TABLE OF CONTENTS

Initial Design Report

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. 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.

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.

3 -) Exit Game: This button is defined as the exiting point from the program. When it is clicked the program will terminate.

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.

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'.

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 current game data will be saved under the name specified.

And finally the 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'.

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. In order to load the game data to the corresponding data structures game engine calls the initialization methods defined.

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 functions of the graphics engine namely OpenGL. Game engine calls an update method which calls the update methods of 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.

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.

3.4 Loader Module

This module handles the interpretation of the 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 3ds format we are going to write our own parser that will extract the information from the corresponding files. The environment data shall also contain the required texturing information.

3.5 Artificial Intelligence Engine

In order handle the up keeping of the game state according to the puzzles defined for levels an AI engine is going to be implemented that will update the next state information according to the information gathered via scripts.

The AI engine shall bear an operator-based knowledge base, allowing programmers to mix and match tactics, behaviors and goals as appropriate for their game. The artificial intelligence engine should support agents that are:

1. Reactive: These are the agents that respond quickly to changes in the environment and those reactions are specific to the current situation. Reactive (stimulus-response) agents just react to the current situation at each time step with no memory of past actions or situations. Calculation of the response of this kind of agents shall be very quick since it will not depend on contextual information.

2. Context Specific: These are the agents those ensure that their actions are consistent with past sensor information and the agent's past actions. These kind of agents shall consider the their current situation within the game state.

3. Flexible: These agents have a choice of high level tactics with which to achieve current goals and a choice of lower level behaviors with which to implement current tactics. In order to make use of contextual information script-based agents are generated which have a number of scripts, or sequences of actions, one of which is selected and executed over a number of time steps. And once a script is selected all the actions performed are consistent with the context and goals of the script.

4. Easy to Develop: The artificial intelligence engine makes agent development easier by using a knowledge representation that is easy to program and by reusing knowledge as much as possible.

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.

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. While generating scripts, not to affect the performance negatively scripting will mostly be done for game functionality and GUI responses. Also, via scripting, except having all

the code in memory at once the will be able to be loaded from the script files. Besides saving memory, scripting engine allows the dynamic loading and unloading of pieces of code which make it possible to over-ride functionality on the fly.

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. As a design issue, we are going to implement a multimedia module that answers these needs. The module consists of a static class which stores and manages the audio files or video file that is played. We will define methods to play menu music (played only in menus), background music (played during the game-play), and play various sound effects (sounds that are caused by the elements in the game - hero, objects and enemies).

Other modules should be able to use these routines so these routines should be packaged an independent module making the required methods public.

We will make use of free libraries such as DevLib and OpenAL to handle these issues easily and efficiently. The Open Audio Library (OpenAL) is a free platform independent library (like OpenGL) which has the capability of playing sound files. On the other DevLib is a relatively new but much more talented library which also provides classes for playing both music files and video files. Indeed DevLib has the power of many popular and widely -especially in the game development area.

Basically DevLib is an object-oriented development framework written in C++. The main advantages of using DevLib is that 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. One may save/load images, export meshes from

LightWave, 3D Studio MAX and Maya, play sound files, show videos, execute LUA scripts and manage consoles which allow updating the value of user-specified keys at run-time. DevLib supports OpenGL and DIRECTX as render system, PNG, JPG, BMP, TGA file extensions in image management, M1V and M2V files which are based on MPEG-1 and MPEG-2 for video management, MP3, OGG, WAV, XM and MOD file extensions for sound management. DevLib is fully compatible with Microsoft's Visual C++ 2003.

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. BSP tree construction is a process which takes a subspace and generates partitions from this subspace by any hyper-plane that intersects the interior of that subspace. The result is two new subspaces that can be further partitioned by recursive application of the method.

A "hyper-plane" in n-dimensional space is an (n-1) dimensional object which can be used to divide the space into two half-spaces. For example, in three dimensional space, the "hyper-plane" is a plane. In two dimensional space, a line is used.

BSP trees are chosen for implementation of spatial data since they are extremely versatile and powerful sorting & classification structures. They have uses ranging from hidden surface removal and ray tracing hierarchies to solid modeling and robot motion planning.

Another important topic for performance issues is managing the level of detail effectively. In order to handle the issue we are going to use mipmaps with continuously increasing level of detail.

5. Level Design & Puzzles

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.

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.

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.

Specific Methods: –

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 boxes 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)

6.14. Book

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

6.14.1. Attributes:

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

6.14.2. Methods:

Inherited Methods:

- CharacterMesh getMesh(void)

- Vertex getOrigin(void)

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

- void useItem(void): This method is for throwing the book.

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

Specific Methods: This class is the class for books and inherited from object class.

- void insert(Closet) : This method inserts the book into the closet.

6.15. Closet

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

6.15.1 Attributes:

- Object *objects : This attribute holds the objects of the closet.

6.15.2 Methods:

Inherited Methods:

- CharacterMesh getMesh(void)

- Vertex getOrigin(void)

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

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

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

Specific Methods: –

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.

6.20.1 Attributes:

Script *script: This is a temporarily variable that holds the temporary information that is going to be used during the game.

6.20.2 Methods:

- void genScript(string): This method writes some specific information related with the current environment conditions into a file.

- void parseScript(FILE *): This method reads the state specific information from a file parses this information as a script.

- Script returnScript(void): This method returns the current information stored in the script object.

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)

- void mouseHandler(void)

- void consoleHandler(void)

- void scriptHandler(void)

- void hande(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 defined files to the specified data structures.

6.22.1 Attributes:

All the data structures that are defined for the program are in the attributes part of this module.

6.22.2 Methods:

These methods loads the specific data identified by their names from the files to the data structures defined by the help of mutimedia modules and parser defined to get the environment and model data from specific files with the identified file formats.

- void loadModelData(void)

- void loadEnvironmentData(void)

- void loadLevelData(void)

- void loadSoundData(void)

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

- Puzzle *levelPuzzles: Carries the information about the puzzles defined for the current level.

- 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.

- Puzzle getPuzzleInfo(void): This method loads the information about the puzzles that are defined for the current level.

