Instituto Superior de Engenharia do Porto

Nostalgia Studio

A game engine for creating collaborative multiplayer online graphic adventure games

David da Luz Gouveia

Dissertação para obtenção do Grau de Mestre em Engenharia Informática, Área de Especialização em Sistemas Gráficos e Multimédia

Orientador: Doutor Carlos Miguel Miranda Vaz de Carvalho

Porto, Outubro 2012

Resumo

No decorrer do projeto *SELEAG* foi desenvolvido um jogo de aventura gráfica educativo com o propósito de ensinar história, cultura e relações sociais aos alunos. Este jogo foi avaliado em contexto de sala de aula em diversos países, obtendo resultados positivos. No entanto, por motivos técnicos, alguns dos objetivos propostos pelo projeto não puderam ser devidamente explorados, como permitir que o jogo fosse extensível por outros educadores ou suportar a colaboração online entre os jogadores.

Nomeadamente, as ferramentas utilizadas para desenvolver o jogo eram demasiado complicadas para serem utilizadas fora da equipa de desenvolvimento, o que limitou a extensibilidade do projeto, e tornou impossível que educadores sem conhecimentos de programação fossem também capazes de traduzir os seus conteúdos educativos para este formato.

Além disso, apesar do jogo possuir algumas funcionalidades de colaboração online, toda a interação era efetuada externamente ao jogo, através de um fórum de mensagens, o que demonstrou ser pouco motivante para os jogadores, pois muitos deles nem se aperceberam que havia uma componente de colaboração no jogo.

O objetivo desta tese incide sobre estes dois problemas, e consistiu em desenvolver um editor e motor de jogo com uma interface simples de utilizar, que não necessita de conhecimentos prévios de programação, e que permite criar jogos de aventura gráfica com uma componente de colaboração online verdadeiramente embebida na jogabilidade.

A aplicação desenvolvida foi testada por um conjunto de utilizadores de diversas áreas, tendo-se obtido resultados que demonstram a acessibilidade e simplicidade da mesma, independentemente do nível de experiência prévio de programação do utilizador. A componente de colaboração online foi também muito bem recebida pelos utilizadores, os quais demonstraram bastante interesse em ver jogos de aventura gráfica com componente de colaboração online serem desenvolvidos no futuro.

Palavras-chave: Editor, Motor, Jogo, Aventura, Gráfica, Point-and-Click, Colaboração, Online, Multiplayer

Abstract

Project SELEAG was responsible for the creation of an educational graphic adventure game with the purpose of teaching history, culture and social relations to students. However, some of the goals that were proposed for the project could not be properly explored, such as allowing the game to be extended and supporting online collaboration between players.

In particular, the tools which were used to develop the game were too complicated to be used by the general public, which limited the extensibility of the project. Also, although the game possessed some online collaboration features, all of the interaction was handled outside of the game through a message forum, and players never saw each other inside the game.

The goal of this thesis was to develop an editor and game engine which allowed the creation of graphic adventure games with a true online collaboration component, using a simple interface that did not require previous programming knowledge.

This application was tested by a group of users from different backgrounds, and the results proved that it was accessible regardless of the their previous programming experience. The online collaboration was also well received, and the users demonstrated a lot of interest in seeing graphic adventure games with online collaboration being developed in the future.

Keywords: Editor, Engine, Game, Adventure, Graphic, Point-and-Click, Collaboration, Online, Multiplayer

Acknowledgments

I would like to express my gratitude to Dr. Carlos Vaz de Carvalho for his guidance, supervision and mentorship throughout the evolution of my thesis. I would also like to thank him for the invitation to participate in project SELEAG which eventually led to the idea behind this thesis.

I would like to thank my parents Miguel and Fátima Gouveia for all of their support not only during the completion of my thesis and my academic life, but in everything else too. I would also like to thank my girlfriend for the patience she's shown while I was busy working on my thesis.

I would like to thank my friend Helena Barbosa who helped me tremendously by creating all of the illustrations and backgrounds for the demo game that was sent to the testers. She did a great job and I wish her a lot of success in her career!

I would like to thank Duarte Lopes who was my colleague when we worked on project SELEAG, and ended up doing most of the scripting for Time Mesh by himself, while I was busy getting started with the engine for Nostalgia Studio.

I would like to thank Jan Nedoma and Thomas Dibke who created the Wintermute Engine and Visionaire Studio respectively and provided me with some of background history on how and when each of these engines were created.

Finally I would like to thank everyone who helped test the application and replied to the survey, including everyone at the Serious Games Network, and Trevor Sundberg from Super Game Programmer 3 - good luck with your game!

Table of Contents

Index of Figures

Index of Tables

List of Acronyms

SELEAG	Serious Learning Games
XML	Extensible Markup Language
ZIL	Zork Implementation Language
AGI	Adventure Game Interpreter
SCUMM	Script Creation Utility for Maniac Mansion
AGS	Adventure Game Studio
WME	Wintermute Engine
DLL	Dynamic Link Library
GUI	Graphical User Interface

1 Introduction

Nostalgia Studio is an extension of project *SELEAG (Serious Learning Games)* [SELEAG, 2011] whose goal was to evaluate the use of serious games for learning history and social relations with an extensible multiplayer online game based on collaboration and social interaction. The game that was developed was a graphic adventure game known as *Time Mesh* [TimeMesh, 2011] which managed to achieve very positive results, but due to technical constraints, some of the initial goals could not be explored in depth.

In particular, the multiplayer online features of *Time Mesh* were mostly limited to the website that hosted the game, instead of being integrated into the gameplay itself. The main reason for this limitation is that although multiplayer online play mechanics have already been used in almost every video game genre, that is not the case with graphic adventure games which have always remained single player games to this day. Therefore none of the existing game engine solutions supported this behavior.

Another shortcoming of project *SELEAG* was its extensibility, because despite the desire to create a platform where new content could be designed and added independently by educators, the technology used to develop *Time Mesh* was too complex to fulfill that goal, since it required some experience with software development tools and concepts, such as *Adobe Flash* and *XML*.

This work is based around the two limitations described above, and aims to develop a graphic adventure game engine known as *Nostalgia Studio* with the following goals in mind:

- **Simplicity** - Explore to which degree a specialized graphic adventure game engine could allow people with little or no software development experience to create their own graphic adventure games.

- **Multiplayer online** - Explore the possibilities of multiplayer online play as a mean of collaboration in graphic adventure games, a genre which has always been focused on single player gameplay.

1.1 Project SELEAG

Project *SELEAG* was an European project created to evaluate the use of serious games for learning history, culture and social relations, through the implementation of an extensible, online, multi-language, multi-player, collaborative and social game platform. The project counted with the collaboration of seven different partners and resulted in the creation of a point-and-click adventure game called Time Mesh.

SELEAG
SERIOUS LEARNING GAMES

Figure 1 - Serious Learning Games

Time Mesh tells the story of a child who stumbles upon a time machine, and uses it to travel back in time in order to correct certain events in history that have occurred differently from what we know. Most of the game is based on actual historical facts, and covers three different time periods: the Age of Discovery, the Industrial Revolution, and the Second World War. It would have been interesting to allow more scenarios besides these three to be added to the game, but with the technology in which the game was created, this was not possible without significant resources and training.

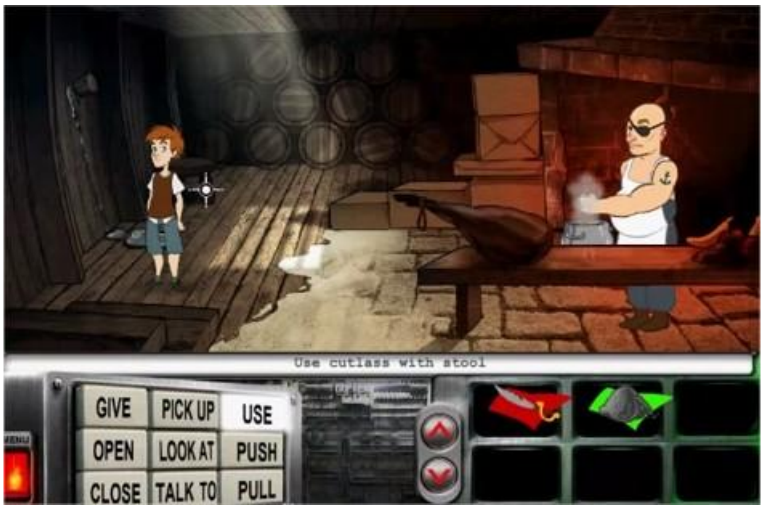
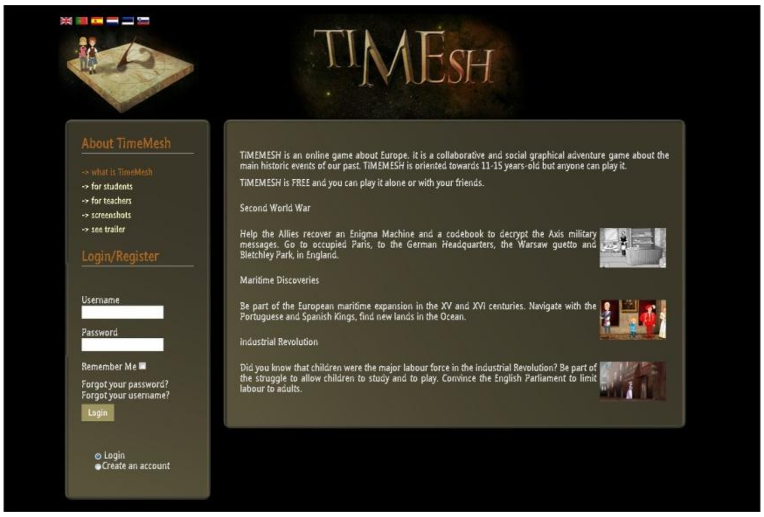


Figure 2 - The *Time Mesh* game

The game was created using an *Adobe Flash* based game engine which allowed it be played directly from within most web browsers. The game was integrated into a webpage that required users to register and create teams in order to collaborate with each other. Collaboration with other teams was necessary because there were two versions of each game scenario, and knowledge from one version was required to advance in the other. However, the game itself had no knowledge about the existence of the other team, and all interactions were handled outside the game, on the web platform.



TIMEMESH is an online game about Europe. it is a collaborative and social graphical adventure game about the main historic events of our past. TIMEMESH is oriented towards 11-15 years-old but anyone can play it.

TIMEMESH is FREE and you can play it alone or with your friends.

Second World War

Help the Allies recover an Enigma Machine and a codebook to decrypt the Axis military messages. Go to occupied Paris, to the German Headquarters, the Warsaw guetto and Bletchley Park, in England.

Maritime Discoveries

Be part of the European maritime expansion in the XV and XVI centuries. Navigate with the Portuguese and Spanish Kings, find new lands in the Ocean.

Industrial Revolution

Did you know that children were the major labour force in the industrial Revolution? Be part of the struggle to allow children to study and to play. Convince the English Parliament to limit labour to adults.

Figure 3 - The *Time Mesh* web platform

Although *Time Mesh* was originally meant to be played autonomously by students in their free times, the game also ended up being tested and used in a classroom context, and has already been played by over 1.000 different users across several European countries, often as part of school activities. In general, results have been positive in terms of children enjoyment.

1.2 Motivation

Besides giving continuation to a research that was initiated with project *SELEAG*, there is also a strong personal motivation behind the creation of this thesis, because it aims to contribute to a video game genre that has been one of my favorites since the times of my childhood.

My first contact with graphic adventure games happened around 1994 with the game *Monkey Island 2* by *LucasArts.* Unlike most of the games that I had played before which usually had very simple goals, such as reaching the end of the level, or beating your opponent in a fight, graphic adventure games were all about the story, the characters, the environment and the puzzles. This concept was extremely new to me, but I was hooked from the first minute, and ended up playing through almost every graphic adventure game that I could get my hands on.

Unfortunately, that period was the golden era of graphic adventure games, and problems within the industry have led the genre through a long period of stagnation, until very recently. It is therefore highly rewarding for me to work on this project, since it aims to contribute to one of my favorite game genres of all time.

1.3 Structure

1. **Introduction** -This chapter introduces the context, goals and motivation behind the implementation of this project.

2. **The Evolution of Adventure Games** - This chapter provides a detailed description of what distinguishes adventure games from other game genres. Then it proceeds to describe the origin and evolution of the genre, and how the subgenre of graphic adventure games was born from that evolution. Knowing this evolution is important because it allows us to understand how the core gameplay mechanics and user interactions have changed, as well as what is expected from a modern graphic adventure game.

3. **Graphic Adventure Game Engines** - This chapter introduces the concept of the game engine as a tool that facilitates the process of developing games, and provides an overview of the current state-of-the-art when it comes to graphic adventure game engines, with three popular graphic adventure game engines being presented and compared.

4. **Analysis and Requirements** - This chapter describes the first steps taken during the implementation of the engine. During this process, a set of guidelines and requirements for Nostalgia Studio were deduced from a extremely detailed comparison of the existing graphic adventure game engines. Not only does it describe which features are essential to the engine, but also how each of them should be implemented in order to make the user experience as simple as possible.

5. **Architecture and Design** - This chapter describes the organization of Nostalgia Studio, starting from a high-level view of its architecture, followed by a more detailed description each of its components. Afterwards, it describes in detail how multiplayer online play was integrated into the gameplay, and how it can be varied to provide different modes of gameplay.

6. **Implementation** - This chapter describe how the most significant components of Nostalgia Studio were implemented from a low-level and technical point of view. This includes information on the techniques, algorithms, data structures and design patterns which were employed to solve individual challenges encountered during the implementation. The chapter ends with an analysis of all results received from a study that was conducted at the end of the implementation.

7. **Conclusion** - This chapter concludes the work by looking back into what went right and wrong during the development of the project, and summarizing all of the work that was done and the results that were obtained.

2 The Evolution of Adventure Games

Video games can be categorized under a multitude of genres, such as action, adventure, role-playing, simulation, or strategy, depending on their gameplay and interaction styles. This work focuses entirely on adventure games, so it makes sense to start by providing some insight into that genre.

There are some elements that can be found in the majority of adventure games [Adams, 2006]:

- They are usually built around a *narrative*...
- ...in which the player participates by *assuming the role of a very small subset of its characters.*
- They are usually driven by *puzzle solving* rather than by a physical effort...
- ...and rely mostly on *exploration*...
- ...*object manipulation...*
- *...or dialogue.*

Some of these elements make adventure games significantly different from other genres:

- *Action* - Instead of relying on the player's reflexes like most action games, adventure games rely on reasoning and logical thinking, which results in a more relaxed experience.
- *Strategy* - Although strategy games also rely on logical thinking, they usually require the player to control a large amount of units, such as an army or entire nation, while adventure games focus on the main character(s) in the narrative.

- *Role-playing* - While both genres have a lot in common, adventure games are deterministic in nature and follow a fixed path, while role-playing games have a statistical background which introduces randomness and variation. They also lack the management components usually present in role-playing games (e.g. party management, skill management) [Barton, 2008].

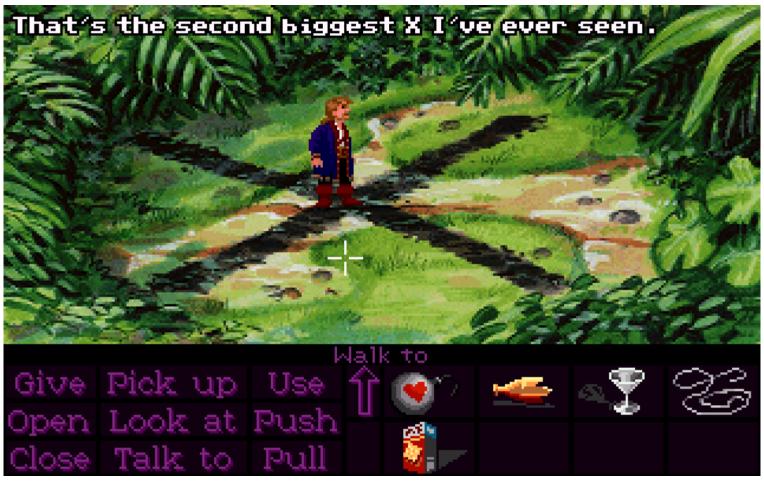


Figure 4 - *Monkey Island 2*, one of the most popular adventure games of all time.

Since adventure games place a lot of emphasis on story, it is common for them to draw inspiration from other narrative-based media - such as literature and film - and to encompass a wide variety of themes, from comedy to horror, or fantasy to science-fiction.

On the subject of action games, there are some adventure games that incorporate action sequences into their gameplay, but they are always treated independently (e.g. as mini-games) and remain secondary to the narrative and puzzles. On the other hand, if a game has some adventure elements but the gameplay is all action based, it is usually categorized under the sub-genre of action games known as the action-adventure, which will not be covered here.

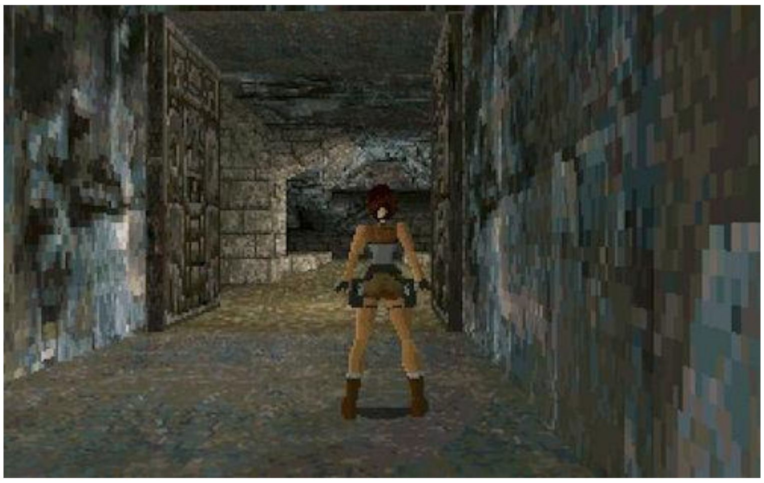


Figure 5 - *Tomb Raider I*, a classic example of an action-adventure game.

Based on the traits described so far, it is also easy to understand why there has not been much effort put into creating adventure games with a multiplayer online component. Most multiplayer online games rely on the game's progression being flexible, and allowing a large number of players to interact simultaneously. In an adventure game, where the entire story progression has been fixed from the start, and the number of controllable characters in the plot is limited, multiplayer game design becomes a lot more challenging. But on the other hand, puzzle solving and exploration are both elements that would benefit from cooperation in a multiplayer environment, at least on a limited scale. Later chapters of this work will address this particular problem.

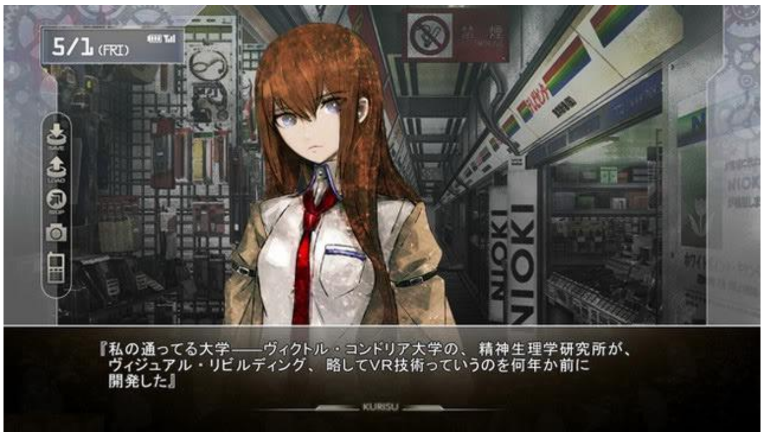
Like almost every other genre in existence, the adventure game genre has a few sub-genres with distinctive features. The most common distinction is made between text adventure games and graphic adventure games. Text adventure games do not have any graphics, so they rely entirely on text to present the narrative and to receive player input. Text adventure games are also frequently known by the synonym, interactive fiction.

Graphic adventure games, on the other hand, provide a visual representation of the world, and can employ a few different control schemes for player interaction. The most common control scheme for a graphic adventure game is the use of a pointing device (e.g. the mouse) or a touch screen interface, and games using that approach are known as point-and-click adventure games.

The ratio between narrative and puzzles can vary widely from game to game, resulting in very distinct experiences. Some adventure games place most of their emphasis on puzzles, while demoting the narrative to a secondary role. These are sometimes known as puzzle adventure games, and a popular example of this type of game is the 1993 best-seller, *Myst*.

Figure 6 – *Gobliiins,* a puzzle adventure game with a very simple plot.

On the other hand, there are also some adventure games that focus almost entirely on narrative, sometimes at the cost of reducing interaction and gameplay to the bare minimum. A common example of this approach is the Japanese visual novel genre, which is a form of interactive fiction with anime style graphics, and minimalistic gameplay that consists mostly of a few branching choices in the game. But unlike most adventure games, visual novels usually have multiple story paths and multiple endings, and player choices have a strong effect in the outcome of the game.



『私の通ってる大学——ヴィクトル・コンドリア大学の、精神生理学研究所が、
ヴィジュアル・リビルディング、略してVR技術っていうのを何年か前に
開発した』

Figure 7 – *Steins;Gate*, a Japanese visual novel that recently received an anime adaption.

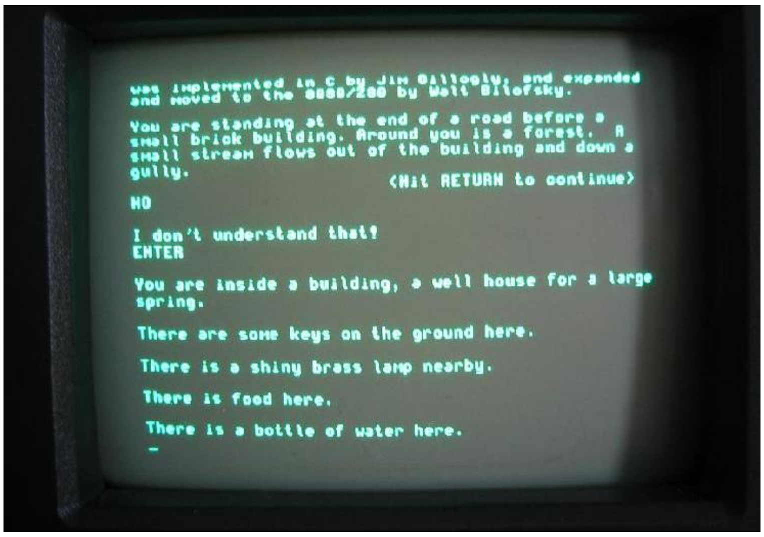
2.1 The Text Adventure

2.1.1 Colossal Cave Adventure

The first adventure game dates back to the mid 1970s when Will Crowther, a programmer at BBN (Bolt, Beranek and Newman) and a caving enthusiast, created a computer game based on his experiences at the Mammoth Cave system in Kentucky. He named that game *Colossal Cave Adventure* but it was often abbreviated to *Adventure*, a name that would later be used to define the entire genre [Jerz, 2007]. The game was built around the exploration of a massive underground realm modeled after the real world, but it also incorporated fantasy elements such as magic and creatures, influenced by the popular non-computer role-playing game *Dungeons & Dragons*.

Wanting to create a game that would be accessible to everyone, Crowther used a natural language interface which gave the illusion that the player was talking with the computer. The game relied on narrative to describe the environment, and let the player interact with it using simple verb-noun instructions such as "get key" or "open door".

Games like this are designated *text adventures*, or *interactive fiction*, because of their similarity to traditional literature with the added interactivity of computer games. It did not take long for *Adventure* to achieve considerable following and inspire others to create their own adventure games, especially after Don Woods, a graduate student at Stanford University, extended the original game and released it on the ARPANET in 1977 [Woods, 2001].



```
was implemented in C by Jim Gillooly, and expanded
and moved to the 8080/Z80 by Walt Bilofsky.

You are standing at the end of a road before a
small brick building. Around you is a forest.  A
small stream flows out of the building and down a
gully.
                              <Hit RETURN to continue>
HO

I don't understand that!
ENTER

You are inside a building, a well house for a large
spring.

There are some keys on the ground here.

There is a shiny brass lamp nearby.

There is food here.

There is a bottle of water here.
_
```

Figure 8 - *Colossal Cave Adventure*, the game that started the adventure genre.

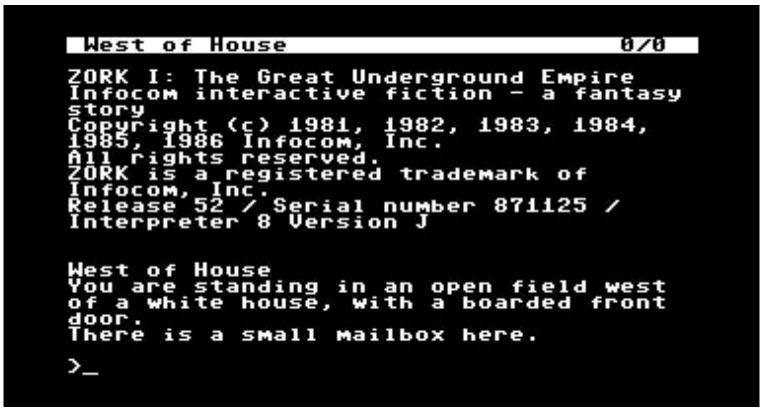
2.1.2 Zork

Once *Adventure* reached the MIT (Massachusetts Institute of Technology), it caught the attention of a group of students: Tim Anderson, Marc Blank, Bruce Daniels, and Dave Lebling (some of which would later found *InfoCom*). After enjoying the game for the first couple of weeks, their thoughts soon turned towards improving it. They identified two major shortcomings with the game: the rudimentary parser system and the lack of extensibility.

With these in mind, the group set forth to develop a brand new game, and by June 1977, the first version of *Zork* had been created [Anderson, 1985]. *Zork* took place in a fantasy world, and managed to excel not only from a technical point of view, but also from the quality of its storytelling. It kept being extended until its commercial release by *InfoCom* in 1980. It had an interpreter capable of handling complex sentences such as "kill troll with sword" or "put crown, scepter and gold into case", which was a huge improvement over its predecessor. It was also significantly easier to add new content to the game.

The first version of the game only ran on the *PDP-10* mainframe computer, which very few people had access to. With personal computers (such as the *Apple II*) gaining popularity, *InfoCom* decided to take advantage of the situation. But compared with the *PDP-10*, personal computers had limited resources, and the multitude of platforms presented a portability challenge. To deal with the memory constraints of personal computers, they split the original game in three: *Zork I: The Great Underground Empire*, *Zork II: The Wizard of Frobozz* and *Zork III: The Dungeon Master*.

They were released one year apart and had several improvements made to their content, causing the overall storyline to be more coherent. To address the portability problem, they developed a virtual platform known as the Z-Machine which was programmed in a specialized scripting language named the Zork Implementation Language (ZIL). All of the game content was specified in ZIL, and porting the game to a different platform only required the creation of a Z-Machine Interpreter for it. Famous graphic adventure developers such as Sierra On-Line and LucasArts would later adopt this concept and develop their own scripting languages, AGI and SCUMM respectively.



```
 West of House                          0/0
ZORK I: The Great Underground Empire
Infocom interactive fiction - a fantasy
story
Copyright (c) 1981, 1982, 1983, 1984,
1985, 1986 Infocom, Inc.
All rights reserved.
ZORK is a registered trademark of
Infocom, Inc.
Release 52 / Serial number 871125 /
Interpreter 8 Version J

West of House
You are standing in an open field west
of a white house, with a boarded front
door.
There is a small mailbox here.

>_
```

Figure 9 - *Zork I: The Great Underground Empire*

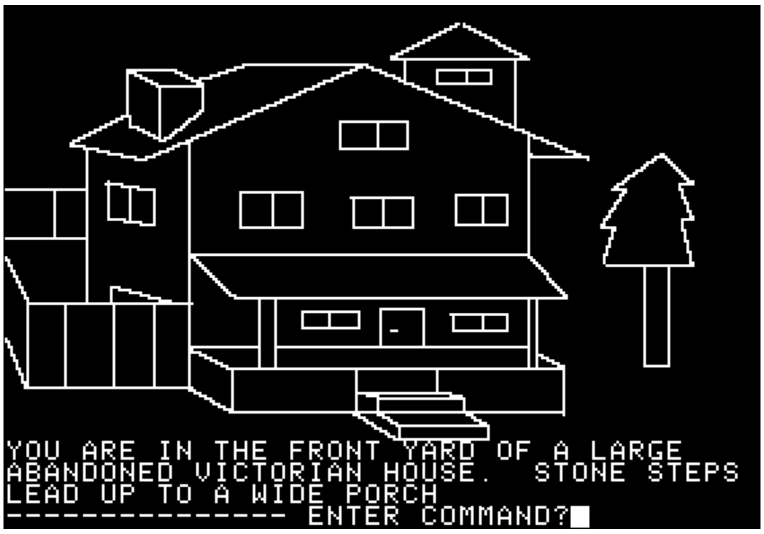
2.2 The Graphic Adventure

2.2.1 Mystery House

Ken Williams and his wife Roberta Williams were also inspired by *Adventure* to create their own games. However, their vision was different from the MIT group, and instead of focusing on text, they decided to take advantage of recent advances in computer hardware and brought graphics into the genre. Their first game was called *Mystery House* and it was released in 1980. The game told an Agatha Christie-like mystery story where the player had to search for treasures in a house while trying to avoid a less fortunate fate. Roberta designed the entire game on her own, and Ken implemented it on an *Apple II* microcomputer.

Mystery House had a very simple parser, but unlike *Adventure*, each room was accompanied by a monochromatic picture. This was ground-breaking given the hardware limitations of the time. For instance, memory was too limited to store a bitmap for each room, so images were stored as sets of line drawing instructions instead. When the time came to present the picture, the system executed those instructions in turn.

The game was a commercial success, and so the couple decided to dedicate themselves to game development. Together they founded what would become *Sierra On-Line*, a company responsible for some of the most famous graphic adventures of all time [Loguidice, 2009].

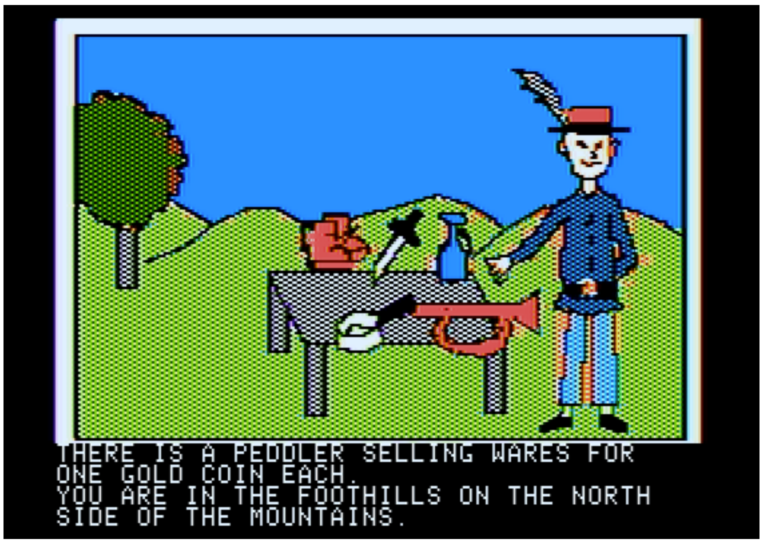


```
YOU ARE IN THE FRONT YARD OF A LARGE
ABANDONED VICTORIAN HOUSE.  STONE STEPS
LEAD UP TO A WIDE PORCH
---------------- ENTER COMMAND?
```

Figure 10 - *Mystery House*, the first graphic adventure game.

2.2.2 Wizard and the Princess

Still in 1980 they released *Wizard and the Princess*, a game inspired by fairy tales and legends that Roberta used to love as a child [ACG, 1999]. Unlike its predecessor, *Wizard and the Princess* featured colored graphics, making full use of the *Apple II*'s graphic capabilities. The system allowed a total of 6 colors to be displayed simultaneously, but through clever use of dithering it was possible to give the illusion that more colors were present. Once again the game was a success, and also served also to set the theme and tone for a game yet to come.



```
THERE IS A PEDDLER SELLING WARES FOR
ONE GOLD COIN EACH.
YOU ARE IN THE FOOTHILLS ON THE NORTH
SIDE OF THE MOUNTAINS.
```

Figure 11 - *Wizard and the Princess*, a prequel to *King's Quest.*

2.2.3 King's Quest

Throughout the following years, Sierra On-Line continued releasing game after game and reinforcing their status as the leading adventure game developers in the market. But their biggest contribution to the graphic adventure genre happened in 1984 when IBM commissioned them to develop a demonstration product for the PCjr [TheDotEaters, 1998]. That product was a graphic adventure game called *King's Quest* which tells the story of Sir Graham, a knight on a quest to retrieve three stolen artifacts and become the next king of the realm.

The game took advantage of the double buffering capabilities of the PCjr and introduced animated characters. Furthermore, for the first time in a graphic adventure, the main character could also be seen on screen and controlled using the arrow keys, although all other actions were still handled through the text parser. It was also on this project that Sierra developed their scripting system called the Adventure Game Interpreter (AGI). AGI simplified the process of porting the game to other platforms, which turned out to be critical to the success of the game.

The PCjr did not sell well because of its experimental keyboard design and a price well above its competition. As a consequence, *King's Quest* was not immediately successful. It was only when the game was re-released for the Tandy 1000 one year later that it became a hit. Afterwards, its popularity grew exponentially, and the game ended up being released on a multitude of platforms such as the Apple II, Amiga, and even the Sega Master System.

The game was not without flaws however, most notably due to its high difficulty. Many puzzles suffered from a very convoluted logic, and the limited text parser introduced many word guessing challenges because you did not know the names of the objects that were on the screen. The character was also not easy to control, and a misstep within the world would frequently mean death. Nonetheless, none of these issues were enough to prevent the game from achieving a massive fan base.

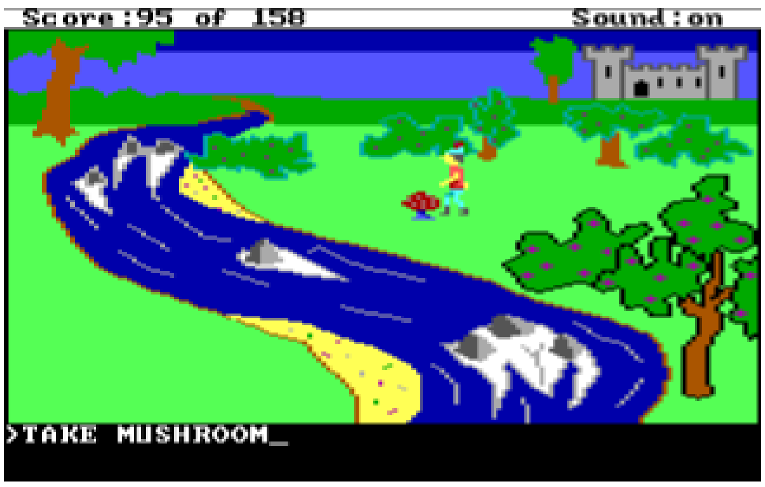


Figure 12 - King's Quest I

Following the success of the first game, Sierra spent the following 10 years devoting themselves to the creation of graphic adventure games. They took the *King's Quest* series up to its eighth installment, and also started other series such as *Space Quest*, *Leisure Suit Larry* and *Gabriel Knight*.

