resources such as fonts, images, 3D meshes, files, xml, zip-archives, sounds, videos. DevLib library itself makes use of DevIL, FreeType 2, LUA, ODE, libjpeg, libmpeg2, libpng, TinyXML, unzip, ZLib, SDL, DirectX 9, FMOD, GLEW and STL libraries to fulfill its requirements.

DirectSound and DirectShow APIs can also be used while implementing the multimedia module. The main disadvantages of these APIs are they are windows platform dependent and harder to learn with respect to other APIs.

The decision of choosing the API that suits our needs most depends on its capabilities, efficiency, ease of use, being free and platform independent, availability of learning material (tutorials, guides, example codes). All of these APIs support MPEG-2, MP3, OGG, WAV file formats. WAW format will be used as audio file format; MPEG-2 will be used as video file format. Previously in the initial design we have considered but during design we have decided not to use Devlib. For just playing audio and video DevLib would cause performance reduction and space redundancy. Instead of DevLib - the higher level library that uses SDL- using SDL itself would be much more convenient. Hence, we plan to make use of SDL in the first place to build the multimedia module.

However, during implementation phase, use of OpenAL and DirectShow can be taken into consideration as a B plan in case SDL stole from efficiency.

3.8 Physics Module

Another factor that affects realism in the game is obviously the behavior of the objects with respect to some identified action. That is in particular the fall of a box must be logical and should respect to the physics rules. This consideration of the behavior of objects requires many numerical computations.

The routines that define the behavior of the objects as they are mentioned above will be encapsulated in a module. These routines will be allowed to be reached from any other module. Fortunately, many of these common calculations for the behaviors of objects in the world are defined in a physics engine named ODE (Open Dynamics Engine) which lightens our workload.

The Open Dynamics Engine (ODE) is a free library that is mostly written in C++. It provides routines for simulating behavior of connected rigid bodies and helps determining the dynamics of motion such a system does. Ode provides efficiency and accuracy in platforms that virtual reality is essential. Its built-in collision detection capability and stable integration that controls the simulation errors makes it a convenient tool to be used in real-time simulations. Ode supports sphere, box, capped cylinder, plane, ray, triangular mesh collision detection primitives and quad tree, hash space, and simple collision spaces. One may model rigid bodies with arbitrary mass distribution and ball-and-socket, hinge, slider (prismatic), hinge-2, fixed, angular motor or universal joint types. Friction can also be modeled by using Ode. Another important feature of Ode is, it has a native C interface and also C++ interface built on top of the C one.

4. Scene Management

In order to implement visibility culling the team is going to use Binary Space Partitioning trees as the data structure. All the level data shall be handled by use of the BSP trees. BSP trees represent a recursive, hierarchical partitioning or subdivision of n dimensional space into convex subspaces.

A culling system renders only the parts of a game level that are not covered by walls or other objects. The BSP tree system is the fastest and most effective, especially for indoor levels. The BSP tree is created and initialized after the game engine creates an instance of the graphics engine, by the graphics engine for the current level using the

current level data. With a BSP tree culling system, the indoor rendering speed is independent of the level size and number of objects, which allows games to run with a decent frame rate even on old PCs.

Constructing BSPs

BSP tree construction is a process which takes a subspace and partitions it by selecting a splitting axis and splitting point for a node interior of that subspace, then dividing the objects into portions that intersect the left and right cells, and after that recursively generating trees for the left and right portions of objects.

For performance issues it is desirable to have a balanced tree, where each leaf contains roughly the same number of polygons. However, there is some cost in achieving this. To materialize this fact the splitting plane is selected by minimizing cost function representing the cost for intersecting a ray with the current cell. The cost function accounts for the surface areas of the new cells as well as the number of objects that they enclose. The construction uses a static depth bound that is determined from the number of objects in the scene to bound the memory usage for the tree.

Intersecting a ray with a BSP tree involves sequentially stepping through the nodes along the path of the ray. Traversing a node of the BSP tree involves choosing which of the two children should be traversed first. The algorithm maintains the entry and exit points for every cell that is traversed, and classifies these against the splitting plane to determine the order in which the 2 cells should be traversed. Our implementation uses an iterative traversal algorithm with a state stack storing untraversed nodes due to performance issues.

Drawing BSPs

In order to draw the contents of the tree, perform a back to front tree traversal. Begin at the root node and classify the eye point with respect to its partition plane. Draw the subtree at the far child from the eye, then draw the polygons in this node, then draw the near subtree. Repeat this procedure recursively for each subtree.

Dynamic Objects and Collision Detection using AABBs

In order to draw a dynamic object which is separated from each static object by a plane, it will be represented as a single point regardless of its complexity. This can dramatically reduce the computation per frame because only one node per dynamic object is inserted into the BSP tree. During tree traversal, each point is expanded into the original object. Inserting a point into the BSP tree is very cheap, because there is only one

front/back test at each node. Points are never split, which explains the requirement of separation by a plane. The dynamic object will always be drawn completely in front of the static objects behind it.

A dynamic object inserted into the tree as a point can become a child of either a static or dynamic node. If the parent is a static node, perform a front/back test and insert the new node appropriately. If it is a dynamic node, a different front/back test is necessary, because a point doesn't partition three dimensional space. The correct front/back test is to simply compare distances to the eye. Once computed, this distance can be cached at the node until the frame is drawn.

We will use AABBs (axis aligned bounding boxes) to bound dynamic objects and to embed them in a method that very quickly checks for collision between such a box and a BSP processed complex level.

The collision detection is accomplished by using the following basic intersection checks: ray/polygon intersection check; ray/AABB intersection check; edge/edge intersection check. The main collision detection function will be called with a local AABB (minimum and maximum points relative to its origin), the current position (p1) - the position the object reached in the left- and the desired destination position (p2) - the position the object wants to move to in the current frame. The function will check if the supplied AABB can move from p1 to p2 and, if a collision is found, it will process it applying the collision response code and recourse to compute the path required by the movement.

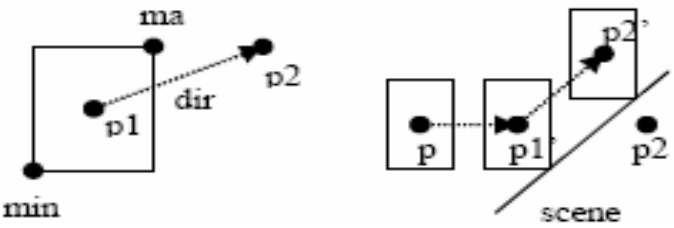


Figure : AABB defined by max and min points moving from p1 to p2 (left). Collision -detection/response recursion. AABB was moving from p1 to p2. Results on collision detection (right).

For a simple box/face intersection, only two loops will be required. The first collision moves the box to p1' and computes the new destination position p2' using the response code. Then it loops again doing collision detection for moving from p1' to p2'

(in the above figure right). As no collision is found between p1' and p2', it will stop the loop and return the p2' as the current position for the AABB.

In some cases more loops are needed as in the case of another collision being found between p1' and p2'. The collision detection method has to find if a box defined by its minimum and maximum points, moving from point p1 to point p2, will collide anything. To achieve this we will need to perform several computations, but fortunately we can cull several of them with simple dot product tests, and thus facilitating real-time performance.

5. Levels & Puzzles

Level Design & The Story

The bride will be a game with a total of five different game levels each having their own maps and environments. The game will also have three different difficulty levels in order to present the end user a changing atmosphere. While progressing between the levels we will emphasize the role of the scenario with the head assassin that is going to be killed to pass the current level. The player will meet with the head assassins according to the order they are killed within the movies of “Kill Bill”.

Now let us revisit the events happened before our game begins before explaining the flow of events in our game. Uma (The Bride), who is a former assassin, betrayed by her boss, is going to kill Bill, her former employer. She is going to take revenge from the assassin circle, for shooting her at her wedding - along with everyone else in attendance - and leaving her for dead. Four years after surviving a bullet in the head, Uma emerges from a coma where our game begins. Uma opens her eyes in hospital in the very beginning of the first level of the game and tries to get her way out of the hospital. Afterwards she passes through streets, houses and reaches to the first head assassin, namely Darly, by the help of the clues she gathers during the game. She has to solve a puzzle to get to Darly and then she enters to second level if she “finishes her”.

In the second level she begins seeking track of Lucy to take her revenge and continues her way in the subway. As expected new challenges like well trained assassin members and promising puzzles wait for her. She is able to enter level three after killing Lucy. The third level is full of puzzles to be dealt with and the most important of them is obviously finding the way to Vivica. Uma should complete many subtasks to find her way to Vivica.

In the fourth level, Uma needs to find the house where master Hanzoi lives and take a sword from him to slay the head assassin Micheal. However doing so is not an easy task and the bridge needs to overcome several obstacles. After obtaining the sword Uma goes to the assassin head quarter and there kills Micheal.

The final level consists of again struggling with assassin servants in the head quarters and then learning the place of Bill. Uma goes to the motel where Bill stays as soon as she learns where he is. There waits Uma a final battle with her former boss Bill.

As we have mentioned before during each level the hero would have to face with various puzzles. The user should solve these puzzles to progress in the game and get closer to the hero's final goal, killing Bill. Generally speaking a puzzle is a problem for which a method for the solution should be figured out and then necessary actions for overcoming it should be taken.

There will be certain places where the hero will be able find key objects or characters that she can speak and take information. All the information, items she collects during the game will be vital for solving puzzles that have been faced but not solved or that have not yet been faced. The information about puzzles is stored within the game level data. We will not cover all of the puzzles in the scope of this document but we will try to outline the basics of building puzzles.

The main objective of the game is helping to our hero namely the bridge to find and then kill Bill. However, on her way to Bill, Uma -that is again another way of calling our hero- will face with lots of obstacles whose aim is nothing but preventing her from reaching her goal. Uma should defeat the evil – that is the members of the assassin circle – by sometimes killing and sometimes deceiving them.

There will be a number of puzzles that is related with deceiving an assassin member or giving what he wants to progress in the game. Finding an item and give it to the guards may be necessary to pass a door protected by these guards. Choosing the correct path from other possible paths may also require solid background information gathered from other characters in the game. Furthermore, guessing the correct combination of actions might be necessary to proceed through the head assassins in the level.

Let us consider a specific instance of the cases mentioned above. The bridge learns from a character she has spoken that she needs to use the sword of master Hattori to defeat head assassin. Therefore Uma needs to find where Hattori Hanzo lives and then

take the sword. However the assistant of master Hattori does not allow the bridge to get near to his master without proving her brevity. Uma should drink the nonpoisonous potion from five existing potions to prove herself. Indeed all five potions is poisonous and therefore fatal, she should make a mixture from these five potions and make the nonpoisonous potion herself. Throughout the game Uma finds empty cab that she will use in this puzzle and collects information about the qualifications of the chemicals from other characters she speaks or books she finds.

Structures

As the example states, interpreting priori information and creative thinking plays a significant role in the solving puzzle process. Since levels consist of many puzzles, from the user's point of view, one of the major effort consuming parts of the game will be puzzles.

In order to handle the game state in an efficient way, we are going to keep a binary vector of all the dynamic objects within the level. A '0' in the vector defines that the corresponding facility for the object has not been realized and a '1' defines the opposite. Ex:

{
s11010111010010110
s00101110101110110
s01101101011010010
s10101101111111000
}, for a level with this data we will be able to determine the dynamic objects for the corresponding level and their initial situations. The length of this vector will be equal to the number of dynamic objects in the level. It can be a long list but we need only one bit for each interactive object in the level. The first line corresponds to the initial state and the last line to the final state.

In the initialization of the game the model & level data will be loaded using the external data for the specified level within the files. During this process the vector for the dynamic objects corresponding to this level, will be also initialized.

The puzzles will also be coded as the binary vectors that show the necessary changes that must be done to solve that puzzle, to the dynamic objects defined for the level. There will be more than one puzzles coded by this way in the level data file. In the following format (Brackets makes the documentation easier.):

{

p011001010101101010

p101010101101010101

p011011010010101010

p011001101111111111

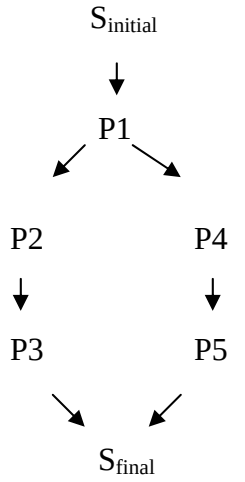
}

For a level file with this data the level will have 4 puzzles defined with the first line identifying the first puzzle.

All the state & puzzle information will be handled by the game engine. Comparison for, if a solution to a puzzle is reached will be done by taking the “Bitwise AND” of the two vectors namely the puzzle & the current state vector, which is updated at any time when an interaction between the hero and the environment occurs, and looking if the state vector has the corresponding bits for the puzzle as ‘1’. And if the puzzle is solved the level state will be updated to the next transition defined within the transition for that level.

We are also going to add a dependency graph for the solution sequence of the puzzles. This graph will enable the user to skip some of the puzzles and also transitions or reach to the final state by different ways.

Ex: Let the user has to solve the first puzzle than, he will either solve the second and third puzzles or the fourth, fifth and sixth puzzles in order to reach the final state to finish the current level. Then we will have a dependency graph as:



And this dependency within the puzzles will also be coded as a transition flow as:

{

S1P1P2P3Sf

S1P1P4P5Sf

}, in the level data file after the data for the transitions and puzzles are given. As it can be understood from the graph there can be also flows in the inner states depending on puzzles, but it will not be possible for the user to reach the end of the game without using one of the paths defined.