- 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

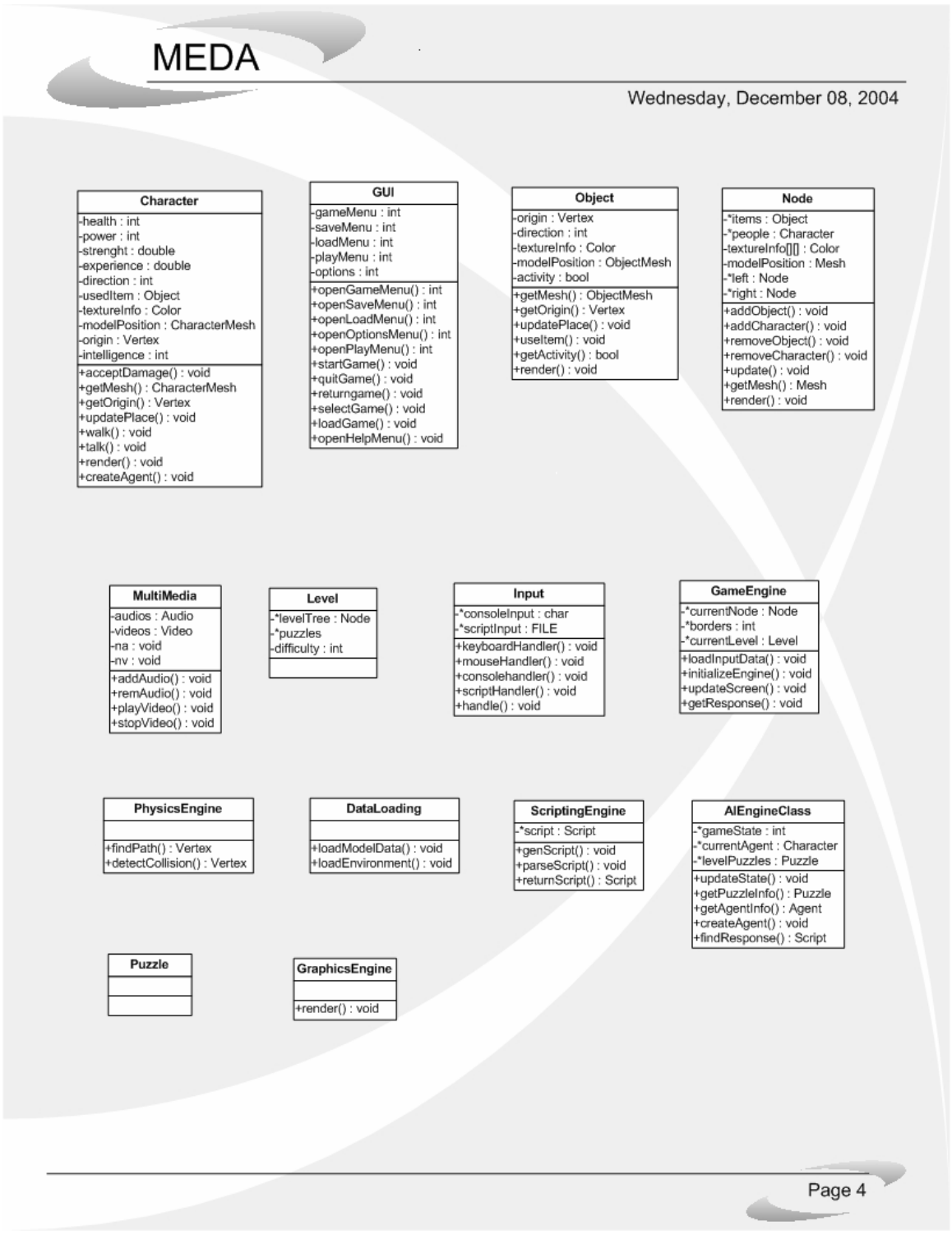
This module is for rendering the Objects, Characters and the Environment.

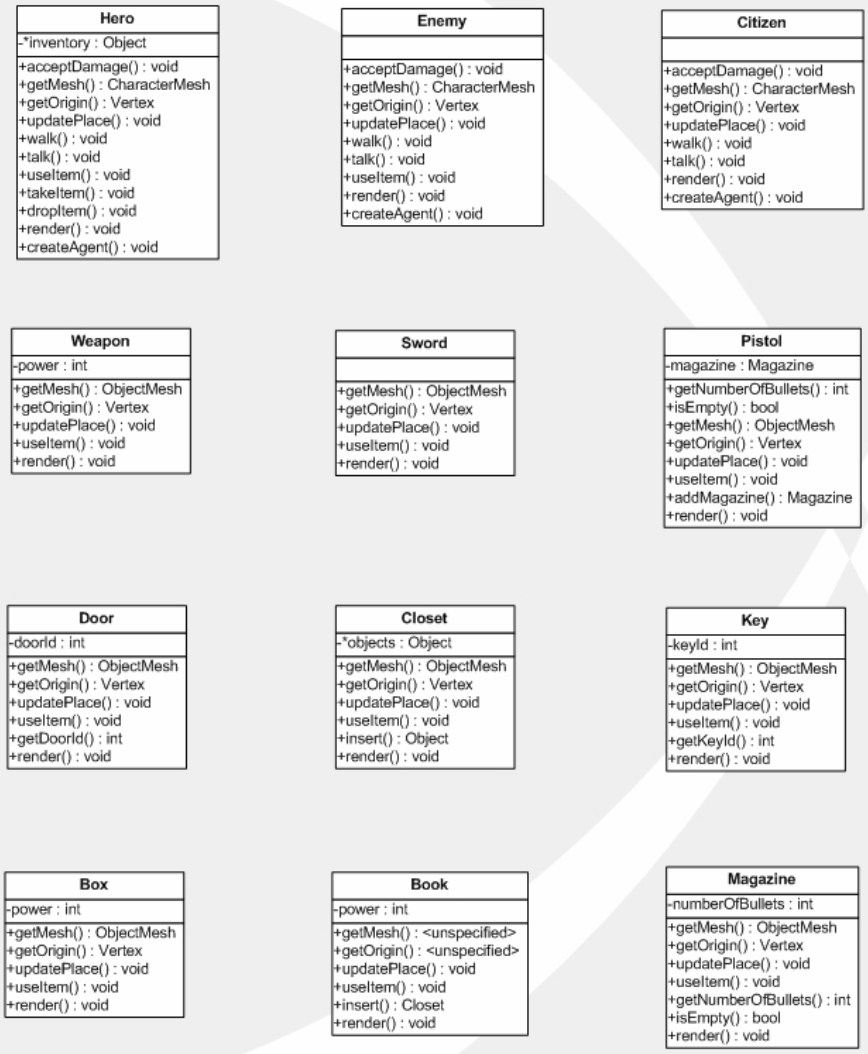
6.25.1 Attributes: –

6.25.2 Methods:

- void render(Node*)