2.3 The Point-and-Click Interface

2.3.1 Apple Macintosh

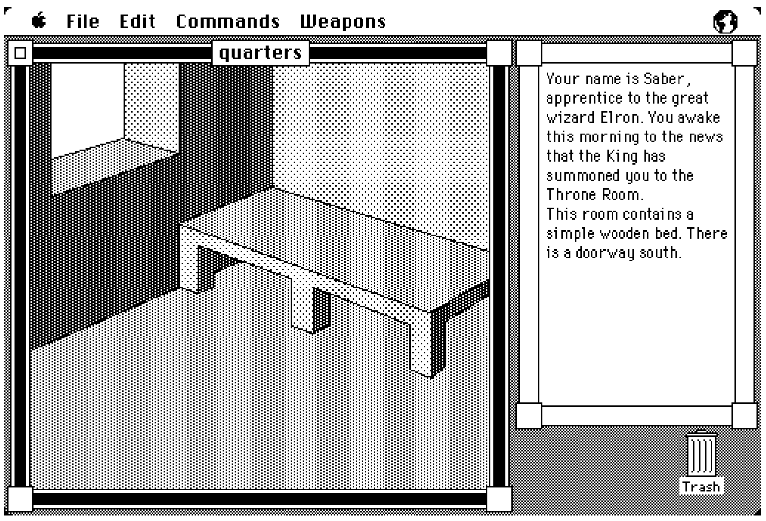
Although the adventure genre had evolved significantly with the introduction of computer graphics and animation, interaction was still handled through the rudimentary parser interface. One disadvantage of the parser interface was that it required players to guess what words to *write*, rather than focusing on what they had to *do*. While some companies poured more effort into their parsers and managed to minimize the problem, others did not, resulting in games that were extremely hard to complete without a user guide. The problem was aggravated with early graphic adventures, where low fidelity graphics created situations where the player would have to look at two pixels on the screen and guess whether it was a rock, an egg, a marble, or part of the scene's backdrop.

Fortunately, a new technology was about to hit the market which would change the universe of computer games forever. In 1984, Apple introduced the world to its new line of personal computers, the Macintosh. The Macintosh was the first commercially successful personal computer based around a graphical user interface rather than a command-line interface. It also came equipped with a pointing device, whose introduction presented many opportunities for graphic adventures - by moving the cursor around the scene, the player was now able to interact with objects directly rather than having to address them by name.

Figure 13 - The Apple Macintosh

2.3.2 Enchanted Scepters

The introduction of this pointing device was revolutionary, and developers were quick to realize the possibilities, with *Enchanted Scepters* by Silicon Beach Software being released that same year (1984) for the Macintosh. While *Enchanted Scepters* did not look too different from early graphic adventures, objects were selected using the mouse, and commands were chosen from a drop down menu. No longer were players required to memorize lists of commands or try to figure out the correct name for an object.



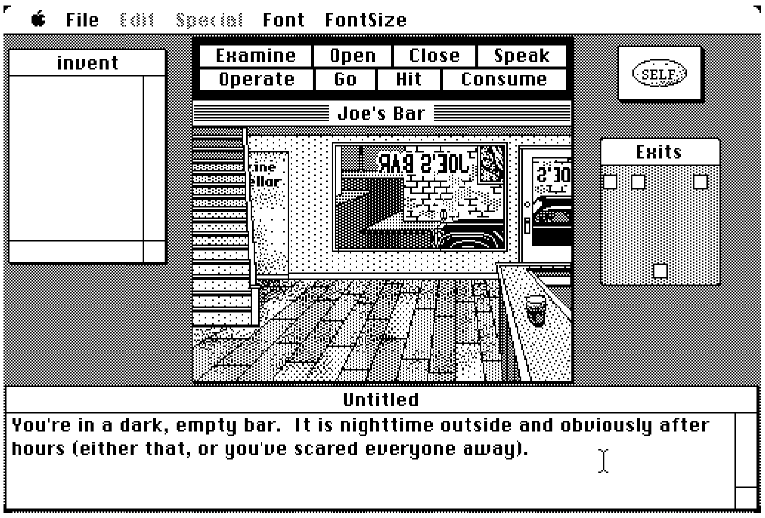
Your name is Saber, apprentice to the great wizard Elron. You awake this morning to the news that the King has summoned you to the Throne Room.
This room contains a simple wooden bed. There is a doorway south.

Figure 14 - Enchanted Scepters

2.3.3 Déjà Vu

That was the birth of the *point-and-click adventure*, a sub-genre of graphic adventures which is still regarded as one of the most popular types of adventure games. And if *Enchanted Scepters* took the first steps in the genre, it was *Déjà Vu: a Nightmare Comes True* (released in 1985 by ICOM Simulations) that really helped establish it [Moss, 2011].

Besides the features introduced by *Enchanted Scepters*, *Déjà Vu* added a command palette that simplified the task of selecting commands, and an inventory window to help the player keep track of his possessions. Fans of the genre will probably recognize these features from most popular adventure games that have been released since then. Sierra also realized the advantages of the point-and-click interface, and started using it in later iterations of their *King's Quest* series.



You're in a dark, empty bar. It is nighttime outside and obviously after hours (either that, or you've scared everyone away).

Figure 15 - Déjà Vu

2.3.4 Labyrinth

Another company that left a deep mark on the history of adventure games was Lucasfilm Games (now LucasArts). In 1986, they released a graphic adventure game called *Labyrinth*, based on the movie of the same name. Although the game starts out as a text adventure, it quickly transitions into a graphic adventure after a few minutes into the game. All interaction was still handled through the keyboard, but it did not require commands to be typed directly. Instead they were selected from a slot machine text interface with verbs shown on the left and nouns on the right.

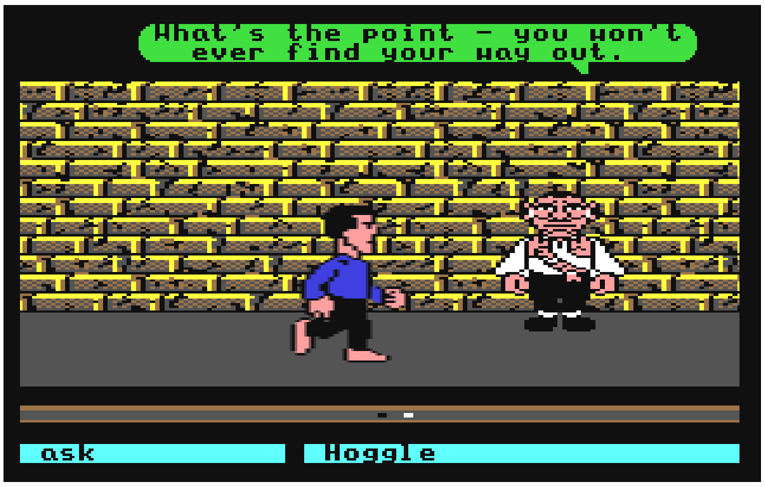


Figure 16 - Labyrynth

2.3.5 Maniac Mansion

But the game that really helped LucasArts establish itself as a strong contender in the graphic adventure field was *Maniac Mansion*, released the following year. The game tells the story of a teenager's attempt to rescue his girlfriend from an evil mad scientist, and was inspired by previous graphic adventure titles such as *King's Quest* and *Déjà Vu*, as well as many B-horror movies from the 40s and 50s with humorous elements [Kalata, 2011].

A lot of the game's merit is based on how it managed to combine and significantly improve upon its predecessors' qualities. *Maniac Mansion* was powered by a system known as the Script Creation Utility for Maniac Mansion (SCUMM) which was developed by Ron Gilbert, a well known game designer, programmer and producer.

The SCUMM engine was controlled through a point-and-click interface, with a command palette and inventory being shown at the bottom of the screen, while the top half presented the game world to the player (clear influences of ICOM and Sierra respectively).

The story was a lot less linear than previous adventure games, as the player could choose between different characters, and there were multiple endings to the game depending on the paths he took. The game was also one of the first examples in video game history to make extensive use of cutscenes in order to advance the plot.

But the most significant change lied in its design philosophy. Ron Gilbert wanted to create a user friendly experience, so he looked into streamlining aspects of the game that would not contribute to the player’s fun. This philosophy was always central in all of LucasArts games, and from game to game, new guidelines were adopted in order to make them more enjoyable [Moss, 2011].

The use of the keyboard was completely replaced by the mouse, even for player movement (through the use of an intelligent pathfinding system). Great care was also put into designing puzzles that actually made sense, and to introduce problems and solutions in a sensible order.

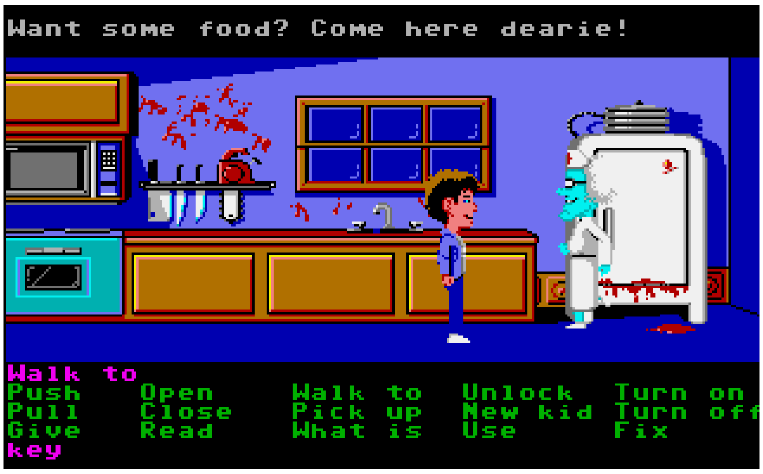

Want some food? Come here dearie!

Figure 17 - Maniac Mansion

More recent games also took great care to prevent the player from being able to get irreversibly stuck, or even to die, an extremely common and annoying occurrence in Sierra's games. The combination of all these traits along with large doses of humor, interesting and likeable characters and involving storylines, made LucasArts into what is perhaps the most memorable graphic adventure game developer to this day and age.

2.3.6 Other LucasArts Games

Although LucasArts released many great games following Maniac Mansion, it is not within the scope of this work to cover each of these games individually, as they were mostly refinements over the existing formula. But to give a brief overview into what their games looked like, Figure 18 shows some of their most famous ones, in order: *Monkey Island 2*, *Loom*, *Day of the Tentacle* (a sequel to *Maniac Mansion*), *Indiana Jones and the Fate of Atlantis*, *Sam & Max Hit the Road*, and *Full Throttle*.

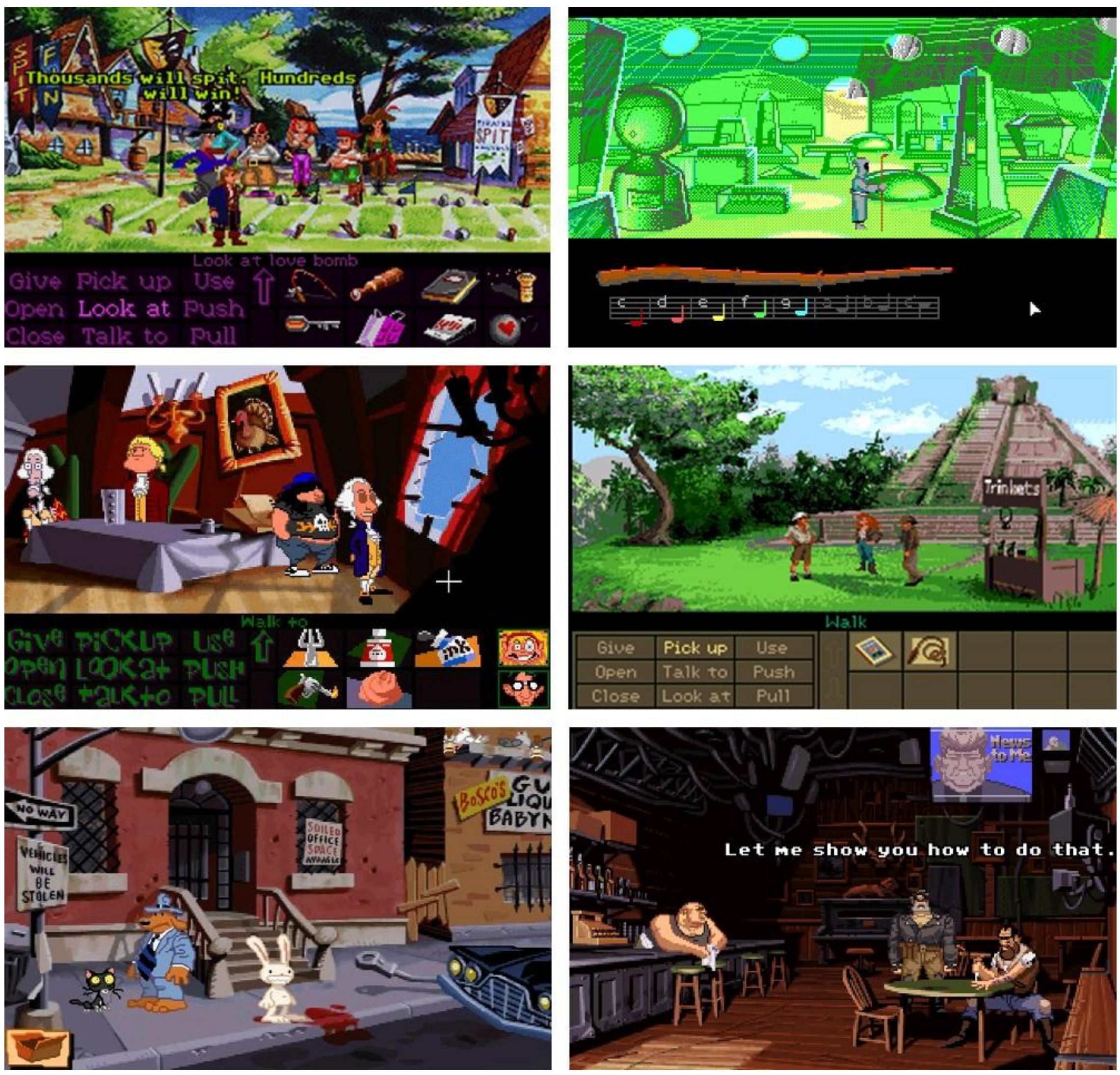


Figure 18 - Some popular games from LucasArts' catalogue.

2.4 Technological Advances

2.4.1 CD-ROM

Technology has always been central to the evolution of video games. Previously we saw how the introduction of the mouse was enough to lead the graphic adventure genre into a brand new direction.

Another significant step in that evolution was the introduction of the CD-ROM. With hundreds of times more storage capacity than its predecessors, it allowed developers the freedom to add voice-overs, full motion video sequences, recorded soundtracks, and higher resolution graphics to their games. The main disadvantage was slower read speeds which resulted in longer load times, but that was a tolerable tradeoff considering the amount of new content it permitted.

A game that contributed greatly to the CD-ROM's popularity was Cyan World's best-seller, *Myst*, released in 1993 [Moss, 2011]. *Myst* is a puzzle adventure, shown from a first-person perspective, and showcasing pre-rendered 3D graphics that were extremely sophisticated for the time. Unlike most adventure games to date, *Myst* did not have any list of actions to choose from or a specialized interface, and all interaction was done by clicking directly in the environment [Loguidice, 2009]. This fact contributed significantly in making the game accessible to a very large range of people, and with over 6 million copies sold worldwide, it remained the best-selling PC game for a long time, until it was surpassed by *The Sims* in 2002 [GameSpot, 2002].

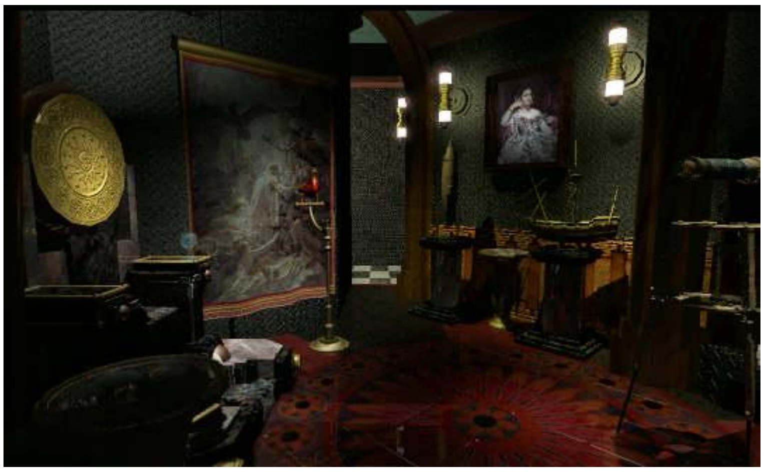


Figure 19 - Myst

In 1995, Sierra On-Line released *Phantasmagoria*, an horror-based graphic adventure which combined computer generated backgrounds with real life actors. The game required such a large amount of assets that it spanned a total of 7 CD-ROMs. *Phantasmagoria* also stirred a lot of controversy for its mature content, such as gore, violence and sex, which led to the game being banned in Australia and several major retailers refusing to sell it. A year later, a sequel was released, this time also using real life footage for most of the backgrounds.

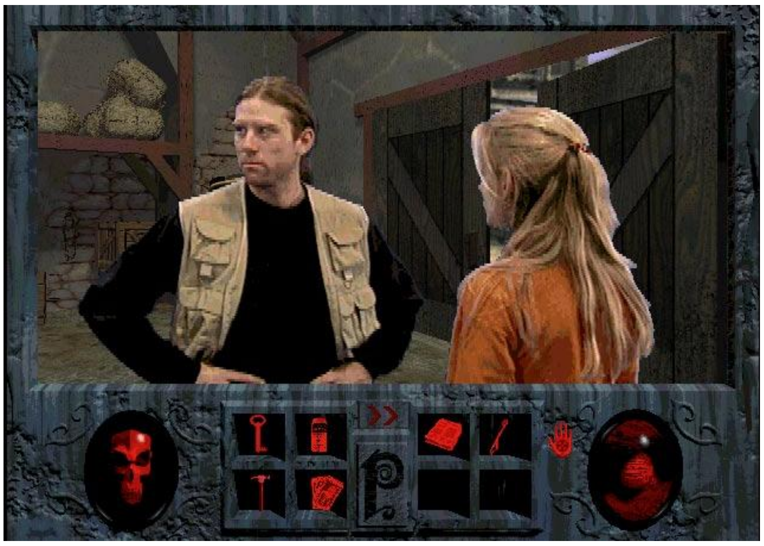


Figure 20 - Phantasmagoria

As for LucasArts, the company also made use of the CD-ROM, but in a more traditional and conservative way, such as improving the graphical and sound quality of their games, and featuring recorded voice tracks for the character dialogues. These changes were applied not only to new releases, but to older games as well, which were upgraded and re-released with new content.

2.4.2 3D Graphics

Another technological advance that had a huge impact on the world of video games, was the development of real time 3D graphics and 3D graphics hardware. Although 3D graphics were considered more relevant to other genres, a few adventure game studios also began making use of 3D graphics in their games.

In 1998, LucasArts switched from their old SCUMM engine into a new 3D engine called the GrimE, and released *Grim Fandango*, a dark comedy adventure game taking place in the Land of the Dead. The game abandoned the classic point-and-click interface in favor of a keyboard-based control scheme, where the player used the arrow keys to control the main character, and used another key to trigger an action with the closest object.

Although the game looked 3D, only the characters were actually being rendered as true 3D models, while backgrounds were still pre-rendered 2D images. But since the style of the backgrounds matched those of the characters, by carefully overlaying the 3D models on top of them, it was possible to give the illusion that the entire world was in 3D.

Unfortunately, despite being received with great acclaim by the critics, and being considered as one of the greatest games of all time, the game sold very poorly. With video games in general becoming increasingly popular with rise of 3D graphics, a very large portion of the market was now looking into other genres, such as action games like *Quake*. This fact had a strong effect on publishers, who started to drop their support for graphic adventure game developers, especially in the United States, leading to a very clear decline of the genre.

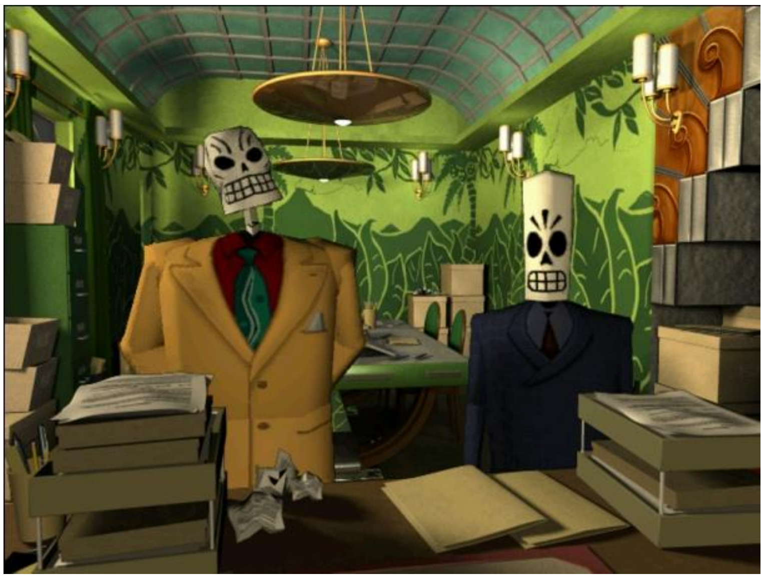


Figure 21 - Grim Fandango

After Grim Fandango, the only graphic adventure game released by LucasArts was the fourth installment in the Monkey Island series, in the year 2000. Every other attempt ended being cancelled. Sierra also underwent major changes, with the company being bought by CUC and Ken Williams stepping down as the CEO as the starting point to the decline of the company.

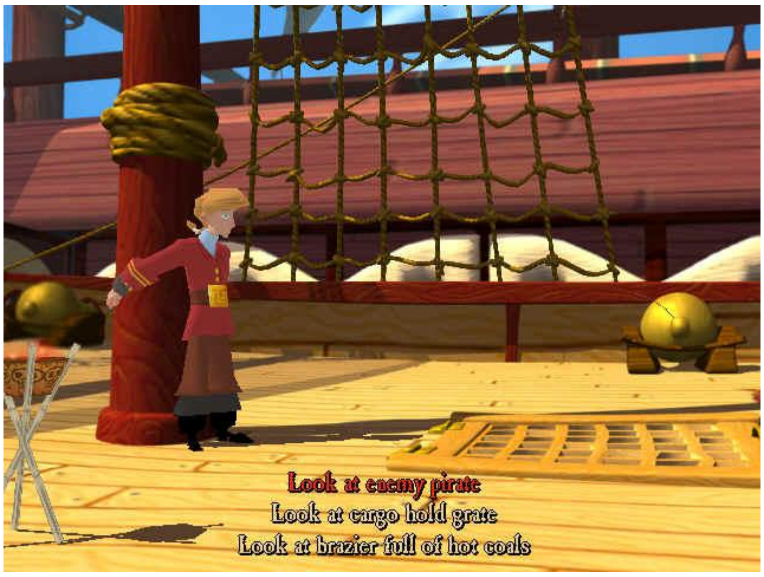


Figure 22 - Monkey Island 4

2.5 The Present

The decline in the graphic adventure genre was mostly felt in the United States, with influential companies such as LucasArts and Sierra retiring from the scene [Crook, 2007]. Even if the developers wanted to keep creating graphic adventure games, there was a generalized lack of support by publishers around the nation, and most new projects ended up being cancelled. This situation was mostly due to the fact that other game genres were rapidly increasing in popularity, while adventure games could not keep up with this growth.

Despite the situation in the United States, the graphic adventure genre was still going strong in the rest of the world, particularly in Europe and Japan. For example, Norwegian developers Funcom released *The Longest Journey* in 1999, while French developers Microïds released *Syberia* in 2001, both of which were well received by the public. Japanese visual novels on the other hand, were undergoing a revolution around this time, with games such as *To Heart* in 1997 and *Kanon* in 1999, and would only be increasing in popularity in the future.

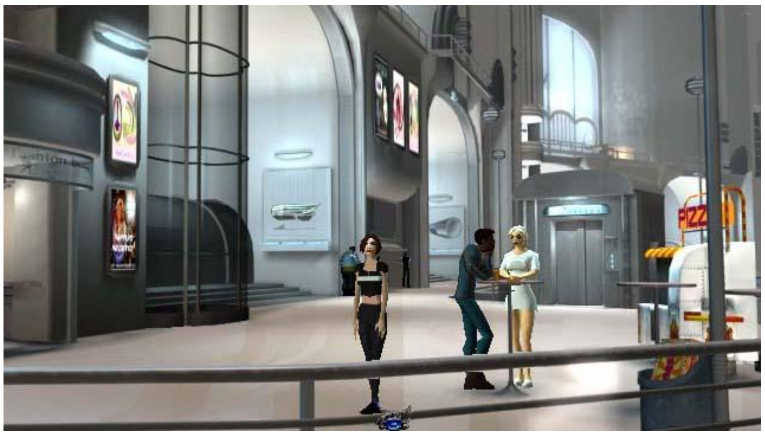


Figure 23 - The Longest Journey

Some American graphic adventure developers also refused to lay down their arms. In 2004, a company known as Telltale Games was founded by a group of former LucasArts' employees that had been working on a cancelled *Sam & Max* sequel. Managing to acquire the rights to the series, Telltale Games started releasing new *Sam & Max* games beginning in 2006, using a business model that consisted of dividing the games into several, small episodes, and releasing them individually with one or two months of interval. The same approach has been reused by the company in several other games such, such as *Tales of Monkey Island*, released in 2009, and *Back to the Future: The Game*, released in 2010 [Moss, 2011].

Another area where graphic adventure games started becoming more popular was in the video game console market. While graphic adventure games have always received ports for video game consoles, such as Maniac Mansion for the NES, the lack of a pointing device usually resulted in a weaker experience with bad controls. But some of the most recent video game consoles have popularized new interaction methods that fit the graphic adventure genre perfectly, such as the touchscreen interface of the Nintendo DS or the Wii Remote of the Nintendo Wii, which among other features, serves as a handheld pointing device. This led to the development of games such as *Zack & Wiki*, released in 2007 for the Nintendo Wii, or *Ghost Trick*, released in 2011 for the Nintendo DS. Smarthphones and tablets with their touchscreen interfaces have also become a popular target for graphic adventure game developers, with one of the most popular examples being *Machinarium*.

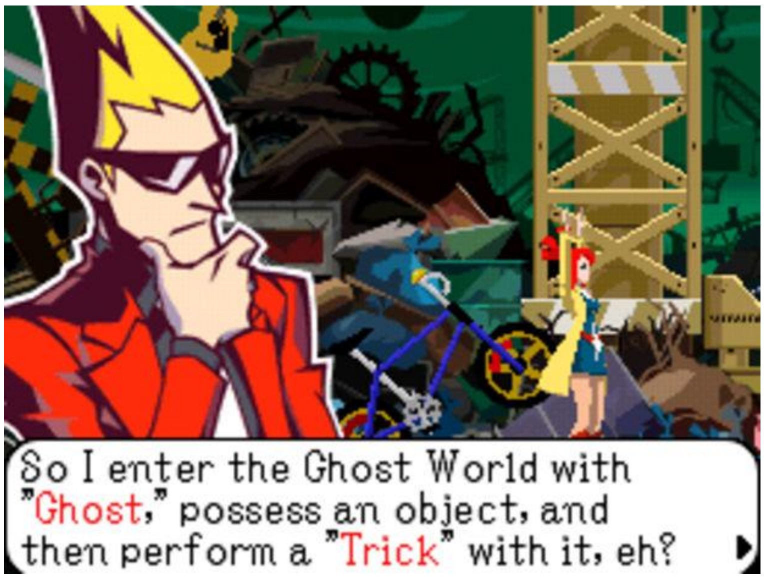

So I enter the Ghost World with "Ghost," possess an object, and then perform a "Trick" with it, eh?

Figure 24 - Ghost Trick

After almost a decade of inactivity in this field, LucasArts also had a change of heart and decided to revisit some of their old adventure titles. From this effort, the first two *Monkey Island* games were remade with an entirely new hand-drawn animated art style, high resolution graphics, re-mastered music presented in full orchestral detail, a new gameplay interface, an in-built puzzle hint system, and even a way to switch back and forth between the new and the old style at any time. The second game also included developer commentary and a behind the scenes art gallery.

Also related to LucasArts, the video game designer Tim Schaffer who played a crucial role in the development of many of their classics, left the company in 2000 to found Double Fine Productions. Although Double Fine Productions started by focusing mostly on action games, such as *Psychonauts* or *Brutal Legend*, they are currently trying to make a big comeback into the graphic adventure genre. Their first step in that direction was the puzzle adventure game *Stacking*, released in 2011, but what really took the world by surprise was their use of the crowdsourcing platform Kickstarter to announce the development of a brand new graphic adventure game in February 2012 [DoubleFine, 2012]. The project is currently under development, after managing to gather over 3 million dollars from fans of the genre. This effort has also caught the attention of other graphic adventure developers who have followed their lead, with projects such as a *Leisure Suit Larry* remake and a new game by *Gabriel Knight*'s creator. All of this proves that era of graphic adventure games might not be over yet.

2.6 Summary

Table 1 - Timeline of the evolution of adventure games

Game	Company	Year	Significance
		Text Adventure	
Colossal Cave Adventure	Crowther, Woods	1976~	First text adventure
Zork	Infocom	1980	Improved Parser
		Graphic Adventure	
Mystery House	Sierra On-Line	1980	First graphic adventure Monochromatic images
Wizard and the Princess	Sierra On-Line	1980	Colored images
King's Quest Space Quest	Sierra On-Line	1984+	Animation Character present on scene
		The Point-and-Click Interface	
Enchanted Scepters	Silicon Beach Software	1984	Point-and-click interface
Déjà Vu	ICOM Simulations	1985	Command palette Inventory
Maniac Mansion Monkey Island Day of the Tentacle Many others	LucasArts	1987+	Fun!
		Technological Advances	
Myst	Cyan World	1993	CD-ROM
Phantasmagoria	Sierra On-Line	1995	Real life footage
Grim Fandango Monkey Island 4	LucasArts	1998+	3D Characters
		The Present	
The Longest Journey	Funcom	1999	Europe while the genre was dead in America
Syberia	Microids	2001	
Sam & Max Tales of Monkey Island	Telltale Games	2006	Episodic content
Zack & Wiki Ghost Trick Machinarium	Several	2007+	Consoles Touchscreen
Unnamed Adventure	Double Fine	2012	Kickstarter ?

3 Graphic Adventure Game Engines

A game engine is a software system designed to assist in the creation and development of video games. This is usually accomplished by providing tools and reusable components that would otherwise need to be implemented by the development teams. Some game engines, such as *Game Maker*, are very generic and flexible in order to be useful for different game genres (e.g. platform games, simulation games, first-person shooters). Other game engines, such as the ones described in the following sections, are designed for a specific game genre and usually provide most of the gameplay constructs required for that type of game, at the cost of abandoning the flexibility needed to create something different from that pattern.

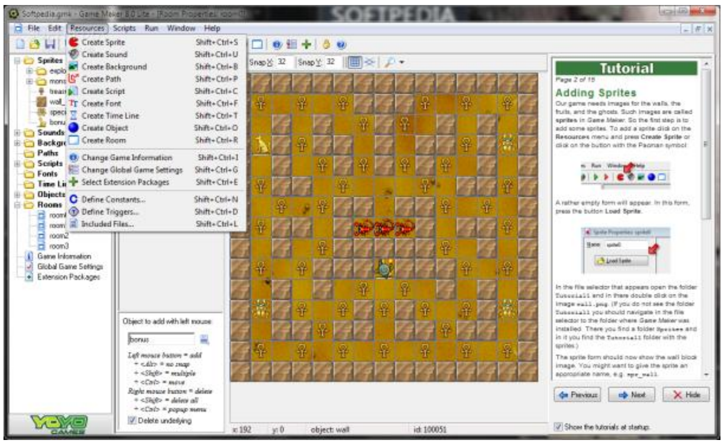


Figure 25 - Game Maker

While many games are still built from scratch or with custom game engines, an increasing number of game development studios make use of game engines that have been developed externally. This allows costs to be reduced dramatically and for teams to concentrate on creating actual game content. Even when game engines are developed internally, they are usually reused in future projects, so the benefits are the same. Some game engines such as the *UDK* or the *CryENGINE* bring the technologies that power some of the most advanced AAA games to the masses, while others such as *Unity3D* are very simple to use and provide a very attractive platform for students, hobbyists and indie game developers to create their projects.

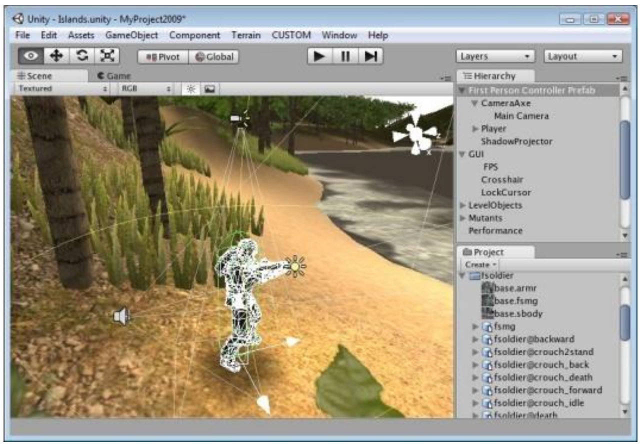


Figure 26 - Unity 3D

Finally we have those game engines that have been tailored for a specific game genre. Due to these constraints, it is possible to find game engines in this category that require no programming experience from the user, allowing people from all areas to create their own games. For example, two popular game engines in this category are *M.U.G.E.N* (for the creation of 2D fighting games) and *RPG Maker* (for the creation of tile-based role-playing games).

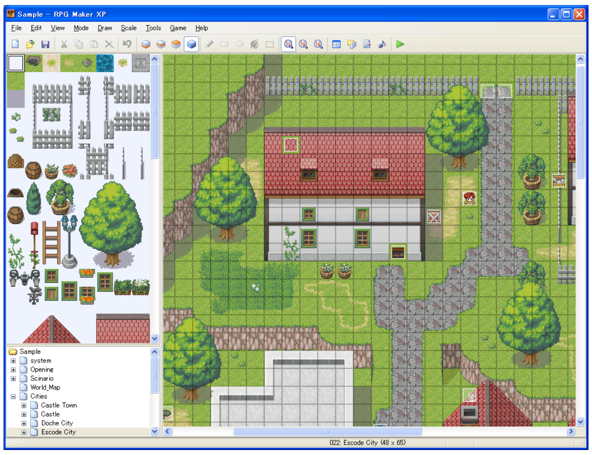


Figure 27 - RPG Maker XP

It should come as no surprise that there are also a few free and commercial game engines focused entirely on the creation of graphic adventure games. The rest of this chapter will be dedicated to the three most popular graphic adventure game engines of the moment: Adventure Game Studio, Wintermute Engine and Visionaire Studio . These three engines were selected because of their popularity and because they have already proven their quality with all the successful commercial games that have been developed and released with them.

3.1 Adventure Game Studio

Adventure Game Studio (AGS) was first released in 1997 as an MS-DOS program called "Adventure Creator". It was developed by British programmer Chris Jones, and highly inspired by Sierra On-Line's graphic adventure games, as the first version of the engine was built specifically for low resolution, keyboard-controlled adventure games.

The initial adoption of the engine was slow, but by 2001 it had achieved widespread popularity, and managed to keep an edge over its competitors (e.g. AGAST and SLUDGE) due to the size of its community and the large number of games being released every year.