6. CLASS DEFINITIONS

6.1 Character

This class is the base class for handling different characters and their actions in the game.

6.1.1 Attributes:

- **int intelligence :** This attribute represents the intelligence level of the character.
- **int health:** This attribute represents the health value of the character.
- **int power:** This represents the attacking force of the character against the opponents.
- **double strength:** This attribute represents the strength coefficient for reducing the damage effect on the character.
- **double experience:** This attribute represents the experience coefficient for improving strength, health and power parameters.
- **int direction:** This is the angle of the character according to the starting position.
- **Object usedItem:** This is the object which is currently held by the character.
- **Color textureInfo[ ][ ] :** This is the color information of all pixels of the texture that will be mapped to the character.
- **CharacterMesh modelPosition:** This attribute keeps the position of the character relative to the origin.
- **Vertex origin:** This is the origin point for the mesh.

6.1.2 Methods:

These are the pure virtual methods for character types.

- **virtual void createAgent(void) :** This method interacts with AIEngine and determines the intelligence level of the character.
- **virtual void acceptDamage(Object &):** This method modifies the health of the character according to the strength of the character and the power of the object.
- **virtual CharacterMesh getMesh(void)**
- **virtual Vertex getOrigin(void)**
- **virtual void updatePlace(Vertex):** This method changes the origin value.

- **virtual void walk(unsigned char axis):** This method is for walking through the given axis.
- **virtual void talk(void):** This method is to make character talk.
- **virtual void render(void) :** This method renders the Character.

6.2 Hero

This class is inherited from Character class

6.2.1 Attributes:

- **Object *inventory:** This attribute holds the items belong to Hero.

6.2.2 Methods: These are the pure virtual methods for character types.

Inherited Methods: These are the methods inherited from the Character class.

- **void createAgent(void) :** This method interacts with AIEngine and determines the intelligence level of the hero.
- **void acceptDamage(Object &):** This method modifies the health of the Hero according to the strenght of the Hero and the power of the object.
- **CharacterMesh getMesh(void)**
- **Vertex getOrigin(void)**
- **void updatePlace(Vertex):** This method changes the origin value.
- **void walk(unsigned char axis):** This method is for walking through the given axis.
- **void talk(void):** This method is to make Hero talk.
- **void render(void) :** This method renders the Hero.

Specific Methods: These are the methods specific to Hero class.

- **void useItem(Object &):** This method is for activating an object.
- **void takeItem(Object &):** With this method, Hero takes the specified item to its inventory and this item is removed from the environment.
- **void dropItem(int):** With this method, Hero drops the item from the inventory.

6.3 Enemy

This class is inherited from Character class

6.3.1 Attributes: –

6.3.2 Methods:

Inherited Methods: These are the methods inherited from the Character class.

- **void createAgent(void) :** This method interacts with AIEngine and determines the intelligence level of the enemy.

- **void acceptDamage(Object &):** This method modifies the health of the Enemy according to the strenght of the Enemy and the power of the object.

- **CharacterMesh getMesh(void)**

- **Vertex getOrigin(void)**

- **void updatePlace(Vertex):** This method changes the origin value.

- **void walk(unsigned char axis):** This method is for walking through the given axis.

- **void talk(void):** This method is to make Enemy talk.

- **void render(void) :** This method renders the Enemy.

Specific Methods: These are the methods specific to Enemy class.

- **void useItem(Object &):** This method is for activating an object.

6.4 Citizen

This class is inherited from Character class

6.4.1 Attributes: –

6.4.2 Methods:

Inherited Methods: These are the methods inherited from the Character class.

- **void createAgent(void) :** This method interacts with AIEngine and determines the intelligence level of the citizen.

- **void acceptDamage(Object &):** This method modifies the health of the Citizen according to the strenght of the Citizen and the power of the object.

- **CharacterMesh getMesh(void)**

- **Vertex getOrigin(void)**

- **void updatePlace(Vertex):** This method changes the origin value.

- **void walk(unsigned char axis):** This method is for walking through the given axis.

- **void talk(void):** This method is to make Citizen talk.

- **void render(void) :** This method renders the Citizen.

Specific Methods: –

6.5 Node

This is the basic element in order to implement BSP class.

6.5.1 Attributes:

- **Object *items:** This attribute is the Object array to keep items in that Node.

- Character *people: This attribute is the character array to keep characters in that Node.

- Color textureInfo[ ][ ] : This is the color information of all pixels of the texture that will be mapped to the environment.

- Mesh modelPosition : This attribute keeps the position of the environment.

- Node *left: This is the left child of the class.

- Node *right: This is the right child of the class.

6.5.2 Methods:

- void addObject(Object)

- void AddCharacter(Character)

- void removeObject(void)

- void removeCharacter(void)

- void update(void)

- Mesh getMesh(void)

- void render(void) : This method renders the Node.

6.6 Object

This class is the base class for handling different objects and their actions in the game.

6.6.1 Attributes:

- Vertex origin : This is the origin point of the mesh.

- int direction : This is the angular value of the direction of the object.

- Color textureInfo[ ][ ] : This is the color information of all pixels of the texture that will be mapped to the object.

- ObjectMesh modelPosition : This attribute keeps the position of the object relative to the origin.

- bool activity: This is the activity flag to show if the object is active or inactive.

6.6.2 Methods:

- virtual ObjectMesh getMesh(void)

- virtual Vertex getOrigin(void)

- virtual void updatePlace(Vertex) : This method changes the origin value.

- virtual void useItem(void): This method is for activating the object.

- bool getActivity(void): This method returns the value of activity flag.

- virtual void render(void) : This method renders the Object.

6.7 Weapon

This class is the class for handling swords and pistols and inherited from object class.

6.7.1 Attributes :

- int power : This represents the attacking force of the weapon against the opponents.

6.7.2 Methods:

These are the pure virtual methods for swords and pistols**.**

Inherited Methods: These are the methods inherited from the Object class.

- virtual ObjectMesh getMesh(void)

- virtual Vertex getOrigin(void)

- virtual void updatePlace(Vertex) : This method changes the origin value.

- virtual void useItem(void): This method is for activating the weapon.

- virtual void render(void) : This method renders the Weapon.

6.8. Sword

This class is the class for swords and inherited from weapon class.

6.8.1 Attributes : –

6.8.2 Methods:

Inherited Methods: These are the methods inherited from the Weapon class.

- ObjectMesh getMesh(void)

- Vertex getOrigin(void)

- void updatePlace(Vertex) : This method changes the origin value.

- void useItem(void): This method is for hitting with sword by changing the position of the sword.

- void render(void) : This method renders the Sword.

Specific Methods: –

6.9. Pistol

This class is the class for pistols and inherited from weapon class.

6.9.1 Attributes:

- Magazine magazine: This attribute holds the magazine of the pistol.

6.9.2 Methods:

Inherited Methods: These are the methods inherited from the Weapon class.

- ObjectMesh getMesh(void)

- Vertex getOrigin(void)

- void updatePlace(Vertex) : This method changes the origin value.

- void useItem(void): This method is for firing the pistol.

- void render(void) : This method renders the Pistol.

Specific Methods:

- void addMagazine(Magazine): This method is for inserting magazine to the pistol.

- int getNumberOfBullets(void)

- bool isEmpty(void)

6.10 Box

This class is the class for general purpose objects such as boxes, tables, etc… and inherited from object class.

6.10.1 Attributes:

- int power : This represents the damage force of the box.

6.10.2 Methods:

Inherited Methods: These are the methods inherited from the Object class.

- ObjectMesh getMesh(void)

- Vertex getOrigin(void)

- void updatePlace(Vertex) : This method changes the origin value.

- void useItem(void): This method is for throwing the box by changing the position.

- void render(void) : This method renders the Box.

Specific Methods: –

6.11. Key

This class is the class for keys and inherited from object class.

6.11.1. Attributes:

- int keyId : This represents the key number.

6.11.2 Methods:

Inherited Methods: These are the methods inherited from the Object class.

- ObjectMesh getMesh(void)

- Vertex getOrigin(void)

- void updatePlace(Vertex) : This method changes the origin value.

- void useItem(void): This method is for inserting the key.

- void render(void) : This method renders the Key.

Specific Methods:

- int getKeyId(void)

6.12. Door

This class is the class for doors and inherited from object class.

6.12.1. Attributes:

- int doorId : This represents the door number.

6.12.2. Methods:

Inherited Methods: These are the methods inherited from the Object class.

- ObjectMesh getMesh(void)

- Vertex getOrigin(void)

- void updatePlace(Vertex) : This method changes the origin value.

- void useItem(Key): This method is for opening the door.

- void render(void) : This method renders the Door.

Specific Methods:

- int getDoorId(void)

6.13 Magazine

This is the class of magazines and inherited from object class.

6.13.1 Attributes:

- int numberOfBullets : This is the number of bullets in the magazine.

6.13.2 Methods:

Inherited Methods: These are the methods inherited from the Object class.

- CharacterMesh getMesh(void)

- Vertex getOrigin(void)

- void updatePlace(Vertex) : This method changes the origin value.

- void useItem(void): This method is for shooting the bullets.

- void render(void) : This method renders the Magazine.

Specific Methods:

- int getNumberOfBullets(void)

- bool isEmpley(void)

6.16. Level

6.16.1 Attributes:

- Node *levelTree: This is the Binary Space Partitioning (BSP) tree.

- Puzzle *puzzles: This is the array of puzzles.

- int difficulty: This is the difficulty value of the current Level.

6.16.2 Methods:

- Node *getLevelTree(void)

6.17. MultiMedia Class

This class is for Multimedia operations.

6.17.1 Attributes:

- static Audio * audios: The array that contains all the audio files played at a moment.

- static Video * videos: The array of videos that are played at a moment.

- static int na: The number of the audios stored in audios array.

- static int nv: The number of the videos stored in videos array.

6.17.2 Methods:

- static void addAudio(char*): Play the audio file with the given name (music, shooting, cry, creature, hit sound of an metalic object, broking sound of glass, ..).

- static void remAudio(int): Stops the identified sound or all of the sound according to the argument.

- static void playVideo(char*, int, int, int, int): Play the video file with the given name in the given frame.

- static void stopVideo(int): Stops the video with the given id.

6.18. PhysicsEngine Class

6.18.1 Attributes: –

6.18.2 Methods:

- Vertex* findPath(int): Calculates the points that are on the path of a given motion

- Vertex* detectCollision (Object, Object): Check collision of two objects

6.19. GameEngine Class

This is the base Class handling the game progress.

6.19.1. Attributes:

- Node *currentNode: This pointer holds the information for the current place of the hero.

- int *borders: Holds the border information that is going to be used in visibility culling.

- Level *currentLevel: This pointer holds the current Level information.

6.19.2 Methods:

GameEngine(// Options Specified Before Game): This is the constructor that creates an instance of the game engine using the data specified in options field.

- void loadInputData(void): This method loads all the needed input data through the DataLoading object.

- void initializeEngine(InputData &): This methods initializes the current state and environment variables using current level data.

- void updateScreen(Node &): This method calls the update functions of the objects that need to be updated for the current screen. In order to update, the graphics engine and when needed, the physics engine work together to calculate the information precisely that is going to be rendered.

- void getResponse(Script &): This method is needed in order to get the response that is generated by the AI engine, thus making it possible to determine the next action.

6.20. ScriptingEngine Class

Base Class handling scripting using the input information. We will use python as the scripting language and call its APIs from C++.

6.20.1 Attributes:

string [] moduleFiles : This is an array of the names of the script files which are written in Python.

PyObject *[] modules: This is an array of PyObject pointer type which is predefined in the Python APIs. This array holds each module which are defined in each script file.

int numberOfModules: This is the number of script files held in the moduleFiles array.

6.20.2 Methods:

- ScriptingEngine(string []) : This method is the constructor of the class. It takes a string array that holds the script file names and initialize the moduleFiles array with it. It also set the number of modules.

- void loadModules() : This method loads the modules defined in the script files held in the moduleFiles array into the modules array by using the Run_Function method of “pyembed” library that we will use to make communication of C++ with Python easier.

- void callFunction(string, string, string [], char * &) : This method calls the specified method given as the first argument in the specified module given as the second argument

with the specified arguments given as the third argument and sets the return value of the method to the (char *) variable given as the last argument.

6.21. Input Class

This is the main class handling all the input possible for a user to specify.

6.21.1 Attributes:

These are the input buffers in order to handle the input data effectively.

char *consoleInput: This attribute holds the inputs from the console.

FILE *scriptInput: This attribute holds the Script files.

6.21.2 Methods:

These methods get all the input entered from the keyboard, mouse shell & files and update the state variables.

- void keyboardHandler(void)

These methods get all the input entered from the keyboard, mouse shell & files and update the state variables. Hero's actions are controlled by the inputs from keyboard. These inputs ('W', 'A', 'S', 'D', ←, →, ↑, ↓) will invoke the method:
updatePlace(Point) of Character class.

Pressing W makes the Hero go forward.
Pressing A makes the Hero go left.
Pressing S makes the Hero go back.
Pressing D makes the Hero go right.
Pressing → makes the Hero turn right.
Pressing ← makes the Hero turn left.
Pressing ↑ makes the Hero look up.
Pressing ↓ makes the Hero look down.
Pressing ‘q’ invokes takeItem(Object &) method of the Hero class.
Pressing DEL invokes dropItem(Object &) method of the Hero class.
Pressing CTRL invokes useItem(Object &) method of the Hero class.