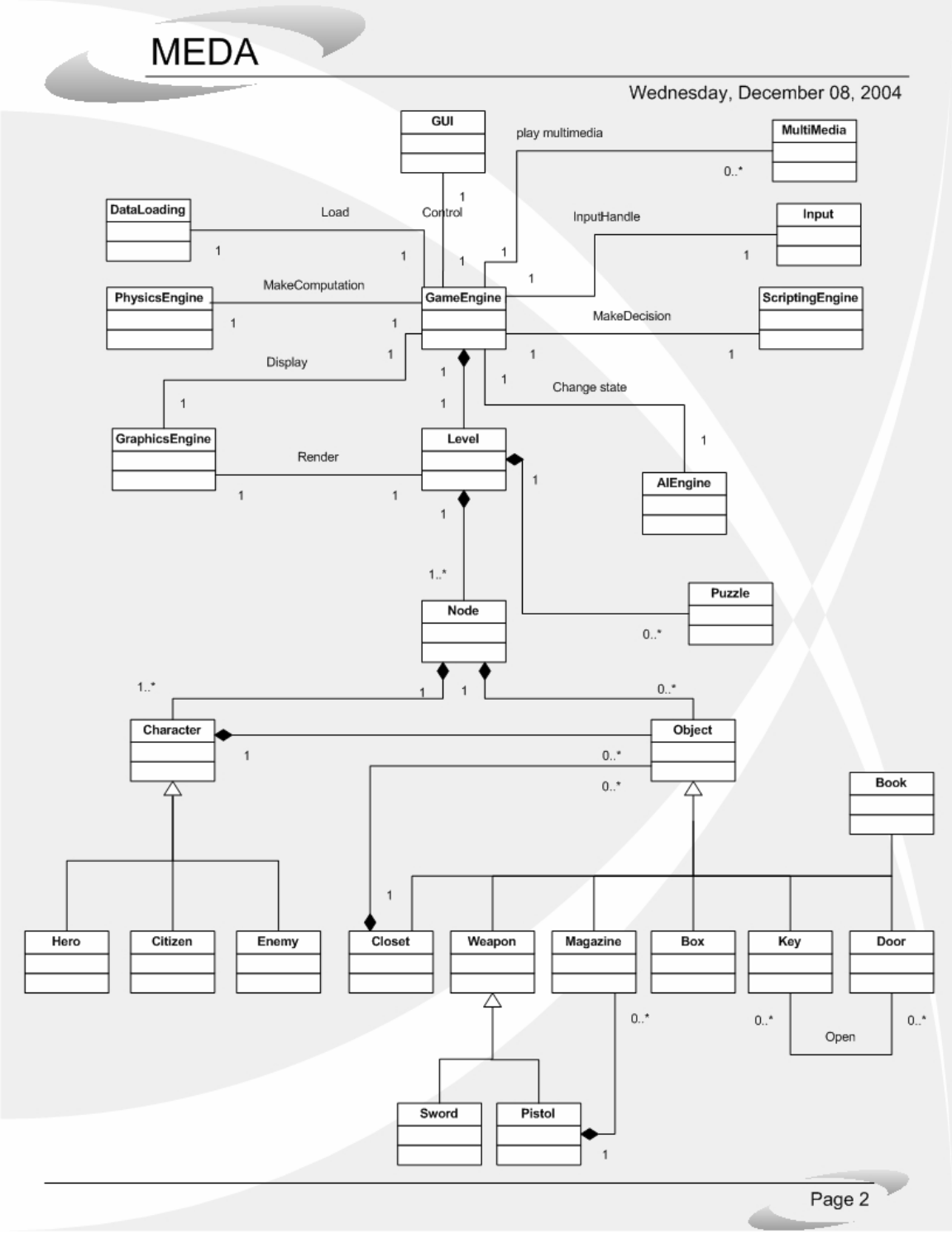
7. DIAGRAMS


MEDA
Wednesday, December 08, 2004
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
Input
-*consoleInput : char
-*scriptInput : FILE
+keyboardHandler() : void
+mouseHandler() : void
+consolehandler() : void
+scriptHandler() : void
+handle() : void
GameEngine
-*currentNode : Node
-*borders : int
-*currentLevel : Level
+loadInputData() : void
+initializeEngine() : void
+updateScreen() : void
+getResponse() : void
PhysicsEngine
+findPath() : Vertex
+detectCollision() : Vertex
DataLoading
+loadModelData() : void
+loadEnvironment() : void
ScriptingEngine
-*script : Script
+genScript() : void
+parseScript() : void
+returnScript() : Script
AIEngineClass
-*gameState : int
-*currentAgent : Character
-*levelPuzzles : Puzzle
+updateState() : void
+getPuzzleInfo() : Puzzle
+getAgentInfo() : Agent
+createAgent() : void
+findResponse() : Script
Puzzle
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
Closet
-*objects : Object
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+insert() : Object
+render() : void
Key
-keyId : int
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+getKeyId() : int
+render() : void
Box
-power : int
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+render() : void
Book
-power : int
+getMesh() : <unspecified>
+getOrigin() : <unspecified>
+updatePlace() : void
+useItem() : void
+insert() : Closet
+render() : void
Magazine
-numberOfBullets : int
+getMesh() : ObjectMesh
+getOrigin() : Vertex
+updatePlace() : void
+useItem() : void
+getNumberOfBullets() : int
+isEmpty() : bool
+render() : void

8. CLASS DIAGRAM

8.1. DIAGRAM



8.2 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:

