Department of Computer Science

Martin Persson

Development of three AI techniques for 2D platform games

D-level Dissertation (20p)

2005:11

Development of three AI techniques for 2D platform games

Martin Persson

This thesis is submitted in partial fulfillment of the requirements for the Masters degree in Computer Science. All material in this thesis which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Martin Persson

Approved, 9 December 2005

Opponent: Jan Eriksson

Advisor: Simone Fischer-Hübner

Examiner: Donald F. Ross

Abstract

This thesis serves as an introduction to anyone that has an interest in artificial intelligence games and has experience in programming or anyone who knows nothing of computer games but wants to learn about it. The first part will present a brief introduction to AI, then it will give an introduction to games and game programming for someone that has little knowledge about games. This part includes game programming terminology, different game genres and a little history of games. Then there is an introduction of a couple of common techniques used in game AI. The main contribution of this dissertation is in the second part where three techniques that never were properly implemented before 3D games took over the market are introduced and it is explained how they would be done if they were to live up to today's standards and demands. These are: line of sight, image recognition and pathfinding. These three techniques are used in today's 3D games so if a 2D game were to be released today the demands on the AI would be much higher then they were ten years ago when 2D games stagnated. The last part is an evaluation of the three discussed topics.

Keywords

Artificial intelligence, AI, Game, 2D, Platform, Line of sight, Image recognition, Pathfinding.

Acknowledgements

I would like to thank the flowing people.

Simone Fischer-Hübner, for all the advise and help throughout the project. And for showing me that there exist people in the computer science world that do not know who Pac-Man is.

Donald F. Ross, for approving this topic as a D-level dissertation. And for setting an example for how code should not be written.

Jan Eriksson, for constructive criticism during the opposition. And for pointing out all the bad things in the dissertation.

Preface

I decided that the topic for my D-level dissertation would be game artificial intelligence when my C-level dissertation was finished and it ended up containing no artificial intelligence. I found this disappointing because AI in an important part of a game. It is especially important to develop advanced AI in platform games because the last game that developed the genre was released eight years ago. And for some reason there are few people in the academic world that even care about games. This I want to change.

Making movies requires no technical knowledge, anyone who can aim a camera and press record can direct movies. But when it comes to games the game makers are always programmers with great knowledge of computers. And musicians do not need to build their own instruments. Imagine that all movie directors and producers had to build their own cameras. Then there would be a lot less movies that were made. And if all musicians needed to build their own instruments themselves there would be a lot less musicians around. But game makers have to make the engine for the game themselves. Compare musicians and directors with game makers, the musicians and movie directors are artists while game makers are technicians. This is one of the reasons why there are so few game programmers around. There are a lot of people who like games and would like to make their own games but few of them have the technical knowledge needed to make games and even fewer who like 2D platform games. Hopefully this will change in the future.

When the movie Star Wars was released it made every science fiction movie before it seem to be obsolete because of its special effects. It is the same thing for games; a really great game makes all other of its genre seem to be obsolete. This is what is needed in order to make 2D platform games return.

The AI techniques presented in the dissertation are nothing but help to anyone that would like to make AI in a 2D platform game. The real result depends on the game makers themselves. As Robert L. Glass put it in *"Facts and Fallacies about Software Engineering"*[2]:

> The most important factor in software work is not the tools and techniques used by the programmers, but rather the quality of the programmers themselves.

I could not agree more with that statement.

Martin Persson, October 2005.