- void mouseHandler(void)

The orientation of the camera, which actually reflects the view of the hero, is controlled by the mouse. We are also going define the functionalities of the buttons as follows: right button will invoke useItem(Object &) method of the Hero class and pressing left button will invoke takeItem(Object &) method of the Hero class.

- void consoleHandler(void)

This method is to handle the commands that are entered through the keyboard, after a return accepted this method evaluates the information entered and will be able to change the game state or the state of the hero if there is a match occurs between this information

and the predefined sentences.

- void handle(int): This function decides which handle method to call according to it's parameter.

6.22. DataLoading Class

This class is for loading the data from the md3 formatted files to the specified data structures.

6.22.1 Attributes:

tMd3MeshInfo mesh : This is the data structure for the animated models that includes the object list(such as arms, legs…), textures and colors for each objects.

6.22.2 Methods:

These methods load the specific data identified by their names from the files to the data structures defined to get the model data from specific files with the md3 file format.

- void loadModelData(string, t3DModel): This method loads the model data from the md3 file into a data structure having the name t3DModel. Mainly, this is a ready-defined class to parse the md3 files.

- void convertDataStructures(t3DModel): This method is for converting the data from the t3DModel into the tMd3MeshInfo class. When this is implemented, the data structure will have the object list having the normals, textures, vertices and the triangles list. It keeps the animation property by keeping the vertices of each of the frames of the objects on each element of the object list.

6.23. AIEngine Class

This class handles the modification of the current game state, accorrding to the behaviours of the Agents.

6.23.1 Attributes

- int *gameState: Carries the game state information which will be analyzed by the inference mechanism.

- Character* currentAgent: To hold the information of the current character.

6.23.2 Methods

- void updateState(void): This method updates the gameState within the progress of the game.

- Agent getAgentInfo(void): This method is required to get the related information of

an agent.

- void createAgent(int type): This method defines the capability of an agent according to the type specified and current game status.

- Script findResponse(void): This method is to find the response of the current agent through the inference mechanism.

6.24. GUI

This module is the implementation of The GraphicalUser Interface.

6.24.1 Attributes

- int gameMenu: This is the context that is created to handle the top-level window which utilizes the accessibility to the game functionality.

- int saveMenu: This window utilizes the functionality that is required in order to save a game.

- int loadMenu: This window utilizes the load facility.

- int options: This window displays the options menu.

- int playMenu: This is the window that displays the game information during the game.

6.24.2 Methods

These methods create menus with the following identified names.

- int openGameMenu(void)

- int openSaveMenu(void)

- int openLoadMenu(void)

- int openOptionsMenu(void)

- int openPlayMenu(void)

- void openHelpMenu(void)

These methods are needed to generate the facility identified within their names.

- void startGame(void): This method creates an instance of a game engine with the level 1 game data.

- void quitGame(void): This method is called whenever the user interrupts to quit.

- void returnGame(void): This method is called whenever the user wants to continue to an interruted game

- void selectGame( ): This method selects the game from the save game menu list.

- void loadGame(): This method creates an instance of the game engine, with the saved game data identified from the load game list.

6.25. GraphicsEngine class

This module is for rendering the Objects, Characters and the Environment.

6.25.1 Attributes: –

6.25.2 Methods:

- void render(Node*)

6.26. Puzzle class

This module is for handling puzzles.

6.25.1 Attributes:

- bool * puzzleInfo: This attribute presents the puzzle information which contains state information.

6.25.2 Methods:

- bool compare(bool *): This method compares the given state with the state information of the puzzle.

7. TESTING ISSUES

7.1. Test Design

First of all, we have decided to test the system modularly. More clearly, we divided our system into different parts and we will test them. As we have mentioned in our Gantt Chart, in the first and second weeks of May we are planning to do unit testing. After uncovering errors of each class and making them function well we will pass to integration testing. In this step we will apply use-based testing. We will integrate collections of classes that respond to the same use-case in a bottom-up manner. In the last week of May we will do validation testing. Finally, after applying performance tests we will come up with the final version of our software.

7.2. Test Cases

7.2.1. Unit Testing

We will start testing with unit testing. We will test each class separately. For example, we will test the methods of an object from the Character class without loading it to the Environment.

Similarly, we will test the Level class without loading any Character or Object in it and just with a camera we will test the empty Nodes of the Level's BSP Tree.

7.2.2. Integration Testing

After unit testing, we will integrate the error-free classes in a bottom-up manner. Namely, we will integrate the atomic modules, test the integrated system and pass to higher level classes and so on. For example, after testing Hero, Citizen and Enemy classes we will integrate them to the Character class. Similarly, after the integration of the Character Class and Object Class, we will inherit these classes from the Node Class. As a developer, in the meanwhile, we will do white-box testing to be sure that the integrated system executes the necessary methods in an expected order.

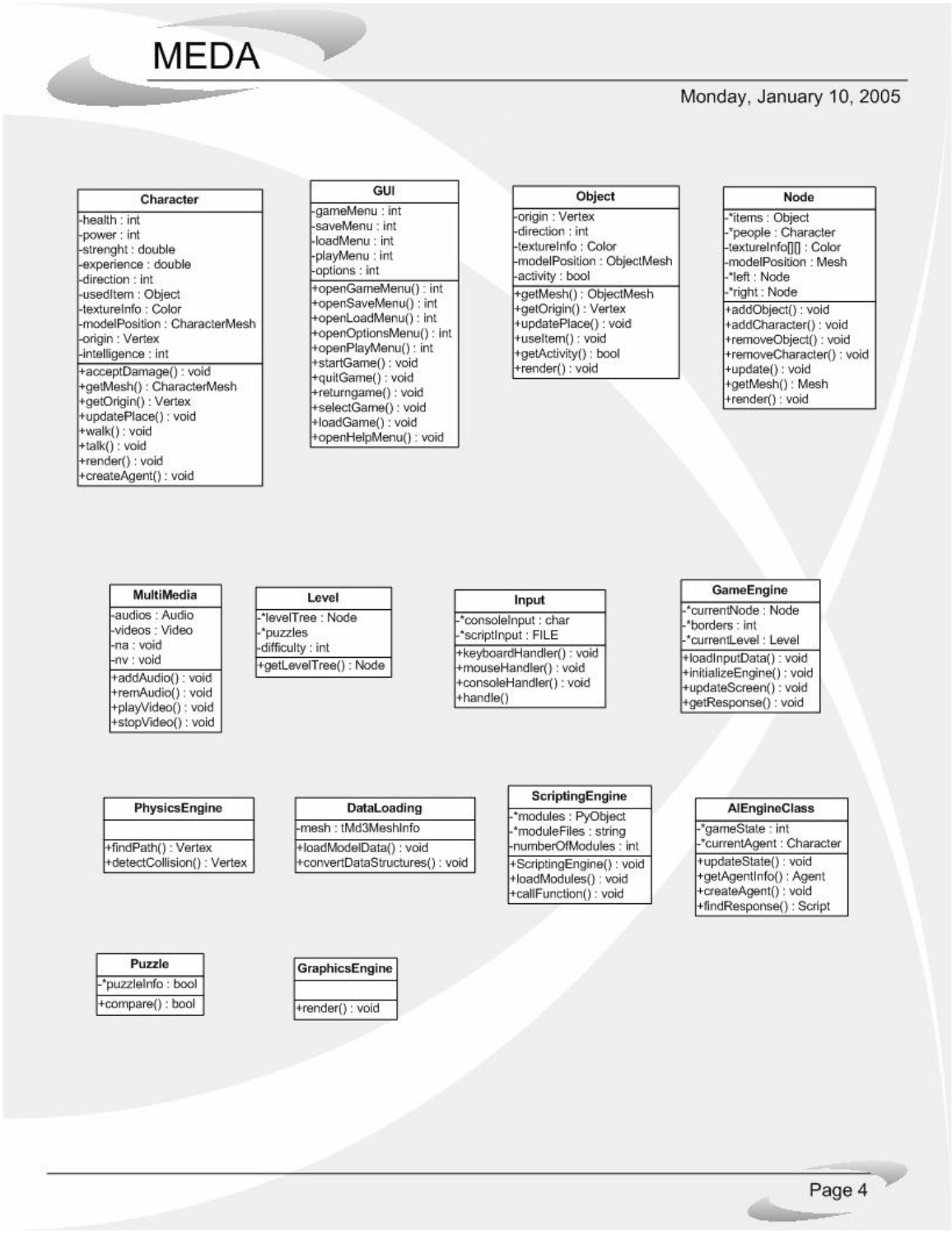
7.2.3 Validation Testing

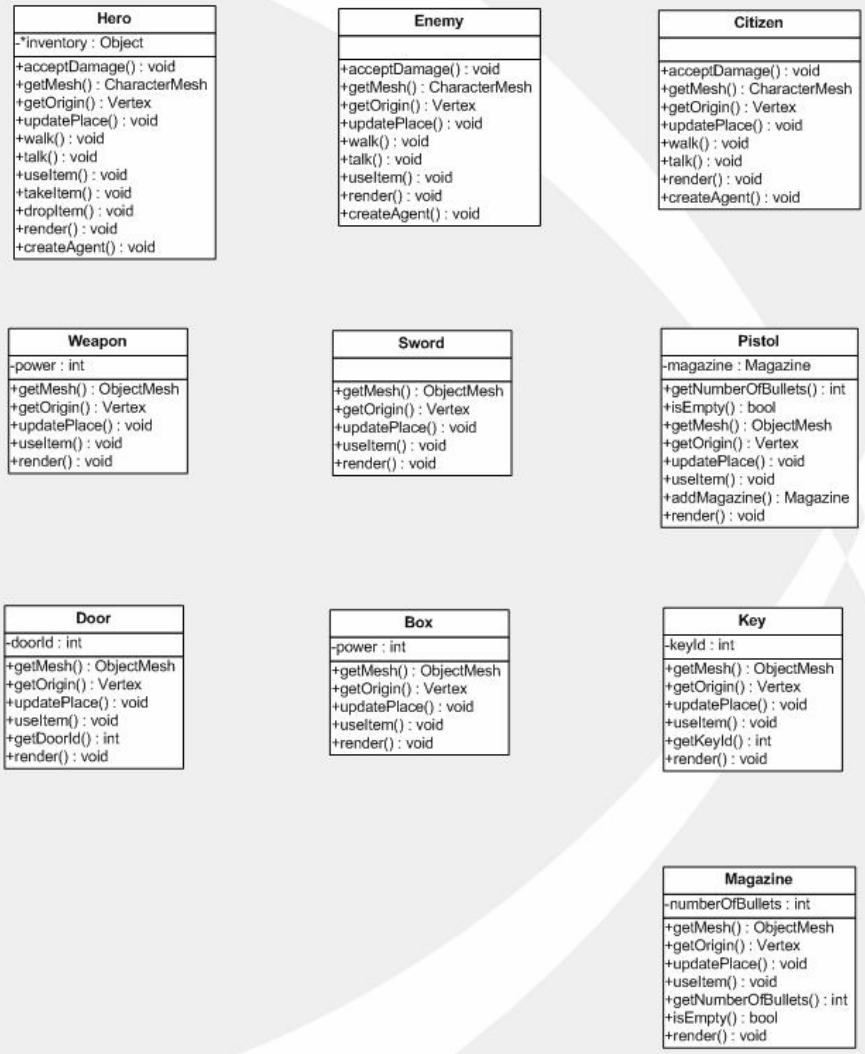
In the validation testing, we thought that beta testing is the best idea to see the bugs of our system with the help of independent users. We will distribute the first version of our software to our friends and make them play our game. In order to shorten the learning process, we will also give the manuals of the game. With the help of the feedbacks coming from the independent users, we will be able to detect and correct the bugs of the system.

7.2.4 Performance Testing

Finally, one of the most important criteria for a successful game is the performance. We will test the run-time performance of our software and if necessary we will make changes and optimizations to get a better performance.