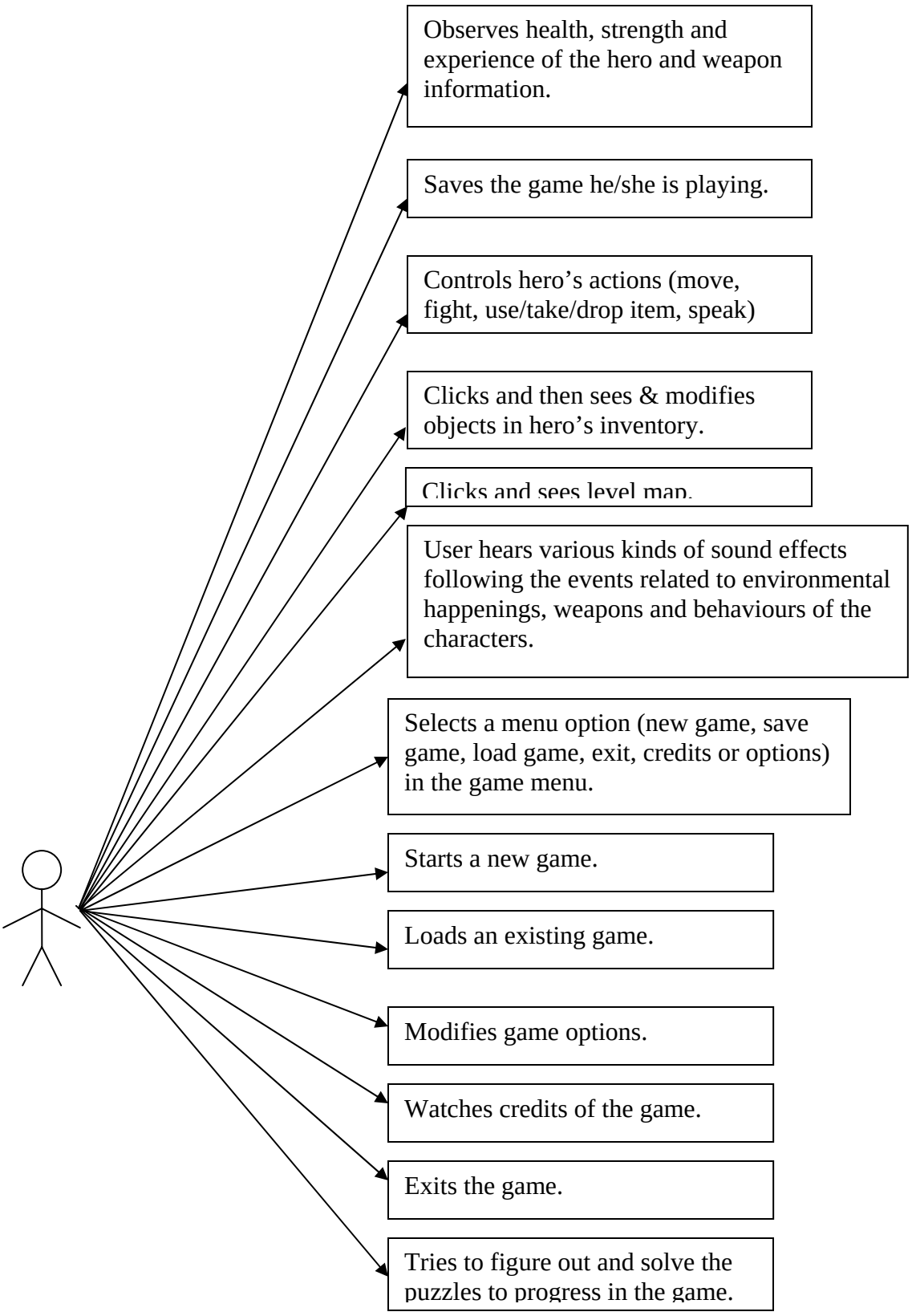
- **Game Engine – Level:** GameEngine class includes a Level object.
- **Level – Puzzle:** Each Level includes zero or more Puzzles.
- **Level – Node:** Each Level includes a pointer to the Node object.
- **Node – Character:** Each Node includes one or more Characters.
- **Node – Object:** Each Node 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, Book, Door, Closet} :** 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.

9. USE CASE DIAGRAM

9.1 DIAGRAM



9.2 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.

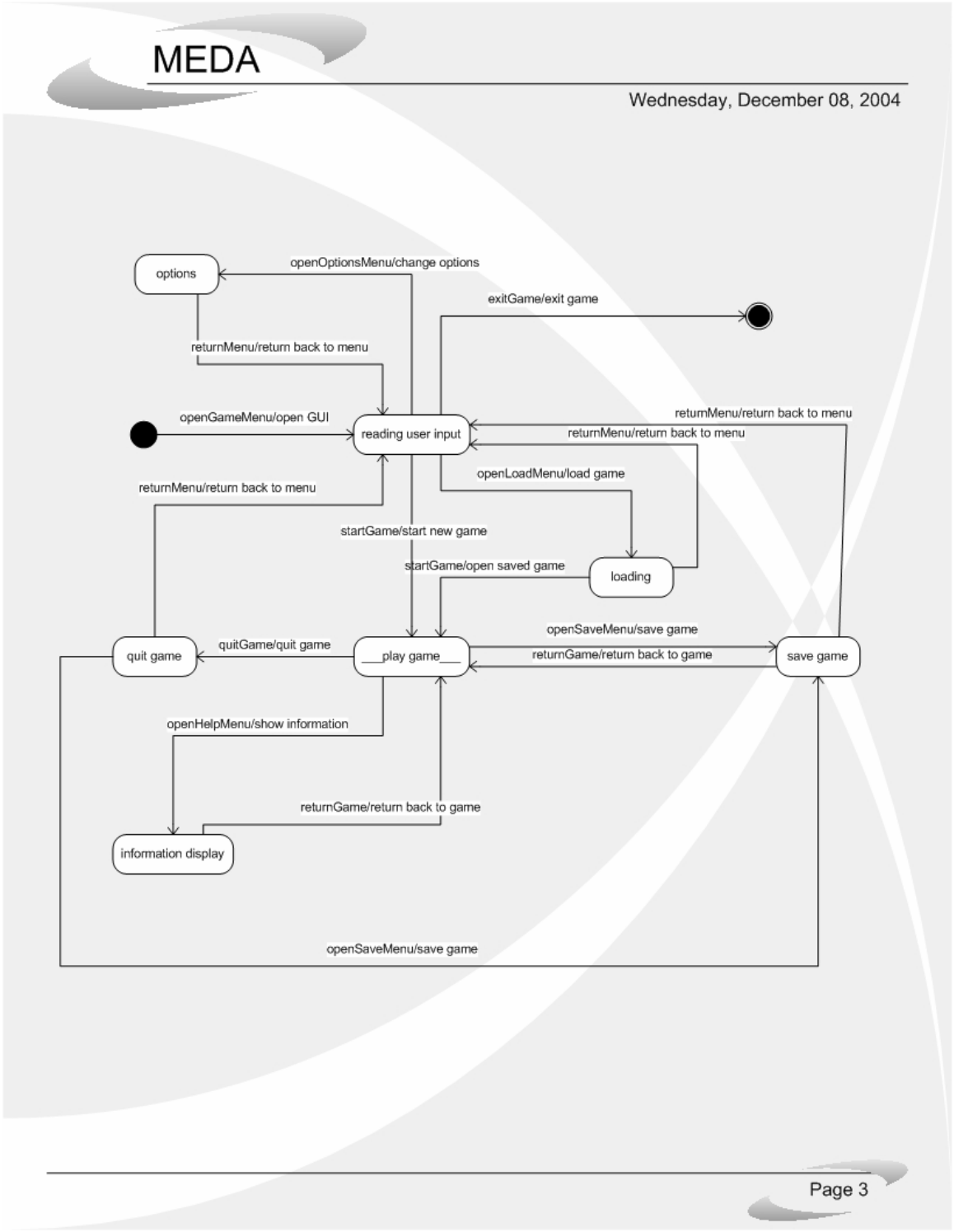
Flow of Events for the Credits Use-case	
Objective	Getting information about the credits of the game.
Precondition	-
Main Flow	1. The user interacts with keyboard and mouse to signal the request to the system 2. The system displays credits of the game.
Alternative Flows	-
Post-condition	Credits of the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