The feature set also expanded significantly, with new additions such as increased screen resolution and color depth support, and hardware acceleration. The editor was also rewritten in 2008 to run on the .NET platform. In 2010, Chris Jones released the source code for the editor, and in 2011, for the runtime engine, both under the Artistic License version 2. AGS is not as modern as the other two engines covered here, but with more than a thousand games under its belt, it still holds a considerable following nowadays.

Website

http://www.adventuregamestudio.co.uk

Current Version

3.2.1 (April, 2011)

License

AGS is in free for commercial use, but there are some factors that need to be considered. In particular, you must comply with the licenses of each individual third party component that is required by AGS, as well as changing the default speech font, which is ripped from Space Quest 4.

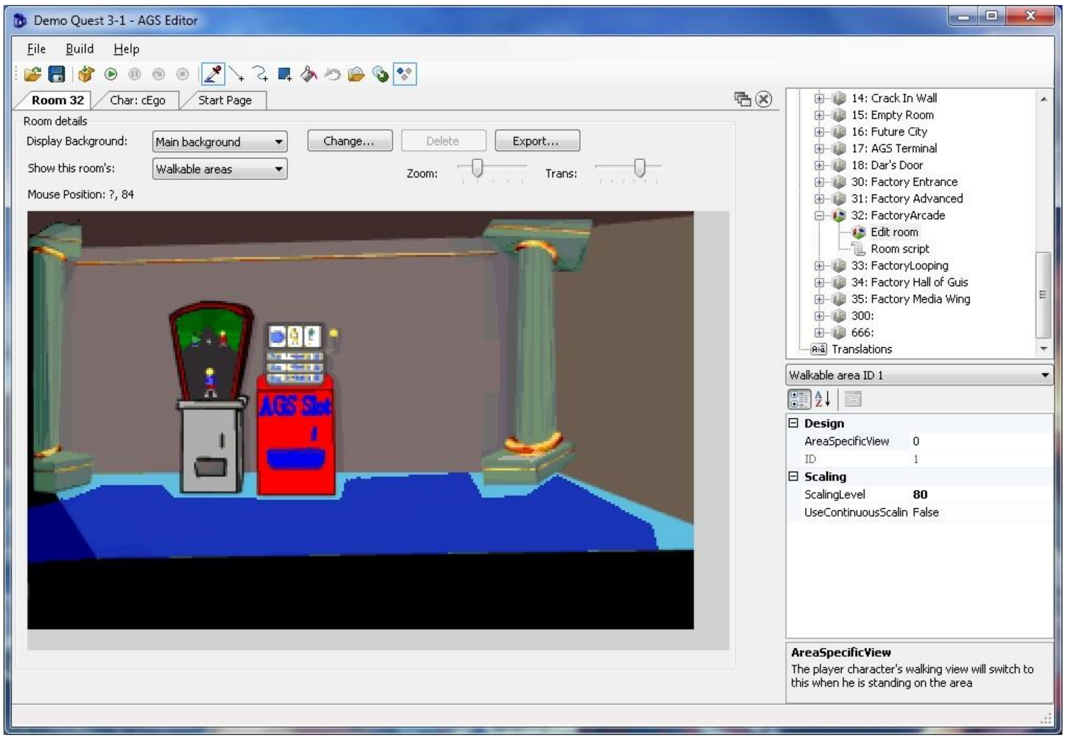
"If you wish to make money from anything you create with AGS, which covers both shareware and commercial releases, there are some factors you need to consider. First of all, the default speech font in AGS is ripped from Space Quest 4. If you are going to distribute your game commercially, you should change this because it may make Sierra unhappy. The default normal font is however drawn by me and freeware. AGS itself, as in the compiled version of the code written by me (Chris) is freeware and therefore may be distributed commercially without penalty. However, AGS uses some third-party components which have different license agreements which you will have to abide by in order to commercially release your game." *(check the reference below for a list of all the third-party components)*

[AGS, 2011]

Features

- Platforms – Only runs on Windows XP and above, although Linux and Mac ports are available for an older version.
- Resolutions – Supports resolutions from 320x200 for retro games up to 1024x768 for a more modern look.
- Color depths - Supports 8-bit (palette), 16-bit (hi-color) and 32-bit (true-color) graphics.
- Graphics – Traditional rendering with DirectDraw5, or hardware accelerated rendering with Direct3D9.
- Audio formats - Supports multiple audio formats such as OGG, MP3, WAV, MOD, XM, and MIDI.
- Voice speech - Speech packs supported with easy system to link lines of text to audio files.
- Video formats – Supports multiple video formats such as AVI, WMV, FLC, and OGV (Ogg Theora).
- Scripting – Simple yet powerful Java/C#-style language.
- Script editor - Built-in with autocomplete, syntax highlighting, calltips and online help.
- Script debugger - Built-in debugger allows placing breakpoints and stepping through script code.
- Translations – Built-in support for translating your game text to different languages.
- Plugins - Plugins are available to enhance AGS with extra features.
- Source control - Integration with any MSSCCI systems such as SourceSafe or Preforce.

Screenshots



Display Background: Main background

Show this room's: Walkable areas

Mouse Position: ?, 84

Walkable area ID 1

AreaSpecificView 0

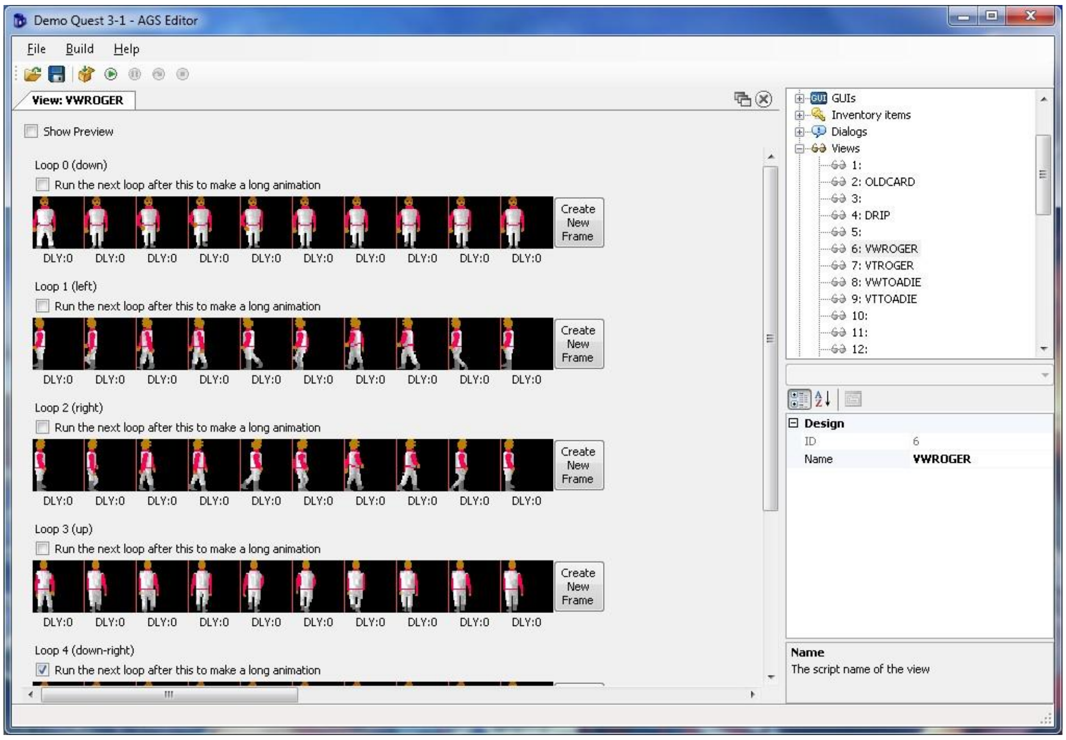
ID 1

ScalingLevel 80

UseContinuousScalin False

AreaSpecificView
The player character's walking view will switch to this when he is standing on the area

Figure 28 - The room editor with the walkable area being displayed in blue.

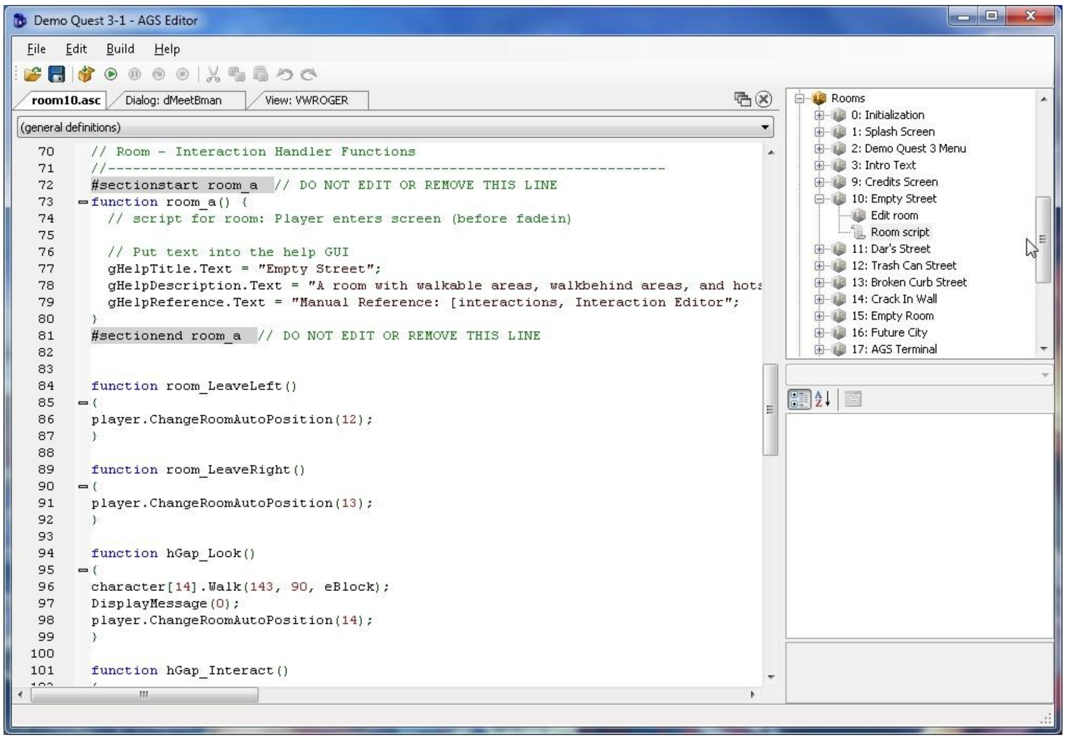


ID 6

Name VWROGER

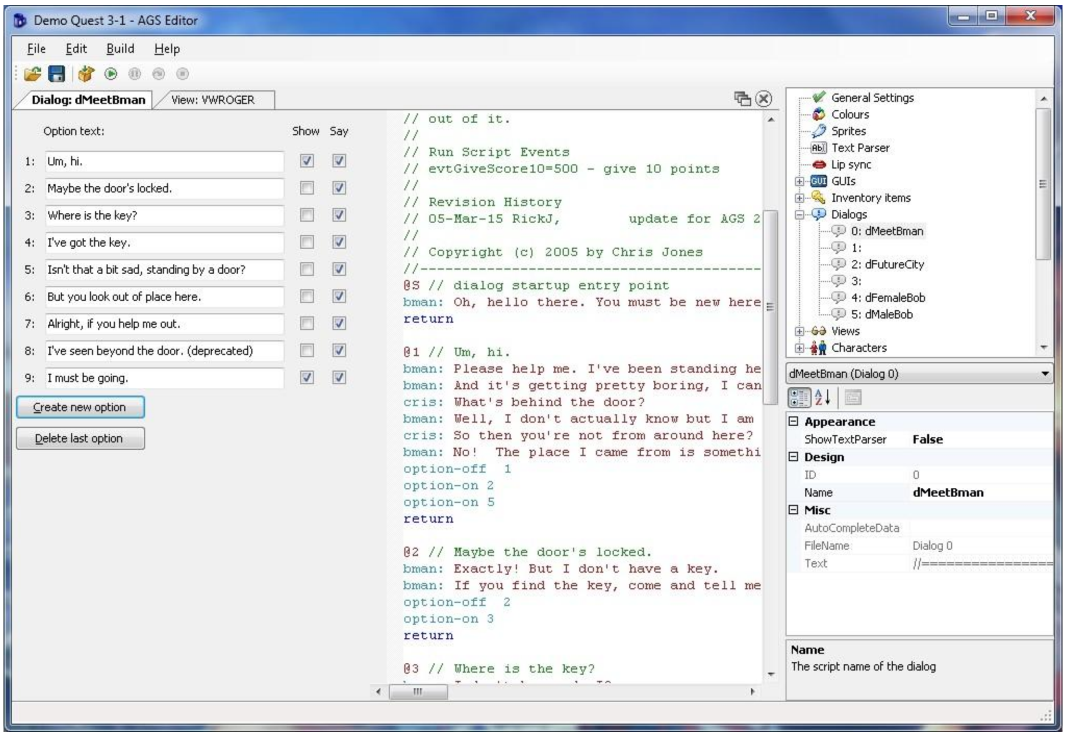
Name
The script name of the view

Figure 29 - The view editor used to setup character and object animations.



```
// Room - Interaction Handler Functions
//-----------------------------------------------------------------
#sectionstart room_a  // DO NOT EDIT OR REMOVE THIS LINE
function room_a() {
  // script for room: Player enters screen (before fadein)

  // Put text into the help GUI
  gHelpTitle.Text = "Empty Street";
  gHelpDescription.Text = "A room with walkable areas, walkbehind areas, and hots
  gHelpReference.Text = "Manual Reference: [interactions, Interaction Editor";
}
#sectionend room_a  // DO NOT EDIT OR REMOVE THIS LINE


function room_LeaveLeft()
{
player.ChangeRoomAutoPosition(12);
}

function room_LeaveRight()
{
player.ChangeRoomAutoPosition(13);
}

function hGap_Look()
{
character[14].Walk(143, 90, eBlock);
DisplayMessage(0);
player.ChangeRoomAutoPosition(14);
}

function hGap_Interact()
```

Figure 30 - A peek at the AGS code editor and its scripting language.



	Option text:	Show	Say
1:	Um, hi.	✓	✓
2:	Maybe the door's locked.		✓
3:	Where is the key?		✓
4:	I've got the key.		✓
5:	Isn't that a bit sad, standing by a door?		✓
6:	But you look out of place here.		✓
7:	Alright, if you help me out.		✓
8:	I've seen beyond the door. (deprecated)		✓
9:	I must be going.	✓	✓

```
// out of it.
//
// Run Script Events
// evtGiveScore10=500 - give 10 points
//
// Revision History
// 05-Mar-15 RickJ,        update for AGS 2
//
// Copyright (c) 2005 by Chris Jones
//-----------------------------------------
@S // dialog startup entry point
bman: Oh, hello there. You must be new here
return

@1 // Um, hi.
bman: Please help me. I've been standing he
bman: And it's getting pretty boring, I can
cris: What's behind the door?
bman: Well, I don't actually know but I am
cris: So then you're not from around here?
bman: No!  The place I came from is somethi
option-off  1
option-on 2
option-on 5
return

@2 // Maybe the door's locked.
bman: Exactly! But I don't have a key.
bman: If you find the key, come and tell me
option-off  2
option-on 3
return

@3 // Where is the key?
```

Appearance	
ShowTextParser	**False**
Design	
ID	0
Name	**dMeetBman**
Misc	
AutoCompleteData	
FileName	Dialog 0
Text	//================

Name
The script name of the dialog

Figure 31 - The dialog editor in AGS which also relies on a scripting language.

Scripting

```
#define DOORPANEL_OPEN_TIME 200

int DoorState;
int BuildingState;
int DoorPanelTimer;

function DoorInit() {
    DoorState = CLOSED;
    oDoor.SetView(24,7);
    DoorPanelTimer=0;
    RemoveWalkableArea(2);
    cPiratebob.ChangeRoom(crmDarsDoor,165,73);
}

function DoorOpen() {
    if (DoorState!=OPENED) {
        DoorState = OPENED;
        aDoorOpen.Play();
        oDoor.SetView(24,7,0);
        Wait(50);
        oDoor.Animate(7,3,eOnce,eBlock,eForwards);
        RestoreWalkableArea(2);
    }
}

function DoorClose() {
    if (DoorState!=CLOSED) {
        DoorState = CLOSED;
        oDoor.SetView(24,7,5);
        aDoorClose.Play();
        Wait(1);
        oDoor.Animate(7,3,eOnce,eBlock,eBackwards);
        RemoveWalkableArea(2);
    }
}
```

```
function hSidewalk_Look() {
    DisplayMessage(1);
}

function hRoad_Look() {
    DisplayMessage(2);
}

function hSkyline_Look() {
    DisplayMessage(3);
}

function hGap_Look() {
    character[14].Walk(143, 90, eBlock);
    DisplayMessage(0);
    player.ChangeRoomAutoPosition(14);
}
```

Showcase

Thousands of games have been produced with Adventure Game Studio over the years. The most notable examples come from Wadjet Eye Games, and independent game development team that created most of their commercial titles using AGS. They are particularly famous for the Blackwell series, which started in 2006 and is currently in its fourth installment. The series is known for its excelent story and voiceovers, having received several awards over the years.

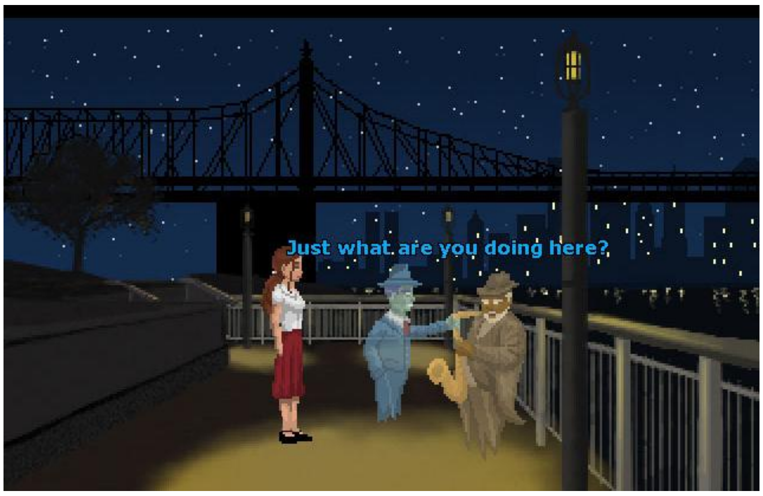


Figure 32 - Blackwell Unbound

Another notable example in their library is Gemini Rue, a science fiction cyberpunk graphic adventure released in 2011. It should be noticeable from these screenshots, that both Gemini Rue, and the Blackwell series, despite being relatively recent games, are built with retro-styled graphics. This is a trait present in most games created with AGS, as the engine does not support the higher resolutions used by modern games, and works better using low resolution graphics.

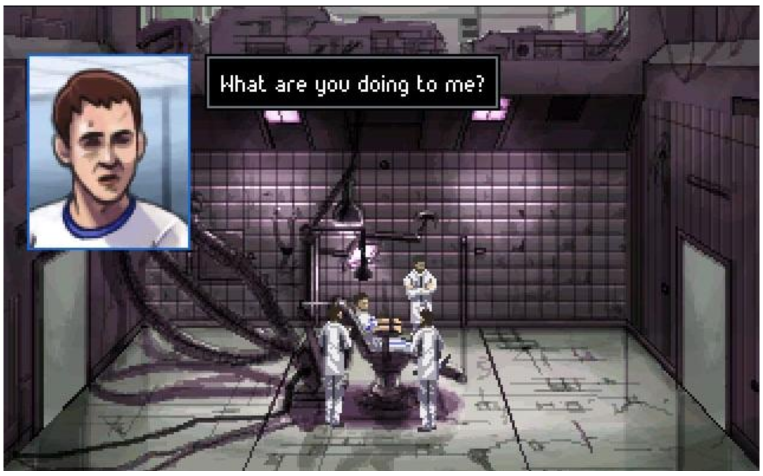


Figure 33 - Gemini Rue

3.2 Wintermute Engine

The Wintermute Engine (WME) is a graphic adventure game engine developed by Jan Nedoma in the Czech Republic, and inspired mostly by the SCUMM engine and the Broken Sword series. The engine started out as a personal project of the author, but was eventually released to the public in 2003. Wintermute had a great reception since the beginning, probably due to its balanced mix between usability and flexibility, and manages to remain a popular game engine to this day.

One of the distinguishing features of this engine is the support for 3D characters, which was added in 2005, and allows the creation of games with graphics like Grim Fandango or The Longest Journey. The engine became open-source in 2007, and a portable version known as WME Lite was released in 2011, making the engine easier to port to other platforms, such as the Mac OSX 10.6 and iOS.

Website

http://dead-code.org

Current Version

1.9.1 (January, 2010)

License

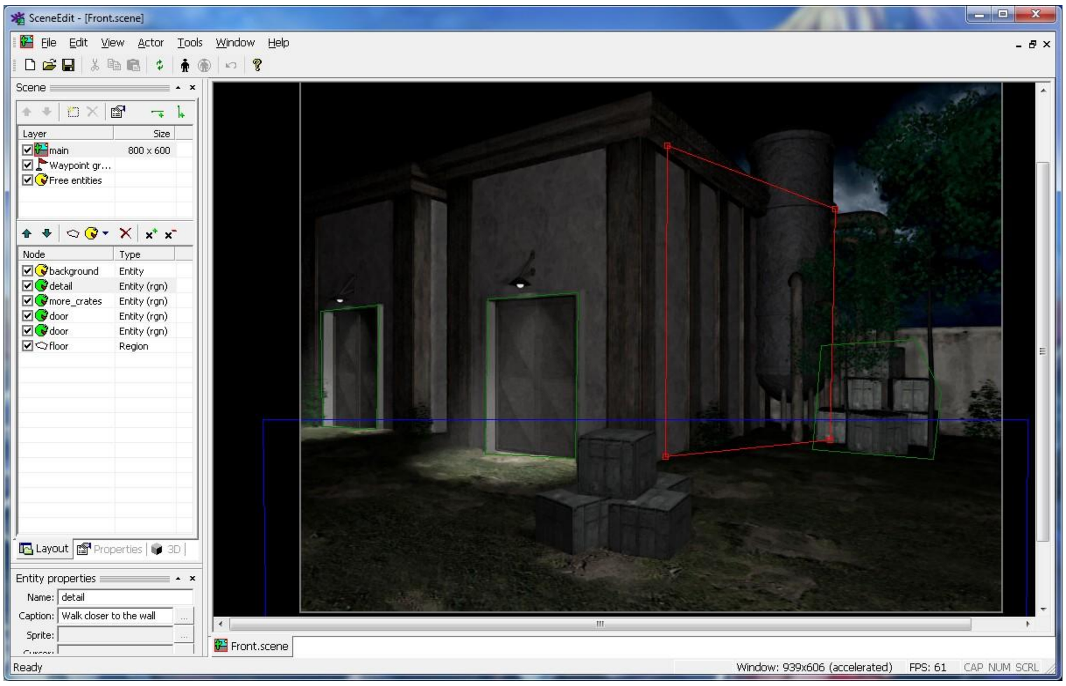
Free for commercial use *and* open-source.

> Wintermute Engine Development Kit is provided for free for both hobby and commercial use. However, if you find it useful and you'd like to support its further development and/or express your appreciation, you're encouraged to make a donation.
>
> Engine source code is available upon request under the terms of GNU Lesser General Public License.
>
> [WME, 2010]

Features

- Resolution – Supports virtually any screen resolution, from retro-style graphics to high-resolution games.
- Color depth – Support for 16-bit (hicolor) and 32-bit (true color) graphics, with automatic conversion if needed.
- Rendering – Supports 3D hardware acceleration to provide fast 2D graphics in high resolutions, with advanced graphical effects such as transparency, alpha blending and antialising. Also has a compatibility mode which runs on older computers without hardware acceleration.
- Supported file formats – The engine supports BMP, TGA, PNG and JPG files for graphics, OGG and WAV for sound, and OGV and AVI with automatic subtitles display for video.
- Scripting language – Provides a flexible object-oriented scripting language, which allows you to add almost any feature or puzzle you need.
- Parallax scrolling – Multi-layered parallax scrolling is natively supported by the engine with no additional scripting required to implement it.
- Particle effects – Has built-in support for particle effects, such as rain or snow, through the scripting language.
- Support for 3D characters – Includes support for real-time rendered 3D characters, that allows 2D environments to be combined with 3D characters to create games like Syberia or The Longest Journey.
- Effect files – Supports the use of custom HLSL effect files for state-of-the-art 3D rendering.
- User interface layer – You can build complex user interfaces using several available controls such as windows and buttons. All the controls are fully skinnable, so you can change their look to fit your game.
- Localization support – Allows the games to be translated into other languages, a feature that is not limited to texts, but also fonts, graphics or even sounds.
- Accessibility support – Provides several options to improve accessibility for vision-impaired players, such as using an automatic text-to-speech synthesizer.
- Plugins – Support for external DLL plguins to add new features to the engine.

Screenshots


SceneEdit - [Front.scene]
File Edit View Actor Tools Window Help
Scene
Layout Properties 3D
Front.scene

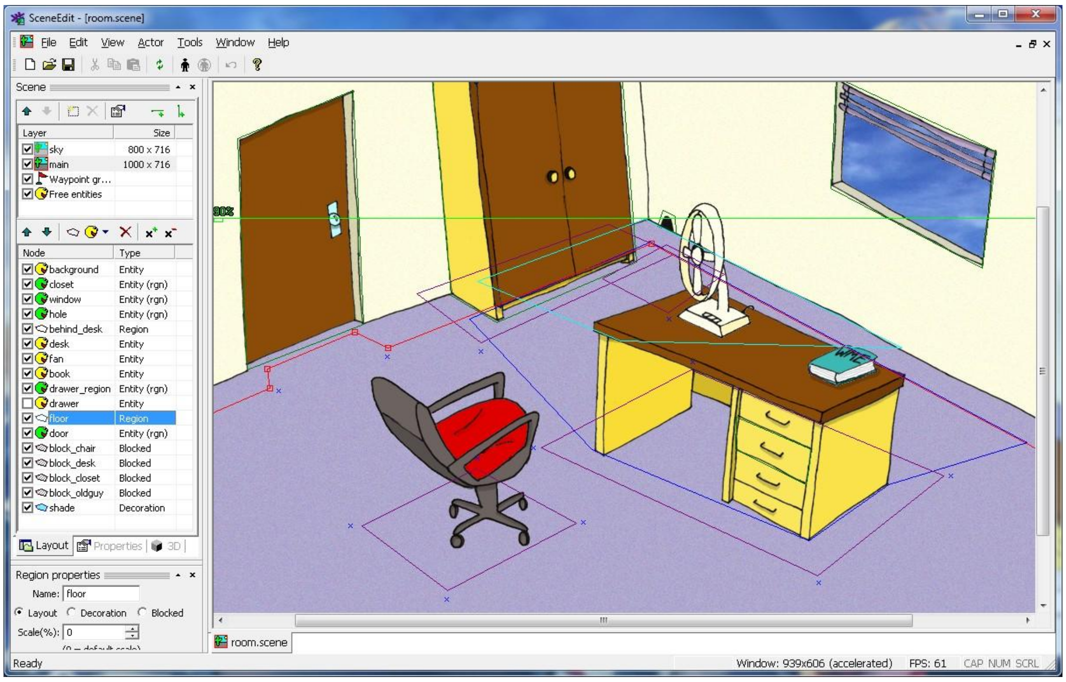
Layer	Size
☑ main	800 x 600
☑ Waypoint gr…	
☑ Free entities	

Node	Type
☑ background	Entity
☑ detail	Entity (rgn)
☑ more_crates	Entity (rgn)
☑ door	Entity (rgn)
☑ door	Entity (rgn)
☑ floor	Region

Entity properties
Name: detail
Caption: Walk closer to the wall
Sprite:

Ready
Window: 939x606 (accelerated) FPS: 61 CAP NUM SCRL

Figure 34 – The scene editor in WME showing a pre-rendered 3D background.



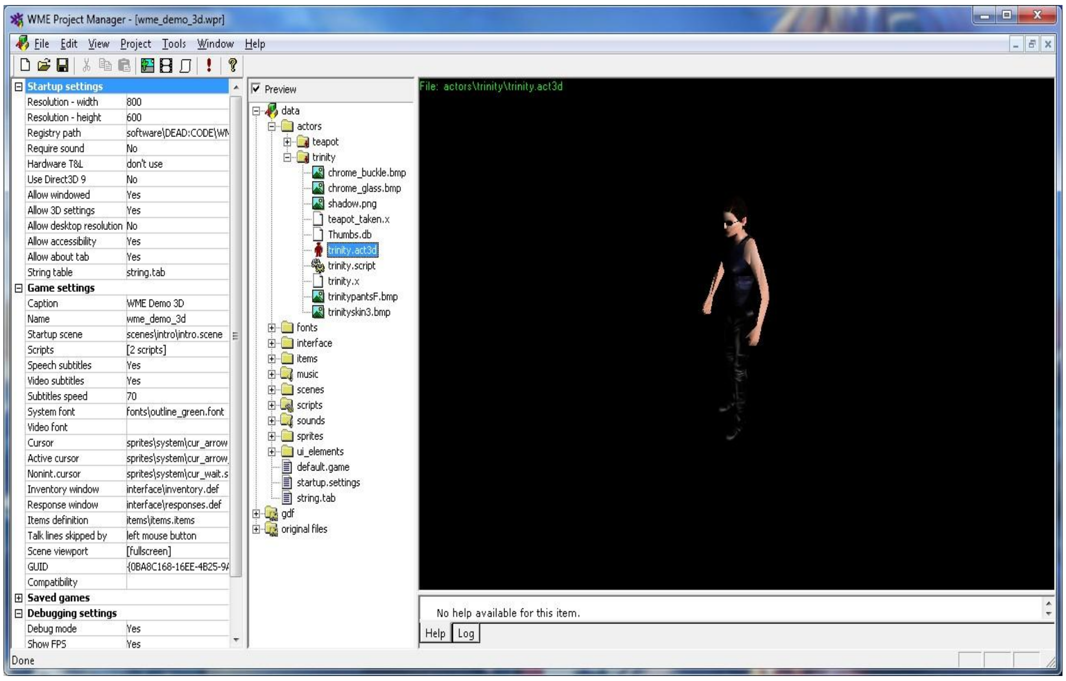
Layer	Size
☑ sky	800 x 716
☑ main	1000 x 716
☑ Waypoint gr…	
☑ Free entities	

Node	Type
☑ background	Entity
☑ closet	Entity (rgn)
☑ window	Entity (rgn)
☑ hole	Entity (rgn)
☑ behind_desk	Region
☑ desk	Entity
☑ fan	Entity
☑ book	Entity
☑ drawer_region	Entity (rgn)
☐ drawer	Entity
☑ floor	Region
☑ door	Entity (rgn)
☑ block_chair	Blocked
☑ block_desk	Blocked
☑ block_closet	Blocked
☑ block_oldguy	Blocked
☑ shade	Decoration

Region properties
Name: floor
(● Layout) ○ Decoration ○ Blocked
Scale(%): 0
(0 = default scale)

Ready
Window: 939x606 (accelerated) FPS: 61 CAP NUM SCRL

Figure 35 - The same scene editor but this time working with a 2D scene.


WME Project Manager - [wme_demo_3d.wpr]
File Edit View Project Tools Window Help
Preview
Help Log
Done

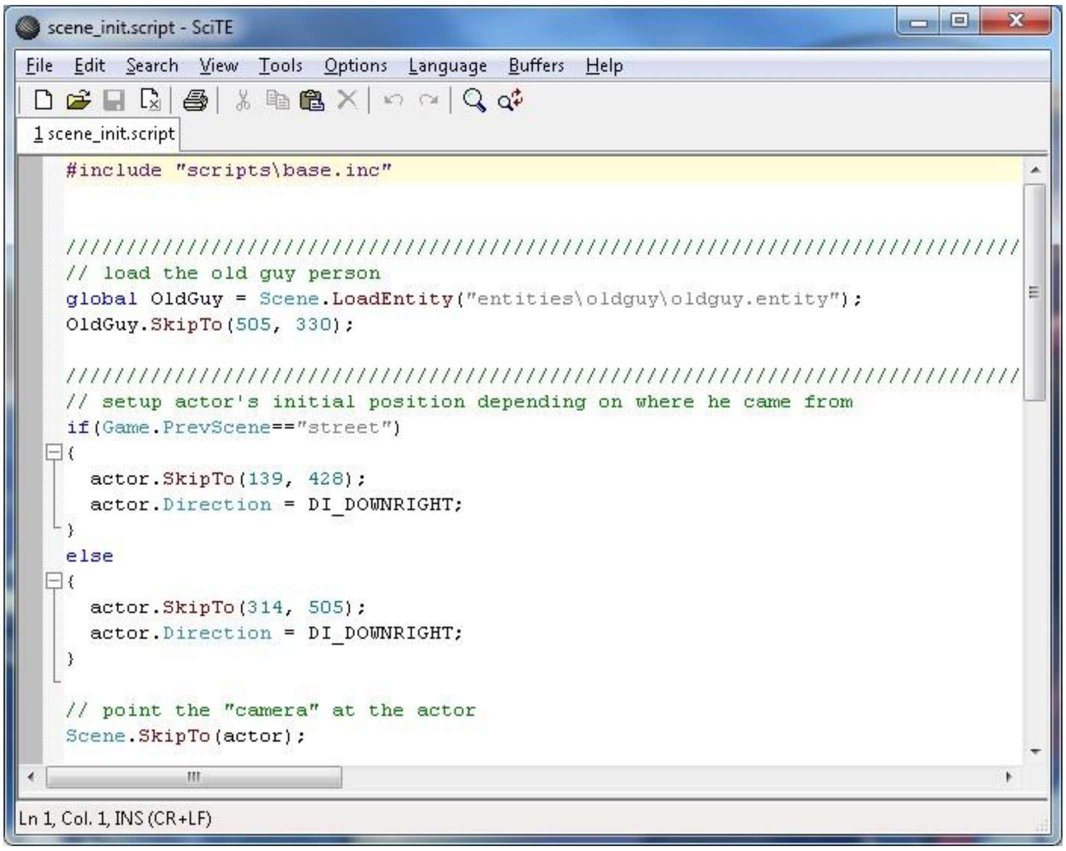
Startup settings	
Resolution - width	800
Resolution - height	600
Registry path	software\DEAD:CODE\WM
Require sound	No
Hardware T&L	don't use
Use Direct3D 9	No
Allow windowed	Yes
Allow 3D settings	Yes
Allow desktop resolution	No
Allow accessibility	Yes
Allow about tab	Yes
String table	string.tab
Game settings	
Caption	WME Demo 3D
Name	wme_demo_3d
Startup scene	scenes\intro\intro.scene
Scripts	[2 scripts]
Speech subtitles	Yes
Video subtitles	Yes
Subtitles speed	70
System font	fonts\outline_green.font
Video font	
Cursor	sprites\system\cur_arrow
Active cursor	sprites\system\cur_arrow
Nonint.cursor	sprites\system\cur_wait.s
Inventory window	interface\inventory.def
Response window	interface\responses.def
Items definition	items\items.items
Talk lines skipped by	left mouse button
Scene viewport	[fullscreen]
GUID	{0BA8C168-16EE-4B25-9A
Compatibility	
Saved games	
Debugging settings	
Debug mode	Yes
Show FPS	Yes

```
data
  actors
    teapot
    trinity
      chrome_buckle.bmp
      chrome_glass.bmp
      shadow.png
      teapot_taken.x
      Thumbs.db
      trinity.act3d
      trinity.script
      trinity.x
      trinitypantsF.bmp
      trinityskin3.bmp
  fonts
  interface
  items
  music
  scenes
  scripts
  sounds
  sprites
  ui_elements
  default.game
  startup.settings
  string.tab
gdf
original files
```

File: actors\trinity\trinity.act3d

No help available for this item.