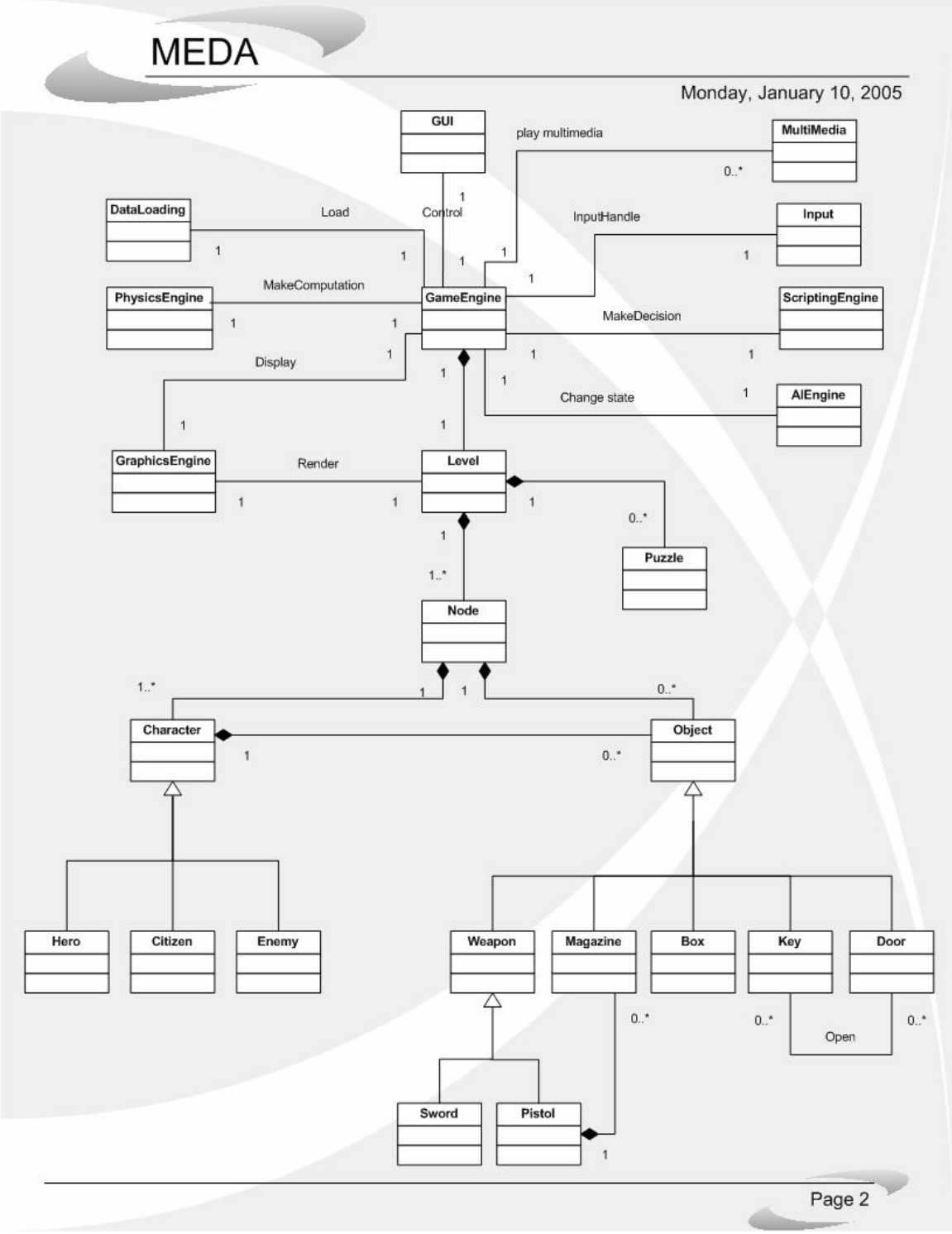
8. DIAGRAMS


MEDA
Monday, January 10, 2005
Character
-health : int
-power : int
-strenght : double
-experience : double
-direction : int
-usedItem : Object
-textureInfo : Color
-modelPosition : CharacterMesh
-origin : Vertex
-intelligence : int
+acceptDamage() : void
+getMesh() : CharacterMesh
+getOrigin() : Vertex
+updatePlace() : void
+walk() : void
+talk() : void
+render() : void
+createAgent() : void
GUI
-gameMenu : int
-saveMenu : int
-loadMenu : int
-playMenu : int
-options : int
+openGameMenu() : int
+openSaveMenu() : int
+openLoadMenu() : int
+openOptionsMenu() : int
+openPlayMenu() : int
+startGame() : void
+quitGame() : void
+returngame() : void
+selectGame() : void
+loadGame() : void
+openHelpMenu() : void
Object
-origin : Vertex
-direction : int
-textureInfo : Color
-modelPosition : ObjectMesh
-activity : bool
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+getActivity() : bool
+render() : void
Node
-*items : Object
-*people : Character
-textureInfo[][] : Color
-modelPosition : Mesh
-*left : Node
-*right : Node
+addObject() : void
+addCharacter() : void
+removeObject() : void
+removeCharacter() : void
+update() : void
+getMesh() : Mesh
+render() : void
MultiMedia
-audios : Audio
-videos : Video
-na : void
-nv : void
+addAudio() : void
+remAudio() : void
+playVideo() : void
+stopVideo() : void
Level
-*levelTree : Node
-*puzzles
-difficulty : int
+getLevelTree() : Node
Input
-*consoleInput : char
-*scriptInput : FILE
+keyboardHandler() : void
+mouseHandler() : void
+consoleHandler() : void
+handle()
GameEngine
-*currentNode : Node
-*borders : int
-*currentLevel : Level
+loadInputData() : void
+initializeEngine() : void
+updateScreen() : void
+getResponse() : void
PhysicsEngine
+findPath() : Vertex
+detectCollision() : Vertex
DataLoading
-mesh : tMd3MeshInfo
+loadModelData() : void
+convertDataStructures() : void
ScriptingEngine
-*modules : PyObject
-*moduleFiles : string
-numberOfModules : int
+ScriptingEngine() : void
+loadModules() : void
+callFunction() : void
AIEngineClass
-*gameState : int
-*currentAgent : Character
+updateState() : void
+getAgentInfo() : Agent
+createAgent() : void
+findResponse() : Script
Puzzle
-*puzzleInfo : bool
+compare() : bool
GraphicsEngine
+render() : void
Page 4


Hero
-*inventory : Object
+acceptDamage() : void
+getMesh() : CharacterMesh
+getOrigin() : Vertex
+updatePlace() : void
+walk() : void
+talk() : void
+useItem() : void
+takeItem() : void
+dropItem() : void
+render() : void
+createAgent() : void
Enemy
+acceptDamage() : void
+getMesh() : CharacterMesh
+getOrigin() : Vertex
+updatePlace() : void
+walk() : void
+talk() : void
+useItem() : void
+render() : void
+createAgent() : void
Citizen
+acceptDamage() : void
+getMesh() : CharacterMesh
+getOrigin() : Vertex
+updatePlace() : void
+walk() : void
+talk() : void
+render() : void
+createAgent() : void
Weapon
-power : int
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+render() : void
Sword
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+render() : void
Pistol
-magazine : Magazine
+getNumberOfBullets() : int
+isEmpty() : bool
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+addMagazine() : Magazine
+render() : void
Door
-doorId : int
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+getDoorId() : int
+render() : void
Box
-power : int
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+render() : void
Key
-keyId : int
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+getKeyId() : int
+render() : void
Magazine
-numberOfBullets : int
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+getNumberOfBullets() : int
+isEmpty() : bool
+render() : void

9. CLASS DIAGRAM

9.1. DIAGRAM



9.2 EXPLANATION

RELATIONS:

- **Control:** This relation is for controlling the GameEngine class according to the requests of the user by the help of GUI class.
- **MakeDecision:** This relation is for parsing the scripts used in the GameEngine class with the ScriptingEngine class to change the flow of the game.
- **MakeComputation:** This relation is for finding new positions for objects or charecters by the help of PhysicsEngine class when a physical effect is applied to them while the GameEngine class is working.
- **Display:** This relation is for displaying the game data by giving the level information in the GamaEngine class to the GraphicsEngine class.
- **Render:** This relation is for rendering the current node elements of the Level class by using GraphicsEngine class.
- **Open:** This relation is for implementing opening action on the Door class by using the information from the Key class.

AGGREGATIONS:

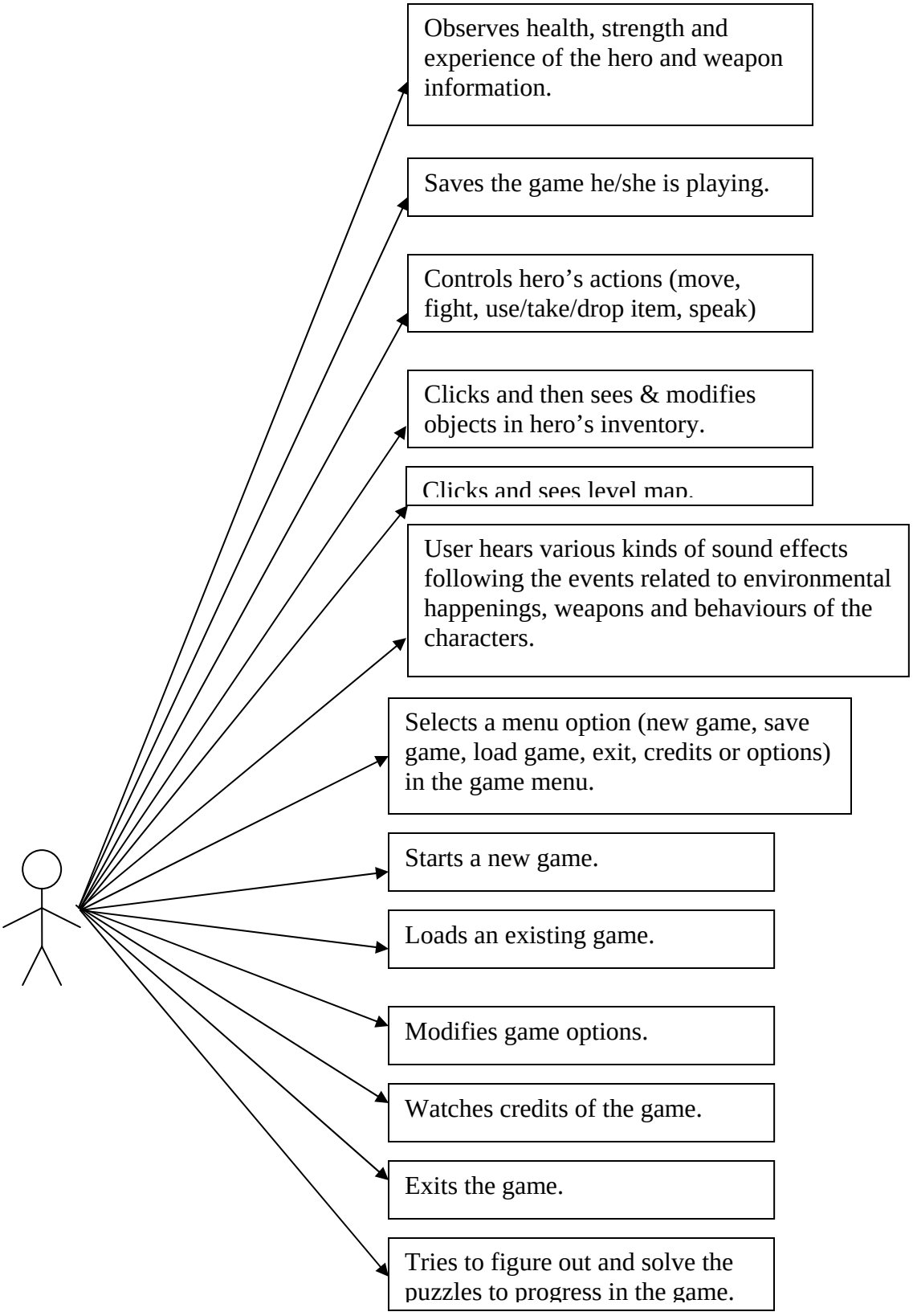
- **Game Engine – Level:** GameEngine class includes a Level object.
- **Level – Puzzle:** Each Level includes zero or more Puzzles.
- **Level – Node:** Each Level includes a pointer to the Node object.
- **Node – Character:** Each Node includes one or more Characters.
- **Node – Object:** Each Node includes zero or more Objects.
- **Character – Object:** A Character may use zero or more Objects.
- **Closet – Object:** A Character may insert some Object into the Closet.
- **Pistol – Magazine:** The Hero can insert a Magazine to her Pistol.

INHERITANCES:

- **Character → {Hero, Citizen, Enemy}:** Character class is the base class for Hero, Citizen and Enemy classes.
- **Object → {Weapon, Box, Key, Magazine, Door } :** Object class is the base class for Weapon, Box, Key, Magazine, Book, Door and Closet classes.
- **Weapon → {Pistol, Sword} :** Weapon is the base class for Pistol and Sword classes.

10. USE CASE DIAGRAM

10.1 DIAGRAM



10.2 EXPLANATION OF USE CASE DIAGRAM

Flow of Events for the Save Game Hero Use-case	
Objective	Saving the current game that is played.
Precondition	User is in game-playing mode.
Main Flow	1. The user interacts with keyboard and mouse to signal the request to the system 2. A save menu appears where user chooses to save current game or quit current game. 3. If save game is selected, the system stores information about the status of the current game.
Alternative Flows	-
Post-condition	Game data is saved.

Flow of Events for the Control Hero Use-case	
Objective	Controlling actions of the hero in the game.
Precondition	User is in game-playing mode.
Main Flow	1. The user interacts with keyboard and mouse to signal the request to the system 2. The system makes the necessary modifications in the system to take the desired action. 2.1. If the action is “move” system updates the positions of the elements in the environment.
Alternative Flows	Step 2.1 can be either 2.2. If the action is open/close/take/drop/use item system updates items location/state. or 2.3. If the action is speak, system initiates conversation with the user and the character identified.
Post-condition	Hero’s place is updated with respect to the defined action.
Description	During the play mode user controls the actions of the hero (main character) by directing her forwards, backwards, rightwards, leftwards, upwards (jumping) or making her open/close/take/drop/use objects or speak/interact with other characters.

Flow of Events for the Query Inventory Use-case	
Objective	Seeing and modifying inventory of the hero.
Precondition	User is in game-playing mode.
Main Flow	1. The user interacts with keyboard or mouse to signal the request to the system 2. The system displays the inventory in a new sub window.
Alternative Flows	In addition to 1&2, 3. The system updates the locations of the objects in the inventory.
Post-condition	Inventory information is displayed to the user and inventory is updated considering the changes made.
Description	When clicked the inventory button the user sees the contents of the inventory of the hero and takes information (name, usage, description, damage rate if applicable) about items in inventory by moving scroll over an item. The user also modifies the inventory by changing the places objects, using or dropping them.