10. STATE TRANSITION DIAGRAM

10.1.DIAGRAM



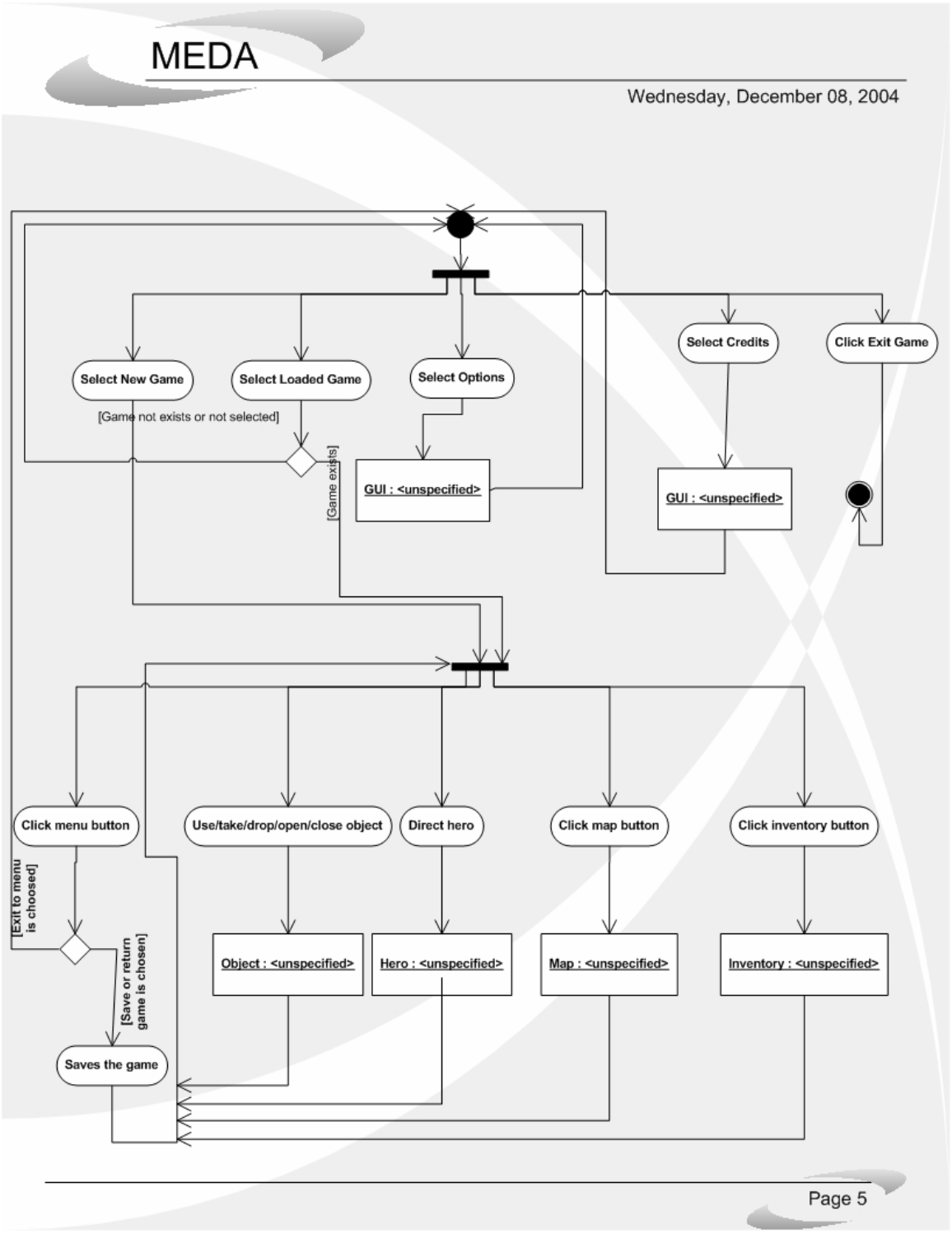
10.2. EXPLANATION

- 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.

11. ACTIVITY DIAGRAM

11.1 DIAGRAM



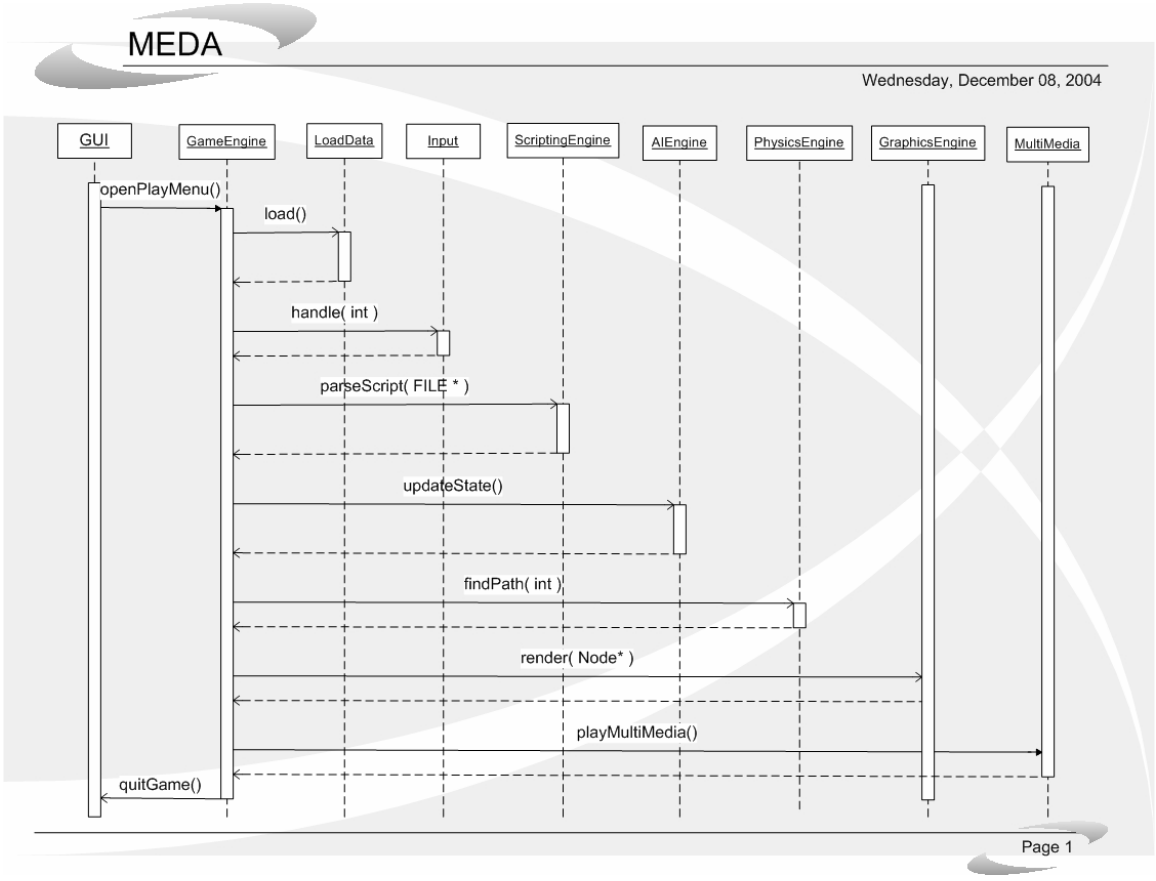
11.2. EXPLANATION

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.