Contents

1 Introduction **1**

1.1 Purpose . . . 1

1.2 Motivation . . . 2

1.3 Limitation . . . 3

1.4 New techniques . . . 4

1.4.1 Line of sight . . . 4

1.4.2 Image recognition of the level . . . 4

1.4.3 Pathfinding . . . 5

1.5 Goal . . . 5

1.6 Disposition . . . 5

I Background **7**

2 Introduction to artificial intelligence **9**

2.1 What is artificial intelligence? . . . 9

2.1.1 AI agent . . . 10

2.2 Heuristics . . . 11

2.3 Weak and strong AI . . . 11

2.4 The Turing test . . . 13

2.4.1 The history of the Turing test . . . 13

2.4.2 Objections to the Turing test . . . 14
2.5 The Foundation of artificial intelligence . . . 15
2.5.1 Philosophy . . . 15
2.5.2 Mathematics . . . 16
2.5.3 Neuropsychology . . . 16
2.5.4 Control theory . . . 17
2.6 The history of Artificial Intelligence . . . 17
2.6.1 The birth . . . 17
2.6.2 The first AI agents . . . 18
2.6.3 Setback . . . 19
2.6.4 Knowledge based systems . . . 19
2.6.5 The AI winter . . . 20
2.6.6 AI becomes a science . . . 20
2.7 AI today . . . 21
2.7.1 Robotics . . . 22
2.7.2 Spam filters . . . 22
2.7.3 Virus scanners . . . 22
2.7.4 Security agents . . . 23
2.8 Image recognition . . . 23
2.8.1 Introduction . . . 23
2.8.2 Reading a image . . . 24
2.8.3 Transforming the image into data . . . 25
2.8.4 Recognizing objects . . . 26
2.9 Deep Blue . . . 28
2.10 Summary . . . 29

3 Introduction to game programming 31

3.1 Game platforms . . . 32

3.1.1 Arcade games . . . 32
3.1.2 Console games . . . 33
3.1.3 Computer games . . . 34
3.2 2D and 3D games . . . 35
3.3 Terminology . . . 38
3.4 Game Genres . . . 43
3.4.1 Platform games . . . 43
3.4.2 First Person Shooter games . . . 44
3.4.3 Role Playing Games . . . 45
3.4.4 Adventure games . . . 47
3.4.5 Strategy games . . . 49
3.4.6 Beat'em up . . . 51
3.4.7 Shoot'em up . . . 51
3.4.8 Simulator games . . . 53
3.4.9 Sim games . . . 53
3.4.10 Sports games . . . 54
3.4.11 Massive Multiplayer Online games . . . 54
3.4.12 Puzzle games . . . 55
3.5 Game Application Programming Interface . . . 56
3.5.1 DirectX . . . 56
3.5.2 SDL . . . 57
3.5.3 OpenGL . . . 57
3.6 Programming games . . . 58
3.6.1 Modifications . . . 59
3.7 Summary . . . 60

4 Introduction to game AI **61**

4.1 Introduction . . . 62

4.2 Mainstream AI and game AI 62
4.2.1 Expert systems and production systems 62
4.2.2 Artificial life 66
4.2.3 Finite state machine 67
4.3 Fuzzy logic 70
4.3.1 Fussy sets 71
4.3.2 Defuzzification 73
4.4 Pathfinding with A* 74
4.4.1 Terms 74
4.4.2 The algorithm 76
4.4.3 Pseudo code for the algorithm 77
4.5 A complete enemy AI agent 78
4.6 Summary 78

II Experiment **79**

5 Line of sight **81**
5.1 Introduction 81
5.2 Theory 82
5.2.1 Visual limit 83
5.2.2 Free sight 85
5.2.3 Bresenham's algorithm 86
5.2.4 Efficiency 89
5.3 Implementation 89
5.3.1 One problem 91
5.3.2 Statistics 91
5.3.3 Solution 93