Flow of Events for the Query Map Use-case	
Objective	Seeing map of the level.
Precondition	User is in game-playing mode.
Main Flow	1. The user interacts with keyboard and mouse to signal the request to the system 2. The system displays the level map in a sub window.
Alternative Flows	-
Post-condition	The visited path information of the level that user is in, is displayed.
Description	After clicking map button the user sees the map information related with the level. The paths that are passed up to that moment are displayed in a sub window that is displayed at the center of the game window.

Flow of Events for the New Game Use-case	
Objective	Start playing a new game.
Precondition	User is in main menu.
Main Flow	1. The user interacts with keyboard and mouse to signal the request to the system 2. The system displays new game information. 3. System emerges level 1of the game and user starts controlling the hero.
Alternative Flows	-
Post-condition	User is enters in game-play mode, level 1 of the game is initiated and hero obeys the commands that the user gives.

Flow of Events for the Load Game Use-case	
Objective	Start playing an existing game.
Precondition	User is in main menu and a game that has been saved before exists.
Main Flow	1. The user interacts with keyboard and mouse to signal the request to the system 2. The system displays existing games. 3. System loads the specified game and user starts controlling the hero.
Alternative Flows	-
Post-condition	The game is initiated with the identified level information and hero obeys the commands that the user gives.

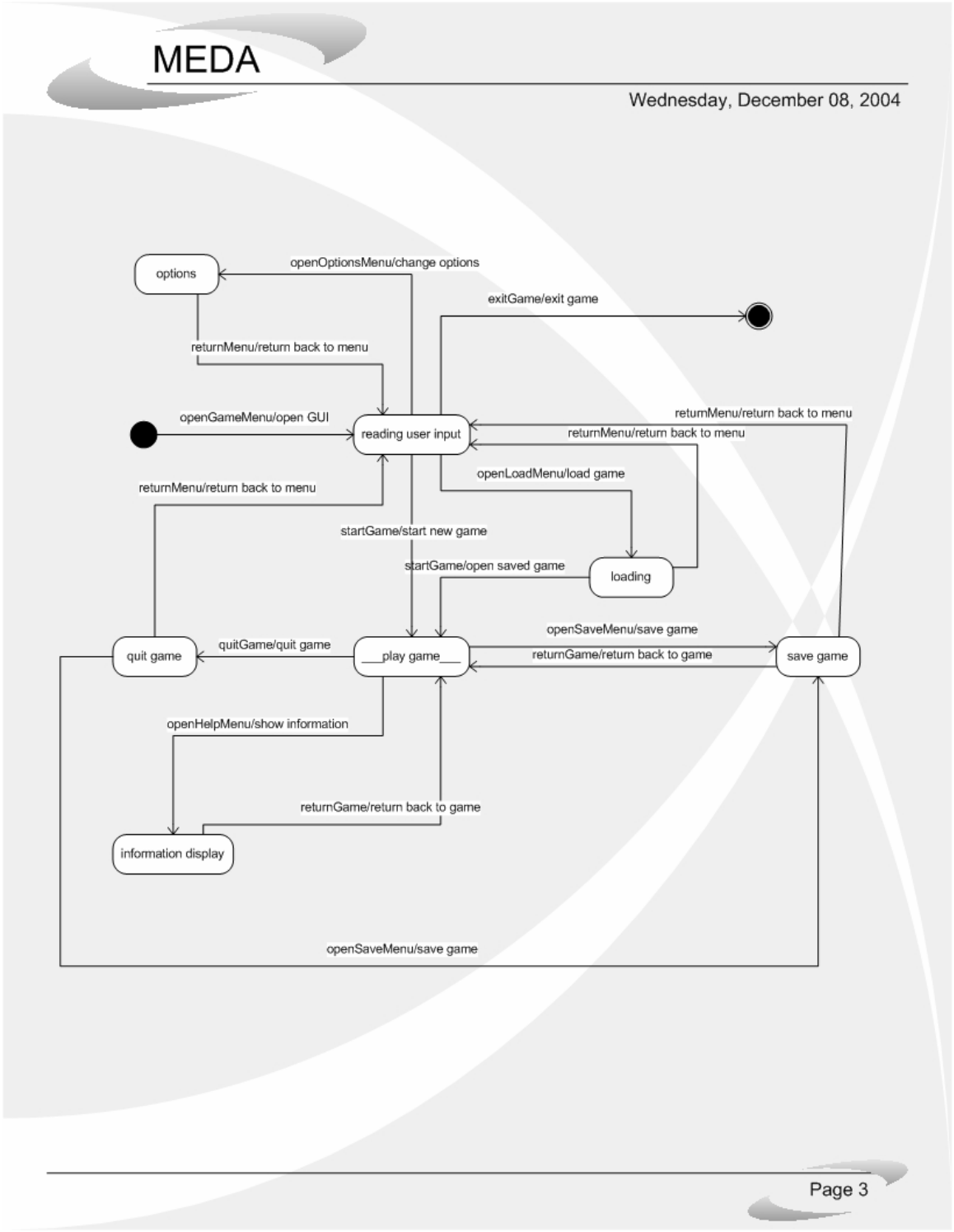
Flow of Events for the Credits Use-case	
Objective	Getting information about the credits of the game.
Precondition	-
Main Flow	1. The user interacts with keyboard and mouse to signal the request to the system 2. The system displays credits of the game.
Alternative Flows	-
Post-condition	Credits of the game are displayed as a new window.

Flow of Events for the Options Use-case	
Objective	Configuring options of the game.
Precondition	User is in main menu.
Main Flow	1. The user interacts with keyboard and mouse to signal the request to the system 2. The system displays options of the game. 3. User sees and modifies the options.
Alternative Flows	-
Post-condition	Options are updated.

Flow of Events for the Exit Game Use-case	
Objective	Quitting the game.
Precondition	User is in main menu.
Main Flow	1. The user interacts with keyboard and mouse to signal the request to the system 2. The system
Alternative Flows	-
Post-condition	System stops running.

11. STATE TRANSITION DIAGRAM

11.1. DIAGRAM



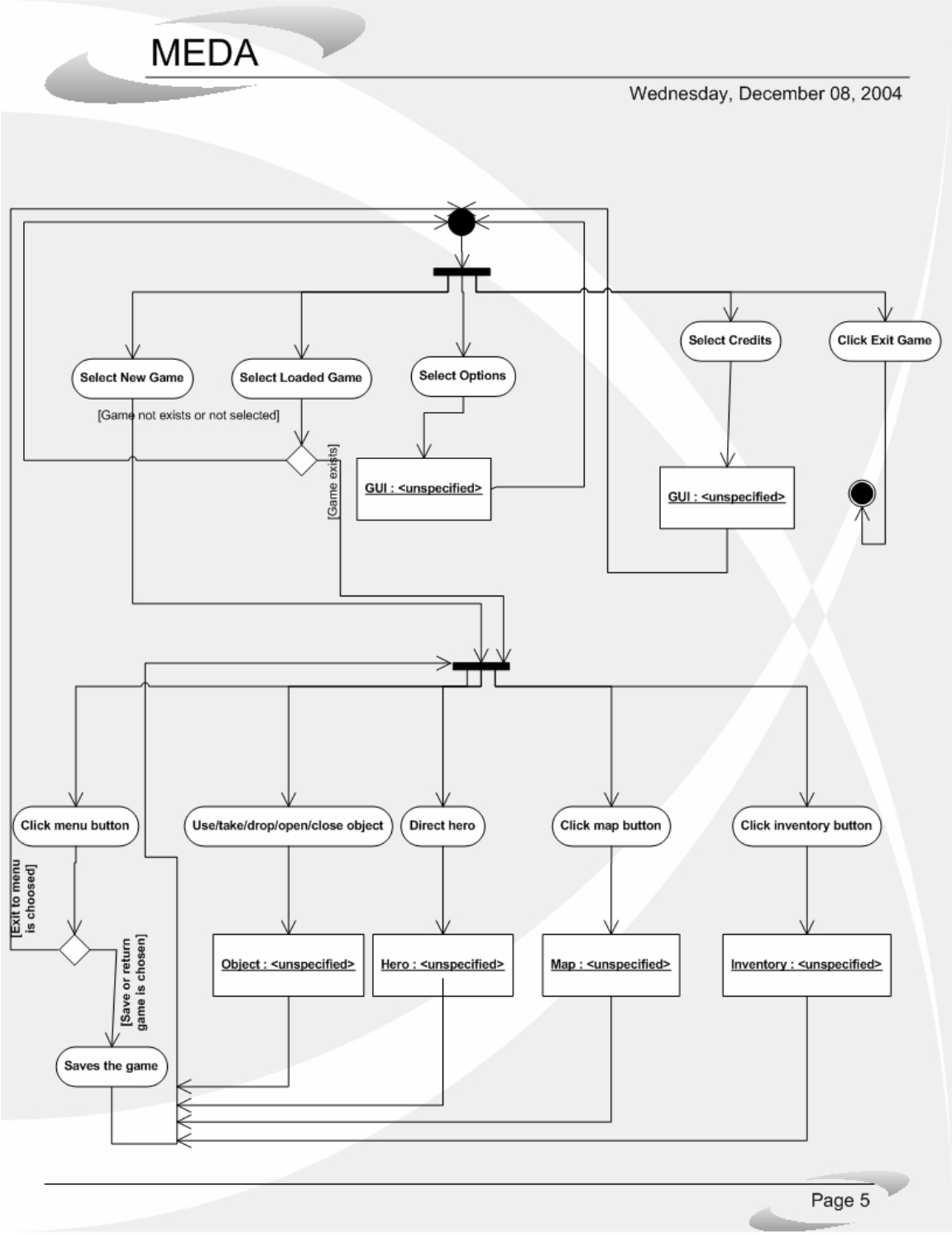
11.2. EXPLANATION

- reading user input : This is a state of GUI which waits inputs from the user. We can change this state to loading state with openLoadMenu event, _play game_ state with startGame event and options state with openOptionsMenu.

- **loading :** This is a state of GUI which loads the saved game data. We can change this state to reading user input state with returnMenu event and _play game_ state with startGame event.
- **_play game_ :** This is the game playing state of GUI. We can change this state to save game state with openSaveMenu event, quit game state with quitGame event and information display state with openHelpMenu event.
- **save game :** This is a state of GUI which implements game saving. We can change this state to _play game_ state with returnGame event and reading user input state with returnMenu event.
- **quit game :** This is a state of GUI which implements quit game. We can change this state to reading user input state with returnMenu event and save game state with openSaveMenu event.
- **information display :** This is a state of GUI which displays the level information. We can change this state to _play game_ state with returnGame event.
- **options :** This is a state of GUI which displays the options and lets the user to change them. We can change this state to reading user input state with returnMenu event.

12. ACTIVITY DIAGRAM

12.1 DIAGRAM



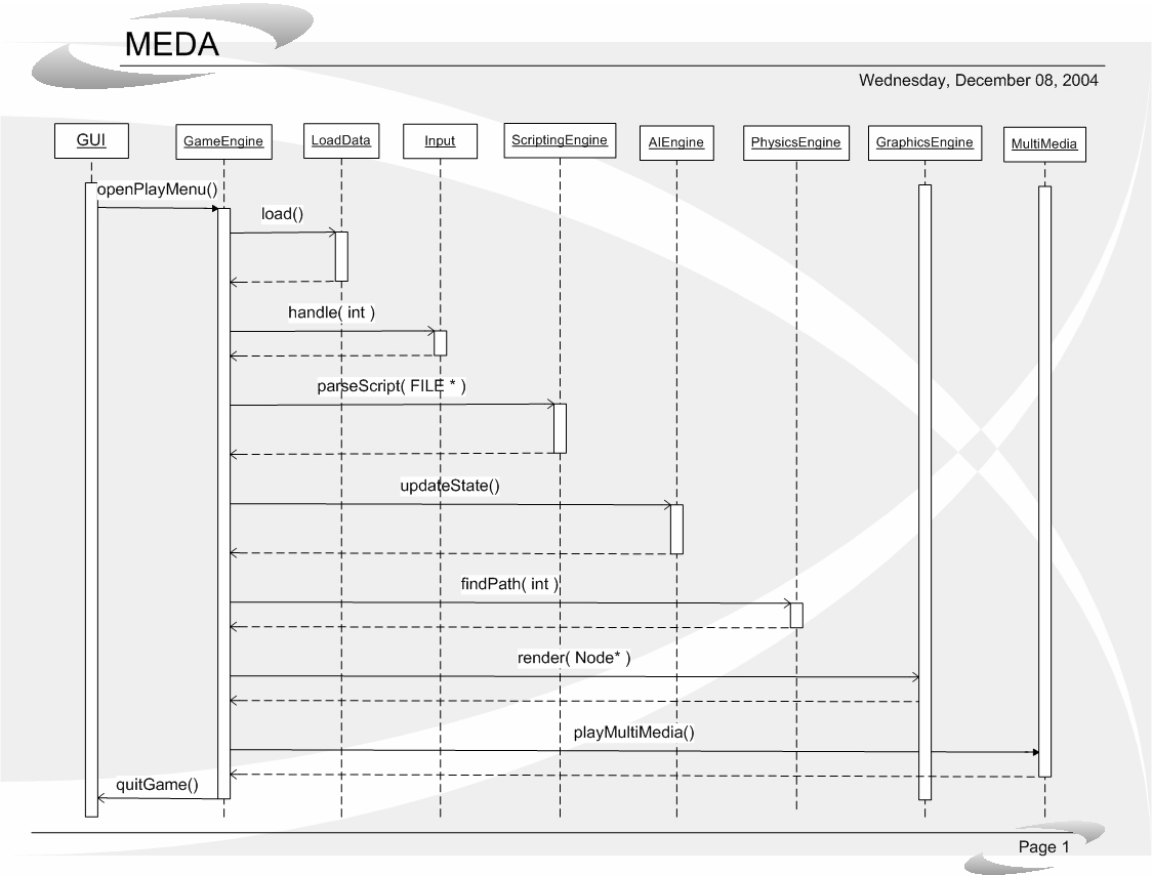
12.2. EXPLANATION

We have 5 activities originated from the starting condition. These are select new game, select loaded game, select options, select credits and click exit game. Select options and select credits returns to the initial condition after implementing their activities. On the other hand, exit game implements exit game and finishes the activities.