Figure 36 – Previewing a 3D actor imported into WME.


scene_init.script - SciTE
File Edit Search View Tools Options Language Buffers Help
1 scene_init.script
Ln 1, Col. 1, INS (CR+LF)

```
#include "scripts\base.inc"


////////////////////////////////////////////////////////////////////////////
// load the old guy person
global OldGuy = Scene.LoadEntity("entities\oldguy\oldguy.entity");
OldGuy.SkipTo(505, 330);

////////////////////////////////////////////////////////////////////////////
// setup actor's initial position depending on where he came from
if(Game.PrevScene=="street")
{
  actor.SkipTo(139, 428);
  actor.Direction = DI_DOWNRIGHT;
}
else
{
  actor.SkipTo(314, 505);
  actor.Direction = DI_DOWNRIGHT;
}

// point the "camera" at the actor
Scene.SkipTo(actor);
```

Figure 37 – The scripting language used by WME.

Scripting

```
#include "scripts\base.inc"

// load the old guy person
global OldGuy = Scene.LoadEntity("entities\oldguy\oldguy.entity");
OldGuy.SkipTo(505, 330);

// setup actor's initial position depending on where he came from
if(Game.PrevScene=="street")
{
  actor.SkipTo(139, 428);
  actor.Direction = DI_DOWNRIGHT;
}
else
{
  actor.SkipTo(314, 505);
  actor.Direction = DI_DOWNRIGHT;
}

// point the "camera" at the actor
Scene.SkipTo(actor);

global StateRoom;
```

```
on "book"
{
  Game.Interactive = false;
  actor.Talk("OK, I'll put the book back.");

  // walk to the desk
  actor.GoTo(782, 645);
  actor.TurnTo(DI_UPLEFT);

  // play "take" animation
  actor.PlayAnim("actors\molly\ul\take1.sprite");

  // show the book entity again and remove "book" item from the inventory
  Game.DropItem("book");
  var EntBook = Scene.GetNode("book");
  EntBook.Active = true;

  // play the second "take" animation
  actor.PlayAnim("actors\molly\ul\take2.sprite");
  Game.Interactive = true;
}

on "LookAt"
{
  actor.GoToObject(this);
  actor.Talk("It's a sturdy looking desk.");
  if(!Game.IsItemTaken("book")) actor.Talk("There's a book lying on it.");
}
```

Showcase

Wintermute has been used to develop a considerable amount of games, but the ones that stand out the most are the ones that make use of the engine's advanced graphical features. One example is Dark Fall: Lost Souls, a first-person horror-adventure game that managed to provide a full panoramic 3D experience by creating every scene in the game as a textured 3D object, with the camera being positioned from the inside. This is typically not possible in a Wintermute game, and serves to showcase how flexible the engine can be.

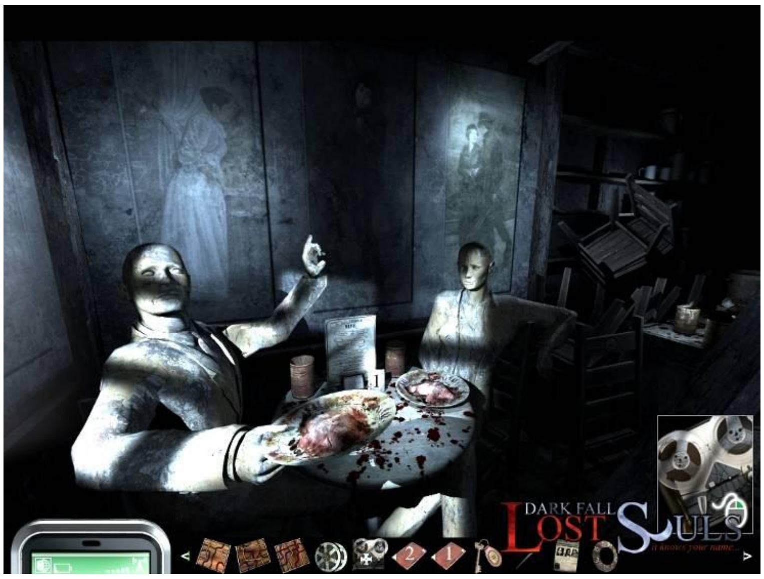


Figure 38 - Dark Fall: Lost Souls

Another good example of the power of Wintermute is from the indie game Alpha Polaris, which also an horror adventure game, but from a third-person perspective and featuring 3D characters over beautiful pre-rendered backgrounds.

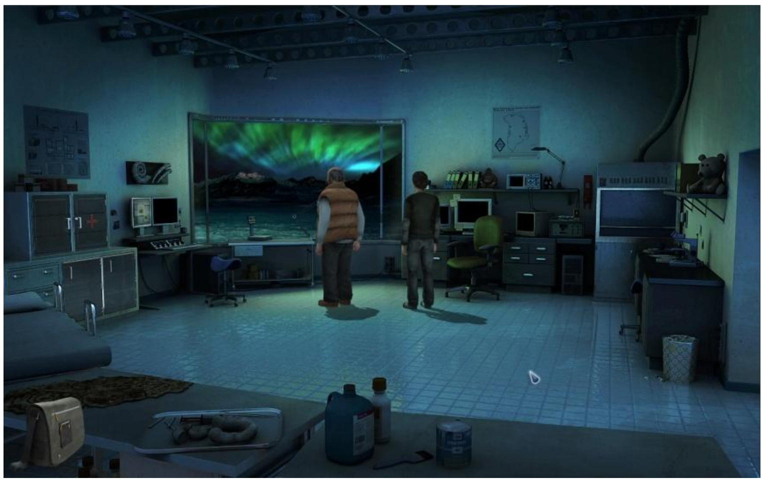


Figure 39 - Alpha Polaris

3.3 Visionaire Studio

Visionaire Studio is a graphic adventure game engine that was primarily developed with simplicity and usability in mind. It started out as a school project by Timotheus Pokorra and Thomas Dibke in 1998, but they were later joined by Alex Hartmann and Robert Neumann as the project grew. The first public version of Visionaire was released around 2001, with the second version being released in early 2004.

At the time of writing, the engine is in its third version and still being actively developed. The biggest selling point of the engine was that it did not require any scripting knowledge by the user, and every action in the game can be created directly through the editor's interface, although advanced users can still write scripts directly in the Lua programming language. The creators are also currently working on new additions to the engine, such as multi-platform support and 3D characters, which should make it an even stronger contender in the field.

Website

http://www.visionaire-studio.net

Current Version

3.6 (August, 2011)

License

There are free and commercial licenses depending on the scope of the game.

"To obtain a license for Visionaire 3.x, the users have 3 different options to choose from:

License Option 1 - Freeware

The freeware version of Visionaire 3.x is available here and does not cost anything. After having downloaded the software, the user can test the software extensively. All features of Visionaire 3.x are available for that - there are no constraints of functionality. But the user of the freeware version has no possibility of compiling his games into one file (Vis game file). Therefore, the "uncompiled game resources" can be still edited with the Visionaire editor, i.e. the files can be seen and edited by other users of Visionaire. Also only freeware-games may be developed. It's prohibited to use the freeware-version of "Visionaire" for developing or distributing commercial games.

License Option 2 - Limited Distribution

The full version of Visionaire 3.x can be purchased for 35 Euros. With this version it is possible to compile the project to a finished game. A finished game consists of one or multiple files and cannot be edited by the Visionaire Editor anymore. With this license you can develop and distribute both freeware and commercial games with limited distribution rights. The games are only to be sold for a maximum price of 15 Euro (or the according value in another currency). It is allowed to sell your games yourself or on a shareware platform. If the game is developed for a third party (e.g. publisher or client for an ad game) then license option 3 is required.

License Option 3 - Unlimited Distribution

The unlimited distribution rights for a game can be purchased for 1000 Euro. The Visionaire 3.x full version is included with this license. If requested it is possible to receive intermediate builds (internal Visionaire versions between two public releases). This allows you to get a version with possible important bugfixes or features faster than with the regular releases."

[Visionaire, 2011]

Features

- Resolution - Supports basically all reasonable screen resolutions, including widescreen.
- Supported formats - Supports most popular multimedia formats and video codecs.
- Action System - Offers an easy to use and flexible action system, so that even users without programming skill can create all of the common elements of the game.
- Scripting - Beyond the action system, Visionaire provides even more flexibility through the integrated Lua scripting language.
- Hardware Acceleration - Uses OpenGL hardware acceleration to achieve optimal speed.
- Flexible Interface - Visionaire's flexible interface system allows the use of many different interfaces for commands, placeholders, switches and much more.
- Special Effects - Provides in-built support for parallax scrolling and particle, without the need for any code.
- Multi-language - Games created with Visionaire can be translated into as many languages as you want. Unicode is used for maximum flexibility and to support even complicated characters like those in Cyrillic or Chinese.

Screenshots

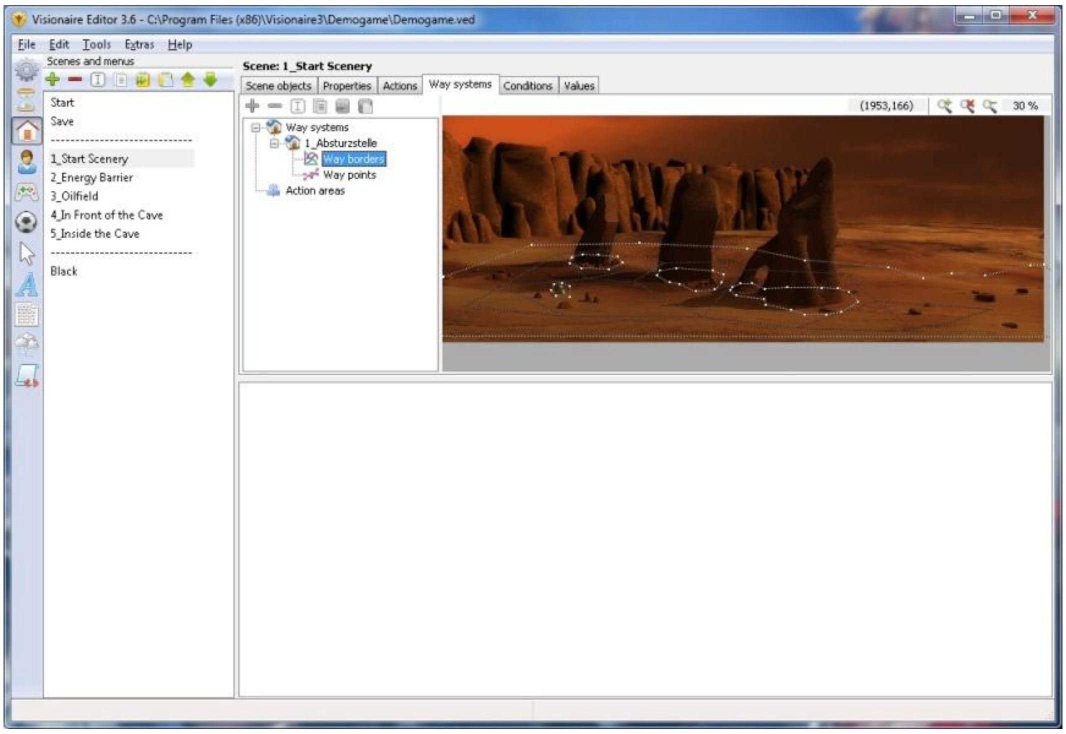


Figure 40 - The Scene and menus editor in Visionaire.

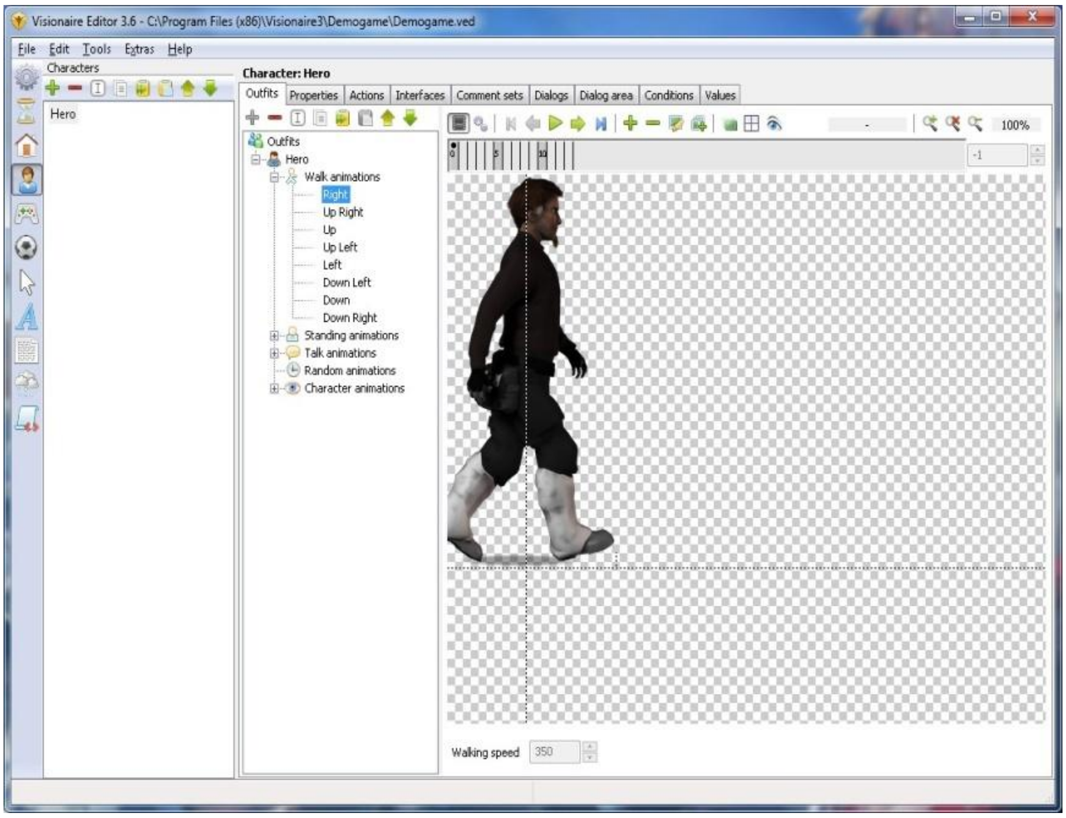
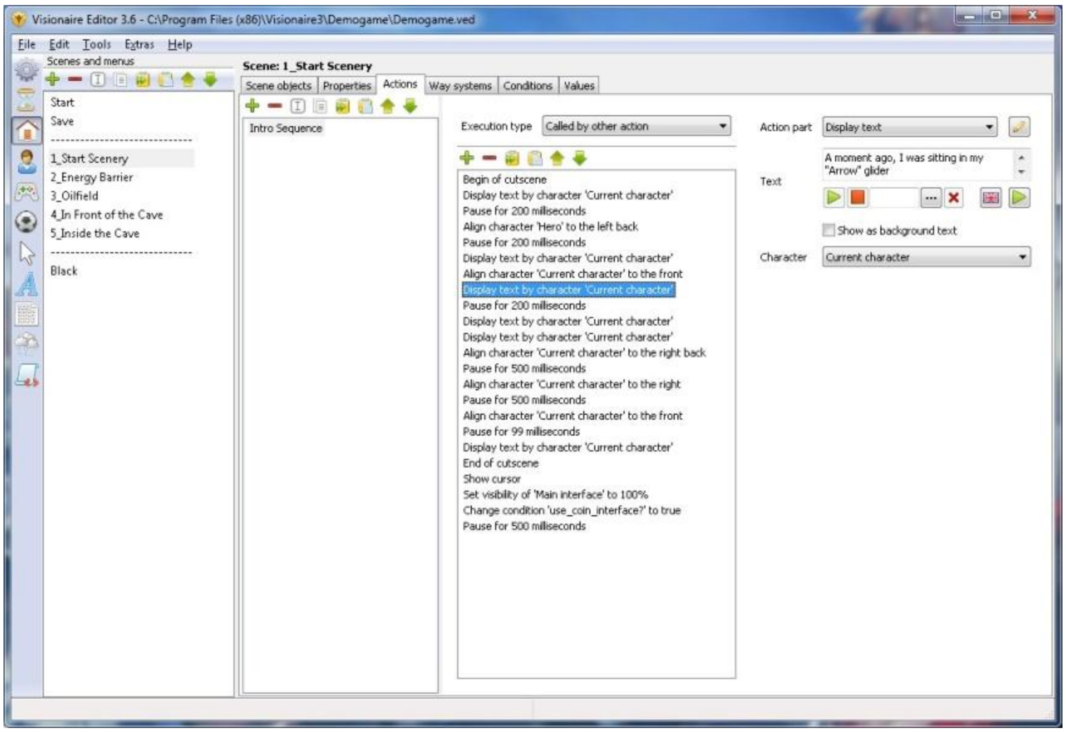


Figure 41 - The Character editor in Visionaire showing the animation timeline.


Visionaire Editor 3.6 - C:\Program Files (x86)\Visionaire3\Demogame\Demogame.ved
File Edit Tools Extras Help
Scenes and menus
Scene objects Properties Actions Way systems Conditions Values

Start
Save
1_Start Scenery
2_Energy Barrier
3_Oilfield
4_In Front of the Cave
5_Inside the Cave
Black

Scene: 1_Start Scenery

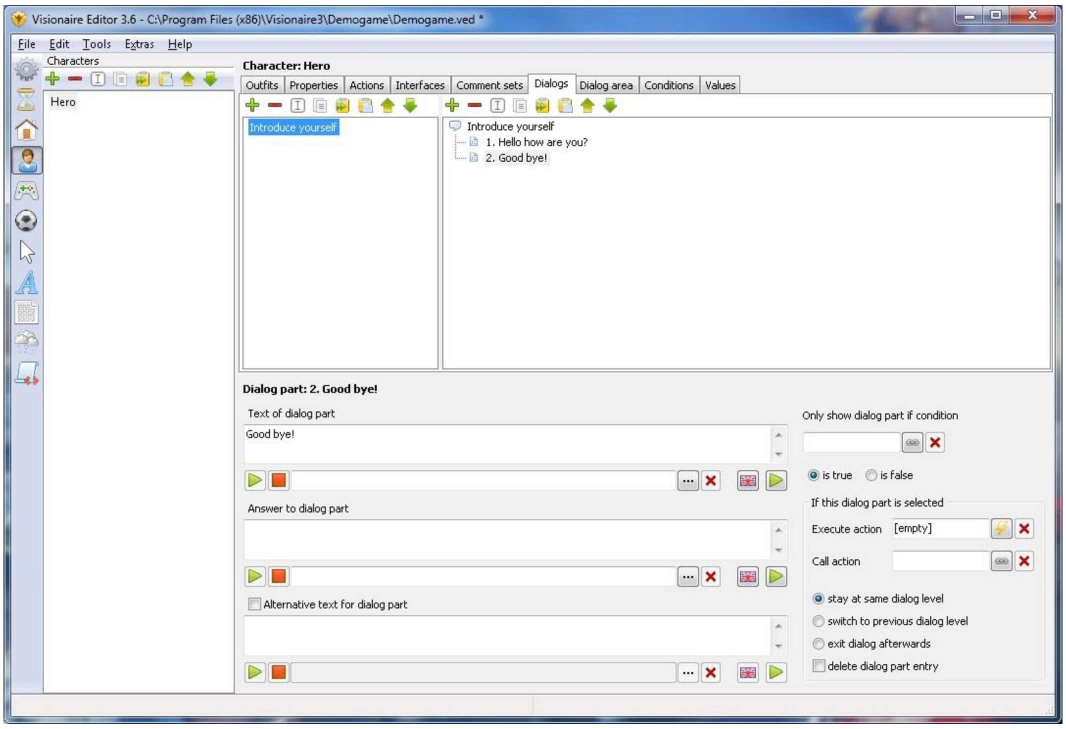
Intro Sequence

Execution type: Called by other action

```
Begin of cutscene
Display text by character 'Current character'
Pause for 200 milliseconds
Align character 'Hero' to the left back
Pause for 200 milliseconds
Display text by character 'Current character'
Align character 'Current character' to the front
Display text by character 'Current character'
Pause for 200 milliseconds
Display text by character 'Current character'
Display text by character 'Current character'
Align character 'Current character' to the right back
Pause for 500 milliseconds
Align character 'Current character' to the right
Pause for 500 milliseconds
Align character 'Current character' to the front
Pause for 99 milliseconds
Display text by character 'Current character'
End of cutscene
Show cursor
Set visibility of 'Main interface' to 100%
Change condition 'use_coin_interface?' to true
Pause for 500 milliseconds
```

Action part: Display text
Text: A moment ago, I was sitting in my "Arrow" glider
Show as background text
Character: Current character

Figure 42 - An example of how actions can be created without programming in Visionaire.


Visionaire Editor 3.6 - C:\Program Files (x86)\Visionaire3\Demogame\Demogame.ved *
File Edit Tools Extras Help
Characters
Outfits Properties Actions Interfaces Comment sets Dialogs Dialog area Conditions Values

Hero

Character: Hero

Introduce yourself

```
Introduce yourself
  1. Hello how are you?
  2. Good bye!
```

Dialog part: 2. Good bye!

Text of dialog part: Good bye!
Answer to dialog part
Alternative text for dialog part

Only show dialog part if condition
(● is true ○ is false)
If this dialog part is selected
Execute action: [empty]
Call action
● stay at same dialog level
○ switch to previous dialog level
○ exit dialog afterwards
☐ delete dialog part entry

Figure 43 - The dialogue editor in Visionaire Studio.

Showcase

Several notable games have been created with Visionaire Studio, most of which tend to feature high-definition 2D graphics. Although there is still no support for 3D characters like Wintermute, most games created with Visionaire manage to have a distinctively modern look to them. One example is The Whispered World, developed by Daedalic Entertainment and released in 2010. The game features gorgeous painted scenery and cartoon characters, and makes extensive use of parallax scrolling, with rooms often having more than ten layers of detail.

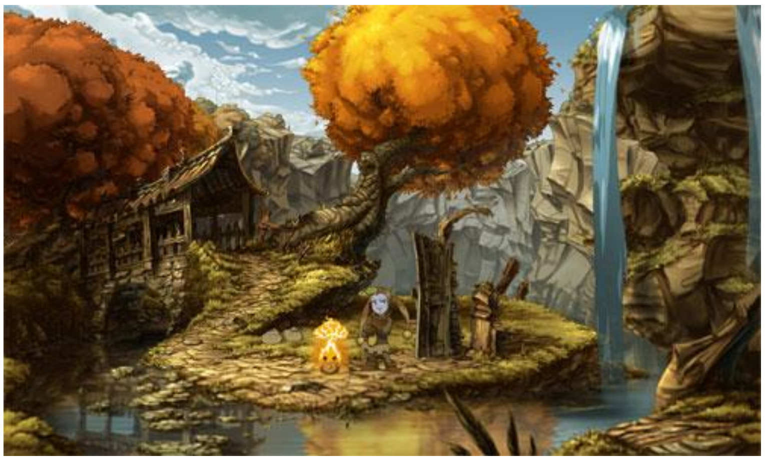


Figure 44 - The Whispered World

Another example by the same developers is Deponia, released in 2012, and featuring the same level of graphical detail as The Whispered World. Furthermore, the game was created in widescreen format, making it a perfect fit for modern computers screens.

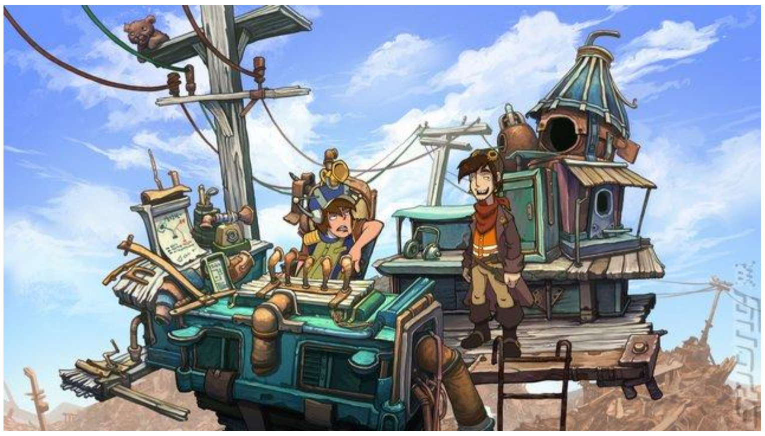


Figure 45 - Deponia

3.4 Summary

All of the engines presented in this chapter are extremely powerful, and capable of creating virtually any graphic adventure game imaginable. At their core however, there seems to be a different philosophy and target audience between the engines, which is worth noting:

- *Adventure Game Studio* is used almost exclusively for creating retro adventure games, and it is still evident that the engine was heavily inspired by Sierra's old AGI engine. The editor is simple, and despite making use of a scripting language, the language is not too difficult to learn, in part due to the built-in auto-complete feature of the code editor.

- *Wintermute* aims to provide modern features and flexibility, and is the only one of the three engines that currently supports 3D characters. All of this power comes at the cost of added complexity, and both the editor and the scripting language are significantly harder to learn and to use.

- *Visionaire* on the other hand aims for absolute ease of use, by making everything available from within the interface, and eliminating the need to learn a scripting language entirely. Despite that, the editor is feature packed and modern, allowing smooth high definition graphics and special effects.

But the differences described above are mostly just guidelines, inferred from the types of games that are usually created with each of them, and not actual limitations of the engines, which aside from a few technical differences, have very similar feature sets. The table below should provide a general comparison between each of the engines from a feature set point of view.

Table 2 - Feature set comparison of graphic adventure engines

	AGS	**Wintermute**	**Visionaire**
Platforms	Windows	Windows	Windows
License	Free	Free	Commercial
Screen Resolutions	Limited	Any	Any
Hardware Acceleration	Yes	Yes	Yes
Scripting	Required	Required	Optional
Parallax Scrolling	Plugin	Yes	Yes
Particle Systems	Plugin	Yes	Yes
3D Characters	No	Yes	No
Localization	Yes	Yes	Yes
Text-to-Speech	No	Yes	No
Plugins	Yes	Yes	No
Latest Update	April, 2011	January, 2010	August, 2011
Philosophy	Classic Experience	Power / Flexibility	Simplicity

4 Analysis and Requirements

In regard to the three graphic adventure game engines presented in the previous chapter, if you ignore the features that distinguish them from each other, it is easy to realize that there is a lot in common between them, such as having rooms, characters, objects, items, actions and dialogue sequences. That's because all of these traits are pretty much required by the majority of graphic adventure games, and therefore must be supported by any graphic adventure game engine.

The goal of this chapter is to take the common ground between these engines and use it to identify the feature set that *must* be implemented in Nostalgia Studio, as well as the best way to implement each of these features in order to maximize the simplicity of the user experience.

In general terms, we could say that the after the game has been designed, the process of implementing a graphic adventure game goes through three different stages. The process is iterative and each of the stages will usually need to be visited multiple times and in different orders. Those stages are:

1. Creating the game world with all of its rooms.

2. Populating the game world with objects and characters.

3. Bring the game world to life with scripted actions and dialogues.

The first stage consists of creating all the environment that composes the game world, which in virtually every graphic adventure game, consists of a set of rooms. Rooms need to carry metadata that allows characters to navigate within and between them. In order for players to navigate within a room, the bounds of the floor and any obstacles in the room must be marked. As for navigation between rooms, that is usually accomplished by adding doors, or invisible regions in the rooms, that teleport the player to a new destination after examining or entering. This information is usually added manually by the developer when creating the room.

The second stage consists of creating characters and objects and adding them to the rooms created on the first stage. Another important concept of most graphic adventure games is the item, which is an object the player can hold in his inventory for later use. Some items are directly related with objects in the world, while others are added directly to the player's inventories without.

The third and final stage consists of adding behavior and interaction to the world. This is usually done by defining chains of automated action sequences that take place when certain events occur. This is the stage where the narrative and puzzles are added to the game.

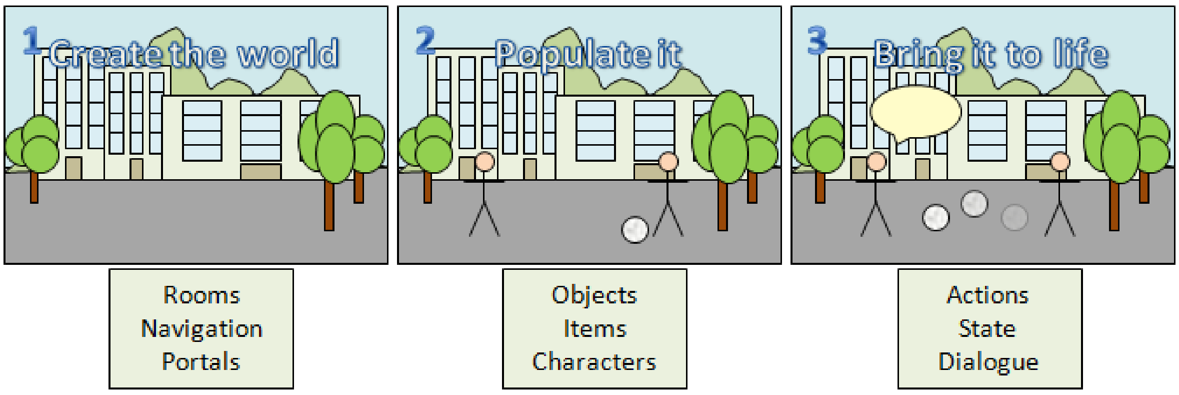


Figure 46 - The three stages of implementing a graphic adventure game.

The method used to gather all the information presented in this chapter was to specify a very simple game scenario that contained all of the minimum requirements of a graphic adventure game, and then implement that scenario in each of the three game engines. Every step of the process was documented, and that information combined and organized into a set of common tasks and categories. The requirements of that scenario were:

- To have two separate rooms, linked by a door, where the character is free to navigate.
- To have two characters, one controlled by the player, and one non-playable character.
- To have an object in one of the rooms that the character could pick up.
- To be able to have a conversation with the non-playable character.
- To be able to give the an item to the non-playable character.
- To have the dialogue with the non-playable character change once the item was given.

4.1 Creating the Game World

4.1.1 Rooms

The first step in creating the game world is to create some rooms. Rooms in graphic adventure games are usually small in size, but packed with detail, and frequently have several layers of imagery. Some layers are drawn behind the characters (i.e. background) while others may appear in front of them (i.e. foreground). In some games, layers scroll by at different speeds (i.e. parallax scrolling) which gives a strong illusion of depth to the scene.

Table 3 - Rooms

	AGS	**Wintermute**	**Visionaire**
Create room	Right-click on "Rooms" node.	Right-click on "Scenes" folder.	Create new entry in the "Scenes and menus" section.
Add background	Choose a single image for the background.	Room can have as many layers as needed. Each layer holds a single image and can be set as either background of foreground.	Choose a single image for the background.
Add foreground	Paint over parts of the background image to mark them as walk-behinds.		The only way is to create objects for the foreground images and disabling interaction.
Choose room size	Deduced automatically from the background image.	Deduced automatically from the main layer.	Deduced automatically from the background image.
Parallax scrolling	Only by using plugins.	Implemented by varying the size of the layers.	Implemented at the object's level, by selecting different scroll speeds for each object.

Comments: The approach used by AGS to create a foreground layer is interesting, but requires you to draw the contour of the foreground precisely, which can be difficult with the mouse. Using separate images for background and foreground would be easier. In this respect, Wintermute has the advantage because it allows rooms to have as many layers as needed, while Visionaire does not have the concept of layer, requiring objects to be used to emulate them.

Solution: Nostalgia Studio will work a bit differently than these three engines. Rooms will always have only two layers, the background layer and the foreground layer, but these layers will be able to hold an unlimited number of images in them. Images will be added to the room using a drag-and-drop interface, and may then be ordered, scaled, rotated and positioned freely within the room. In practice, this allows infinite layers like Wintermute. Parallax scrolling will not be implemented at this time for the sake of simplicity.

4.1.2 Navigation

At this point, the main look of the room is already specified, but it is simply a collection of images, with no metadata associated. For the characters to be able to walk around in this room, it is necessary to mark which areas should be treated as the floor, and which areas are obstacles. Another important thing to decide is the scale and depth of the room, which basically corresponds to how large the characters should look in each portion of the room.

Table 4 - Navigation

	AGS	Wintermute	Visionaire
Define floor	Draw floor region by using a set of tools very similar to MS Paint.	Draw floor polygon starting from a rectangle and shaping it by adding new vertices on the borders.	Draw floor polygon sequentially, vertex by vertex, right clicking at the end to close it.
Define obstacles	Draw using the right mouse button to erase parts of the floor.	Add new regions for the obstacles, but mark them as blocked.	Start drawing a new polygon while holding down the ALT key.
Define paths	Deduced automatically from the floor, but has performance issues at higher resolutions.	Need to manually place waypoints around convex vertices of obstacles.	Draw waypoints inside the floor polygon, and then link them manually to form a graph.
Define scale	Select min and max scaling values for each walkable area.	Add horizontal lines to the room known as scale levels.	Select scaling factor for each node in the waypoint graph.

Comments: Overall, the easiest approach is the one used by AGS, where you simply draw the floor area, and then erase the portions that correspond to obstacles. However, because it allows such fine grained control over the shape of the floor, the implementation suffers from performance problems. In particular, pathfinding in AGS works almost at the pixel level, using a grid representation, and so the amount of nodes increases exponentially with the size of the rooms and the resolution of the game. Wintermute and Visionaire both use points-of-visibility pathfinding with a polygonal representation of the floor, which performs a lot better than AGS, but both require the designer to create the graph manually. In this regard, Wintermute only requires nodes to be placed on the floor, with edges being deduced automatically, while Visionaire requires both nodes and edges to be placed manually, a process which is very flexible, but cumbersome.

Solution: Nostalgia Studio will also use a polygonal points-of-visibility pathfinding, but trying to construct the underlying graph automatically from the given floor shape. As for scaling, it will use a simplified version of Wintermute's approach, where the designer only needs to choose the scaling level for the top and the bottom of the room.

4.1.3 Portals

Even after players have the information required to navigate within a room, they still need a way to travel between rooms. This is usually done by creating some sort of invisible region on the floor that reacts to the player stepping on it and teleports the character to another location in the world. In some cases, the teleportation could happen in response to examining some object on the scene, such as a door, but the overall process is still similar enough, so it will not be described separately.

Table 5 - Portals

	AGS	**Wintermute**	**Visionaire**
Define region	Create a hotspot on room-A with the drawing interface.	Create a region on room-A with the polygon interface.	Create an action area on room-A with the polygon interface.
React to character entering	Add a new event to the hotspot that reacts to the player standing on it.	Assign a new script file to the region. Add code to the script file to handle the actor entry event.	Add *action* to the action area that executes when a character enters.
Teleport to destination	On the event, add code to send the player to room-B. Destination coordinates are input manually.	On the event add code to change room. On room-B's script file add code to position the character when he appears.	Add empty object to room-B where the character should appear. Add *action part* to the action that teleports character to that object.