5.4 Summary . . . 93

6 Image recognition of the level **95**

6.1 Introduction . . . 95

6.2 Collision detection for making AI agents "see" . . . 96

6.3 Jump over gorges . . . 96

6.3.1 To know when to jump . . . 97

6.3.2 To know if it is possible to jump . . . 98

6.3.3 To know how far to jump . . . 99

6.3.4 Summary . . . 99

6.4 Free jump trajectory . . . 100

6.4.1 To know if there is jump area is free . . . 100

6.4.2 To know if the trajectory path is free . . . 101

6.5 Jump over objects . . . 103

6.5.1 To know the obstacle is not too high . . . 105

6.5.2 To know if it is possible to jump over a obstacle . . . 107

6.5.3 To know if the opening is big enough . . . 108

6.6 Triggers on the map . . . 109

6.6.1 Fixed points on the map . . . 111

6.7 Summary . . . 112

7 Pathfinding **113**

7.1 Introduction . . . 113

7.2 Theory . . . 114

7.2.1 The graph of nodes . . . 114

7.3 2D platform games need only few nodes . . . 114

7.4 Implementation . . . 117

7.5 Summary . . . 118

III Discussion **123**

8 Evaluation **125**

8.1 Introduction . . . 125

8.2 Achievements . . . 126

8.2.1 Line of sight . . . 126

8.2.2 Image recognition of the level . . . 127

8.2.3 Pathfinding . . . 128

8.3 Cell phones and palm pilots . . . 129

8.3.1 Line of sight . . . 129

8.3.2 Image recognition of the level . . . 130

8.3.3 Pathfinding . . . 130

8.4 Summary . . . 130

9 Conclusions **133**

9.1 Conclusion . . . 133

9.1.1 Line of sight . . . 133

9.1.2 Image recognition of the level . . . 134

9.1.3 Pathfinding . . . 134

9.2 Problems . . . 134

9.2.1 Line of sight . . . 135

9.2.2 Image recognition of the level . . . 135

9.2.3 Pathfinding . . . 136

10 Plans for Future work **137**

10.1 Line of sight . . . 137

10.2 Image recognition of the level . . . 138

10.3 Pathfinding . . . 139

References **141**

IV Appendixes **143**

A Line of sight **145**

A.1 Bresenham's algorithm 146
A.1.1 Code 146
A.2 Line of sight 147
A.2.1 Code 148
A.3 Character on screen 150
A.3.1 Code 150
A.4 Collision test 150
A.4.1 Code 151
A.5 Free sight 151
A.5.1 Code 152
A.6 Simple ground collision 153
A.6.1 Code 154

B Image recognition of the level **155**

B.1 Reached left edge 155
B.1.1 Code 156
B.2 Reached right edge 156
B.2.1 Code 156
B.3 Block within jumprange 157
B.3.1 Code 157
B.4 Measure gap width 159
B.4.1 Code 159
B.5 Free jump area 160

B.5.1 Code 161
B.6 Free jump trajectory 162
B.6.1 Code 162
B.7 Block in path 163
B.7.1 Code 163
B.8 Obstacle low enough 164
B.8.1 Code 164

List of Figures

1.1 Bubble Bobble, a plattform game released in 1986. The characters and objects all stand on platform, can move left and right and fall downward. 2

1.2 Castlevania: Aria of Sorrow, a 2D platform game released 2003. But the genre have not developed since 1997. 3

2.1 A photograph of a stapler (left) and edges computed from the photograph (right). 26

2.2 The same stapler as in Figure 2.1 but with the adjacency points between the lines (left) and the lines of the object known by the AI drawn out(right). 27

3.1 Super Mario Bros, the first screen scrolling platform game. 34

3.2 Pitfall!, the worlds first platform game. 36

3.3 Ultima Underworld, one of the first real 3D games. 37

3.4 Mario^{64}, a third person perspective game. Notice the reflection of the little guy hovering on the cloud with the camera in the mirror. . 38

3.5 Guybrush Threepwood from The Secret of Monkey Island^{TM}. The rectangle will disappear when it is sprited on the screen. 40

3.6 Wolfenstein 3D, the worlds first First Person Shooter. 45

3.7 The Secret of Monkey Island^{TM}, a classic point and click game. . . 49

3.8 Dune II: The Battle for Akkaris, the first Realtime Strategy game. . 50
3.9 Gradius, a side scrolling 2D shot'em up game. 52
3.10 Tetris, a simple but popular puzzle game. 56
3.11 Diagram of a main event loop. 58

4.1 The ghosts from Pac-Man. Blinky, Pinky, Inky and Clyde. 61
4.2 A basic production system. 63
4.3 A simple finite state machine for a enemy AI agent. 69
4.4 A fuzzy trapezoid set. 72
4.5 Graph displaying the membership in fuzzy sets of a enemy AI agent depending on how much health it has. 73

5.1 Blackthorne, released in 1994, a 2D platform game where the enemies only reacted on what they saw. 83
5.2 This picture shows a screenshot of a level with the a character in the center. 84
5.3 This picture shows the same level as Figure 5.2 but is shows what would be on the screen if everything were moved a bit to the left. . 85
5.4 Free sight example 1. Two characters on the same screen. 86
5.5 Free sight example 2. Highlighting the area of the screen that is within the line of sight of the character below the block. 87
5.6 Line Approximations[1]. 88
5.7 Illustration of that several lines can be drawn between two characters in a game. 90
5.8 Summary of line of sight, the shaded area within the rectangle is the area of the level that the enemy AI agent can see. 94