Select loaded game and select new game passes another condition after their activities. 5 activities originates from that condition. These are click menu button, use/take/drop/open/close object, direct hero, click map button and click inventory button. After their activities, use/take/drop/open/close object, direct hero, click map button and click inventory button are returned to the condition from which they are originated. Click menu button creates another activity named saves the game or returns to the starting condition according to the result of the exit or save game selection. If it creates the “saves the game” activity, “saves the game” activity returns the condition from which the click menu button activity is originated.

13. SEQUENCE DIAGRAM

13.1 DIAGRAM



13.2 EXPLANATION OF SEQUENCE DIAGRAM

We represented the time-method sequence relationship and the class relationships in this diagram.

- Firstly, GUI class has a relationship with GameEngine class with openPlayMenu method.

- When the play-game status occured, GameEngine class associates with the LoadData class with load event to implement the loading of models, textures, level information etc.

- After that, GameEngine class relates with input class using handle event to implement user interaction.

- Then, it is the turn of scripting engine. Against the action of the user, ScriptingEngine class reads the script related to that action, parses the script and gives the answer for the action to the GameEngine class. GameEngine class and ScriptingEngine class associates with each other by the help of parseScript event.

- After that the action parsed by scripting engine is given to AIEngine class with updateState event. AIEngine implements the deciding action and gives the answer against the action of the user.

- Then, the GameEngine class associates with PhysicsEngine class by the help of findPath event. PhysicsEngine makes the calculations for the action generated by AIEngine class and returns the effect of action to the GameEngine class.

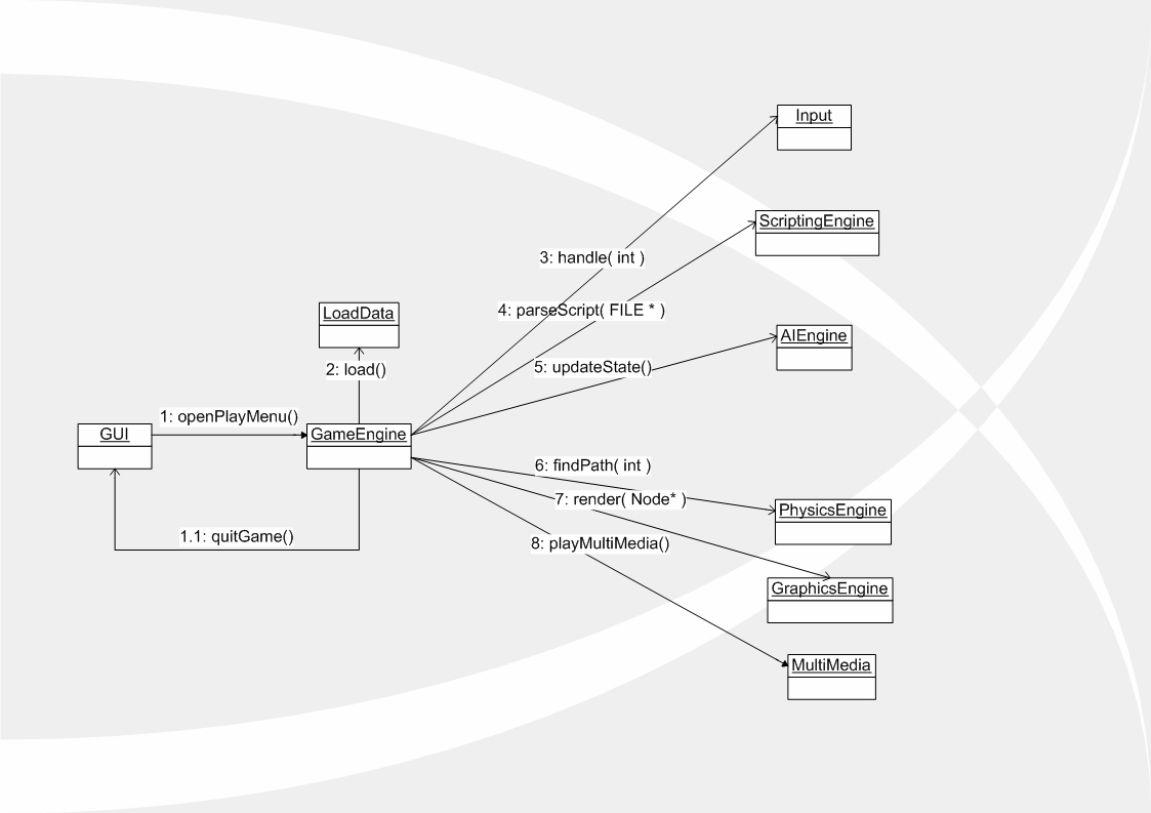
- After that, another important phase comes. It is rendering phase and implemented by the GraphicsEngine. GameEngine associates with the GraphicsEngine class by the help of render event. GraphicsEngine class renders the graphics and gives the control to the GameEngine class again.

- Then, MultiMedia class is associated with GameEngine class with the playMultiMedia event. The related medias are played and the control again bellongs to GameEngine cass.

- Lastly, GameEngine class is related with the GUI again by the help of quitGame event.

14. COLLABORATION DIAGRAM

14.1. DIAGRAM



14.2. EXPLANATION

We have shown the relationships of the classes in this diagram.

GameEngine class:

LoadData with load();

Input with handle(int);

ScriptingEngine with parseScript(FILE *)

AIEngine with updateState();

PhysicsEngine with findPath(int);

GraphicsEngine with render(Node *);

MultimediaEngine with playMultiMedia();

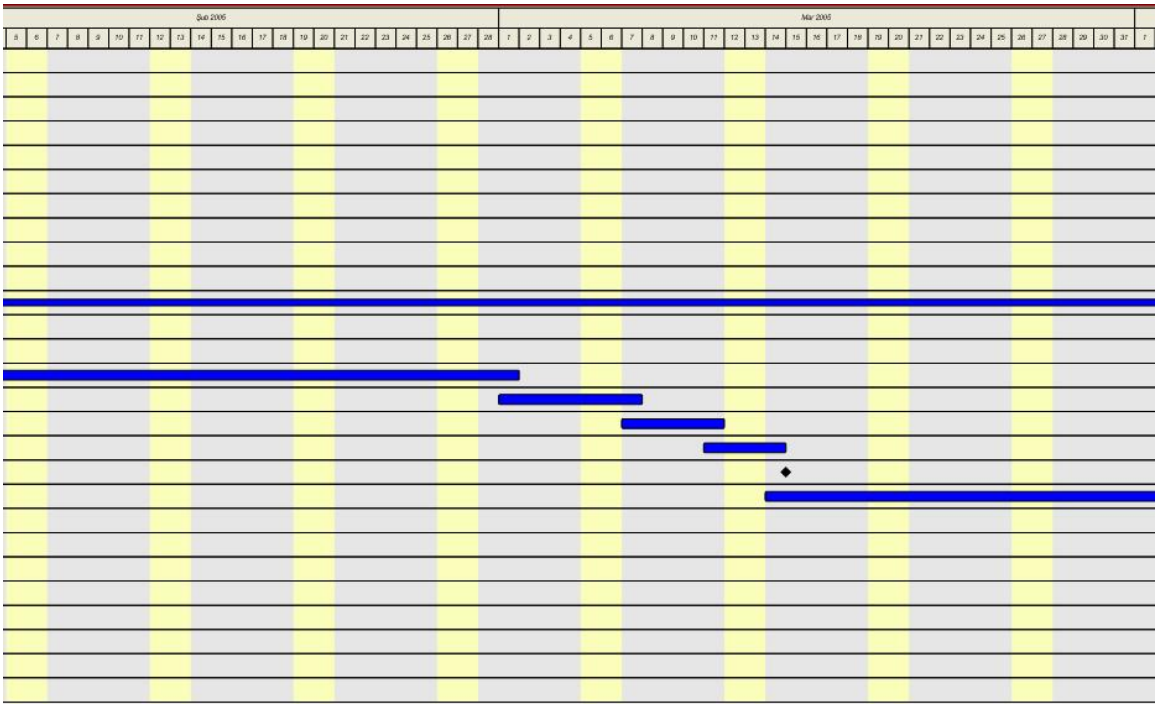
GUI with quitGame();

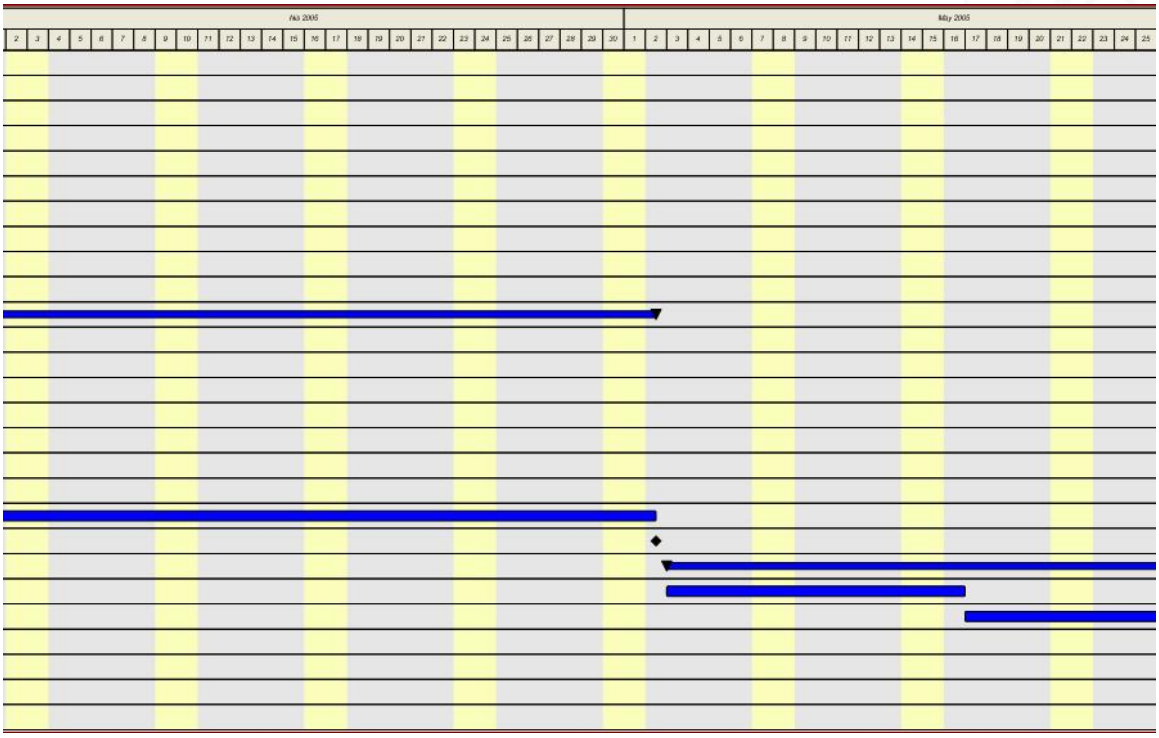
GUI class:

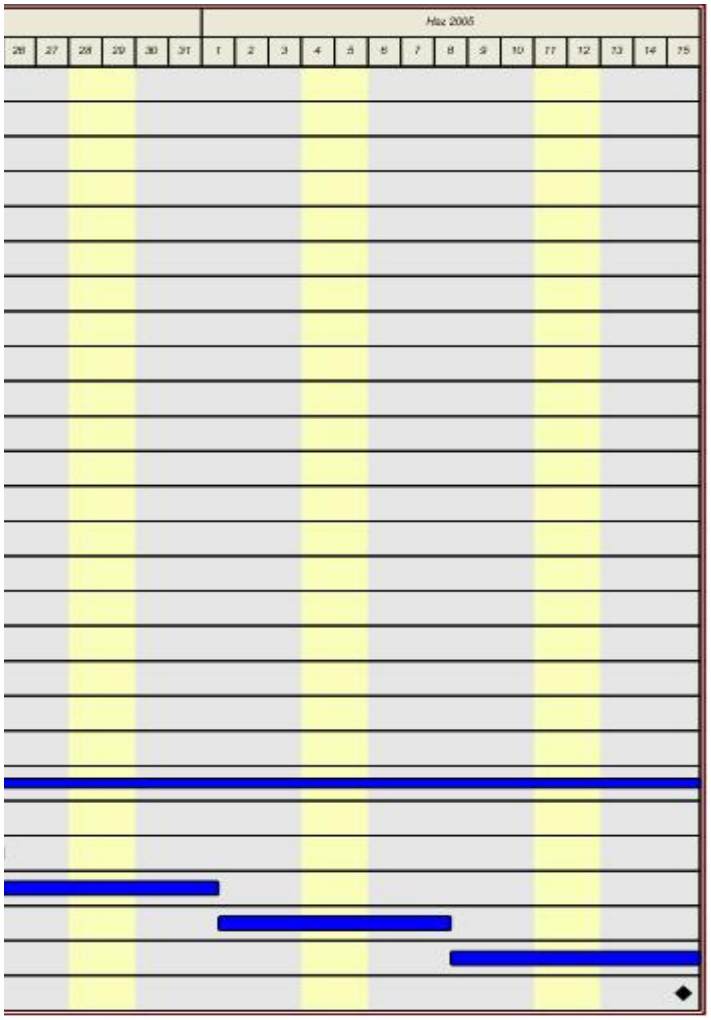
GameEngine with openPlayMenu();