Comments: Defining a region and making it react to a character stepping on it is very similar in all three engines, except that Visionaire does not require any code to be written. However, there's quite a bit of variation in how the teleportation action is defined, and each of the solutions has its own set of quirks. AGS has the simpler solution, the only disadvantage being that you need to check the X and Y coordinates of the destination, and write them down manually. An interface to help with this selection would be helpful. Wintermute is the most contrived, because the act of changing room, and the act of positioning the character in the new room, are handled separately and in completely different script files. As for Visionaire, the only downside is needing to create an object to serve as the target of the teleportation.

Solution: Since Nostalgia Studio is aiming for a codeless experience, it will work similarly to Visionaire Studio, but also allowing the destination to be chosen directly without the need to create an extra object. In other words, a new dialog will appear allowing the character to choose one of the rooms and click directly where the character should appear, or select one object from the room. Additionally, it will also provide a specialized type of region just for portals which already groups the character enter event and the teleport action, since it's such a common combination.

4.2 Populating the Game World

4.2.1 Objects

An object in a graphic adventure game can be considered as any part of the rooms that is interactive or animated. Objects usually have a name assigned to them, and can be the target of player actions, although sometimes they exist only to provide animation to the rooms. Another characteristic of objects is that their position and depth in the scene can be dynamic, allowing characters to walk both behind and in front of them depending on their relative positions.

Table 6 - Objects

	AGS	Wintermute	Visionaire
Create object	Choose objects category when editing a room and right-click directly in the room.	Add a sprite entity while editing a scene. Can optionally be placed in a special layer for free entities.	Add a new object in the "Scenes and objects" tab of the room.
Define graphics	For simple objects choose image directly. For animated objects create and assign a view.	For simple objects choose image directly. For animated objects create and assign a sprite.	Simple objects and animated objects are both handled in the objects properties, but on separate tabs.
Define interactivity	All objects are interactive unless disabled in code.	There's an interactive property which can be specified on the objects.	Objects need a collision region to be interactive.
Define origin	By default the bottom of the object is used as the origin, but it can be overridden with a baseline property.	For simple images the center is used as the origin, while for sprites there's a cross marking it.	By default the center of the object is used as the origin, but it can be overridden with a centerline property.
Object sorting	Objects are automatically sorted by their origins.	Regular entities appear in the specified order. Free entities appear along with the region the object is in.	Objects are automatically sorted by their origins.

Comments: There is not much of a difference between how objects are handled in the three engines, since all of them consider objects to be a part of the rooms. All of them also use separate parts of the interface to handle static and animated objects, which is a bit redundant, since static images could just as well be treated as animations with a single frame. As for the depth of an object in the room, there's always a point within the object that acts as its origin and is used to determine the drawing order of object. Players usually have control over the object's origin. But while AGS and Visionaire sort objects by their vertical positions, Wintermute requires additional regions to be created, and has the distinction between regular and free entities. Another interesting difference is that in AGS and Wintermute, objects are interactive by default, while in Visionaire the user must remember to draw a collision region for the object before any interaction becomes possible.

Solution: Nostalgia Studio will keep the general framework used by these engines, but try to simplify the process where possible. There will be no distinction between static images and animations, and interactivity will be controlled by a simple property. As for the order of objects, it will be possible to choose whether an object should be treated as background, foreground or dynamic. When the selected positioning is dynamic, the objects will be automatically sorted by their origin, which is marked by a cross when defining the object graphics.

4.2.2 Items

One major part of graphic adventure games is to collect items and use them or combine them in several situations to advance the plot. There's usually an inventory window where a grid of the items currently in possession of the player is displayed. Sometimes these items correspond to an object in a room, so that when the player picks up the object, it disappears from the room and an item appears in the player's inventory. Other times there's no relation to any object, and the item exists only in the inventory.

Table 7 - Items

	AGS	**Wintermute**	**Visionaire**
Create item	Create in the inventory items section by right-clicking.	Write new entry in a global text file with all the items in the game.	Add a new entry to the items section.
Define graphics	Choose images for the item in its properties.	Write path to images directly in the text file.	Choose image or animation in the item's properties.
Link to object	There's no actual link.	Can be specified in the object's properties.	There's no actual link.

Comments: Items are very simple, and can almost be considered as being just a name and a image. But Wintermute is harder to use than the alternatives because instead of providing an interface for creating items, it uses a definition text file that users have to edit manually. An interesting bit is that Wintermute allows objects to be associated with an item, which allows the act of picking up an object and adding that item to the inventory to be automated.

Solution: Nostalgia Studio will have a separate section of the editor dedicated to creating and editing items like AGS and Visionaire.

4.2.3 Characters

Similarly to objects, characters are also interactive and animated entities of the game world. However, there are many additional actions that a character can perform, which an object cannot. A character can walk around the game world, often changing the room that they're in. A character can talk, which is usually implemented by displaying the text above their heads. You can often engage in conversation with non-playable characters, so they need to have dialogue sequences associated with them. Characters can change the direction that they're facing. All of these require a large number of animations to be defined, for each state and direction combination that the character could be in. Finally, some characters can be controlled by the player, and need to have an inventory to carry items with them.

Table 8 - Characters

	AGS	**Wintermute**	**Visionaire**
Create character	Right-click in the characters node	Right-click either in the actors or entities folder. Actors can walk while entities cannot.	Add a new entry in the characters section.
Define graphics	Create a view for each of the character states and assign them in the character's properties. Each view has one animation for each direction.	When creating an actor or entity, a set of sprites is automatically created for each state and direction. Edit those sprites.	Character has a set of outfits. Each outfit has a set of states. Each state has a group of animations for each direction.
Switch costume	Either change all views in code or switch to another character.	Either load a separate actor file, or load new animations in code.	Simply change the current outfit of the character.
Place in world	In the character's properties, select the starting room and starting position.	Edit the room's script file and load the actor there, while the position is set in the actors definition file.	Create a mock object in the room, and link the character's "Character stands at" property to it.
Set main character	Press a button labeled "Make this the player character" in the character's properties.	The main character is the one assigned to the global actor variable in the game's init script.	Select as the active character in the game properties.

Comments: Compared to the other two, Wintermute is a lot harder to use because it requires editing definition files and code for almost every step in the process. On the other hand, Visionaire is very simple, especially because it has the concept of outfit which helps to group every related animations together.

Solution: Nostalgia Studio will follow Visionaire's approach, except not requiring an object to be created to mark the starting position of the character.

4.3 Bringing the Game World to Life

4.3.1 Actions

Perhaps the most important feature required to add behavior to a graphic adventure game, is the ability to react to certain events in the game, usually triggered by the player, and execute a series of actions in response. These actions encompass everything that can be done in the game, such as making characters speak, changing the room that is currently on display, picking up an item, or triggering a dialogue sequence.

Table 9 - Actions

	AGS	**Wintermute**	**Visionaire**
Attach action to entity	All rooms have a script file containing all the actions for that room and any object, region or hotspot in that room. Actions for characters and items go in a global script file. Each action is a separate function in those files.	Rooms, characters, objects, and almost everything gets its own separate script file with the actions related to then. Some are created automatically while others are attached manually on the properties of the entity.	Almost everything in the editor has an actions tab that allows new actions to be created with an interface.
Select trigger reason	The name of the function decides under which conditions the action should execute. But the events interface for each entity already creates the functions for you.	Add an event construct such as "on event name" to the correct script file, with all the action code inside. Must know the correct event name.	When editing an action, there's an execution type property that allows the trigger reason to be selected from a list of possibilities.
Define action content	Add code to the function created for the action in the step above.	Add code to the event created in the step above.	Add new action parts to the action, and select what they should do from a list.

Comments: This is one of the aspects where each of the engines handles it very differently. Both AGS and Wintermute require users to write code in a special scripting language, but between the two, AGS is significantly easier to use, not only because there are a lot less files to deal with, but because the code editor has many features to help you program. Visionaire on the other hand, doesn't require you to write any line of code, and all actions in the game can be simply selected from a list in the interface and configured accordingly.

Solution: Since one of the goals of Nostalgia Studio is to be used by people without programming knowledge, the approach followed will be very similar to the one used by Visionaire.

4.3.2 State

Another important feature required to add behavior to the game world is the ability to store information for later use, and the ability to follow different paths of execution depending on the content of that information. This information can vary from simple boolean, yes or no values (used for example to signal whether an event has already taken place) to a numerical value (used for example to limit the amount of times that a player can perform an action). At the minimum, the application should be able to compare the contents of that information with another value, and based on the result decide what should be executed.

Table 10 - State

	AGS	**Wintermute**	**Visionaire**
Storing state	Local variables are defined directly in the script files. Global variables have a separate interface just for them.	Local and global variables are defined directly in the script files. Only global variables are persistent.	All entities in the game world have their own set of conditions (boolean) and values (number or string) and everything is created in the interface.
Branching on state	With a typical if-else programming construct.	With a typical if-else programming construct.	There are action parts such as "If condition" and "If value" that allow conditional behavior.

Comments: The way AGS and Wintermute both handle state and branching is typically the same as in most programming languages, using variables, assignments and "ifs". The only exception to this are the global variables in AGS which are created separately with an interface, although their manipulation is still done through code. Visionaire on the other hand, doesn't require any programming and handles everything through the interface. The key difference however is that almost every entity in the game can hold its own set of conditions and values, instead of having a global pool of variables. While the way Visionaire stores its state is interesting, it also requires splitting that state between everything in the world which could become harder to manage. The distinction between a condition and value also adds one more concept that could be avoided by keeping both in the same category.

Solution: While the encapsulation provided by Visionaire in having variables stored inside any of the game's entities can be useful, it can also make the task of managing data more complicated because it ends up being split among an unbound number of entities. Therefore, Nostalgia Studio will opt for the most rudimentary solution which is having a single global pool of boolean values, known as flags.

4.3.3 Dialogue

Almost every graphic adventure game allows the player to engage in conversation with the non-playable characters that populate the game world. During these conversations, the player is usually presented with a list of responses to choose from, and the game can have different outcomes depending on what choices are made.

Table 11 - Dialogue

	AGS	**Wintermute**	**Visionaire**
Create dialogue	Right-click on "Dialogs" node. Uses a different but more simple scripting language than regular actions.	Program the dialogue sequence manually in the character's script file, using loops and conditionals, along with the AddResponse and GetResponse functions.	Go to the character and add a new entry in the dialogue tab. Dialogues are basically a tree of dialogue parts, and there's an interface to help building them.
Start dialogue	In a script, address the dialogue by id and call Start on it.	Put all of the dialogue code in a function inside the character's script, and call it when needed.	Add an action part of type "Start dialog" pointing to the dialogue you want to start.
End dialogue	Use the stop command inside the dialogue.	Manually exit the function on the code.	Select a property on the dialogue part that ends the dialogue.
Branching state	There are commands to disable dialogue options. More complex behavior must be coded.	Everything must be programmed manually.	Individual dialogue parts can be made to appear only when a certain condition is true.
Triggering actions	Add script code inside dialogue but indent it.	Add them directly in the code where needed.	Each dialogue part can be linked to an action.

Comments: On one end of the spectrum we have Wintermute which does not provide any specialized support for dialogues and ends up requiring every conversation to be coded manually. This approach is obviously extremely flexible, but requires the same boilerplate code to be repeated every time a normal conversation should take place. On the other end there's Visionaire which organizes dialogues as a tree of choices and provides an interface for editing these trees, without a single line of code having to be written. Finally there's AGS which is somewhere in between, because although it also requires scripting, it uses a separate simplified language created just for dialogue, which is significantly simpler than the regular scripting language used by actions. There's also an interface for adding new entries to the dialog and controlling some of its options.

Solution: Nostalgia Studio will follow the approach used by Visionaire and provide a specialized interface for editing dialogue trees.

4.4 Summary

To summarize all of the analysis performed in this chapter, the following guidelines will be followed when developing Nostalgia Studio:

- **Rooms** - Rooms will be divided into two layers, one for background and one for foreground, and users will be able to drag and drop any number of images into these two layers, as well as translate, rotate, scale and reorder them as needed.

- **Navigation** - Walkable areas will be created by drawing a polygon with the shape of the floor. Obstacles will be represented by erasing parts of the floor. Pathfinding will use the points-of-visibility technique, but unlike the other engines, will construct the underlying graph without any additional aid from the user.

- **Portals** - Portals will be created by drawing a polygon region on the room, and either adding a teleport action when a character enters, or using a built-in portal type which only requires the destination to be chosen.

- **Objects** - Objects will be created within the rooms interface, and placed either on the background or foreground layers, or set to be dynamically sorted along with characters. This sorting will be performed based on the object's origin which will be marked by a cross when defining the object's graphics.

- **Items** - Items will be treated independently from objects and have their own section in the editor.

- **Characters** - Characters will be able to have multiple costumes, each costume being a grouping of animations for each possible state and direction of the character.

- **Actions** - Action sequences will be created by choosing and chaining actions from the editor's interface, without any need for code to be written by the user. It will be possible to add actions to most of the game's entities, such as rooms, characters, objects, regions and items.

- **State** - State will be kept globally, in the form of a collection of boolean values known as flags. There will be actions to branch execution depending on the current state of a flag.

- **Dialogue** - Dialogue will be stored as part of the characters to which they belong, and will be organized as a tree of choices with its own interface for editing.

5 Architecture and Design

When looking into the target audience for *Nostalgia Studio*, it is clear that there are two very distinct groups of users: those who will be using it to create games, and those who will be playing the completed games. Nostalgia Studio handles this distinction by providing two separate applications: the *Nostalgia Editor* aimed at the first group, and the *Nostalgia Client* aimed at the second group. In order to reduce code duplication and improve the organization of the framework, both the editor and the client work on top of a third component known as the *Nostalgia Engine*, which contains all the shared functionality [Gregory, 2009]. Also, although the client can be executed independently from the editor, the editor does take advantage of the client in order to allow its users to preview the games they are creating. To summarize, *Nostalgia Studio* is composed by three separate components:

- **Engine** - The *Nostalgia Engine* is a dynamic library that provides most of the functionality required by the editor and the client. In other words, it does most of the heavy lifting for *Nostalgia Studio*. It is divided into two layers, one for low-level game-agnostic features (such as graphics, audio, input, networking) and one for high-level game-specific features (such as the gameplay, actions, rooms and characters).

- **Editor** - The *Nostalgia Editor* is the application used to create new games in *Nostalgia Studio*. It is based around a drag-and-drop interface and strives to insulate the user from the implementation details that are usually required to create a game. Its goal is basically to create content that the engine and client knows how to execute, and to provide a way to edit that content.

- **Client** - The *Nostalgia Client* is the application used to play games created with *Nostalgia Studio*. Most of its features are actually implemented in the *Nostalgia Engine*, and the client only behaves as a front-end for the player to have access to those features. It is also used by the editor as a previewer when creating games.

The figure below demonstrates the relationship between each of these components. Notice that the engine is internally divided into two separate layers, labeled *Core* and *Adventure* respectively. More information on this organization will be provided in the next section. Notice also how the relative sizes of each component varies, in order to reflect the fact that almost all of the functionality is provided by the engine, while the client does little more than act as a front-end for the engine.

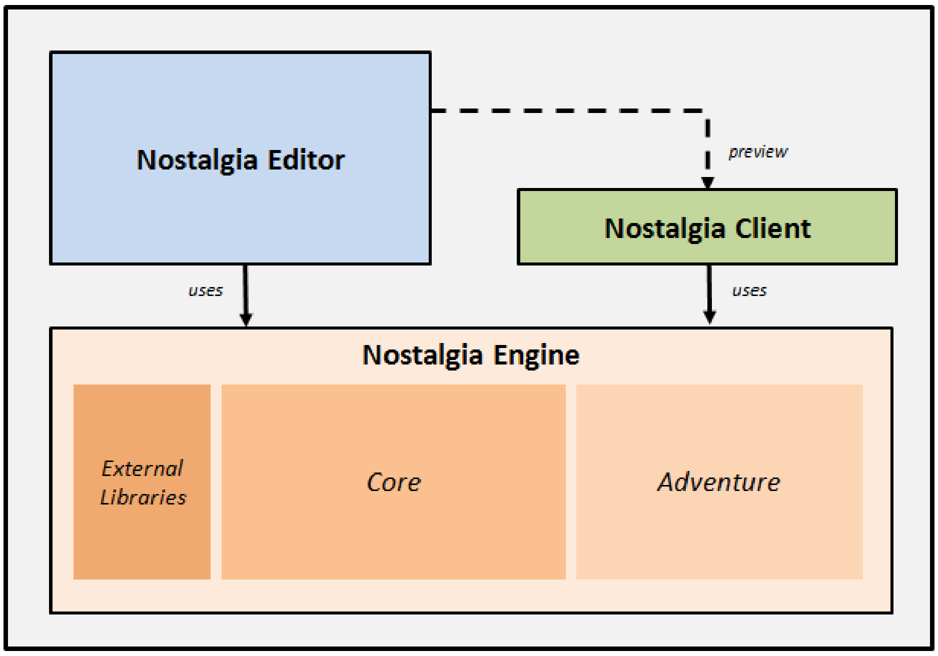


Figure 47 - Nostalgia Studio Architecture

5.1 Engine Architecture

Nostalgia Engine is the component that provides most of the functionality present in *Nostalgia Studio*. The internal architecture of the component is represented in the following figure.

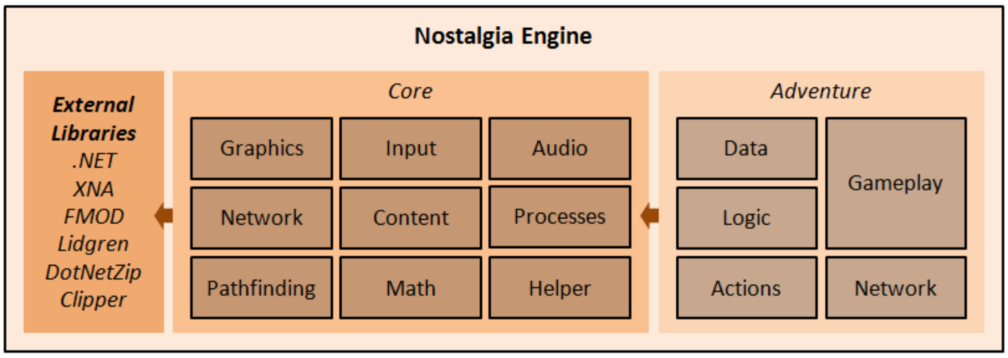


Figure 48 - Nostalgia Editor Architecture

The most prominent detail in the engine architecture is that it is divided into two layers. Also, Notice the arrows in the diagram which represent the direction in which these layers interact with each other. The main goal of each of these layers is described below.

- **Core Layer** - Provides general game-agnostic functionality which might be useful for almost any game. This layer relies on several external libraries for support of low-level operations such as graphics, audio playback and networking.

- **Adventure Layer** - Adds functionality that is specific to the implementation of graphic adventure games. This layer is built on top of the core layer and does not communicate with any of the external libraries directly.

Each layer exists in separate library files (i.e. separate DLLs) which are included as references by both the engine and the client. This separation was extremely helpful in reducing dependencies between modules, and making the code more manageable and reusable [Martin, 2008].

5.1.1 Core Layer Architecture

Despite none of the features present in the core layer being directly related to the gameplay portion of graphic adventure games, this is still where most of the processing in *Nostalgia Studio* takes place. The functionality provided in this layer is extensively used by the rest of the application, and covers a very large spectrum of different technical areas.

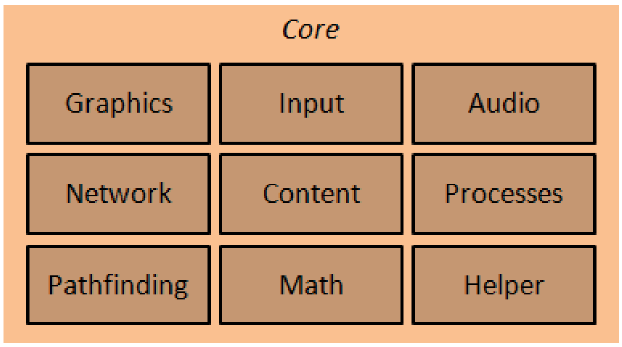


Figure 49 - Modules of engine core

The figure above shows that the engine core is organized into a series of modules, each being implemented as a separate namespace. These modules are designed to be fairly independent from each other, except for a few minor exceptions.

Graphics Module

The *Graphics* module is responsible for all the rendering in the application. This module implements a retained mode renderer, where all the data required to render an entity to the screen is stored inside of a separate object, known as a *Primitive*. This means that instead of the game entities knowing how to draw themselves, they delegate that work to a primitive object from the *Graphics* module. This object is then automatically rendered by the engine each frame without direct interference from the game entity. The engine supports five basic types of primitives:

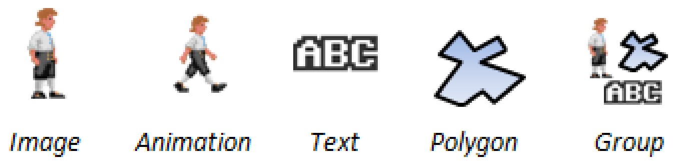


Figure 50 - Types of graphical primitives supported by the engine.

But although it is possible to render primitives independently from each other, it is far more common for an entire scene to be rendered at once. The *Graphics* module contains a custom scene graph data structure [Akenine-Moller, 2008] known as the Stage for this purpose. The stage is divided into layers, has a controllable camera, and interacts with the *Input* module in order to handle mouse interactions with the primitives.

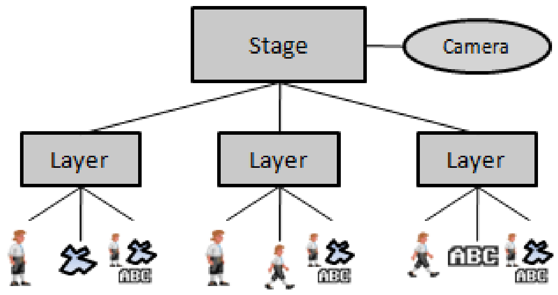


Figure 51 - The scene graph data structure used by the engine.

Besides being capable of rendering primitives there are also other features in the graphics module such as:

- Resolution independence rendering
- Rendering graphics with a retro look
- Rendering of geometric shapes

Input Module

The *Input* module is just an extension to the already existing Microsoft XNA Input Library. The most important feature it adds is the capability to distinguish between when a key was pressed only once from when it is being continuously held down. Other features include detecting which of the mouse buttons was pressed, and how much the mouse has moved since the previous frame.

Audio Module

The *Audio* module is responsible for both the loading and playback of songs and sound effects in the engine. It adds an abstraction layer on top of the FMOD API which simplifies the process of playing audio by introducing a distinction between:

- **Songs** - Sound files which are automatically looped and where only of them can be playing at any time. When one song ends and another one starts, the engine also takes care to cross-fade their volumes.

- **Sound Effects** - Sound files which are played only once and have no limit to how many instances can be playing at once.

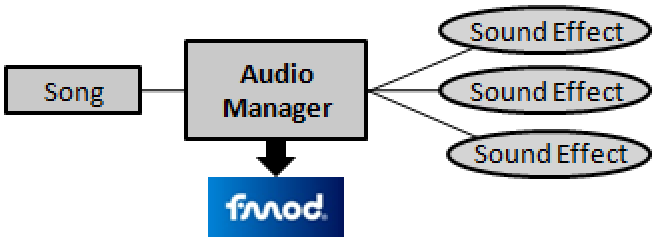


Figure 52 - Architecture of the Audio module

Network Module

The *Network* module is responsible for the exchange of data over a network. It adds an abstraction layer on top of the Lidgren Networking Library which simplifies all interactions down to three components:

- **Server** - Object that awaits connections from one or more clients. Can receive messages from any connected client, and can send messages to individual clients or to all clients at once.

- **Client** - Object that connects to a server. Once connected, it can both send and receive messages from the server.

- **Message** - An object that encapsulates any type of data to be sent over the network.

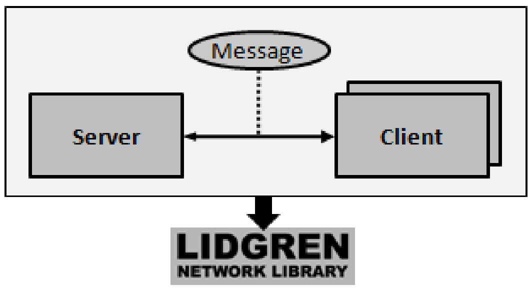


Figure 53 - Architecture of the Network module

Content Module

The *Content* module is responsible for loading images and fonts at *runtime*, unlike the existing Microsoft XNA Content Pipeline which only works at *design time*. It is also capable of loading files from inside ZIP files, courtesy of the DotNetZip Library. This feature is used extensively by the editor in order to package all the images loaded by the user into a single file.

Processes Module

The *Processes* module provides a framework for scheduling and executing sequences of actions. This system is actually responsible for controlling most of the gameplay in *Nostalgia Studio*. Since these actions execute over a certain period of time, the terminology *Process* is used to distinguish them from simple method calls which have immediate execution [McShaffry, 2012]. The system allows processes to be chained sequentially or in parallel.

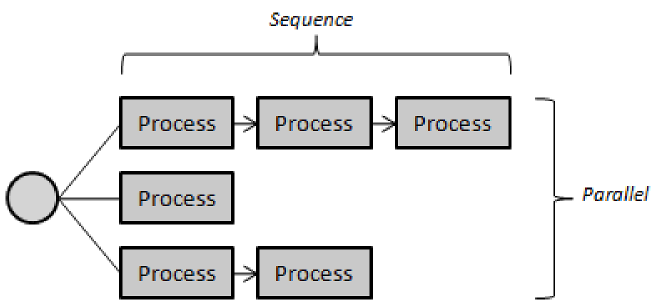


Figure 54 - Processes scheduled sequentially and in parallel.

Although most processes are implemented specifically for a particular game, there are a few general purpose processes provided in the module, such as the ability to suspend execution for a certain period of time, or an utility to convert any regular function call into a process.

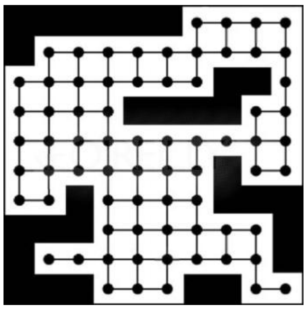
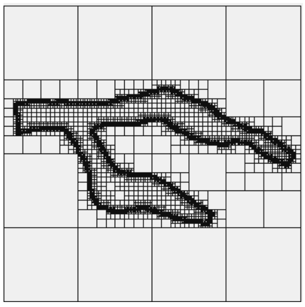
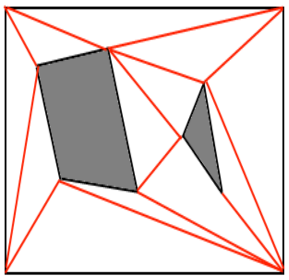
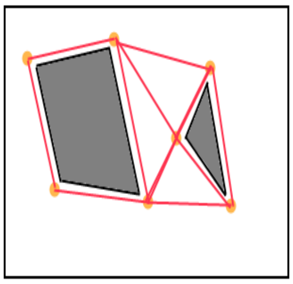
Pathfinding Module

The *Pathfinding* module is capable of calculating the shortest path between any two locations in the world. This process can be considered as two separate problems:

- Calculating the shortest path between a pair of nodes in a graph data structure. This is a very common computer science problem and there are many algorithms available to do it. The one used here is the A* search algorithm [DeLoura, 2000], and the graph uses an adjacency list representation [Dickheiser, 2006].

- Building a graph that accurately represents the game world. This is where game specific knowledge is required. This step boils down to partitioning the world into a finite series of nodes, with edges representing locations that are accessible from each other. There are however, many possible ways to do this, as shown in the table below [Rabin, 2002].

Table 12 - Space partitioning methods

Regular Grid	Partition the world into a regular grid of squares, and create one node for each unobstructed cell in the grid. Then add edges between adjacent nodes.	
Quadtree	Similar to the previous method, but instead of partitioning the space into regular sized cells, a recursive process of subdivision is used which allows finer cells to be created near the obstacles, while using coarser cells where detail is not important. This decreases the number overall number of nodes compared to the regular grid.	
Navigation Mesh	Partition the walkable area into a mesh of interconnected convex polygons. Each polygon becomes a node in the graph, and shared polygon edges become the edges of the graph. The most popular way to create a navigation mesh is by triangulation.	
Visibility Graph	One way to create a visibility graph is to add a graph node for each convex vertex of the obstacles in the world, and connect every pair of nodes that are in line-of-sight. Another approach is to have the level designer create the graph manually, which is usually referred to as a waypoint graph.	

The approach chosen to be implemented in Nostalgia Studio was the visibility graph, because it is the easiest to construct for any arbitrary shape, and the resulting paths are guaranteed to be as direct as possible. This also means that the user only needs to draw a polygon corresponding to the walkable area of the world, and the graph can be automatically generated from it.

The *Pathfinding* module encapsulates both of these operations into an object called the *Floor*, making it extremely simple to use. The adventure layer only needs to provide the module with a polygon of the floor, and everything else is handled automatically.

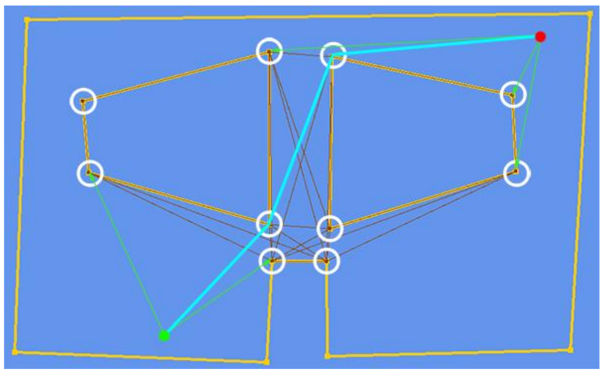


Figure 55 - Example of a visibility graph generated from a polygon

Math Module

The *Math* module is an extension to the math library provided by XNA. In addition to a few general purpose functions that are not present in the original library, most of the new functionality belongs to the field of geometry, including:

- Axis-aligned and oriented bounding rectangles
- Polygons and related operations
- Line segments and related operations

Helper Module

The *Helper* module aggregates all the functionality that does not fall within one of the other categories. This includes additional data structures and algorithms which are not provided by the .NET framework but are required for the implementation of other modules, and other general extensions to the existing functionality of .NET and XNA. It also provides several debug utility classes which were extremely useful during the development process, including:

- **Framerate Monitor** - Automatically outputs the number of frames per second at which the game is currently running to the top of the screen.
- **Logger** - A global class that can be used to output errors or other messages from anywhere within the application into an output file. Uses HTML for formatted output.
- **Messenger** - Similar to the logger, but instead of outputting to an external file, messages are printed directly to the top of the screen. It is also possible to define the duration of each message, before they are automatically removed.
- **Profiler** - Allows precise measurement of the execution time for a specific block of code.

5.1.2 Adventure Layer Architecture

Unlike the core layer which focuses mostly on general-purpose features, the adventure layer only provides functionality that is *directly* related to the implementation of graphic adventure games. It relies heavily on the core layer and ties all of the pieces together in order to create a working game. It is divided into the five modules represented below.

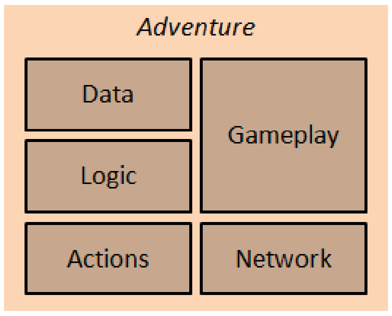


Figure 56 - Modules of the adventure layer of the engine.

Data Module

The *Data* module contains structures to store all of the data that defines a game. This is usually referred to as the game object model. All of these structures have in-built support for XML serialization which allows them to be written and read to the disk in a readable format. In a Model-View-Controller architecture, this module would be roughly equivalent to the model component, but without any rules attached to the data. This separation of data and logic is particularly useful for the editor, which never needs to touch any other module of the engine besides this one.

The game object model in *Nostalgia Studio* is composed by the following types of entities:

- **Rooms** - Any of the environments that the player can navigate in.
    - **Objects** - Any interactive or animated portion of the rooms.
    - **Regions** - Invisible areas on the ground that may trigger certain events.
    - **Floors** - The shapes that define where the characters can walk.
- **Characters** - Interactive entities that can walk, talk, carry items. There are playable characters and non-playable characters.
    - **Costumes** - A group of animations for each state that a character can be in.
    - **Dialogues** - Interactive dialogue sequences related to a character.
- **Items** - Anything that the characters can pick up and store in their inventories.

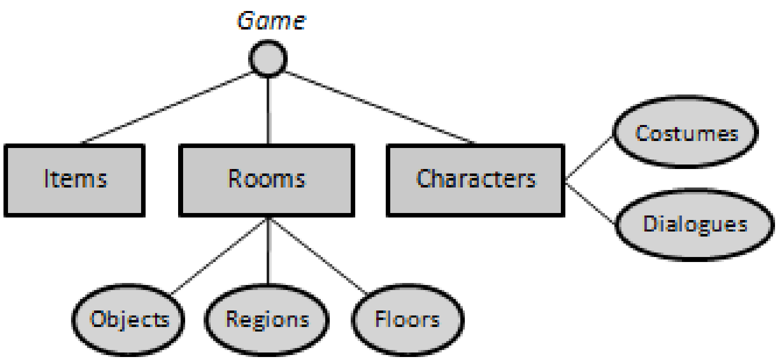


Figure 57 - The general game object model in Nostalgia Studio.

Logic Module

The internal structure of the *Logic* module is almost the same as the *Data* module, but instead of storing only data, it defines how each of the game entities should be displayed, and how they should behave. Every entity in the *Logic* module is designed in the same way, which consists of combining one object from the *Data* module with one primitive from the *Graphics* module, and adding all the operations that are expected from that entity (e.g. letting a character know how to talk or to walk).

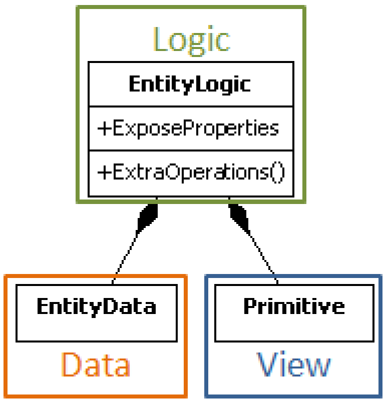


Figure 58 - General architecture of an entity from the Logic module

Once again there is a strong resemblance to the Model-View-Controller architecture, with the primitive acting as the View component, while the logic object acting simultaneously as the remaining portion of the Model component where rules and behavior are enforced, and as the Controller component which ties everything together.

Actions Module

The *Actions* module allows sequences of game events to be attached to almost every type of entity in the game. These sequences of game events are known as *Actions* and each of them is composed by a *Condition* and a list of one or more *Action Parts*:

- **Condition** - The condition of an action is *what* the player has to do for the action to be triggered. The available conditions can vary from entity to entity, and there are many different types of conditions, such as performing a command, using an item, or clicking the entity directly. It is also possible to limit this action to a specific subset of playable characters.

- **Action Part** - Actions are divided into several atomic steps known as action parts, with each part representing an individual change that should be applied to the state of the game. There are action parts for many different types of situations, such as making an object disappear, or teleporting a character to another room.