6.1 The enemy AI agent standing in front of a gorge. 97
6.2 The enemy AI agent checking the ground in front of it. 97

6.3 The enemy AI agent checking if there is something to stand on within the distance of how far the enemy AI agent can jump. 98
6.4 The enemy AI agent checking how far it have to jump in order land on a platform. 99
6.5 Collision detect used by a AI agent in a game. 100
6.6 A scenario where a bock i blocking the jump trajectory of the AI agent. 101
6.7 The AI agent checking the area that it will jump through over the gorge. 102
6.8 The AI agent checking the approximate path it will take in the jump trajectory. 103
6.9 A gorge where it is possible for the enemy AI agent jump over it even though there it blocks in the large check. 104
6.10 The AI agent and a obstacle with height. 105
6.11 Displaying the path the AI agent will take if it do not predict when to jump. 106
6.12 The AI agent is checking the path in front of it in case it is a obstacle in the way. 107
6.13 Path of the AI agent if the jump hade been predicted before it reached the wall. 108
6.14 A enemy character standing in front of a wall with a opening in it. . 109
6.15 An AI agent and a jump trigger. 110
6.16 An AI agent and a turn left trigger. 111

7.1 A part of a navigation mesh covering a staircase. Each convex polygon (with an 'X' in the center) is a node of the navigation mesh. Courtesy of Paul Tozour and Ion Storm Austin[4]. 115

7.2 A map in a 2D platform game with the pathfinding nodes marked as ellipses. 116
7.3 The same map as in Figure 7.2 but with the node indexes and connections. 119

List of Tables

3.1 Diagram showing how different genres often and seldom are combined. 43

4.1 The knowledge database for a system expert system 64
4.2 The production rules for a system expert system 64

5.1 Table over the statistics for average time it took to complete a frame with visible debug lines. 91
5.2 Table over the statistics for the same level as in Table 5.1 but no lines are drawn. 92
5.3 Table of the average time between frames with the number of object in the level greatly increased. 92

7.1 The list for node 22 in Figure 7.3. 120

Chapter 1

Introduction

For a long time computer games with two dimensions, 2D games, were the only games available and they were made in large scale until the middle of 1990s when three-dimensional computer games, 3D games, made their breakthrough. Artificial intelligence in these 2D computer games was not very advanced because of lack of techniques, hardware and motivation. There were no scientific papers written about Artificial intelligence for games in that time. The computers were so slow that too advanced artificial intelligence would make the game run too slow to be playable. And since computer games were a relatively new thing and had a small audience compared to today, the player's demands on the games were lower, hence, the effort to make the enemies in the game look smart were not so great.

1.1 Purpose

When the last 2D games were released almost ten years ago the development of 2D games have stood almost still. Particularity AI in 2D platform games has been left undeveloped and is in need of improvement.

Figure 1.1: Bubble Bobble, a plattform game released in 1986. The characters and objects all stand on platform, can move left and right and fall downward.

Even the last of the 2D platform games had very simple AI. With this dissertation AI for 2D platform games will give the game makers some of the tools and techniques needed for making more advanced AI for 2D platform games.

The purpose of this dissertation is to give programmers who plans to make 2D platform games three techniques that they can use in their games.

1.2 Motivation

When the string "2D platform game AI" was searched with google scholar[28], at the start of the work of this dissertation, it got three hits. None of them contained useful information. This alone is motivation enough to write a dissertation about artificial intelligence for 2D platform games. But that reason mainly applies to hardcore gamers[1], so today's large scale audience will most likely encounter 2D

[1]People that played 2D games ten years ago and miss their old games.