15. Gantt Chart

ID	Task Name	Start	Finish	Duration
1	**Development of Prototype**	**03.01.2005**	**18.01.2005**	**96h**
2	Development of Character class	03.01.2005	07.01.2005	40h
3	Development of Environment	03.01.2005	07.01.2005	40h
4	Loading a Character to the Environment	10.01.2005	14.01.2005	40h
5	Integrating ODE	14.01.2005	18.01.2005	24h
6	Integrating Phyton	14.01.2005	18.01.2005	24h
7	Integrating Open AL	14.01.2005	18.01.2005	24h
8	milestone1	19.01.2005	19.01.2005	0h
9	Update Web Page	20.01.2005	21.01.2005	16h
10	Planing of Configuration Management	20.01.2005	25.01.2005	32h
11	**IMPLEMENTATION**	**20.01.2005**	**02.05.2005**	**580h**
12	Development of GUI class	20.01.2005	01.02.2005	72h
13	Development of Object cass	20.01.2005	01.02.2005	72h
14	Development of remaining classes	02.02.2005	01.03.2005	160h
15	Integrating the Modules	01.03.2005	07.03.2005	40h
16	Integrating Puzzles	07.03.2005	11.03.2005	40h
17	Development of The First Snapshot	11.03.2005	14.03.2005	16h
18	milestone2	15.03.2005	15.03.2005	0h
19	Improving the Modules	14.03.2005	02.05.2005	284h
20	milestone3	02.05.2005	02.05.2005	0h
21	**Testing**	**03.05.2005**	**17.06.2005**	**272h**
22	Unit Testing	03.05.2005	16.05.2005	80h
23	Integration Testing	17.05.2005	25.05.2005	56h
24	Validation Testing	26.05.2005	01.06.2005	36h
25	Performance Testing	01.06.2005	08.06.2005	40h
26	Prepare Test Document	08.06.2005	17.06.2005	60h
27	milestone4	15.06.2005	15.06.2005	0h