The logic for each action part is implemented as a separate object from the *Processes* module of the core engine. Then every time the player interacts with an entity, the engine searches its *Actions* for a matching *Condition*, and generates the correct process for each of its action parts.

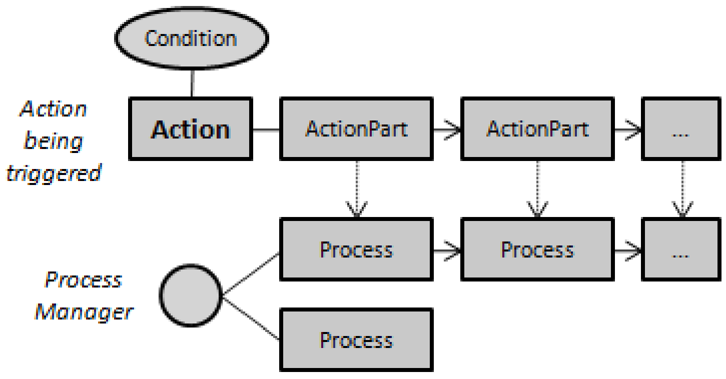


Figure 59 - The correlation between Actions and Processes.

Not every *Condition* is supported by every entity type The tables below list all of the conditions and the action parts supported by the engine.

Table 13 - Action conditions and respective entities

Condition	***Description***	***Room***	***Object***	***Region***	***Character***	***Item***
Command	*The player has performed a command on an entity.*		*X*		*X*	
Item	*The player has used an item on an entity.*		*X*		*X*	*X*
Click	*The player has clicked on an entity.*		*X*	*X*		
Enter	*The character has entered a room or a region.*	*X*		*X*		
Leave	*The character has left a room or a region.*	*X*		*X*		

Table 14 - List of action parts

Type	***Entities***	***Entities***	***Description***
Change Floor	*Room*	*Room*	*Change the floor that is currently being used by a room.*
Change Name	*Object Region Character Item*	*Properties*	*Change the name that is displayed when hovering the mouse over an entity.*
Change Interactive	*Object Region*	*Properties*	*Determine whether an entity should be interactive.*
Change Visible	*Object*	*Properties*	*Show or hide an entity.*
Change Direction	*Character*	*Properties*	*Change the direction a character is facing.*
Change Speed	*Character*	*Properties*	*Change the walking speed of a character.*
Change Animation	*Object*	*Animations*	*Change to a custom animation stored in an object.*
Play Animation	*Object Character*	*Animations*	*Play a custom animation stored in an object or character once.*
Change Costume	*Character*	*Animations*	*Change the costume that is currently being used by a character.*
Teleport To	*Character*	*Moving*	*Teleport a character to any other location in the world.*
Walk To	*Character*	*Moving*	*Make a character walk to another location within the same room.*
Speak	*Character*	*Talking*	*Make a speech bubble appear above a character's head with the specified text.*
Start Dialogue	*Character*	*Talking*	*Initiate the dialogue sequence associated with a character.*
Add Item	*Item*	*Item*	*Add an item to the inventory of a character.*
Remove Item	*Item*	*Item*	*Remove an item from the inventory of a character.*
Add Flag	*Game*	*Logic*	*Change the value of a global flag to true.*
Remove Flag	*Game*	*Logic*	*Change the value of a global flag to false.*
Conditional	*Game*	*Logic*	*Branch execution based on the value of a flag.*
Wait	*Game*	*Logic*	*Wait a certain amount of time.*
Play Song	*Game*	*Sound*	*Change the song that is currently being played in the background.*
Stop Song	*Game*	*Sound*	*Stop the song that is currently being played in the background.*
Play Sound Effect	*Game*	*Sound*	*Play a sound effect once.*

Gameplay Module

The *Gameplay* module is responsible for taking all of the entities defined in the previous modules, and making them work together as a whole. It is divided into a series of controllers, each responsible for a different portion of the game, such as the inventory window, or the command interface.

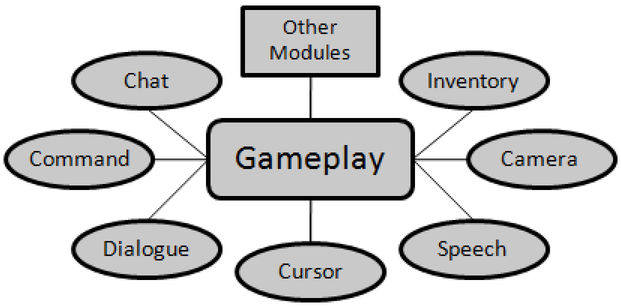


Figure 60 - Architecture of the Gameplay module

Another important task executed by the *Gameplay* module is to make sure that all the input that comes from the player, is not executed immediately but instead sent to the *Network* module first so that it can be shared with all other clients. This is the basis for all the multiplayer gameplay.

Network Module

The *Network* module of the adventure layer provides another level of abstraction on top of the *Network* module of the core layer. This abstraction was given the name of *Player* and its main goal is to process input from the user which is provided by the *Gameplay* module, and decide what do with it. The two possible outcomes are to ask the *Gameplay* module to execute the input immediately, or relay that information over the network so that every player can execute the same operation. This depends of course on the role of the player:

- **Single Player** - If the game is running in single player mode, neither a server nor a client is created. Instead, all user input is returned and executed immediately.

- **Multiplayer Client** - If the game is running in multiplayer mode, but the player was not the one who created the game, only a client instance is created. Then, whenever new input arrives, this input is first sent to the server. Once a response is returned, the user input is executed.

- **Multiplayer Host** - If the game is running in multiplayer mode, and the player was the one who created the game, both a server and a client are created on the same process. The server limits itself to receiving messages and broadcasting them to every client that is connected to him. The client works exactly the same as in the example above.

5.2 Editor Architecture

Nostalgia Editor is the application which allows users to create their own graphic adventure games in *Nostalgia Studio*. It relies heavily on the functionality of the *Nostalgia Engine*, but all of the interface was created by using Windows Forms and Microsoft XNA directly. The editor is divided into two logical halves:

- **Sections** - A *Section* is a portion of the editor which can be accessed directly from the section toolbar, and is usually responsible for a major task within the editor, such as editing a room or a character. Only one *Section* can be visible at a time (except for two types which open in a separate window) but the user can easily switch between them. *Sections* are implemented as a combination of one or more *Components*.

- **Components** - *Components* are the building blocks used to create each *Section*. Most are focused on a very specific task, such as editing actions or animations. They are also designed to be reusable in more than one *Section* when needed. Some components can also appear in a separate window.

Unlike the two big layers of the *Nostalgia Engine* which were stored in separate library files, *Sections* and *Components* both live within the same application, and the separation is merely conceptual.

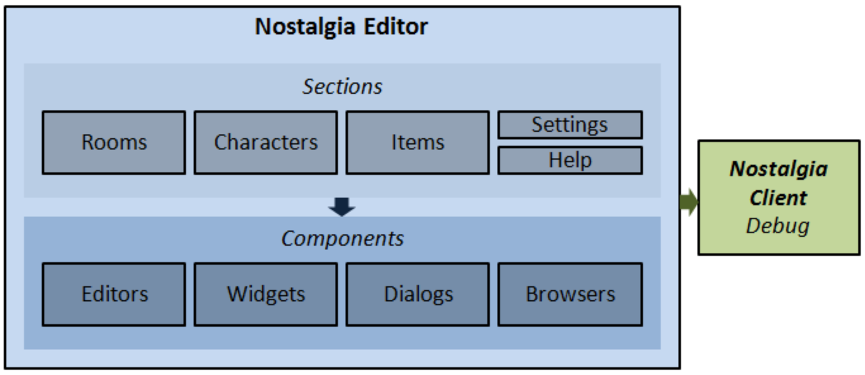


Figure 61- General architecture of the editor.

The *Nostalgia Editor* allows users to preview and debug the games that they are creating. But in order to avoid code repetition, this is accomplished by invoking the same *Nostalgia Client* application that is used by the players, but with a special flag that indicates that it should be executed in *debug mode*.

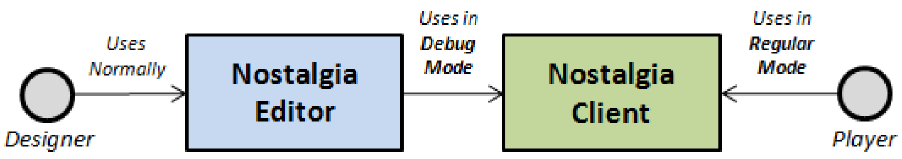


Figure 62 - Interaction between the editor and the client.

5.2.1 Sections

The editor is divided into five different sections. The first three deal with editing major entity types and can only be opened one at a time, while the last two appear in a separate window. All of these sections can be accessed through the section toolbar at the top of the application.

- **Rooms Section** - Create and edit all of the *rooms* in the game, including *scenario*, *objects*, *regions* and *floors*, along with their respective *actions*.
- **Characters Section** - Create and edit all of the *characters* in the game, including their *costumes*, *dialogues* and *actions*.
- **Items Section** - Create and edit all of the *items* in the game, along with their respective *actions*.
- **Settings Dialog** - Opens a new window that lets you change the general settings of the *game*.
- **Help Dialog** - Opens a new window showing help for the section currently being displayed.

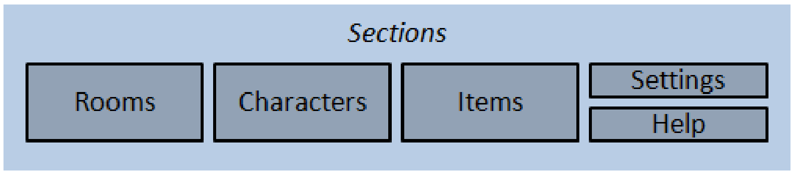


Figure 63 - The sections of the editor

The section toolbar also has a preview button, but this is not considered a section because all it does is launch the *Nostalgia Client* application with the correct parameters.

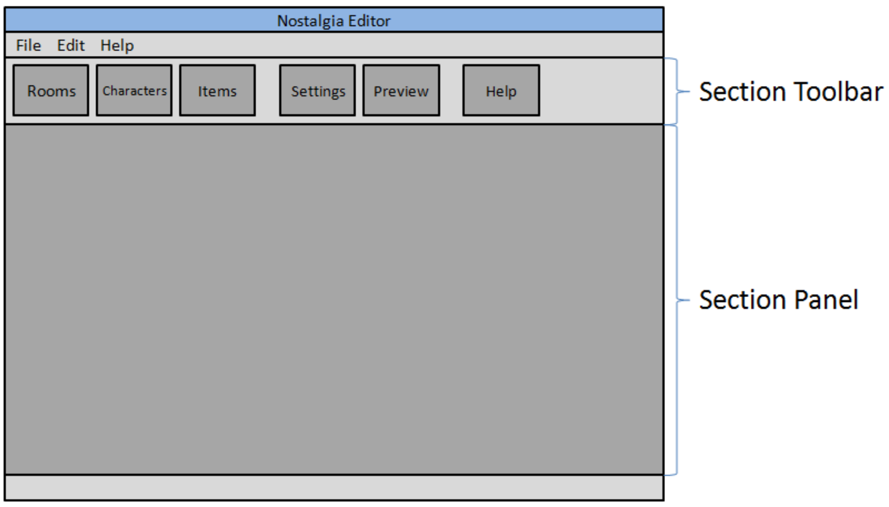


Figure 64 - Basic wireframe of section organization in the editor.

5.2.2 Components

There are a lot of *Components* in the *Nostalgia Editor*, and each is designed to fulfill a different task. Most of the work performed by the *Nostalgia Studio* happens inside of some *Component*. *Components* are not only combined and used by each of the four main *Sections*, but some of them are also used by other *Components*.

Although there is no common architecture that correlates all of the *Components*, they can generally be sorted into one of the four categories below:

- **Editors** – Components that allow the user to create or modify a specific portion of the game, such as a room or a character.
- **Widgets** – Small components with a very specific goal, that serve as the building blocks for larger components.
- **Dialogs** – Components that appear on separate window and require the user to give his feedback before returning control to the main application.
- **Browsers** – Components that presents a set of data in a meaningful way and allow the user to select part of that data.

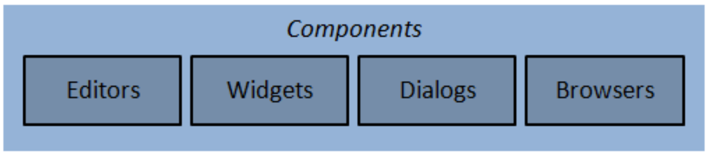


Figure 65 – Categories of Components in Nostalgia Editor

Editors

All of the *Components* in this category allow the user to create or modify some portion of the game, such as a room or a character. Most of them are complex enough that they require assistance from other *Components*.

- **Room Editor** - One of the most complicated *Components* which allows all of the content of a room to be created. This includes the scenario, objects, floors and regions. Because of this multiplicity, the *Room Editor* requires the use of many other *Components*.
- **Character Editor** - Another large *Component* which allows characters to be created, along with their costumes, actions and dialogues. Also requires the use of several other *Components*.
- **Item Editor** - Allows the creation of items, a process which is significantly easier than the previous two examples.

- **Action Editor** - Provides an interface for creating and editing actions. The interface is automatically generated from the contents of each action part, so new action parts can be added easily to the engine without having to modify the editor.

- **Dialogue Editor** - Provides an interface for creating and editing dialogue sequences. These sequences are presented to the user as a tree-like structure of speech lines and responses.

- **Stage Editor** - Lets the user interact and modify the *Primitives* contained in a *Stage* (see *Graphics* module of the *Engine Core*). Some of its features include translating, rotating, scaling, flipping, sorting, creating, deleting and copying *Primitives*. It is useful in many contexts such as creating the scenario for a room, determining the placement of objects, or editing animations.

- **Polygon Editor** - Lets the user construct or modify a polygon on a vertex to vertex basis. It supports two different modes of operation, one for adding and one for subtracting regions from the polygon. It is useful when creating floors and regions for the rooms.

- **Animation Editor** - A component for creating and editing animations, with a drag-and-drop interface and a timeline similar to Adobe Flash. It is useful when editing costumes for a character or when defining the animations of an object.

Widgets

Unlike *Editors* which are generally large and capable of performing many different tasks, *Widgets* are small and reusable components with a very focused goal. Most of them are generic and could even be used in some other applications.

- **Compass** - Lets the user select one of the four cardinal directions by clicking with the mouse on the sides of a compass image.

- **Timeline** - Displays a control for selecting frames in an animation sequence, which is very similar to the timeline in Adobe Flash.

- **XNA Control** - An important part of the editor which provides a way to use the XNA API for rendering inside of a Windows Forms application. It is used for most of the graphical rendering in the editor.

- **Canvas** - Renders a canvas with a look inspired by *Adobe Photoshop*, where a checkered pattern is used to represent transparency, and anything outside the canvas is painted dark gray.

- **Multiple Pages Panel** - An extension of a regular tab control where the tab handles are not visible to the user, but can be switched programmatically.

Dialogs

Dialogs are components that appear in a separate window, and require a response from the user before returning control to the main application. Some of these *Dialogs* are simple wrappers around existing *Components*, while others are designed from scratch to run on a separate window.

- **Location Picker** - Allows the user to pick a location inside one of the game rooms. Optionally, it can display a list of all the rooms in the game, allowing the user to select both the room and the location at the same time.
- **Sprite Cropper** - Allows the user to define the boundaries of any sprite. Cropping can be performed by manually dragging the corners of the crop area, but there's also an option to automatically divide the image into a regular grid, and pick one of the cells to be used.
- **Object Animations Editor Dialog** - Shows a new window with a list of animations for the selected object, and an animation editor for modifying them.
- **Action Editor Dialog** - Wraps the regular action editor so that it can appear in a separate window.
- **Settings Dialog / Help Dialog** - Although they were described earlier as being treated as sections because they are accessible directly from the section toolbar, internally they are constructed just like any other component.

Browsers

Browsers are components which serve to display a set of information to the user, allowing to select a subset of that information. Some *Browsers* also incorporate the option to add, remove, reorder, or rename entries in that set.

- **Generic Tree** - An extension of a tree view control which maps to any collection in the game object model and allows objects to be created, renamed, reordered or removed.
- **Room Tree / Character Tree / Costume Tree / Action Part Tree** - Specialized tree view controls providing access to other parts of the game object model which the generic tree cannot handle, as well as adding custom behavior.
- **Image Browser** - Shows a set of image thumbnails organized into folders which the user can browse, select, and modify. Also allows new images to be imported.
- **Sound Browser** - Shows a set of sound clips that the user can browse, select, modify and preview. Also allows new sounds to be imported.

5.3 Client Architecture

The *Nostalgia Client* is the smallest of the *Nostalgia Studio* components, but only because most of its functionality exists in the engine instead. It is the application used to play the games created with Nostalgia Studio. It is divided into the following three modules:

- **Game** - The heart of the client which is basically a XNA Windows Game application acting as a front-end for the features provided by the engine. Other than that, it is also responsible for loading the game data, and checking if the game should run in debug mode or not (used by the editor).
- **Lobby** - Provides an interface to create and join multiplayer game sessions. Only shown when the game is not running in debug mode.
- **Debug** - Provides an interface to directly control the game state in ways that would otherwise not be possible. Only shown when the client is running in debug mode.

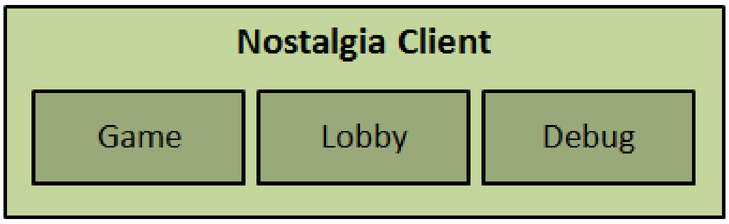


Figure 66 - Architecture of the Nostalgia Client

Like described above, whether the lobby or the debug interface are shown depends on the mode in which the client is running. This can be controlled by a command line argument when running the application. The flowchart below shows the general execution flow since the application starts, until the game is running in either single player or multiplayer mode.

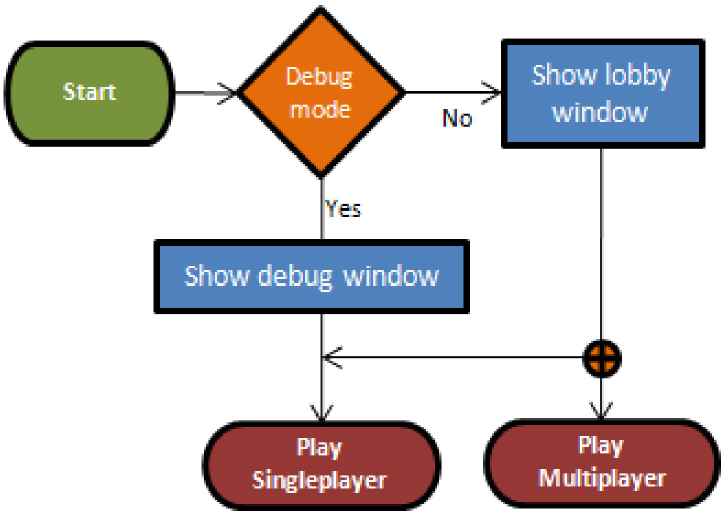


Figure 67 - General client execution flow

5.4 Multiplayer Design

Concluding this chapter will be a discussion about the design of the multiplayer portion of *Nostalgia Studio*, for which a few decisions had to be made. In particular, the four main points that required consideration were:

- How can players collaborate in the game?
- How many players can be playing simultaneously?
- Will all the players have the same role?
- Is there any room for variation in the gameplay?

5.4.1 Types of collaboration

The concept of *collaboration* boils down to having multiple parties working together to achieve a common goal. Collaboration is frequently employed in multiplayer games where players work together instead of against each other.

But collaboration can also happen in single player games. For instance, the graphic adventure genre is single player by nature, but that did not prevent other people, such as family or friends, from also watching the game and give their own suggestions to the player. There's no reason why this type of collaboration could not be adapted to an online multiplayer game environment.

This leads us to distinguish between two types of collaboration in multiplayer video games:

- **Active Collaboration** - This type of collaboration occurs when players possess direct control over the game world. This is usually accomplished by giving the player control over a character or avatar within the world, although in some cases the game world can be modified directly. In this model, players collaborate by actually performing the *actions* they deem fit for each occasion.

- **Passive Collaboration** - This type of collaboration occurs when players exchange *thoughts* and ideas with each other, but don't have direct control over the game world. This exchange of ideas can occur, for instance, in a chat window or over a VoIP connection. It also requires having at least one player with an active role in the game, to put into practice the ideas suggested by everyone.

When implementing active collaboration into a game, it is also a good idea to design the gameplay so that it *requires* collaboration. In a graphic adventure game this could be implemented for instance, by giving the ability to solve different puzzles to different characters, so that everyone is needed to progress in the game.

5.4.2 Number of players

Another aspect that needs to be taken into consideration, is the expected size of our game environment. There are many multiplayer online games that handle hundreds of players simultaneously, but most of them have huge open worlds for the players to inhabit without interfering with each other negatively.

On the other hand, graphic adventure games are typically made up from small interconnected rooms with a large density of detail. Imagine a hundred players sharing one of these rooms while trying to solve a puzzle, and most of the benefits of collaboration are soon to fly out of the window. For the usual size and scope of a graphic adventure game, an upper limit of 3 or 4 active players would be a much more reasonable limit.

But that's mostly a problem with an active collaboration scheme, due to our characters' physical boundaries overlapping each other and/or the environment, making the whole experience too cluttered and confusing.

With passive collaboration though, players do not occupy a physical space in the game world. This means that more of them could be playing at the same time without causing problems to the gameplay experience (although there's still a risk of confusion due to information flooding). The drawback is that passive collaboration is arguably much less interesting than active collaboration, and would not be captivating enough to satisfy the average player.

5.4.3 Player roles

In order to cope with this situation, it's necessary to find a middle term between a fully active collaboration scheme and a fully passive collaboration scheme. The approach taken by *Nostalgia Studio* is to provide both types of collaboration simultaneously, which is possible by identifying two different types of players:

- **Actor** - A player that has direct control over one playable character, and is free to act by himself within the game. He is free to communicate with all other players in the same game session by using the in-game chat.

- **Spectator** - A player that can watch the course of the game from the point of view of any *Actor*, being able to decide at any point which *Actor* he'd like to follow. He can also communicate with all other players in the same game session by using the in-game chat.

Then, when creating a game with *Nostalgia Studio*, the developer specifies the exact amount of actors required to play it. On the other hand, the maximum number of spectators is arbitrary, and can be left for the host of each game session to decide.

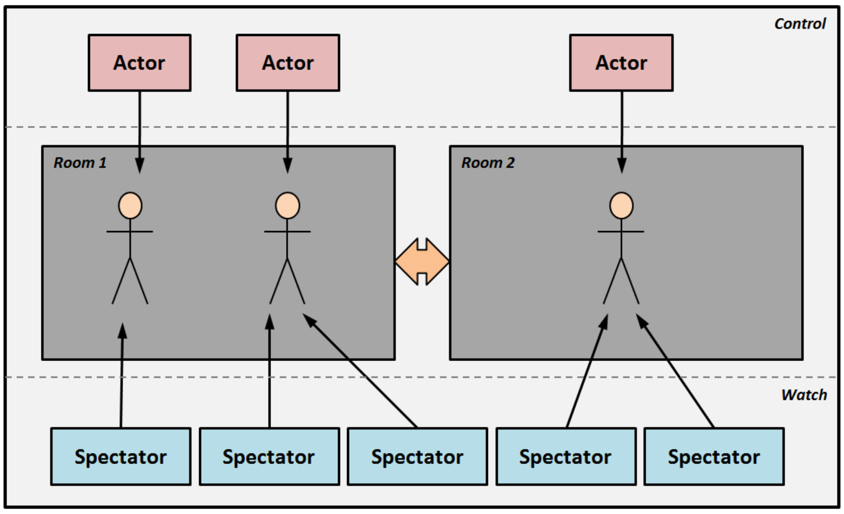


Figure 68 - Actors and spectators

5.4.4 Game Modes

By varying the control mechanism that decides who gets to be an actor and who gets to be a spectator, it is possible to create different types of game sessions. However, I should start by clarifying that by *host* I refer to the person who initiates the game session, and as such holds additional administrative rights.

This implies that in any of these game modes the hosting player is always capable of controlling the role allocation directly, or to expel offending players from the game session. What changes is what all of the *other* players are allowed to do. Here are some examples of possible game modes:

- **King** - In this mode only the host has the ability to decide who plays as an actor and who plays as a spectator. It is worth noting that being a host does not force him to be an actor. For instance, a teacher moderating a game session in a classroom may choose King mode and have control over who plays at each moment, without playing an active role in the game.

- **Tag** - In this mode any of the actors may choose to revoke their own responsibilities and pass them on to a spectator, without requiring the host's intervention.

- **Timed** - In this mode, everyone gets a turn at being an actor. At certain intervals, roles are cycled, and the next spectators in the list become actors while the actors go back to being spectators.

6 Implementation

Due to the size of Nostalgia Studio, it will be impossible to cover all of the implementation details in such a limited amount of space. Instead, this chapter will try to focus on the most important details of the implementation. For the engine, the inner workings of the most relevant modules will be described together with a few class diagrams, while for the implementation of the gameplay and the editor, screenshots of the application will also be shown. But to start the chapter, we'll see an overview of all the different technologies which were used to implement Nostalgia Studio.

6.1 Engine Implementation

As seen in the previous chapter, the engine is divided into two layers. Each of these layers was implemented as a separate project inside the Visual Studio solution, and both of them were set to compile as library files (i.e. as DLLs) instead of as applications (i.e. as EXEs).

6.1.1 Core Layer Implementation

Graphics Module

The heart of the Graphics module is the Primitive which is implemented an abstract base class from which each concrete type of primitive inherits. The base class provides a common set of features for all primitives, such as having a position and knowing how to draw itself. Then every specific type of primitive inherits from this base class to add its own implementation for rendering images, text, polygons or animations.

There is also a special type of primitive known as the group primitive, which is capable of containing other primitives within itself. The group primitive was implemented by applying the *composite design pattern* [GoF, 1995] to the class hierarchy. But the most important detail about this type of primitive is that it behaves like a node in a scene graph in the sense that all of its children's properties become relative to the parent, allowing complex primitive hierarchies to be created.

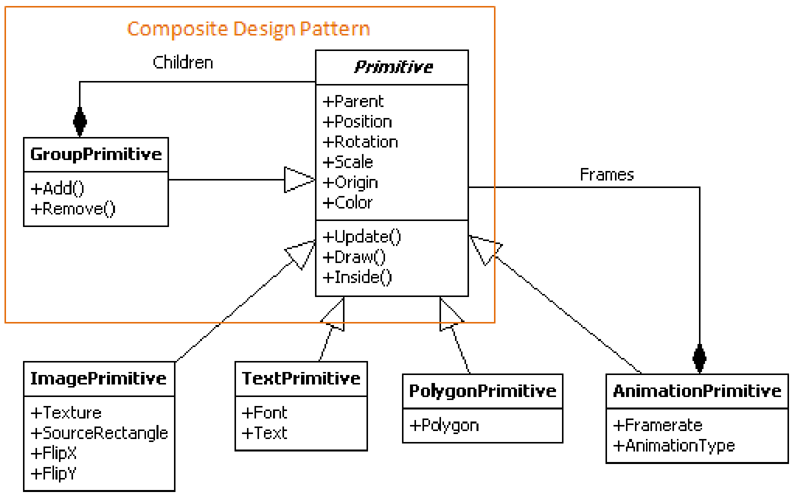


Figure 69 - Primitive class diagram

Because the group primitive already handles all of the parent-children transformation process, implementing a scene graph becomes extremely easy. The stage class is simply a list of layers, with each layer encapsulating a group primitive as its root. Most of the processing is then delegated to the group primitives. There is also a camera class associated with the stage which works by constructing a view transformation matrix which can then be applied by the XNA Graphics Library when rendering.

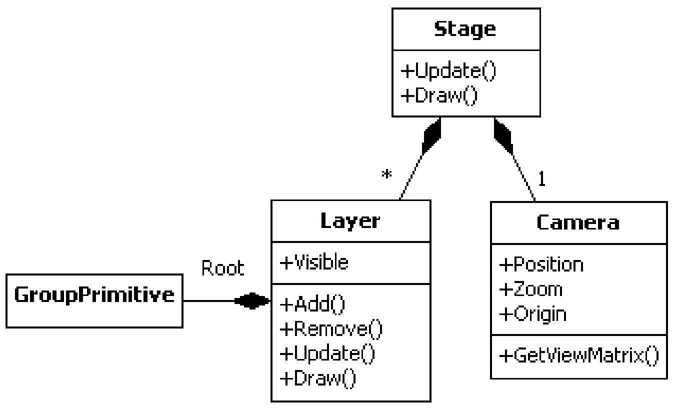


Figure 70 - Scene graph implementation

The engine achieves independence resolution in the games by letting the user define a virtual resolution for the game. Then a render target of the chosen resolution is created and everything is rendered to it first. Then at the end the contents of the render target are rendered to a fullscreen quad making it match whatever resolution the user is actually using. The aspect ratio of the game is preserved during this last operation by adding a letterbox or pillarbox as needed. Notice how the exact same portion of the room is visible in both images below despite the different window sizes.

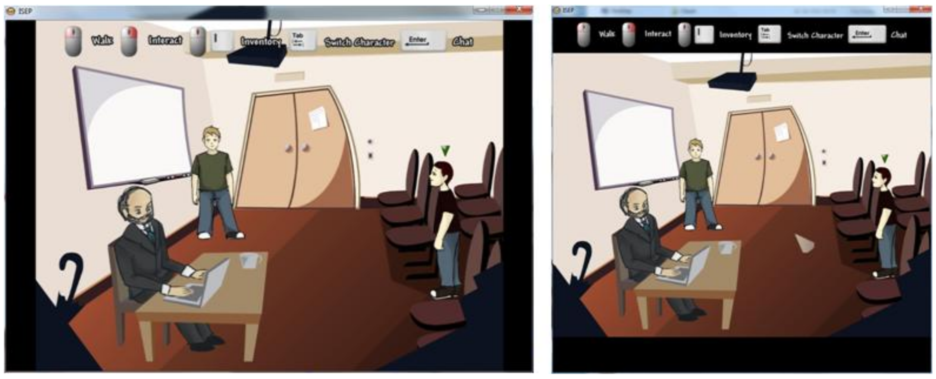


Figure 71 - Pillar box (left) vs letter box (right)

The retro rendering mode in Nostalgia Studio is achieved by setting the texture filtering mode to point when rendering. This works especially well when working with a very low virtual resolution, because it allows pixels in the images to stand out, as shown in the image below:

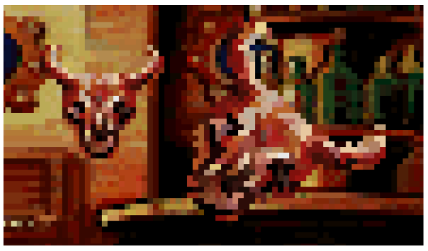


Figure 72 - Scene rendered using retro mode with a low virtual resolution

Audio Module

The audio module is composed by a single class known as the AudioManager which encapsulates and manages a System object and one or more Sound objects from the FMOD API. The interface is extremely simple providing one method to load new sound files, three methods to start and stop playing these sound files, and an update method which is used to service the FMOD audio system, and to implement volume fading between songs.

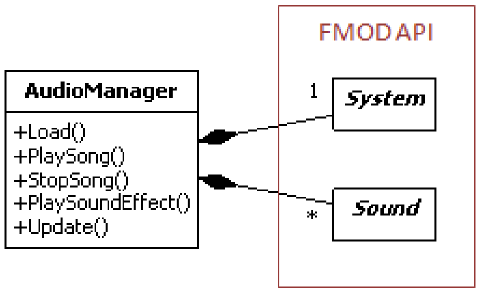


Figure 73 - Audio module implementation

Network Module

The Network module encapsulates some of the native classes from the Lidgren Network Library, and exposes a very simple interface based around an abstraction known as the message. Users can define new types of messages simply by implementing the IMessage interface, and the Server and Client classes will automatically know how to send or receive them over the network.

The message interface requires two methods that know how to read and write the message data into a network packet, and a unique type identifier which is required for the message object to be reconstructed on the destination. This reconstruction required the use of C#'s reflection features in order to determine the type of the message and recreate it at run-time.

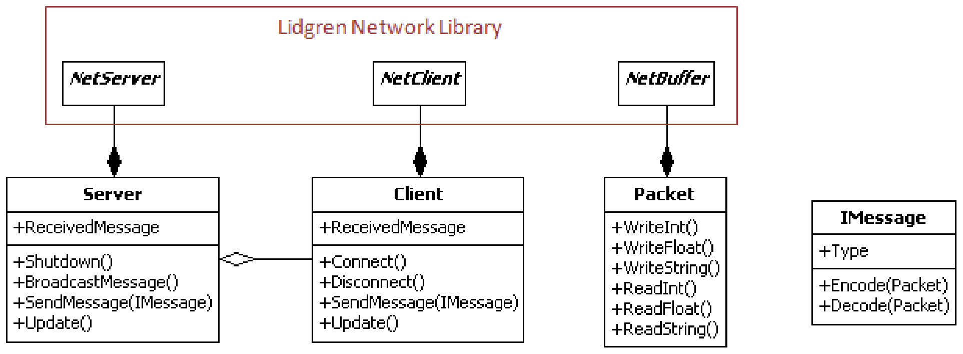


Figure 74 - Implementation of the Network module

Processes Module

The core of the Processes module is an abstract base class called the Process which is basically a variation of the *command design pattern* [GoF, 1995], in the sense that it encapsulates a request or operation inside of an object, but in this case that operation has a lifetime and gets executed over time instead of immediately. Concrete types of processes are then inherited from this base class [McShaffry, 2012]. The game itself provides most of the implementations, but there are also two generic types of processes provided directly in the engine:

- **Wait process** - Process that waits for a certain period of time before flagging itself as finished.
- **Action process** - Process that holds a delegate to a method. That method is executed once the first time the process is updated, and the process immediately exits afterwards.

But a single process is not too useful by itself. The real power of the system comes when we allow more than one process to be linked together and executed in a meaningful order, such as in parallel or sequentially. In order to remain as flexible as possible, the *composite design* *pattern* [GoF, 1995] was once again used here to represent the concept of a process which hosts other processes.

There are two types of composite processes in the engine. The first type is used to execute processes in parallel, which means that every process it contains gets executed every frame, independently from every other process. The second type creates a sequence of processes, but only the first process in the sequence gets executed each frame. The remaining processes need to wait for the first one to finish before their turn arrives. Furthermore, since parallel and sequence are processes themselves, they can be combined in order to create more intricate execution flows.

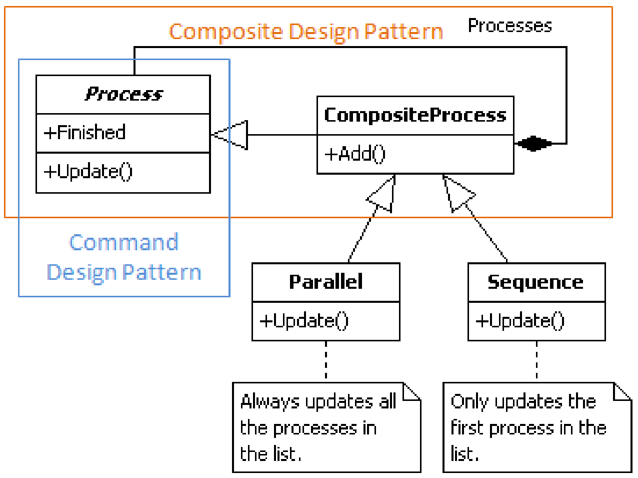


Figure 75 - Process module implementation

Pathfinding Module

The graph data structure used by the engine for pathfinding uses an adjacency list representation, where each node in the graph contains a reference to each of its neighbors. The graph is undirected so when an edge is created the connection is performed on both nodes involved. There is also no explicit object to represent an edge because there was no need to store a weight in this scenario (each node holds a position and the weight is always the distance between these positions).