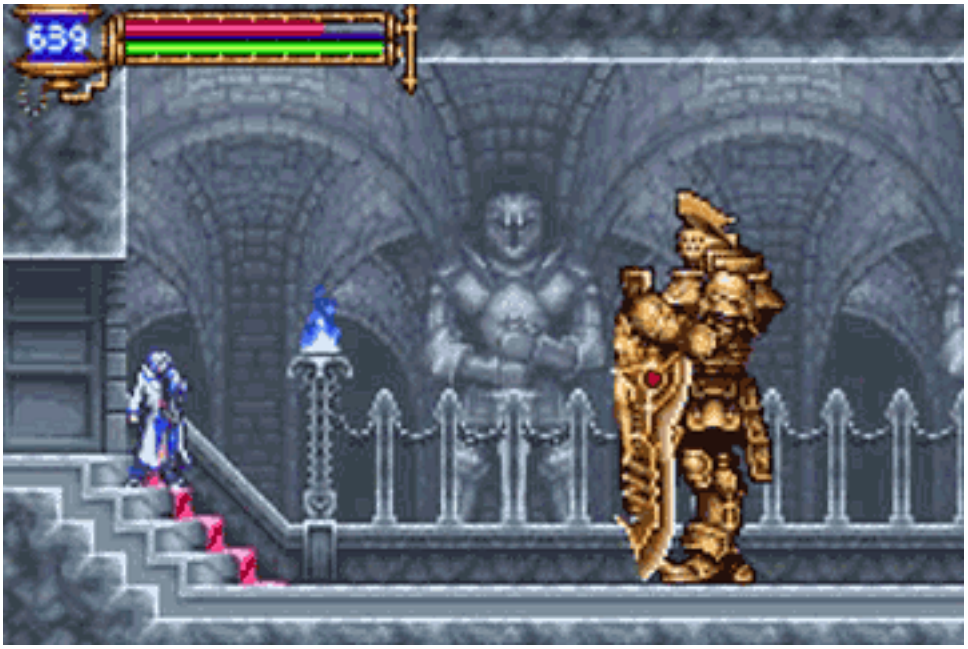

Figure 1.2: Castlevania: Aria of Sorrow, a 2D platform game released 2003. But the genre have not developed since 1997.

games in their cell phoned and/or palm pilots. Although a few new 2D games have been recently released for GameBoy Advance, like *Castlevania: Aria of Sorrow*, See Figure 1.2, they have been either remakes of old games or have contributed nothing new to the genre.

1.3 Limitation

To completely describe a working AI system for a 2D platform game that was to be released today is a too great subject to cover in 20 a points academic dissertation. The experiment in this dissertation focuses on three techniques that were never properly implemented when 2D platform games were “the big thing”.

1.4 New techniques

Three techniques have been developed for this dissertation to address the following problems: line of sight, image recognition of the level and pathfinding.

1.4.1 Line of sight

Line of sight is what the enemies can see. In moast games made before the 3D era, and even in some 3D games, the enemies can see the player the entire time. Even when there is a wall between them. If a 2D platform game was to be released today this would be unacceptable which is why this technique is necessary.

The problem is to decide if the enemy can see the player or not. If there is a wall between the enemy and the player, the enemy should not react to the player. This is a problem because the enemy is part of the game system and has direct access to the values of all the variables in the game. Thus knows where the player is all the time.

1.4.2 Image recognition of the level

In old games the enemies moved only on small places or with extremely simple patterns. The enemies that moved in large areas were known to jump randomly and fall down cliffs. In order to make the enemies know what they are doing, a technique that makes them analyze their surroundings is needed.

The question is: how can the enemies see what is around them? There is no screen that displays it to them and they have no real eyes that can see what blocks are displayed and not. They have access to everything in the game, since they are a part of the game system. But what can they see, and hence, react on? This has to be solved.

1.4.3 Pathfinding

In old games the enemies were often confined to small areas or moved in random on the map. In newer games the enemies cannot wander the level like confused rats. They need a technique that makes them less predictable.

Pathfinding is quite advanced in 3D games and only need to be applied to 2D platform games in order or work. This is not really a problem, it is more of an obstacle, the theory is there; it only have to implemented in 2D games.

1.5 Goal

In this dissertation the three techniques line of sight, image recognition of the level and pathfinding will be explained in detail. Line of sight will limit the things the enemies reacts on to what they see, just like a player only can react to what is displayed on the screen. Image recognition of the level will make the enemies dynamic enough to move around large areas of the level without falling down cliffs and analyze its soundings in a similar way that a human does. Pathfinding will make the enemies find their way on large levels.

The goal is also to implement as many of the techniques as possible. Of course, this work is meant to be the programmers reading this dissertation. But for evaluation purposes it is good to have implemented at least two of the techniques. Line of sight and image recognition of the level will have higher priority then pathfinding because they control small scale behavior and pathfinding control large scale behavior. And small scale behavior is more important.

1.6 Disposition

Part I gives the background information needed for the topics in this dissertation.