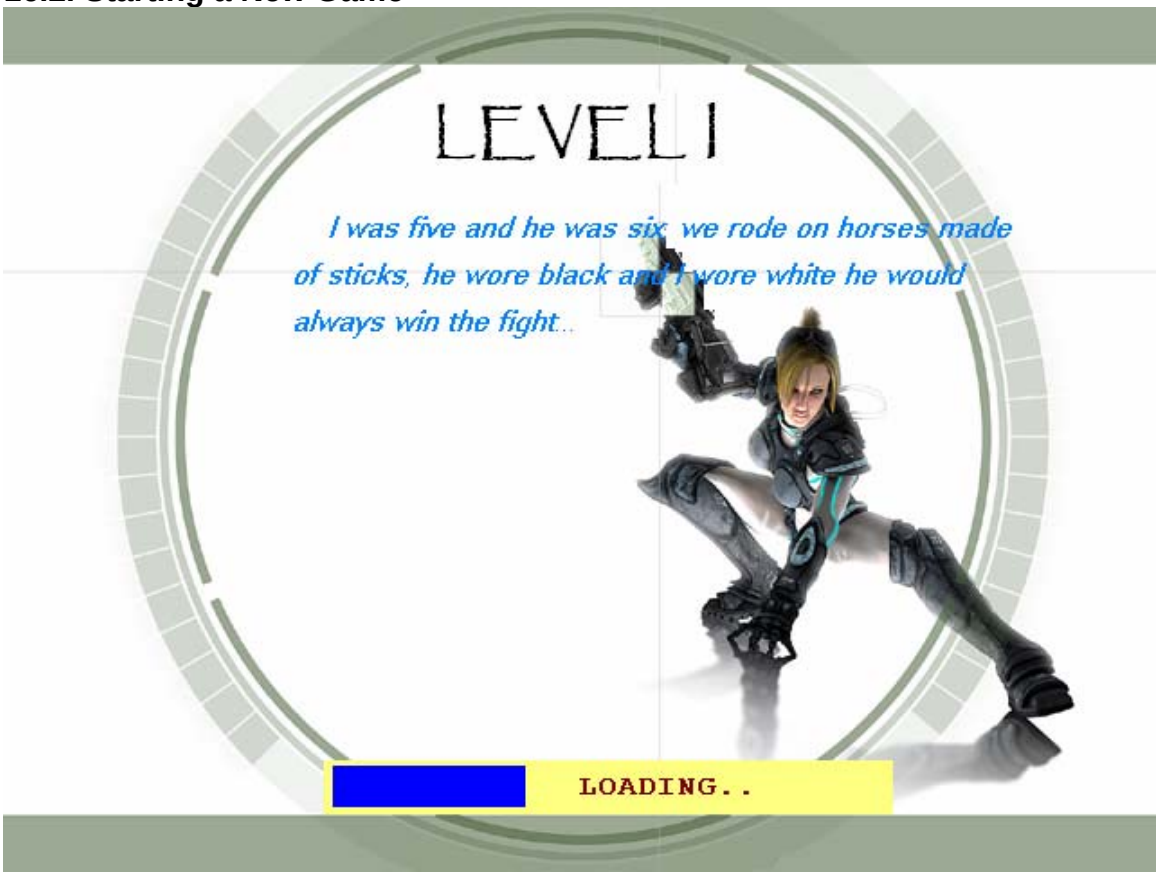




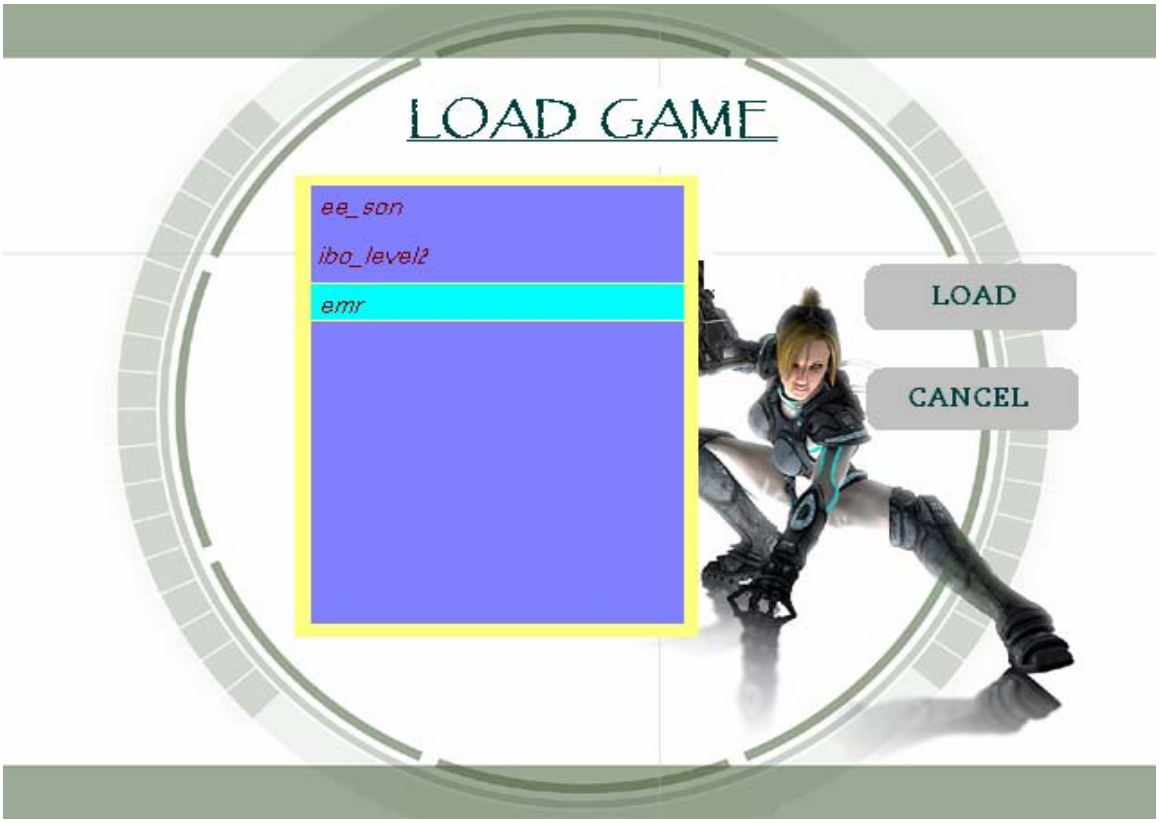
16. Appendix

16.1. Game Menu

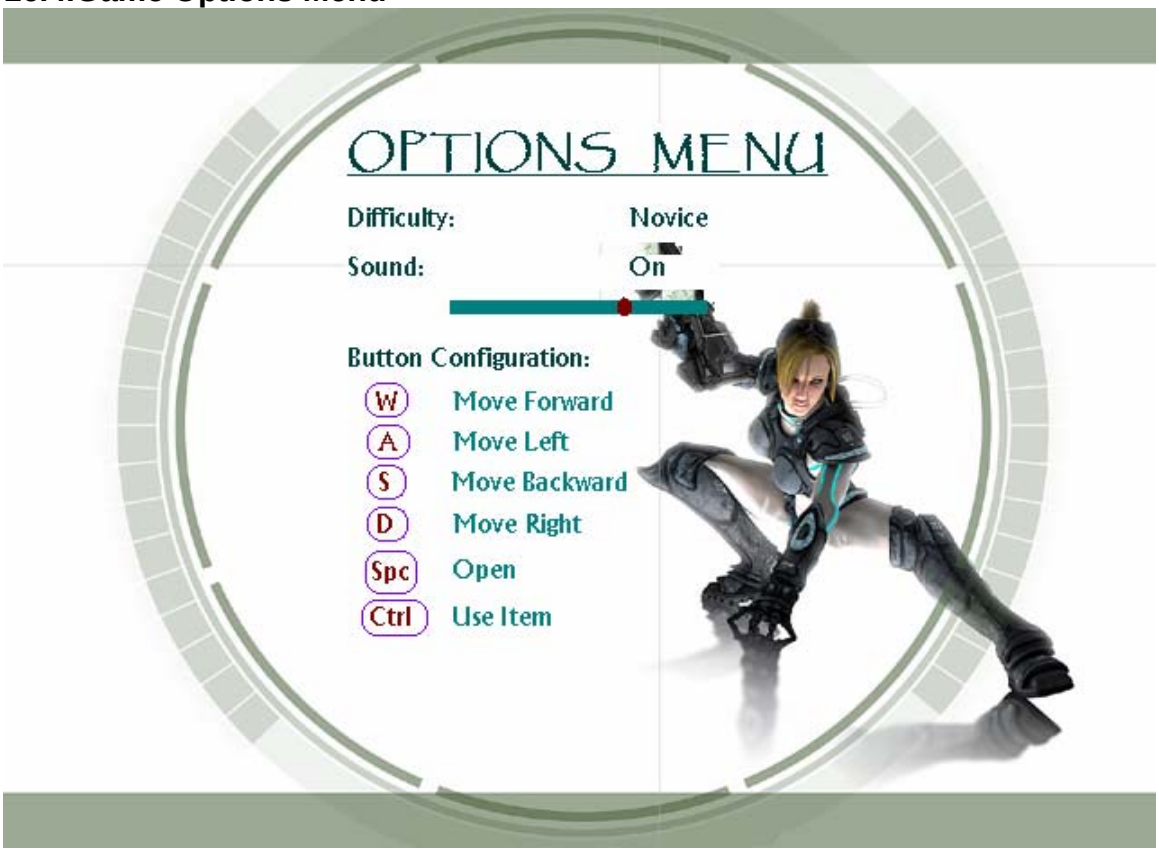
16.2. Starting a New Game



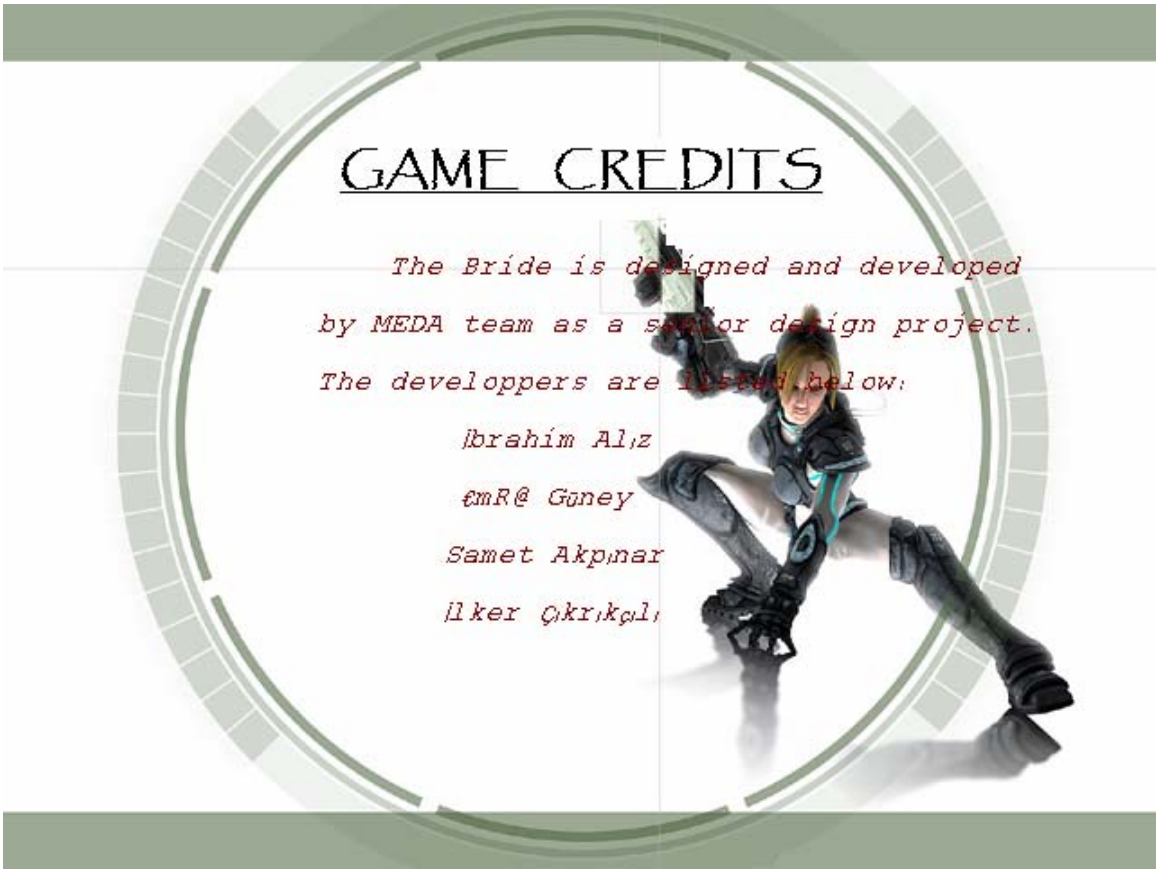
16.3.Load Game Menu



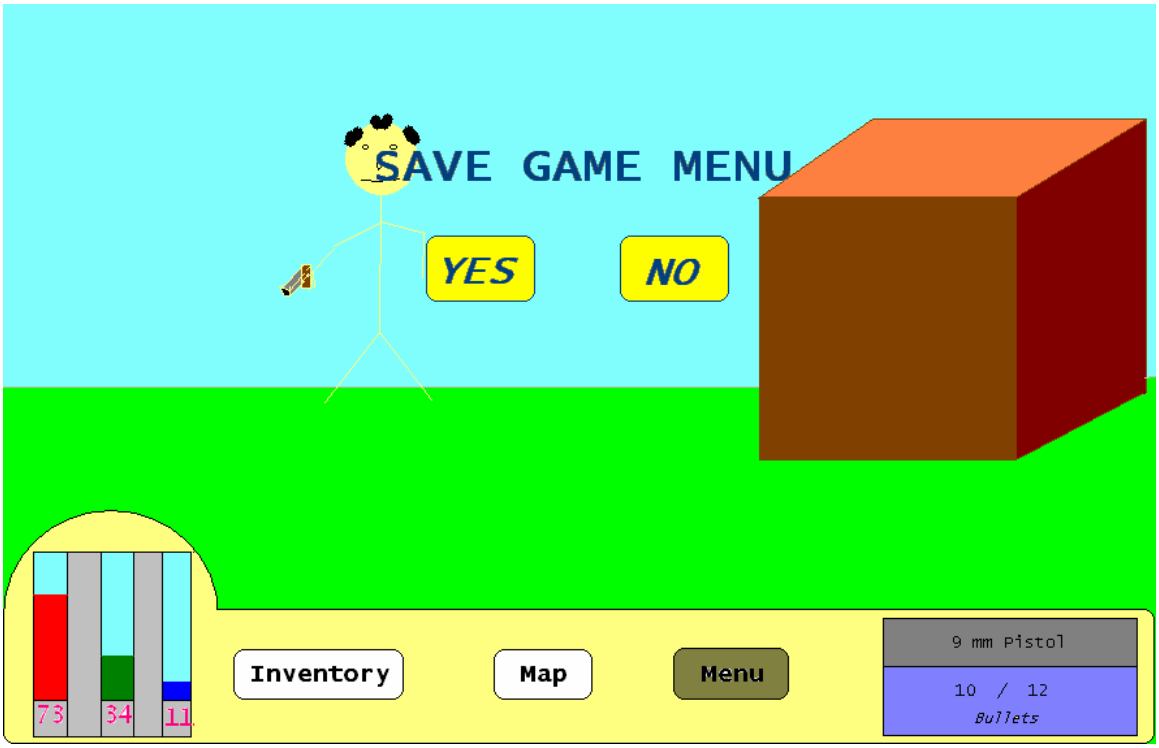
16.4.Game Options Menu



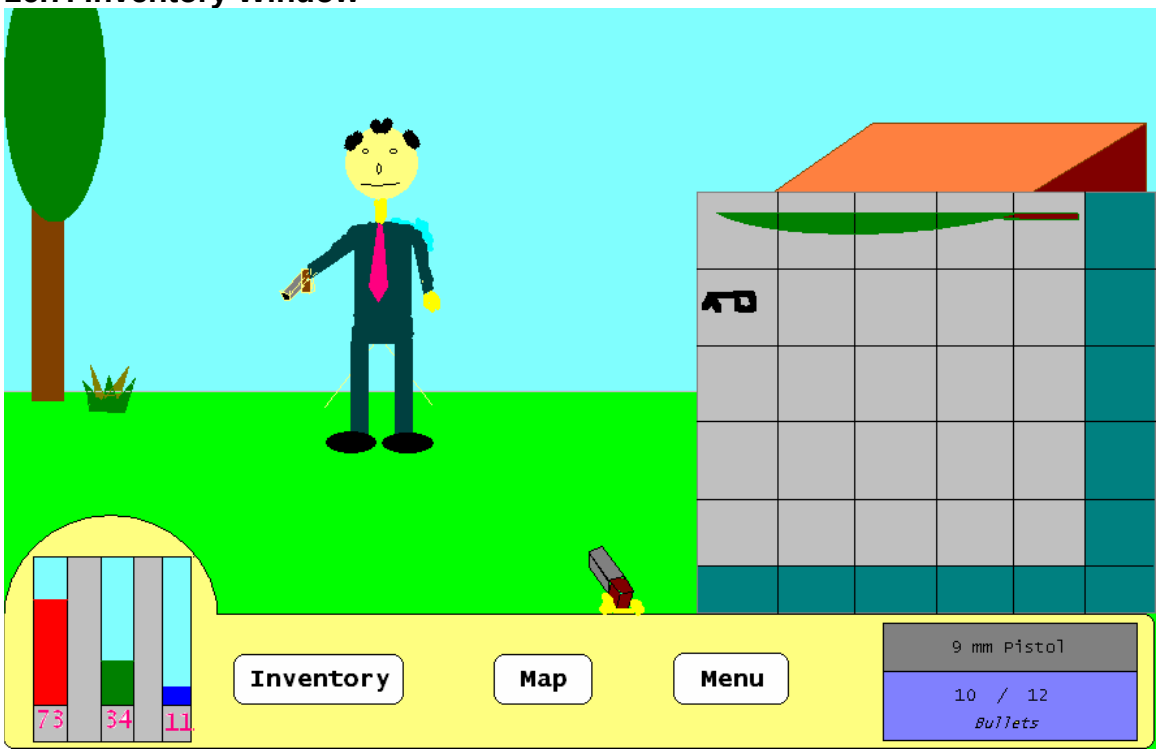
16.5.Game Credits



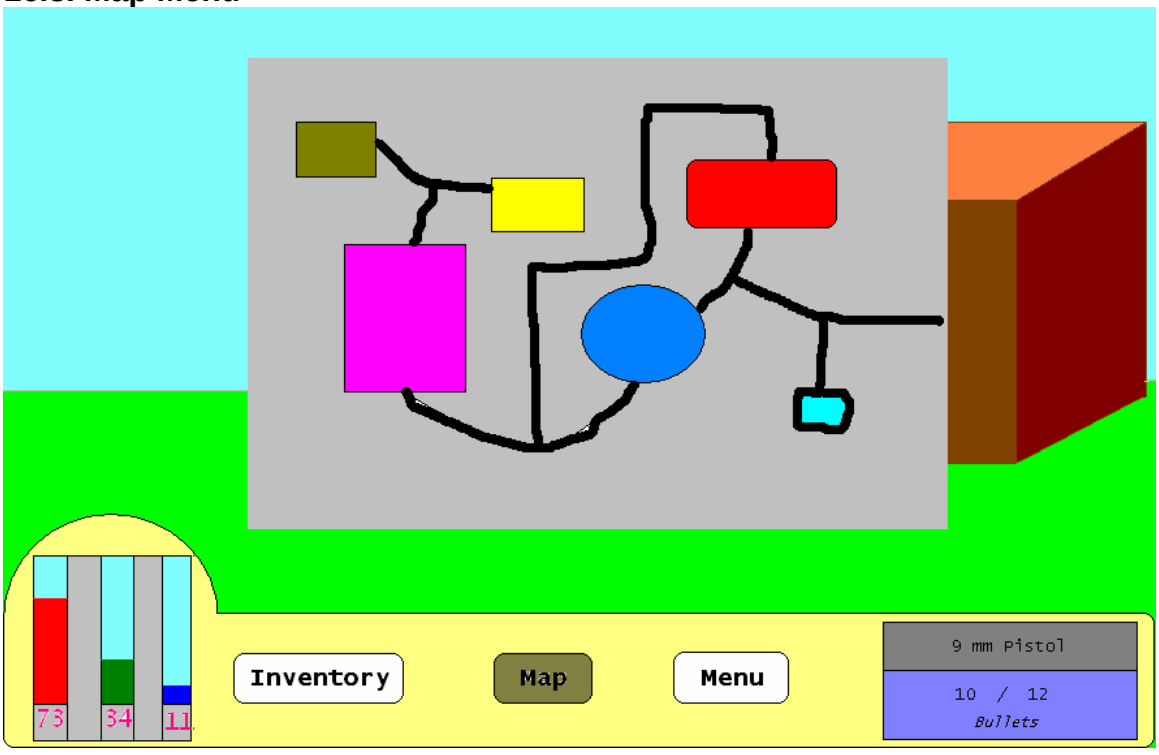
16.6.SaveMenu



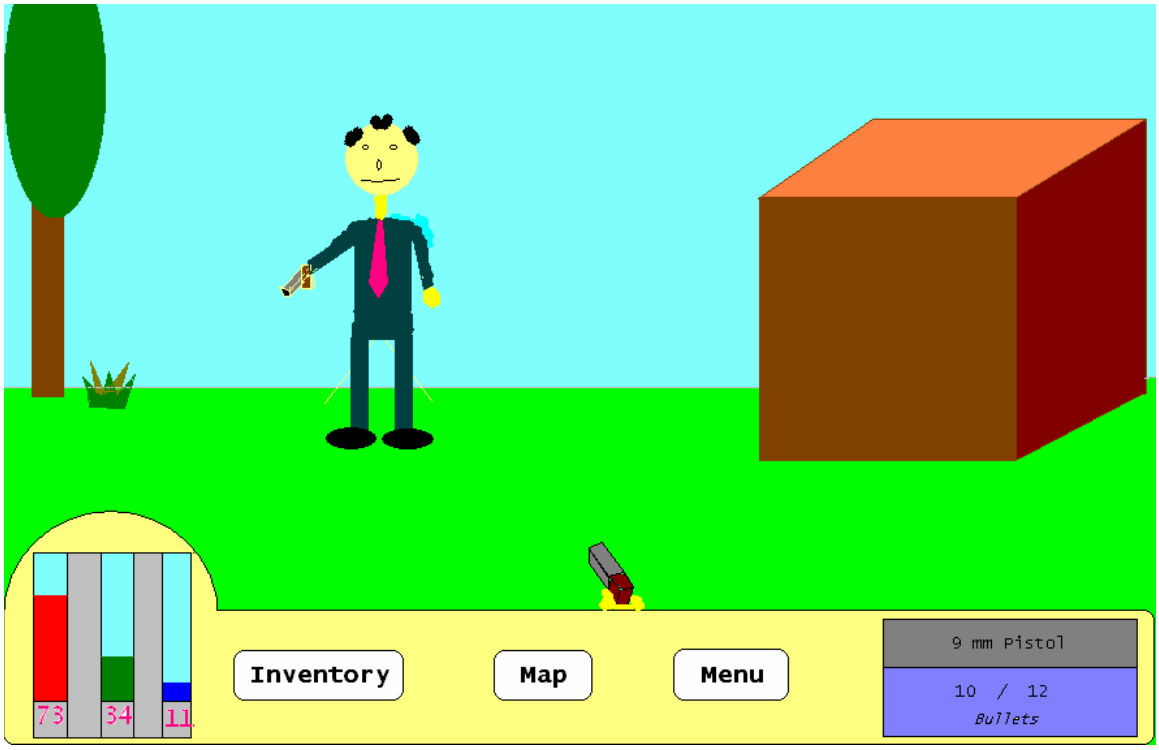
16.7. Inventory Window



16.8. Map Menu



16.9. First Person View



16.10. Third Person View