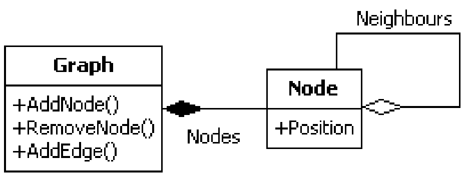


Figure 76 - Graph implementation

The pathfinding algorithm used was the popular A* search algorithm which performs a best-first search on the graph using an heuristic function that combines the cost to any given node with an estimate from the cost from that node to the goal. The implementation used was based on the one provided by Eric Lippert [Lippert, 2007] and can be found below:

```
List<Vector2> FindPath(Node start, Node end)
{
  // The set of nodes already evaluated
  var closed = new HashSet<Node>();
  // The set of nodes to be evaluated next, sorted by priority
  var queue = new PriorityQueue<double, Path>();
  // Add start node to the queue
  queue.Enqueue(0, new Path(start));
  // Repeat until there are no more nodes to evaluate or path found
  while (!queue.IsEmpty) {
    // Get next node with highest priority from the queue
    Path path = queue.Dequeue();
    // If node was already evaluated ignore it and skip ahead
    if (closed.Contains(path.LastStep)) continue;
    // If we've reached the destination end here and return the path
    if (path.LastStep.Equals(end)) return path.ToList();
    // Mark the current node as having been evaluated
    closed.Add(path.LastStep);
    // Enqueue neighbors of the node using the A* heuristic as priority
    foreach (Node n in path.LastStep.Neighbours) {
      Path newPath = path.AddStep(n, Distance(path.LastStep, n));
      queue.Enqueue(newPath.TotalCost + Distance(n, end), newPath);
    }
  }
  return new List<Vector2>(); // If no path was found return empty list
}
```

The Path data structure in the implementation is an helper class which basically behaves like a normal graph node, but also keeps a list of all previous nodes traversed to get there, making it easier to construct the complete path when the algorithm ends.

As described during the previous chapter, Nostalgia Studio uses a visibility graph representation which is built from the shape of the floor. The process of constructing the visibility graph is divided into two separate phases:

1. **Create Nodes** - For each vertex in the floor polygon, run a *concavity test* on the vertex, and add a new node to the graph containing the position of that vertex whenever the test returns true.

2. **Create Edges** - For each node in the graph, run a *line-of-sight test* against every other node in the graph, and add a new edge whenever the test returns true.

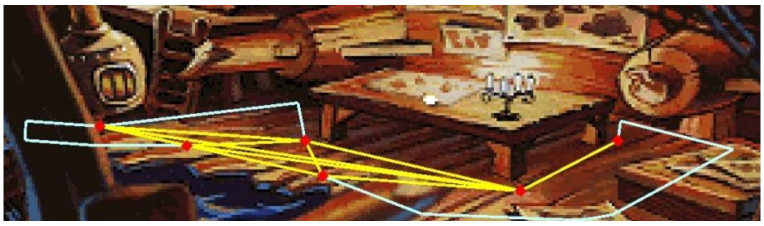


Figure 77 - Visibility graph created from the floor region

Another important detail that should be mentioned is that in order to run the A* search algorithm on a visibility graph, the ending points of the path must first be added to the graph as temporary nodes, and only removed after the algorithm ends. The process of creating these temporary nodes is exactly the same as the one described above.

The *concavity test* and the *line-of-sight* tests are both provided by the **math module**. The first one is a simple cross product calculation between the vertex and its adjacent ones. The line-of-sight test relies on a geometrical line segment intersection test, and its implementation can be seen below:

```
bool InLineOfSight(Polygon polygon, Vector2 start, Vector2 end)
{
  // Not in LOS if any of the ends is outside the polygon
  if (!polygon.Inside(start) || !polygon.Inside(end)) return false;

  // In LOS if it's the same start and end location
  if (Vector2.Distance(start, end) < epsilon) return true;

  // Not in LOS if any edge is intersected by the start-end line segment
  foreach (var vertices in polygon) {
    var n = vertices.Count;
    for (int i = 0; j < n; i++)
      if (LineSegmentsCross(start, end, vertices[i], vertices[(i+1)%n]))
        return false;
  }

  // Finally the middle point in the segment determines if in LOS or not
  return polygon.Inside((start + end) / 2f);
}
```

6.1.2 Adventure Layer Implementation

Data Module

The data module contains objects to store all of the state of the game. At its core there is a base class known as EntityData which guarantees that every unity gets a unique Id and has in-built XML serialization support.

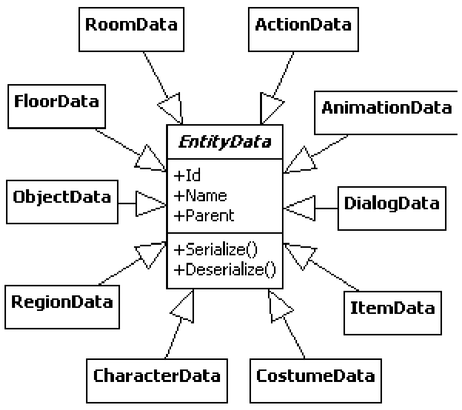


Figure 78 - Basic entity data implementation

All types of entities inherit from this base class and add their own data to it, forming the hierarchy represented below. The inheritance above was omitted from this second diagram in order to reduce the clutter. Some entities have also been marked with the Actions stereotype, which means that they are capable of holding actions within them.

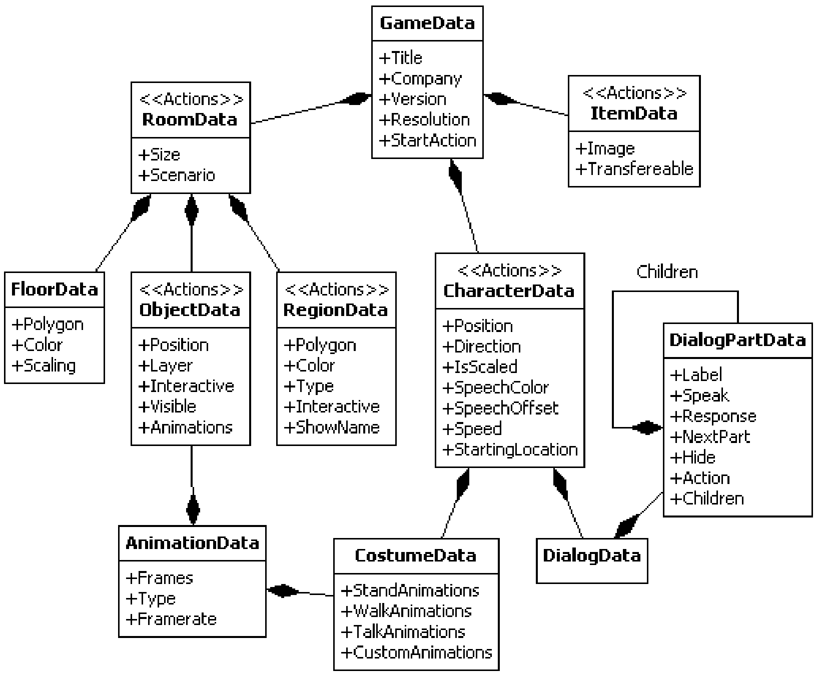


Figure 79 - Game object model hierarchy

Logic Module

The logic module takes every type of entity that supports actions and creates a new class for each of them. These classes bind the data together with a primitive for the view and all the operations that should be supported by the entity. The full hierarchy can be seen below.

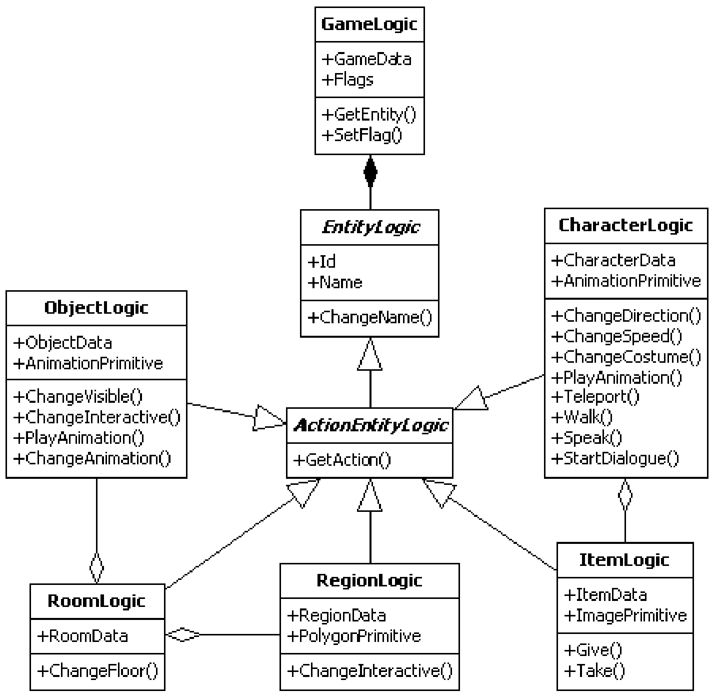


Figure 80 - Logic module implementation

Actions Module

The actions module contains both the data and logic related to actions. Every action is composed by a series of action parts. Each action part contains the same number of parameters, but a type property controls their meaning. Then the logic half consists of a class that takes an action as input and generates a sequence process containing one specialized process for each different type of action part.

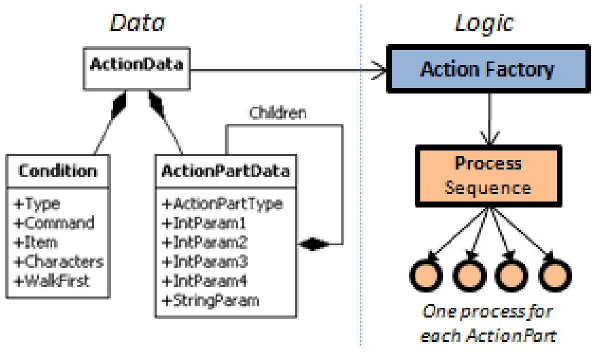


Figure 81 - Action module implementation

Gameplay Module

The gameplay module is divided into a series of controllers each responsible for a different portion of the game. All of these controllers are aggregated in a single controller known as the GameplayController. This controller is also responsible for managing all of the game state, and communicating user input to the network module. Below you can see the general architecture of the module, and an overview of what each of the controllers looks like.

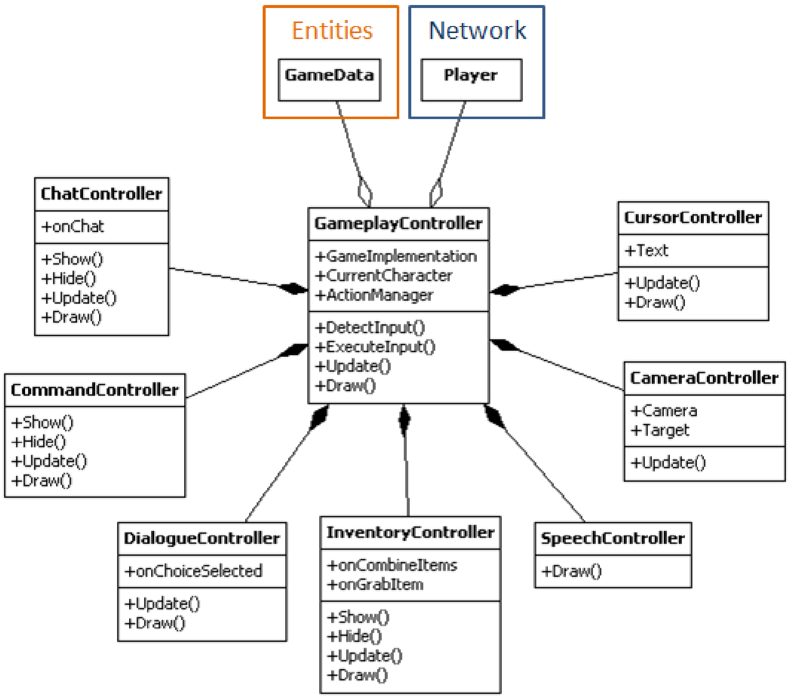


Figure 82 - All the controllers of the gameplay module

The inventory controller is responsible for drawing and handling interactions with the current character's inventory. When this controller is visible, all other types of interactions with the game are disabled.

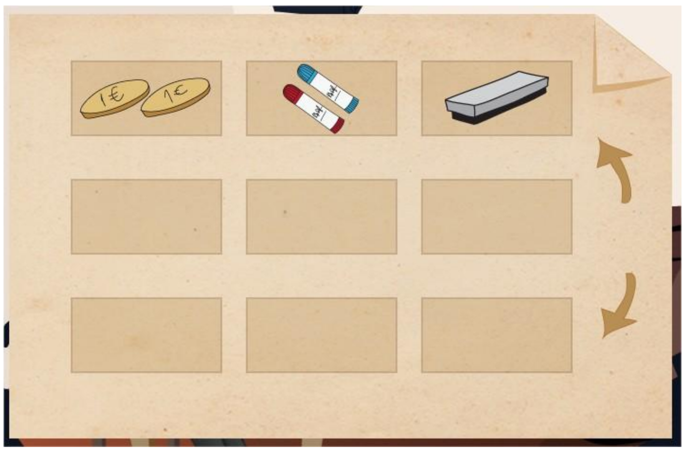


Figure 83 - Inventory controller

The dialogue controller is responsible for displaying a list of dialogue choices and allowing the player to select one of them. When this controller is visible, all other types of interaction with the game are disabled. Once a dialogue choice is selected, the controller automatically hides.

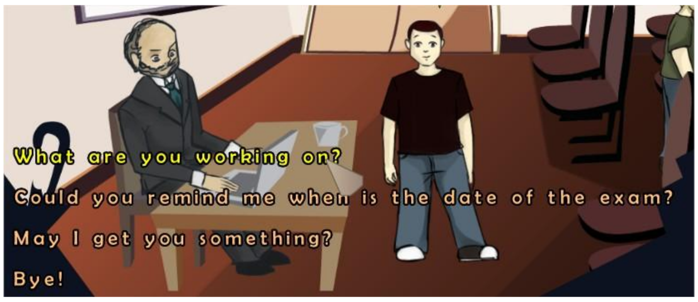

What are you working on?
Could you remind me when is the date of the exam?
May I get you something?
Bye!

Figure 84 - Dialogue controller

The chat controller is responsible for receiving text input from the user, which is then sent to every other player by the engine. It darkens the screen slightly and places a blinking placeholder symbol at the bottom of the screen to indicate that the game is currently in chat mode. When this controller is visible, all other types of interaction with the game are disabled.

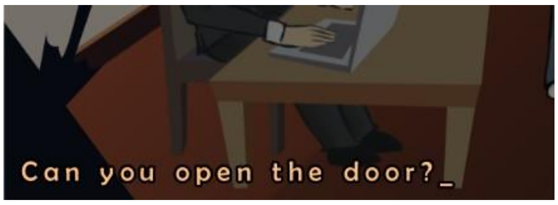

Can you open the door?_

Figure 85 - Chat controller

The command controller displays a circular formation of icons corresponding to each command that the player can perform, and allows him to select one of the commands with the mouse.

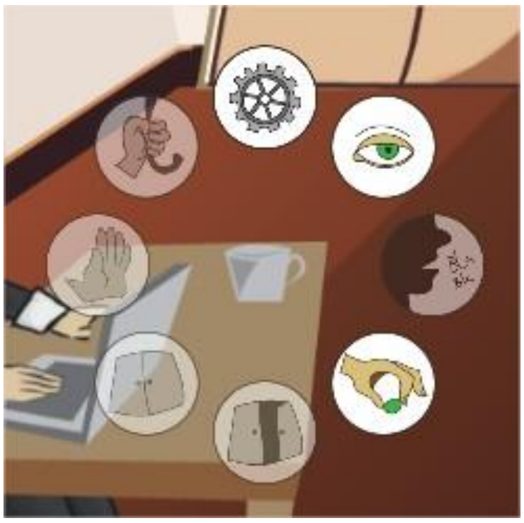


Figure 86 - Command controller

The cursor controller is responsible for drawing the mouse cursor at the correct position. It is also capable of displaying an item image and a text message on the cursor. The image is used by the engine when the player grabs an item from the inventory, and the text is used to display information whenever the mouse is over a game entity.

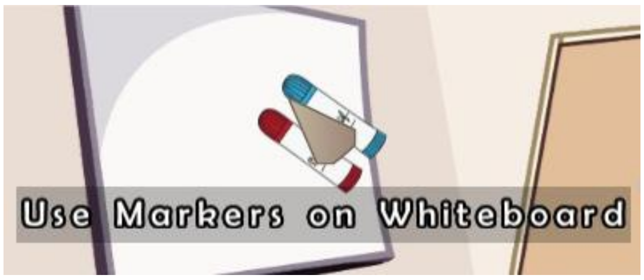


Figure 87 - Cursor controller

The speech controller is responsible for checking if any character is speaking in the current room, and drawing what they are saying to the correct location on the screen, with the correct speech color.

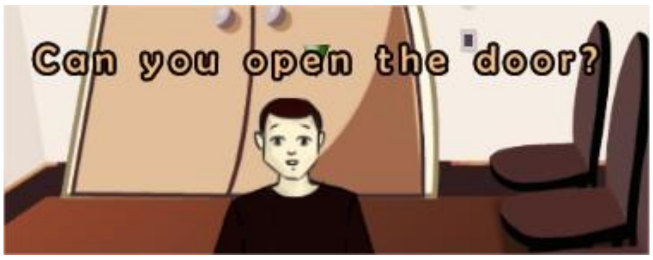


Figure 88 - Speech controller

The camera controller makes sure the camera is focused on the current character, and automatically scrolls the camera whenever the character reaches the edges of the screen, in order to try to get the character to appear at the center of the screen once again.

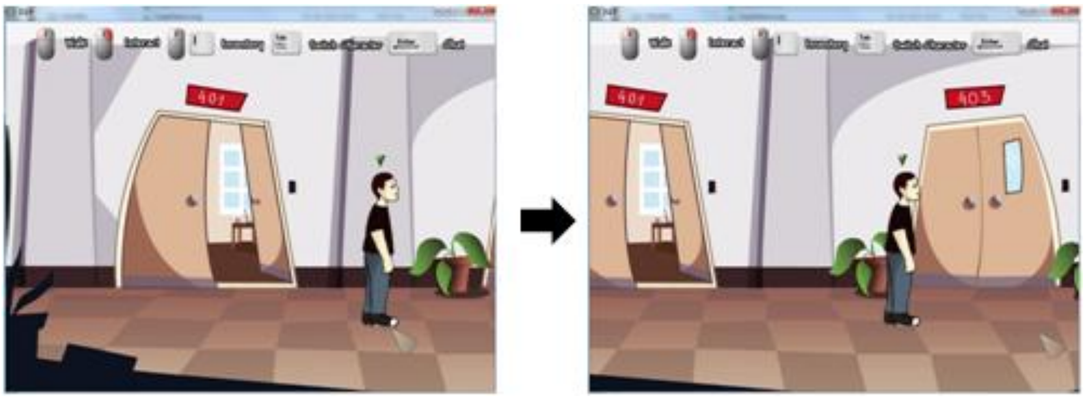


Figure 89 - Camera controller

Network Module

The main purpose of the network module is to communicate every single action that a player does to every other player in the game. In order to pass this information around, a special structure known as InputData was created which is capable of representing every type of user interaction within the game.

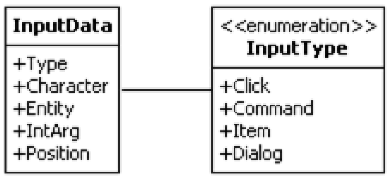


Figure 90 - InputData structure that encapsulates all user input

All of the communication code is stored inside of a player object which the gameplay controller holds a reference to. Then whenever some input is detected, that input gets sent to the player object first instead of being executed immediately. The sequence diagram below demonstrates this process, and how the player class handles the single player mode differently from the multiplayer mode.

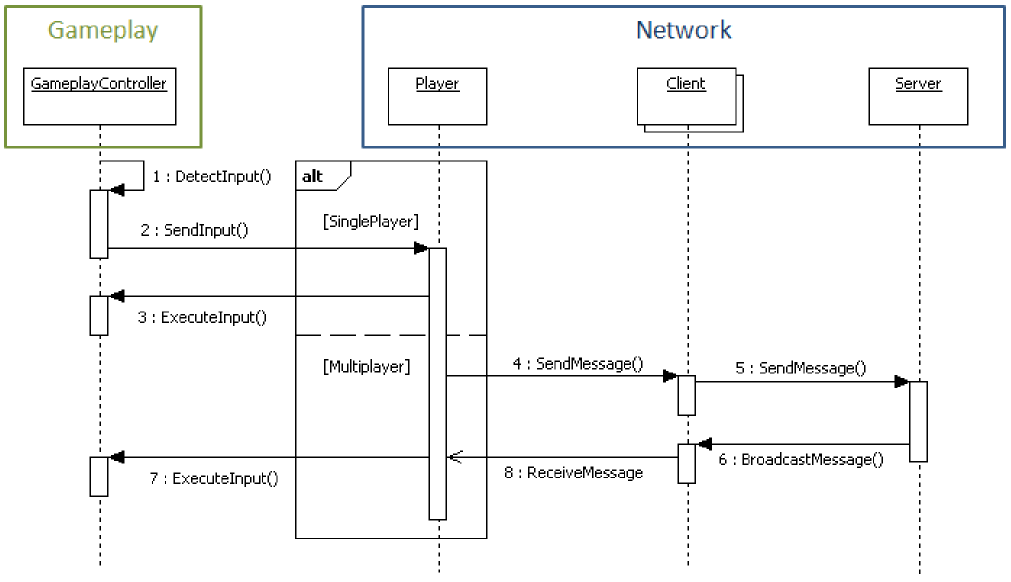


Figure 91 - Sequence diagram that shows network flow

The network module works with four custom types of messages:

- **Input Message** - Message containing all the details of a user interaction with the game.
- **Chat Message** - Message containing the text the player has written in the chat mode.
- **Ready Message** - Sent by the client when the game has finished loading.
- **Play Message** - Sent by the server when all clients have sent their ready messages.

6.2 Editor Implementation

The editor was implemented using Windows Forms as the main GUI Toolkit, which is flexible enough to allow custom controls to be developed [MacDonald, 2005]. However, it also uses Microsoft XNA to render some of the most graphical controls. This is not supported natively by Windows Forms, so a custom control had to be implemented. Unlike the previous section about the implementation of the engine which was a lot more technical, this section will be mostly focused on what the editor ended up looking like. For starters, here's the look of the section toolbar from which every other section can be reached.

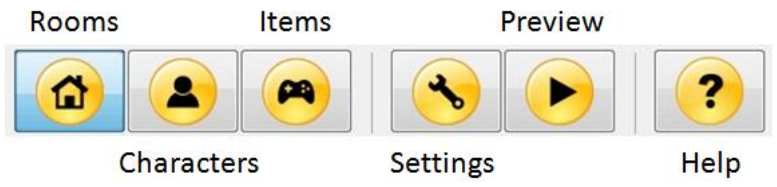
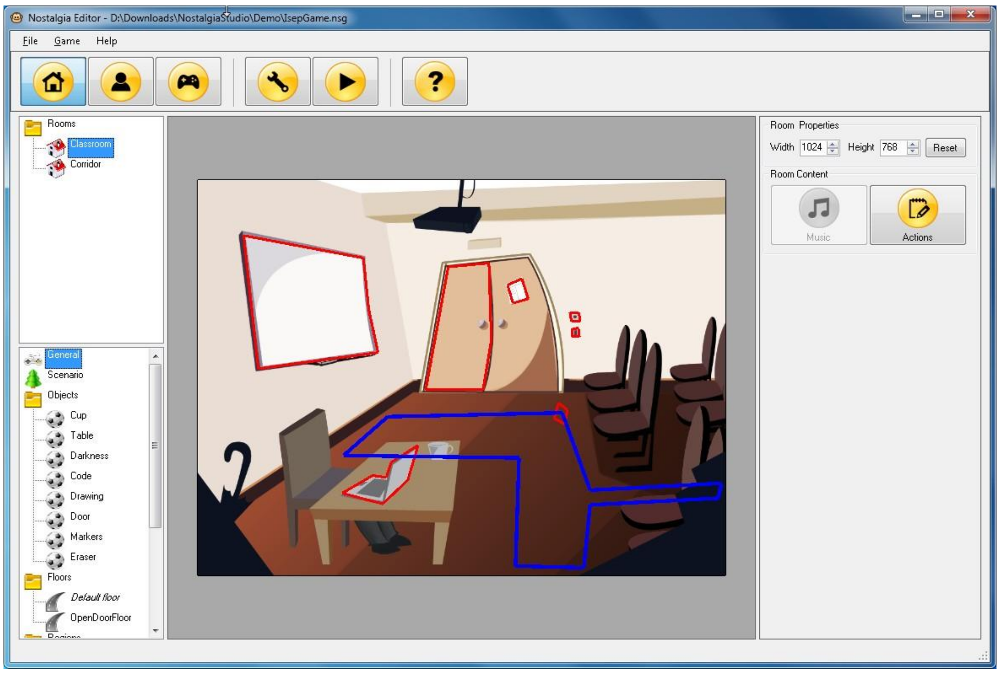


Figure 92 - Section toolbar

6.2.1 Sections

The first section shown when a new game is created is the rooms section, and it is also the most complex of all of them. The room tree on the bottom-left of the screen also allows objects, floors and regions to be accessed and modified. The room section makes use of several components, such as the stage editor, the image browser, the polygon editor and the canvas, among others.


Nostalgia Editor - D:\Downloads\NostalgiaStudio\Demo\IsepGame.nsg

Room Properties: Width 1024, Height 768

Figure 93 - Rooms section

The second session on the toolbar is the characters section. This section is also very complex since not only does it allow the general properties of a character to be changed, it also takes care of his placement inside the rooms, his costumes and animations, and his dialogues. All of these tasks are handled by separate components, and are all combined inside of the character section.

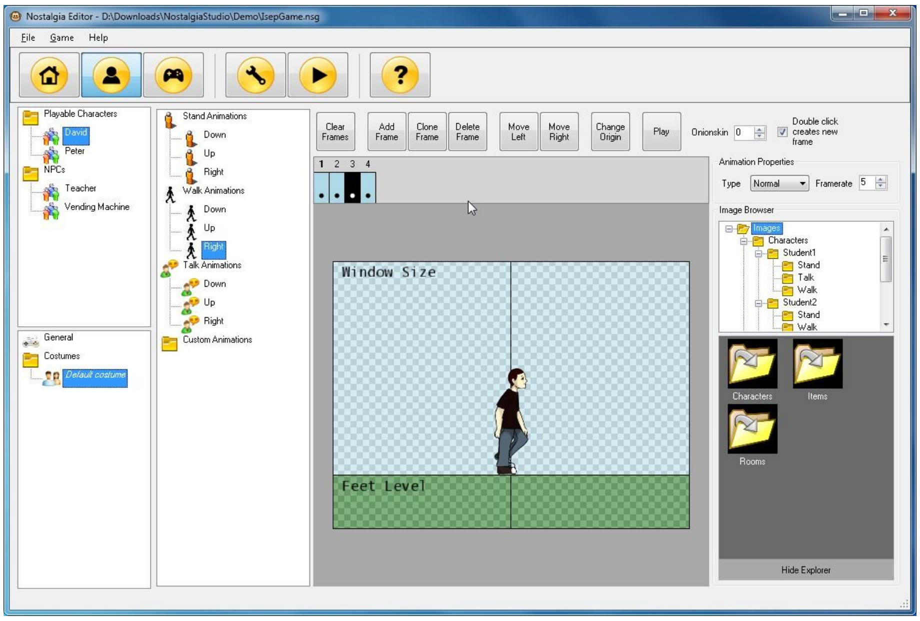
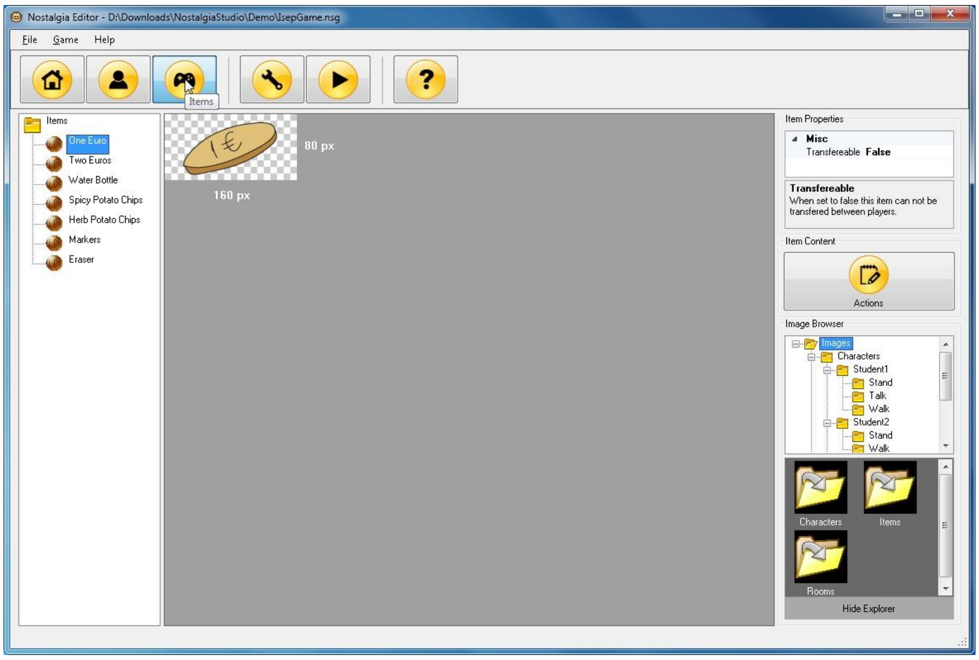


Figure 94 - Characters section

The items section on the other hand is one of the simplest, containing only a generic tree, a canvas and an image browser.

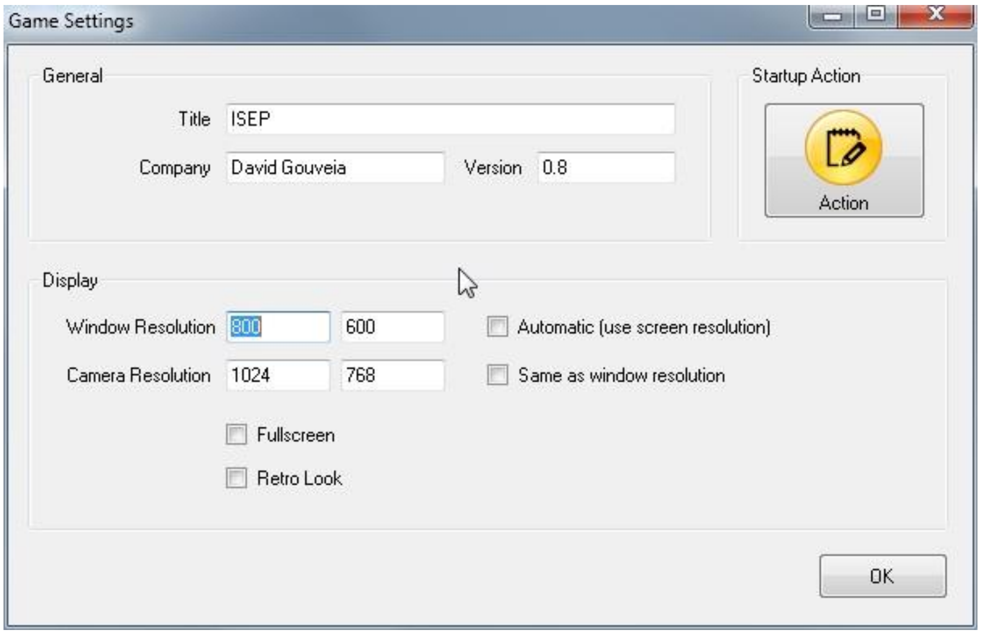


Transfereable False

Transfereable
When set to false this item can not be transfered between players.

Figure 95 - Items section

The settings dialog is also very simple, providing a few simple controls to the edit the game's general properties. There is also a button that launches an action editor in order to set any actions that should be executed when the game first starts.



Game Settings

General

Title: ISEP

Company: David Gouveia   Version: 0.8

Startup Action

Display

Window Resolution: 800 600   ☐ Automatic (use screen resolution)

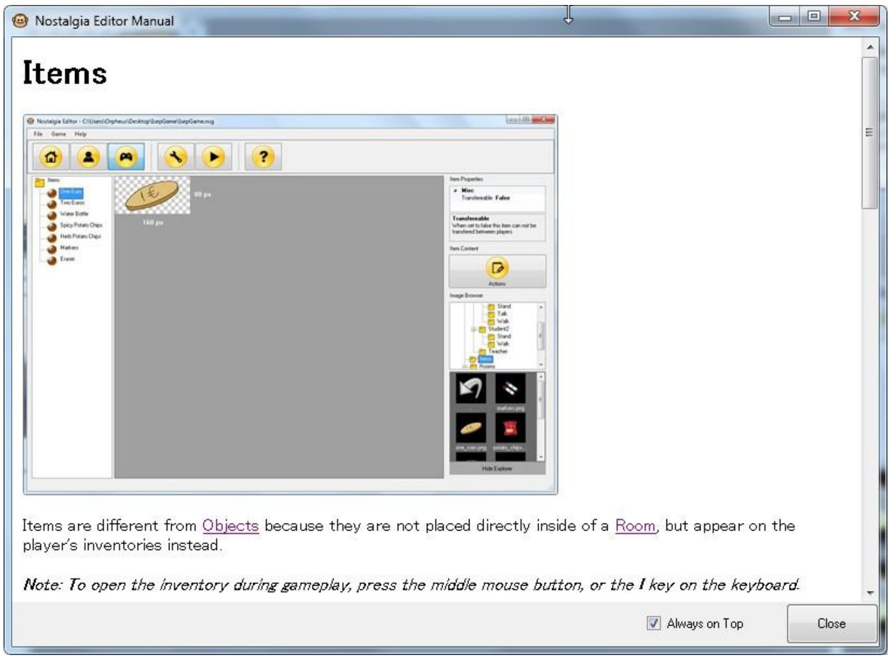
Camera Resolution: 1024 768   ☐ Same as window resolution

☐ Fullscreen

☐ Retro Look

Figure 96 - Settings dialog

The help dialog uses a web browser control to display the manual which is bundled with the editor in HTML format. It is also capable of detecting which section or sub-section the user is currently viewing, and showing the corresponding documentation accordingly.



Nostalgia Editor Manual

Items

Items are different from Objects because they are not placed directly inside of a Room, but appear on the player's inventories instead.

Note: To open the inventory during gameplay, press the middle mouse button, or the I key on the keyboard.