12. SEQUENCE DIAGRAM

12.1 DIAGRAM



12.2 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.

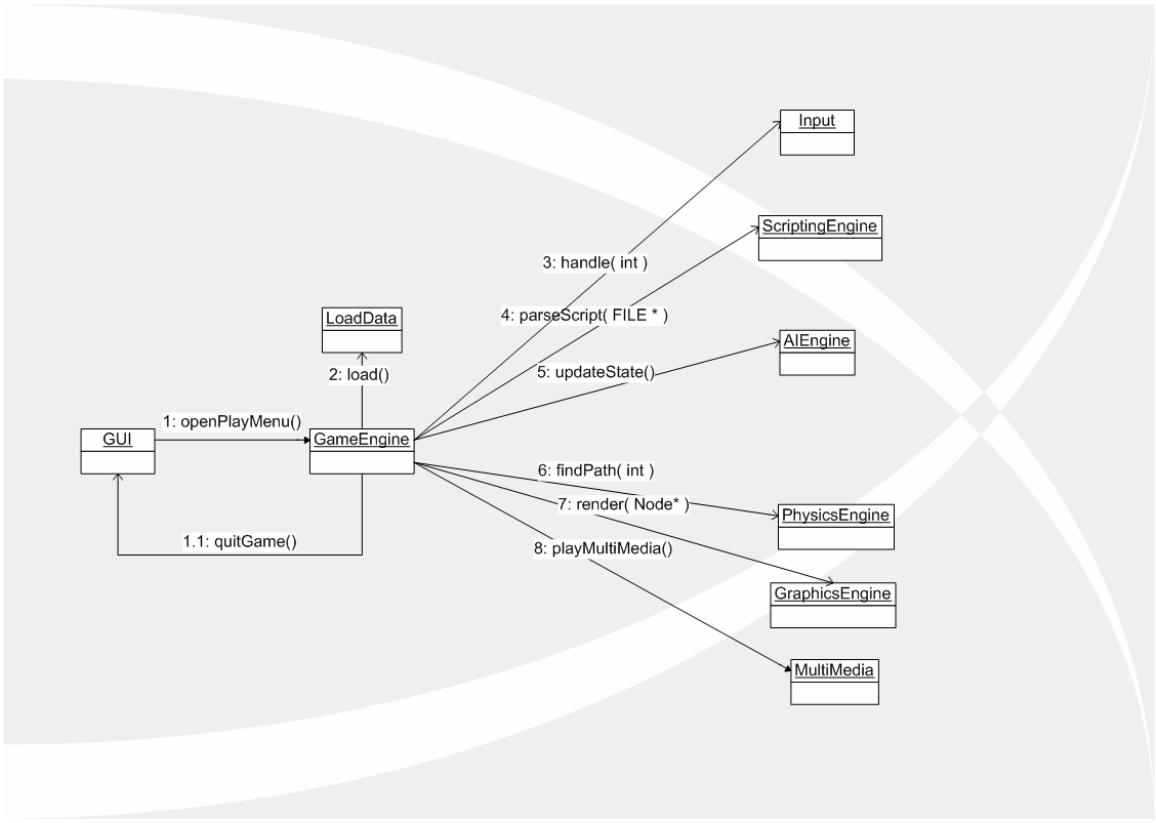
- After that, another important phase comes. It is rendering phase and implemented by the GraphicsEngine. GameEngine associates with the GraphicsEngine class by the help of render event. GraphicsEngine class renders the graphics and gives the control to the GameEngine class again.

- 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.

13. COLLABORATION DIAGRAM

13.1. DIAGRAM



13.2. 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();

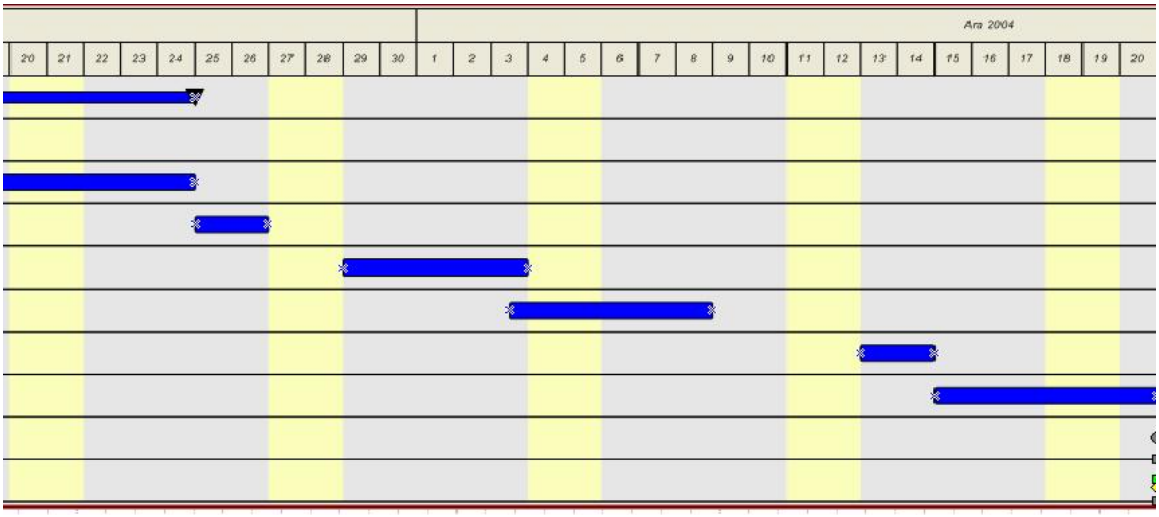
GUI with quitGame();