Chapter 2 gives a introduction to artificial intelligence to provide enough information to understand game AI. It is basically summarizing what you would find in every text book or paper about AI.

Chapter 3 is an introduction to computer games and how games are programmed. If you are planning to hold a fake lecture[2] about games you should read this chapter.

Chapter 4 is an introduction to the tools and techniques commonly used in game AI. If you are planing to extend the fake lecture about games to include game AI you should read this chapter as well.

Part II discusses the three topics that this dissertation tries to improve.

Chapter 5 explains the line of sight technique.

Chapter 6 explains how an enemy can visually analyze the virtual game world.

Chapter 7 explains the pathfinding technique that makes the enemies find their way along the level and not to get lost.

Part III discusses the results from Part II.

Chapter 8 will describe the main results.

Chapter 9 provides the conclusion, it will describe the problems.

Chapter 10 will describe the plans for future work.

Part IV is the appendix which contains the code for the implemented techniques.

Appendix A is the detailed description and the code for the implementation of the line of sight technique.

Appendix B is the detailed description and the code for the implementation of the image recognition of the level technique.

[2]Fake lecture is a term used by Donald Ross. It means to hold a lecture about a topic you just learned.

Part I

Background

Chapter 2

Introduction to artificial intelligence

Making computers act like humans in a given situation is subject to a special area in computer science called **Artificial intelligence**. This chapter will give an overview to the history of artificial intelligence and will cover the very basic theory behind it. It will explain the basic terms and concepts of artificial intelligence and will give a few examples, including Deep Blue. At the end of the chapter a small description of the status of artificial intelligence today will be given.

2.1 What is artificial intelligence?

The purpose of artificial intelligence, **AI** for short, is to make computers mimic the human characteristics creativity, self-improvement and language use. But it is this author's opinion that it is enough to make a computer function independently within the context it was designed for.

"American Heritage Dictionarie"[21] described AI as:

> The ability of a computer or other machine to perform those activities that are normally thought to require intelligence.

To define AI is like trying to define computer science, every programmer/scientist will give a unique in depth definition. However they are divided into two groups, one group defines AI as computers acting and thinking like humans and the other group defines AI as computers acting and thinking rational. The computer is said to act rational when it does the "right thing" given what it knows[6]. The main problem with making computers think is that neuropsychology does not yet know exactly how we think. And since it is really hard to read each others mind we cannot know if anybody other then ourselves is thinking. Human consciousness is an axiom of empistemology. But measuring how computers act in given situations is much simpler, it is just to observe and compare it to how humans acted in that situation.

2.1.1 AI agent

A module that uses AI is called an **AI agent**. The word agent means:

> A means by which something is done or caused[24].

This definition of agent can also be applied to any other program but to be considered an agent the program has to exhibit the behavioral qualities of agenthood. Essentially this means perceiving the software environment through sensors and acting on the environment throughout actions and this must be done autonomously[6].

The AI agent can be the entire system or just one module in the system. There is nothing that says that one system can have just one AI agent, a system can have as many AI agents as the designers want it to have.

The AI agent simply gets input data and produces output data. It can be one big process in the entire system or small independent agents acting alone. Deep Blue, see Section 2.9, was a system with one AI agent occupying the entire system and enemies in most computer games are examples of a system containing many small AI agents.

In the book *"Artificial Intelligence: A Modern Approach"*[6] many different types of agents that handle AI are described. But they are all AI agents

2.2 Heuristics

The ordinary goal in computer science it to find the optimal algorithm that solves a given problem. It means that the solution if found in the shortest amount of time or the optimal solution to a given problem[6].

The word **heuristics** comes from the Greek word "eureka". The word eureka means "I have figured it out" and is most famous from the the incident when Archimedes figured out buoyancy and ran naked through the city.

Heuristic algorithms are algorithms that do not have the goal to find the best possible solution or run in the shortest amount of time. Heuristic algorithms find a solution that is good enough in short enough time[6]. There are however situations where heuristics will give a very bad result in very long time but these situations can be avoided in games by good level design.

Heuristics are often used in search algorithms by using a special function called heuristic function that is used to estimates values[4].

2.3 Weak and strong AI

The field of AI can be divided into two parts, **strong AI** and **weak AI**.