☑ Always on Top

Figure 97 - Help dialog

6.2.2 Components

As seen on the previous chapter, there are many different components in the editor. However most of them were already visible on the screenshots shown before, so they won't be repeated here. Instead we'll see the two most important components that have not been shown yet. First we have the action editor, which is shown here inside of a dialog. Almost every section in the editor provides access to the action editor through a button labeled Actions.

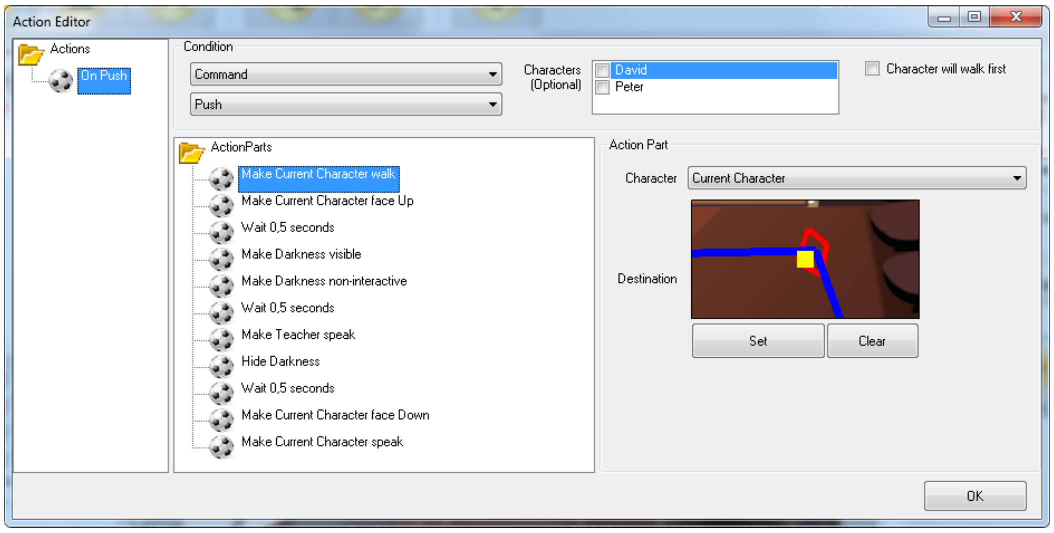


Figure 98 - Action editor

Adding new action parts to the game is done through a context menu which appears when the user right-clicks on the Action Parts folder. This context menu is divided into categories, and provides a logical hierarchy of action parts, making it easier to navigate to the type of action part that we want to create.

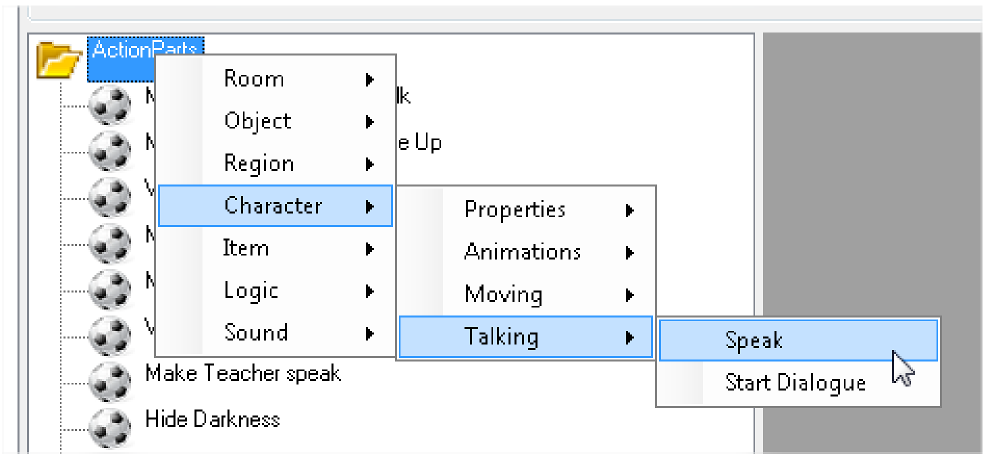
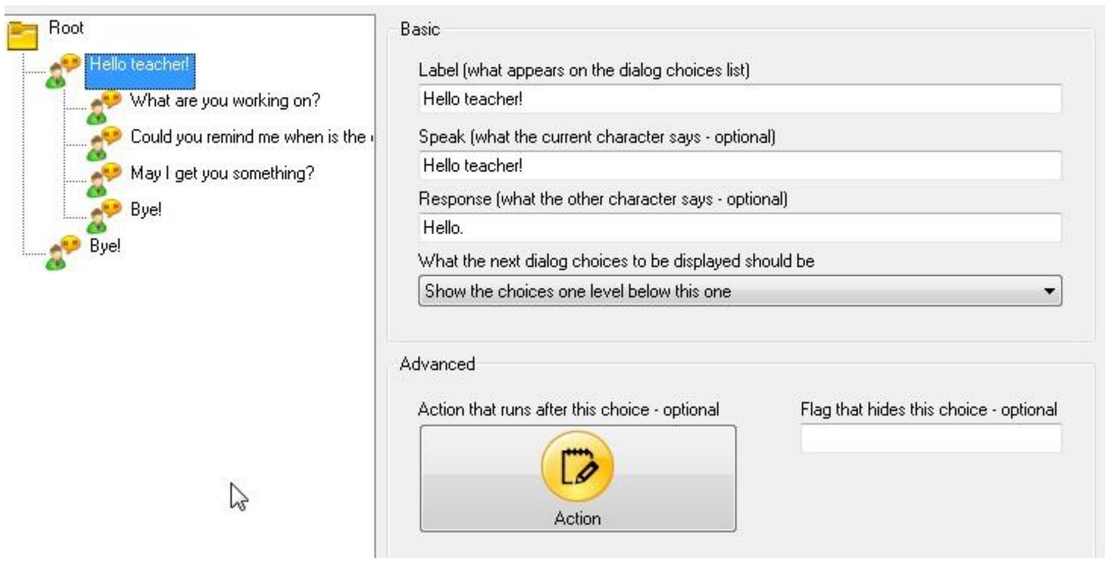


Figure 99 - Adding new action parts

And here's the dialogue editor, which can only be accessed when editing a non-playable character. The dialogue is shown as a tree of choices, with an interface on the right to edit the content of each of these choices.



Root
- Hello teacher!
  - What are you working on?
  - Could you remind me when is the
  - May I get you something?
  - Bye!
- Bye!

Basic

Label (what appears on the dialog choices list): Hello teacher!

Speak (what the current character says - optional): Hello teacher!

Response (what the other character says - optional): Hello.

What the next dialog choices to be displayed should be: Show the choices one level below this one

Advanced

Action that runs after this choice - optional

Flag that hides this choice - optional

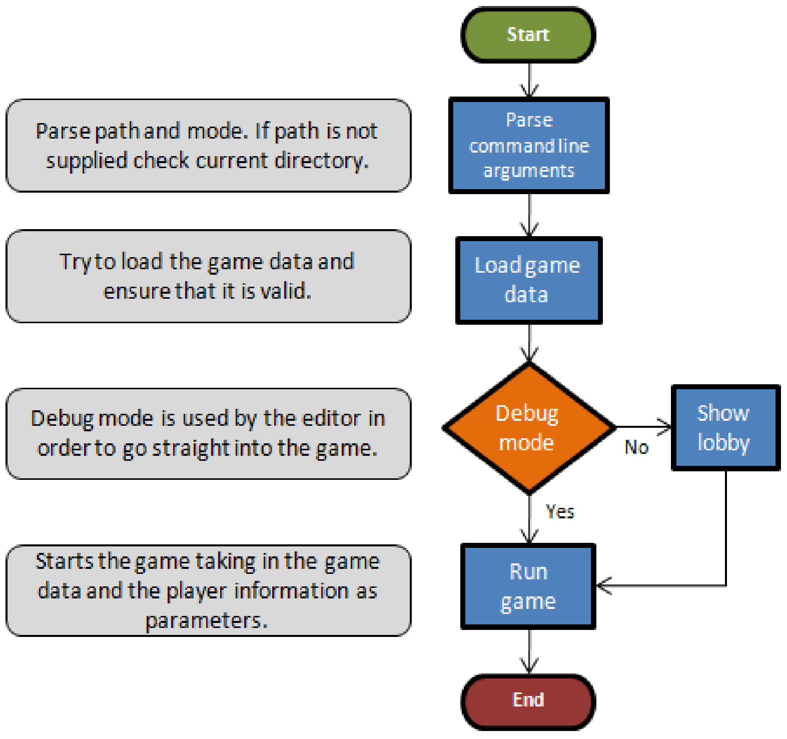
Figure 100 - Dialogue editor

The top half of the interface allows the basic properties of the dialog choice to be edited, such as deciding what each of the characters should say when the choice is selected, and what should be the next dialog choices displayed after this one is selected.

The bottom half of the interface is used for setting more advanced properties of the dialog choice, such as executing an action when the choice is selected, or hiding this choice from the menu whenever a certain flag is set.

6.3 Client Implementation

The Nostalgia Client is a small Microsoft XNA Windows Game project which delegates almost all of its work to the Nostalgia Engine. The main responsibilities of the client are to load and validate the game data, display the lobby interface, and finally start the game. The flowchart below shows the main execution flow implemented in the client.



Start
Parse path and mode. If path is not supplied check current directory.
Parse command line arguments
Try to load the game data and ensure that it is valid.
Load game data
Debug mode is used by the editor in order to go straight into the game.
Debug mode
No
Show lobby
Yes
Starts the game taking in the game data and the player information as parameters.
Run game
End

Figure 101 - Execution flow of the Nostalgia Client

The client can be started with the following two optional command line arguments:

```
NostalgiaClient.exe -p path -d
```

- The first argument (-p) specifies which game data file should be loaded. If this argument is missing, the client tries to search the current directory for a data file with the same name as the executable.
- The second argument (-d) can be added to run the client in debug mode. In this mode the game starts immediately as single player, without presenting the lobby to the player. This argument is used by the Nostalgia Editor when previewing the game that is being created.

An external library was used to facilitate the parsing of the command line arguments.

6.3.1 Game Module

The game module limits itself to instantiating a GameplayController with the loaded game data and the Player instance returned from the lobby.

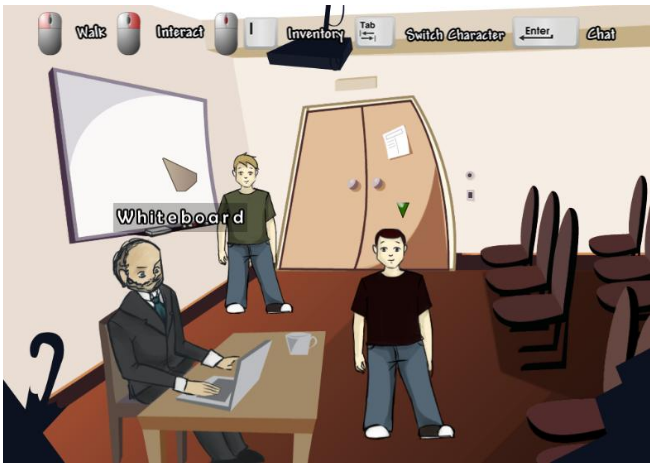
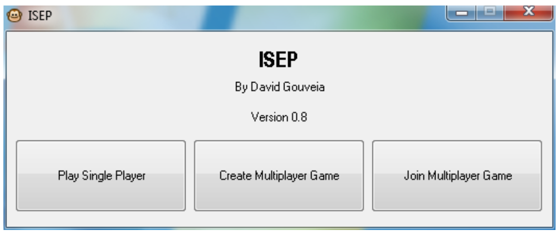


Figure 102 - A game created with Nostalgia Studio running

6.3.2 Lobby Module

The lobby is a Windows Forms form which is shown automatically when the client starts, and is mostly responsible for selecting between playing in single player or multiplayer modes, creating a Player instance of the appropriate type for the selected mode, assigning a unique index to the Player which determines the characters he controls, and marking the Player as a spectator if he joined after all the character slots had been filled.

In this version of the prototype, the lobby is extremely minimalistic, and does not provide all the different game modes and features suggested in the previous chapter.


ISEP

ISEP

By David Gouveia

Version 0.8

Figure 103 - The main lobby interface in the prototype

6.3.3 Debug Module

When running in debug mode, an extra window should open with an interface that allows the game state to be directly manipulated. This interface was not implemented in the prototype.

6.4 Technologies

Nostalgia Studio was created using Microsoft Visual Studio 2010 and the C# programming language. It also makes use of several other technologies as described in the previous chapter:

- .NET Framework (http://msdn.microsoft.com/netframework/)
    - The .NET Framework (pronounced dot net) is a software framework developed by Microsoft that runs primarily on Microsoft Windows. It includes a large library and provides language interoperability (each language can use code written in other languages) across several programming languages.
    - Windows Forms is a GUI toolkit provided by the .NET Framework, and was used to create most of the interface for Nostalgia Editor.
- Microsoft XNA (http://create.msdn.com)
    - Microsoft XNA is a set of tools with a managed runtime environment provided by Microsoft that facilitates video game development and management. The version of Microsoft XNA used by Nostalgia Studio was version 4.0.
- FMOD API (http://www.fmod.org/)
    - FMOD is a proprietary audio library made by Firelight Technologies that plays music files of diverse formats on many different operating system platforms, used in games and software applications to provide audio functionality. The library is already distributed with a C# wrapper which made it easy to integrate into Nostalgia Studio.
- DotNetZip (http://dotnetzip.codeplex.com/)
    - DotNetZip is an easy-to-use, fast, free class library and toolset for manipulating zip files or folders in .NET. All the images imported into the Nostalgia Editor were stored inside zip files with a modified file extension. Then the engine proceeds to load these images at runtime directly from within the zip files.
- Lidgren.Network (http://code.google.com/p/lidgren-network-gen3/)
    - Lidgren.Network is a networking library for .NET framework which uses a single UDP socket to deliver a simple API for connecting a client to a server, reading and sending messages.
- Clipper (http://sourceforge.net/projects/polyclipping/)

    - The Clipper library primarily performs boolean clipping (intersection, union, difference and XOR) on polygons in 2D space. There are no restrictions on either the number nor the type of polygon that can be clipped. They can have holes, be self-intersecting and even have coincident edges. The library also performs polygon offsetting.

- Command Line Parser Library (http://commandline.codeplex.com/)
    - The Command Line Parser Library offers to CLR applications a clean and concise API for manipulating command line arguments and related tasks. It was used only by the Nostalgia Client.
- Notify Poperty Weaver (https://github.com/SimonCropp/NotifyPropertyWeaver)
    - Uses IL weaving to inject INotifyPropertyChanged code into properties. This made the task of implementing data binding in the editor a lot less cumbersome, because the same boiler plate code did not have to be repeated everywhere.
- JetBrains Resharper (http://www.jetbrains.com/resharper/)
    - A refactoring and productivity extension for Microsoft Visual Studio which was also invaluable during the development process.

6.5 Results

After a complete prototype of Nostalgia Studio was implemented, it was sent to a selected group of users, together with a few instructional videos and a demo game that was built using the editor. This demo made use of the multiplayer features of the engine and was designed so that it required the two characters to collaborate in order to perform certain actions.

After the users had experimented with the editor and the demo, they were asked to answer a survey about their general opinion of the editor and of the multiplayer component of the game. Below is an analysis of those results gotten from a total of 12 different users.

6.5.1 Questions about the target public

First the subjects were asked how many adventure games they had already played before, in order to ascertain their familiarity with the genre. Two of them had never played any adventure game before, but the majority had played at least two adventure games before.

As an optional comment to this question, users could also add the name of their favorite adventure game. The names that were referenced the most were some of the classic

LucasArts adventure games such as Monkey Island, Day of the Tentacle and Grim Fandango (which also happen to be favorites of mine).

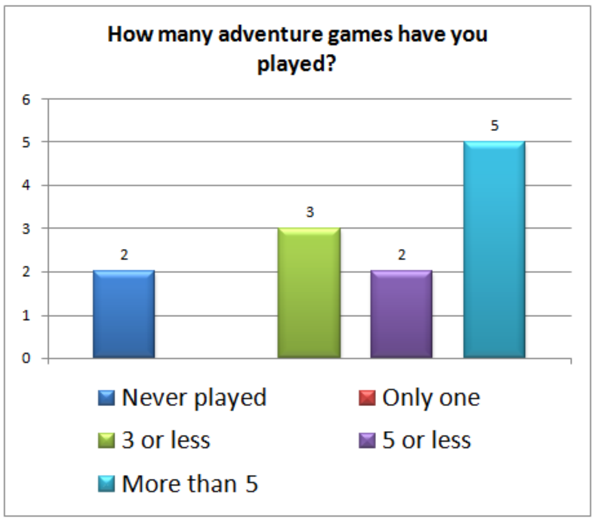


Figure 104 - How many adventure games the users played

Then their previous experience with computer programming was evaluated. Results were varied, with some of the subjects having no previous programming experience, while others had moderate or considerable experience. However, results from the next section showed that there was not a predictable correlation between having previous programming experience and being able to use the application.

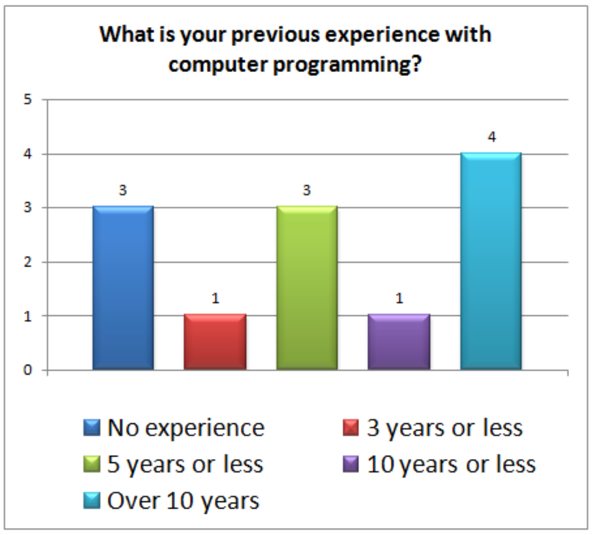


Figure 105 - Previous programming experience of the users

Finally the subjects were asked if they had ever designed or developed an adventure game before. Two thirds of them replied that they had never done that before. Curiously, none of the subjects who replied affirmatively used one of the three game engines covered in this work. This is probably due to the fact that most of the subjects came from an academic background, while these three game engines are most widely used by the game development industry.

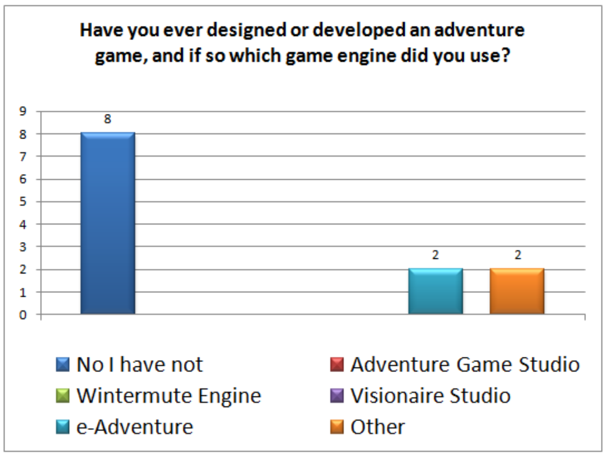


Figure 106 - Previous experience with creating adventure games

Finally this opportunity was also used to determine which aspects of adventure games were considered most important. As we'll see later when discussing the aspects of the multiplayer component, there turned out be a strong correlation between what users liked in a regular adventure game, and what users would like to see in a multiplayer adventure game.

Users were asked to select two out of six different categories related to the gameplay in an adventure game. The results show that the most important portions of an adventure game in the eyes of the users were the puzzles and the story, although a few users also favored exploration and characters.

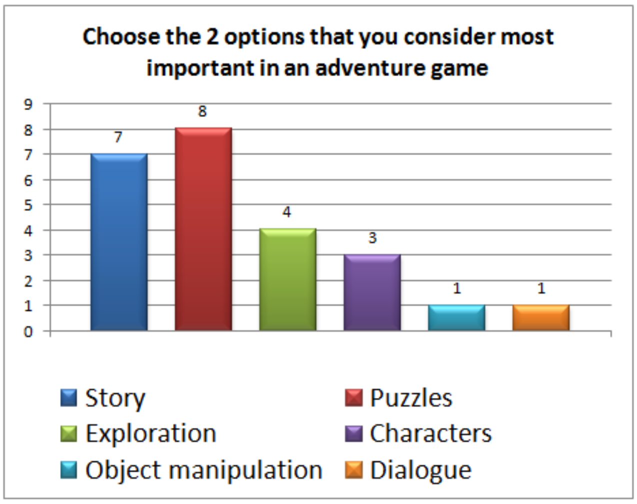


Figure 107 - Most important parts of adventure games

6.5.2 Questions about Nostalgia Studio

On the second page of the survey, users were asked to rate their experience with the Nostalgia Studio application on seven different categories. Results were extremely positive with every category getting an average score above 4 values on a scale of 1 to 5. All of the averages were very close to each other it is hard to determine which category the users enjoyed the most, although statistically speaking, performance and visual appearance were the categories with the highest score.

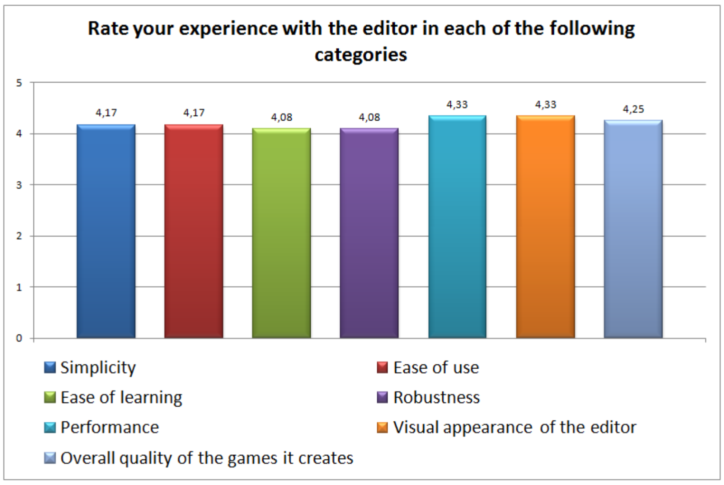


Figure 108 - Nostalgia Studio score averages

Interestingly though, despite the average being roughly the same in every category, opinions were extremely divided in some cases. For example, there were a few subjects who gave simplicity, ease of learning or visual appearance a score of 2, while the majority rated these categories with a score of 5.

Cross referencing this information with the users background information did not give conclusive results though. For instance, the user who had the most difficulty learning how to use the editor was one of the users who had considerable programming experience. However, there were also users in the same situation who thought the editor was very easy to learn.

Table 15 - Nostalgia Studio score distribution

Rating	1	2	3	4	5	Average
Simplicity	0	1	2	3	6	4,17
Ease of use	0	0	3	4	5	4,17
Ease of learning	0	1	2	4	5	4,08
Robustness	0	0	3	5	4	4,08
Performance	0	0	2	4	6	4,33
Visual appearance	0	1	1	3	7	4,33
Games quality	0	0	1	7	4	4,25

Besides this rating system there were also two optional questions where the users could add a comment about what they liked the most and the least about the editor. Once again this proved that opinions were extremely divided. For example, one user said that his favorite part of the editor was the visual appearance, while another user said he did not like the visual appearance of the application.

In general though, the aspect about the editor that most users commented positively about was the editor's simplicity and ease of use, in particular among people with no programming experience, who were surprised that it was possible to create a game as easily as this. This result is extremely positive since that was one of the main goals that this project tried to achieve.

6.5.3 Questions about the multiplayer component

The third and final page of the survey was dedicated to evaluating the usage of multiplayer collaboration in adventure games. The demo game provided along with Nostalgia Studio was used as a tool for the users to experience this type of gameplay and form their opinion. They were then asked to rate the importance of five different features in the context of a multiplayer adventure game.

This time the results were a lot more stable, and the feature that was considered most important for a multiplayer adventure game was having puzzles that required collaboration between players. This choice makes make perfect sense when we consider that the feature of adventure games that users voted on the most on the first page was the puzzles.

Every other feature that directly affected this collaboration such as having characters with different skills, allowing players to trade items and the ability to chat with each other were also considered very important. Conversely, there was also one feature that almost every user gave little importance to, which was allowing spectators to watch the game, although it was not completely disregarded either.

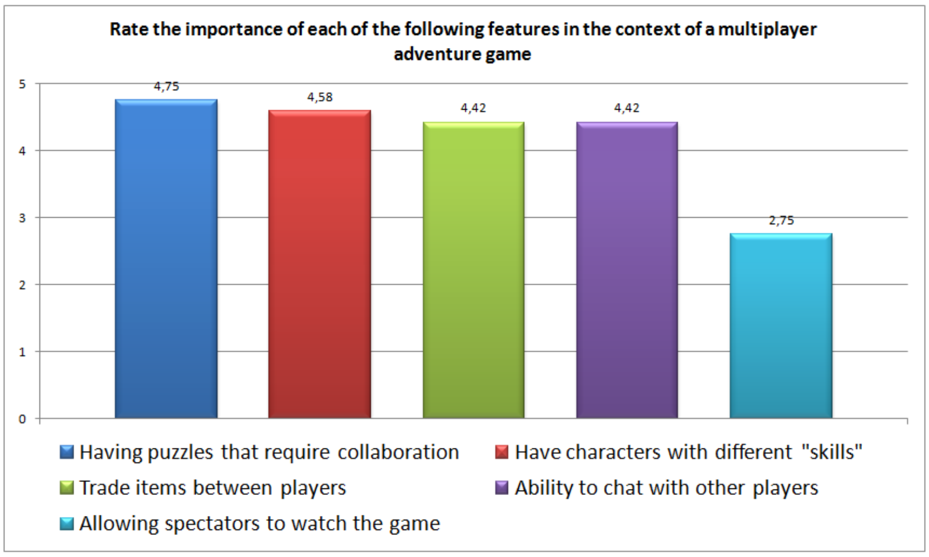


Figure 109 - Multiplayer component features score averages

Table 16 - Multiplayer component features score distribution

Rating	1	2	3	4	5	Average
Puzzles that require collaboration	0	0	0	3	9	4,75
Characters with different "skills"	0	0	1	3	8	4,58
Trade items	0	0	1	5	6	4,42
Chat with other players	0	0	1	5	6	4,42
Spectators	2	2	5	3	0	2,75

Finally, the interest of the users in seeing a multiplayer online adventure game being developed in the future was measured. In general, all of the subjects were very interested in seeing this happen, with an average interest rating of 4.5 out of 5. This result is motivating and shows that multiplayer online collaboration in graphic adventure games is an area which should be further explored.

7 Conclusion

This project started out as a way to address some of the limitations which were found during the development of Time Mesh. In particular, the tools which were used to develop the game were too complex for the general public, making it difficult for other people outside of the development team to create their own educational scenarios for the game. Furthermore, the multiplayer collaboration which was supposed to be one of the central points of the game, ended up being too disconnected from the gameplay itself, and was not given the proper attention.

It was because of those limitations that Nostalgia Studio was created. Nostalgia Studio provides a simple way to create graphic adventure with a drag and drop, "what you see is what you get" type of interface, and at the same time allows those games to be played by multiple players over a network. There are also several elements embedded into the gameplay which motivates collaboration between player, such as the ability to create puzzles that only certain characters can solve, the ability to trade items between players, and a chat system for player communication.

Thinking back on what the biggest problems during the development of Nostalgia Studio were, the main problem was underestimating the amount of work required to create a complete game engine and editor like this from scratch. This process ended up taking one year longer than initially planned, and many features had to be cut from the final version of the prototype in order to finish it on time. Nonetheless, the prototype still ended up containing all of the essential features which were required for it to be fully evaluated.

The results from that evaluation were extremely positive, both regarding the simplicity and ease of use of the application itself, and also about the prospects of incorporating multiplayer gameplay into future adventure games. One of the users who participated in the evaluation, coming from an educational background and having absolutely no knowledge of computer programming, has already shown interest in using the application in a classroom environment. Another user, which was a professional game developer, also gave very good feedback on the application and was particularly excited about the multiplayer component of the game, which he felt was capable of making any type of game more interesting. Overall I believe that both of the goals which were initially set for Nostalgia Studio were successfully achieved.

On a personal level, what I considered to be the most positive aspect about the development of Nostalgia Studio was all the experience I gained from it, as it forced me to research and implement all of the systems that go into creating a complete game engine. This led me to learn about graphics, audio, input, resource management, networking, artificial intelligence scripting, data structures, algorithms, design patterns, and many others. All of this experience will certainly be invaluable to me in the future since I intend to start working as a professional game developer.

Also, as a long time graphic adventure fan, it was extremely rewarding to study all of its history and evolution, as well as to get acquainted with some of the best graphic adventure game engines that are currently available to the general public. Surprisingly, when I first started working on this thesis, the genre of graphic adventure games was still in the same lukewarm state it had been in since the end of the 90s. However, with the successful Double Fine Adventure Kickstarter campaign on February 2012, new awareness has been given to the genre, and many of the masterminds behind the old classic graphic adventure games are now trying to make a comeback. These are certainly great news to all the fans of the genre worldwide.

There is room for new features to be added to Nostalgia Studio in the future, such as adding localization support or a save game system. The interface of the action editor could also be modified to use a visual programming scheme instead of the more traditional approach, which might make the experience even more appealing to the general public.

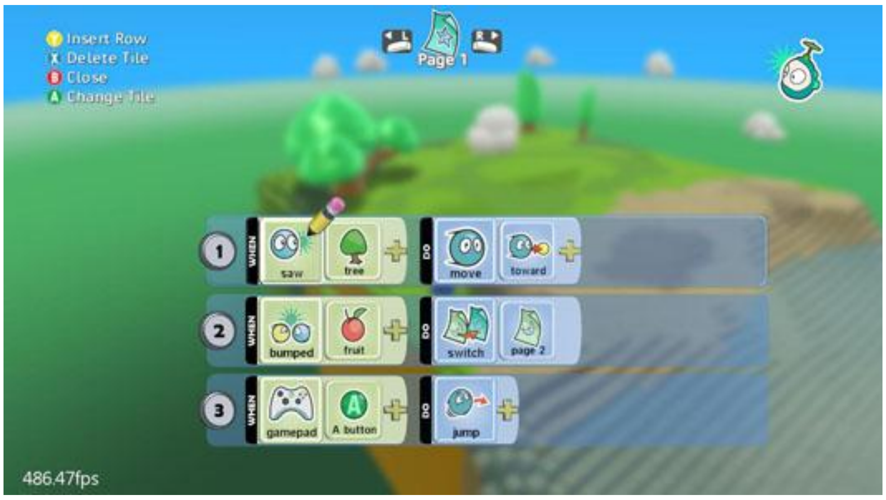


Figure 110 - An example of visual programming from the Kodu project

Another relevant idea for future development would be replace some of the technologies used with cross platform alternatives such as switching to an OpenGL renderer and using the cross platform Mono framework instead of the Microsoft .NET Framework which is currently being used. This would make Nostalgia Studio accessible to a larger audience instead of running only on the Windows platform. There are also a few routes of improvement for the multiplayer component of the game, such as extending the lobby system to introduce the different game modes described in chapter five.

To conclude this thesis, I would like to urge graphic adventure game developers to take inspiration from this work and consider the benefits of incorporating online collaboration into their future games. The majority of the users who evaluated Nostalgia Studio stated that they were very interested in the idea of playing a collaborative graphic adventure game, and I believe we are looking at a subject which holds a lot of potential and could evolve the genre of graphic adventure games even further.

8 References

ACG. (1999, October). *Roberta Williams - Sierra On-Line - Interview*. Retrieved August 2011, from http://www.adventureclassicgaming.com/index.php/site/interviews/127/

Adams, E. (2006). *Fundamentals of Game Design.* Prentice Hall.

AGS. (2011). *Adventure Game Studio Legal Information*. Retrieved January 2012, from http://www.adventuregamestudio.co.uk/aclegal.htm

Akenine-Moller, T. (2008). *Real-Time Rendering, Third Edition.* AK Peters.

Anderson, T. (1985, Winter). *The History of Zork - First in a Series*. Retrieved August 2011, from http://www.ifarchive.org/if-archive/infocom/articles/NZT-Zorkhistory.txt

Barton, M. (2008). *Dungeons and desktops : the history of computer role-playing games.* Wellesley: A K Peters.

Crook, D. (2007, May). *The Circle of Life: An Analysis of the Game Product Lifecycle*. Retrieved September 2011, from http://www.gamasutra.com/view/feature/1453/the_circle_of_life_an_analysis_of_.php

DeLoura, M. (2000). *Game programming gems.* Boston: Charles River Media.

Dickheiser, M. (2006). *C++ For Game Programmers.* Charles River Media.

DoubleFine. (2012, February). *Kickstarter Double Fine Adventure*. Retrieved August 2012, from http://www.kickstarter.com/projects/doublefine/double-fine-adventure

GameSpot. (2002, March 22). *The Sims overtakes Myst*. Retrieved from http://www.gamespot.com/pc/strategy/simslivinlarge/news_2857556.html

GoF. (1995). *Design patterns : elements of reusable object-oriented software.* Reading: Addison-Wesley.

Gregory, J. (2009). *Game engine architecture.* Wellesley: A K Peters.

Jerz, D. (2007, Summer). *Somewhere Nearby is Colossal Cave: Examining Will Crowther's Original "Adventure" in Code and in Kentucky*. Retrieved August 2011, from http://www.digitalhumanities.org/dhq/vol/001/2/000009/000009.html

Kalata, K. (2011). *The guide to classic graphic adventures.* Hardcoregaming101.net.

Lippert, E. (2007, October 2). *Path Finding Using A* in C# 3.0, Part One*. Retrieved February 2012, from Eric Lippert's Blog: http://blogs.msdn.com/b/ericlippert/archive/2007/10/02/path-finding-using-a-in-c-3-0.aspx

Loguidice, B. (2009). *Vintage games : an insider look at the history of Grand Theft Auto, Super Mario, and the most influential games of all time.* Boston: Focal Press/Elsevier.

MacDonald, M. (2005). *Pro .NET 2.0 Windows Forms and Custom Controls in C#.* Apress.

Martin, R. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall.

McShaffry, M. (2012). *Game coding complete.* Boston: Course Technology, Cengage Learning.

Moss, R. (2011). *A truly graphic adventure: the 25-year rise and fall of a beloved genre*. Retrieved October 2011, from http://arstechnica.com/gaming/2011/01/history-of-graphic-adventures/

Rabin, S. (2002). *AI game programming wisdom.* Hingham: Charles River Media.

SELEAG. (2011). *Serious Learning Games*. Retrieved September 2011, from http://www.seleag.eu

TheDotEaters. (1998). *Computer Game History*. Retrieved August 2011, from http://www.thedoteaters.com/p4_stage2.php

TimeMesh. (2011). *Time Mesh*. Retrieved September 2011, from http://timemesh.eu

Visionaire. (2011). *Visionaire Studio License Information*. Retrieved January 2012, from http://www.visionaire-studio.net/cms/licences.html

WME. (2010). *Wintermute Engine License Information*. Retrieved January 2012, from http://dead-code.org/home/index.php/license/

Woods, D. (2001, June). *Interactive Fiction? I prefer Adventure*. Retrieved August 2011, from http://www.avventuretestuali.com/interviste/woods-eng