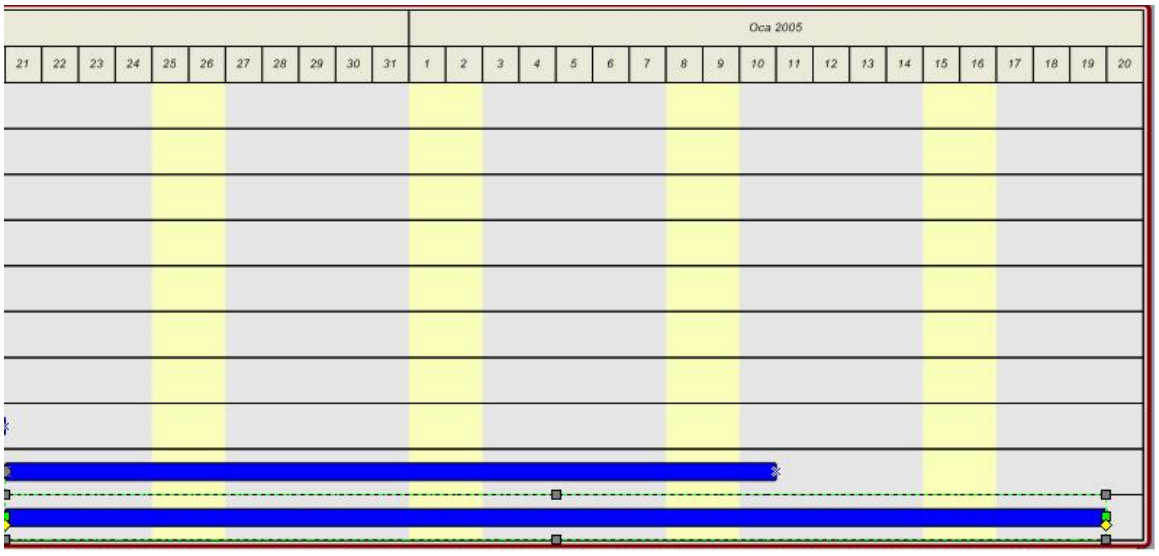
GUI class:

GameEngine with openPlayMenu();

14. Gantt Chart

ID	Task Name	Start	Finish	Duration
1	**Design the modules**	**08.11.2004**	**24.11.2004**	**2,6w**
2	Prepare class definitions	08.11.2004	17.11.2004	1,6w
3	Prepare the attributes and methods of the classes	18.11.2004	24.11.2004	1w
4	Design User Interface	25.11.2004	26.11.2004	,4w
5	Draw UML Diagrams	29.11.2004	03.12.2004	1w
6	Prepare Initial Design Report	03.12.2004	08.12.2004	,7w
7	Prepare Porject Presentation	13.12.2004	14.12.2004	,4w
8	Make corrections on the Initial Design Report	15.12.2004	20.12.2004	,8w
9	Prepare Final Design Report	21.12.2004	10.01.2005	3w
10	Develop Prototype	21.12.2004	19.01.2005	4,4w

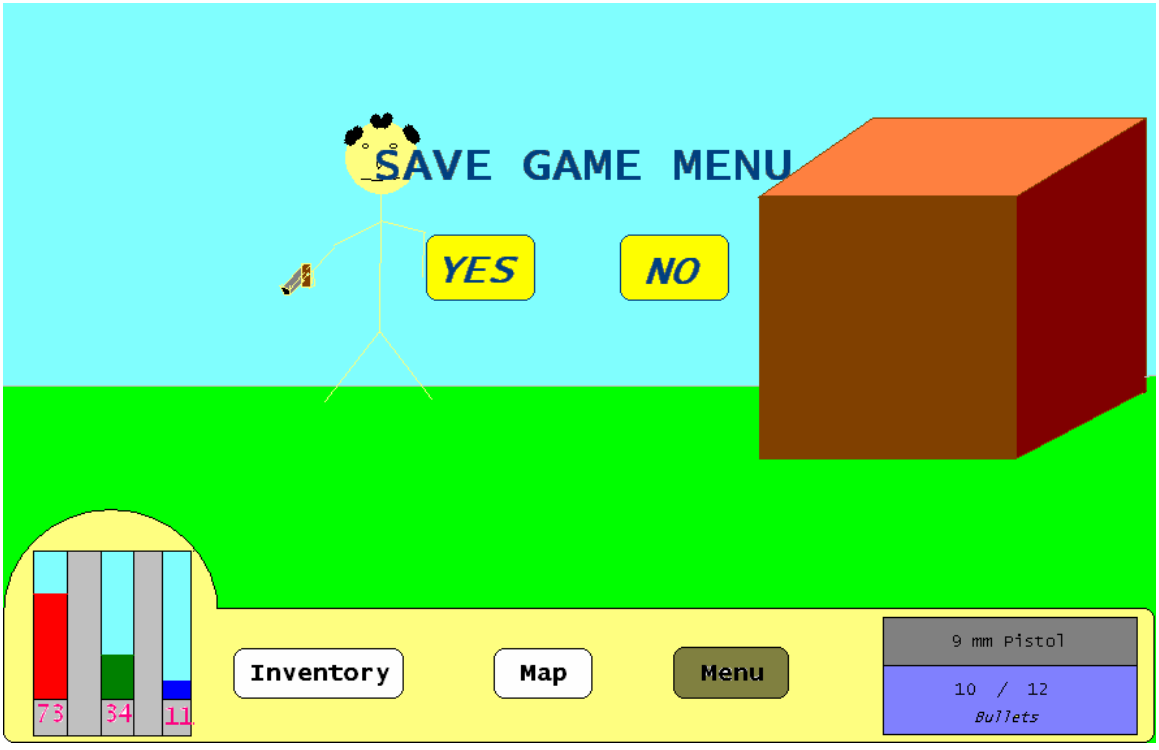




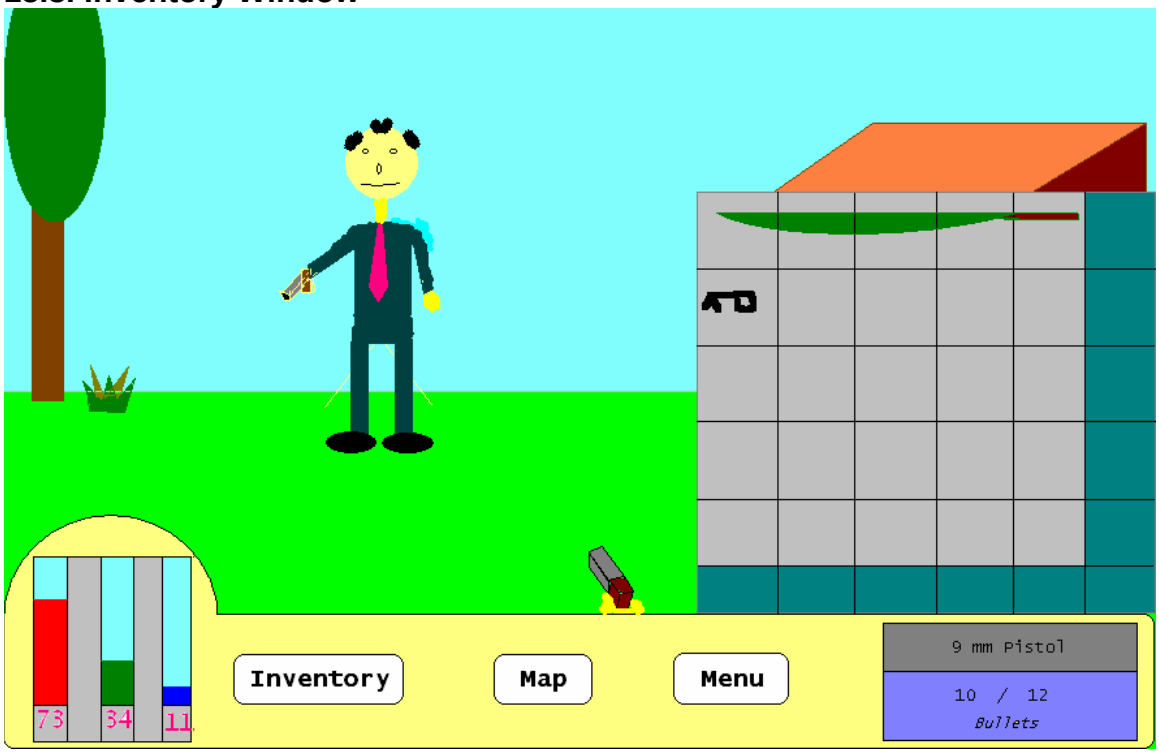
15. Appendix

15.1. Game Menu

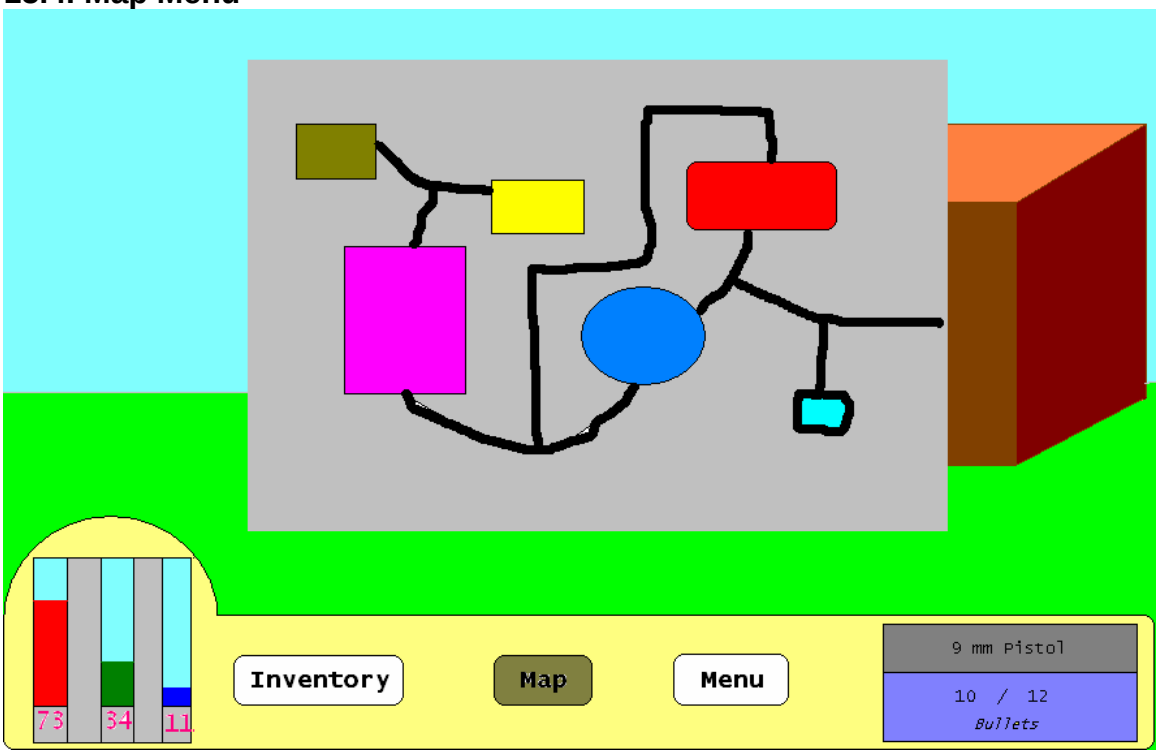
15.2.SaveMenu



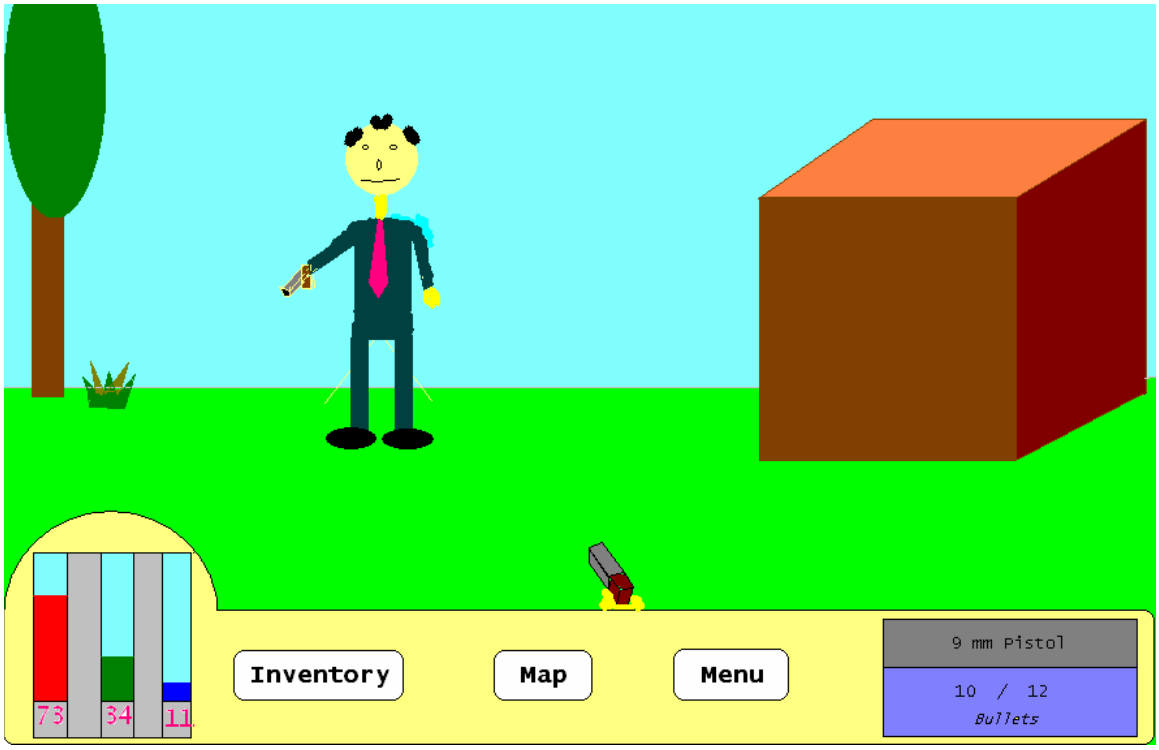
15.3. Inventory Window



15.4. Map Menu



15.5. First Person View



15.6. Third Person View