Strong AI are systems that use real reasoning and rationality when making decisions. Strong AI could be described as computers that act as they have actual minds or machines that really think[6]. Some claim that emotions and consciences of itself is required for a computer to have strong AI[3], meaning that they are aware about their own existence and can figure out new ideas themselves. Strong AI must also be able to adapt to all new situations. So logically we can assume that when strong AI has been created the research is done and all AI scientists can retire. Since when a computer has strong AI, it really thinks. But there is still a long way to go before we are there.

Weak AI on the other hand are just systems that act like humans in the given situation. It has been designed for having human capabilities[6]. Weak AI is much easier to achieve because only the part of the system that is necessary to consider is the output of the system, as long as the system acts like an human it is not important if the machine really thinks or not. Weak AI is also defined as the process of a computer program repeating a algorithm once invented by a human, in this case the differences between a ordinary computation and weak AI is dim, in fact, there is no explicit difference defined. A human sorting integers and a computer sorting integers do a similar thing. But the computer program would not be defined as a AI system by computer scientists but everybody would agree that the human needs intelligence to sort.

Since weak AI "gets the job done" all AI systems in the industry are weak AI systems. Examples of weak AI systems are the speech recognition, parsing and recognition of natural languages and visual recognition of handwritten text. It does not really require any intelligence to preform these tasks but you need to mimic intelligence to do it.

It is arguable if some of these systems really are AI systems but as mentioned above the definition is dim.

2.4 The Turing test

The Turing test is a way of determining the quality of a AI agent developed by Alan Turing in 1950. The test is very simple. A test subject is put in an isolated room with only a computer to communicate with the outside world.[6] The communication could be text conversation or, more related to this dissertation, the test subject could play a multiplayer game against an AI Agent. But the original and most common form of the Turing test is pure text conversation. If the test subject is unable to determine if he/she interacted with a computer or with a human or an AI agent the test is considered successful.

Since the Turing test only measures the behavior of the AI agent, it can only check the quality of weak AI. Passing the Turing test does not mean that the AI agent has strong AI

2.4.1 The history of the Turing test

The test was inspired by a party game known as the "Imitation Game", in which a man and a woman go into separate rooms, and guests try to tell them apart by writing a series of questions and reading the typewritten answers sent back. In this game, both the man and the woman aim to convince the guests that they are the woman[22].

Turing originally proposed the test in order to replace the emotionally charged and for him meaningless question "Can machines think?" with a more well-defined one[22].

One interesting part of his proposed test was that the answers in conversation would have to be delivered at controlled intervals and rates. He believed that this was necessary to prevent the observer drawing a conclusion based on the fact the computer potentially would answer so much faster than the human operator,

especially on mathematical questions[22].

2.4.2 Objections to the Turing test

There have been many objections to the Turing test during the years. But fortunately Allan Turing had foreseen many of them and gave answers to them in the original article. Here are some of the replies he gave to the objections[27].

Mathematical Objections

This objection uses mathematical theorems, such as Gödel's incompleteness theorem[23], to show that there are limits to what questions a computer system based on logic can answer. Turing suggests that humans are too often wrong themselves and pleased at the fallibility of a machine[27].

Argument From Consciousness

This argument, suggested by Professor Jefferson Lister states:

> Not until a machine can write a sonnet or compose a concerto because of thoughts and emotions felt, and not by the chance fall of symbols, could we agree that machine equals brain.

Turing replies by saying that we have no way of knowing that any individual other than ourselves experiences emotions, and that therefore we should accept the test[27].

Lady Lovelace Objection

One of the most famous objections, it states that computers are incapable of originality. Turing replies that computers could still surprise humans, in particular where the consequences of different facts are not immediately recognizable[27].

This objection was inspired by Ada Lovelace's notes about Babbage's differential engine.

Informality of Behavior

This argument states that any system governed by laws will be predictable and therefore not truly intelligent. Turing replies by stating that this is confusing laws of behaviour with general rules of conduct[27].

2.5 The Foundation of artificial intelligence

The foundation of artificial intelligence is a very broad base. It uses logic, mathematics, economy, statistics and psychology. This section will present a overview of the basics needed.

2.5.1 Philosophy

The basic techniques used to create artificial intelligence can be traced back to the ancient Greek times[6]. To be specific it is Aristotelian principles and laws of **logic** that is still used in todays applications. One more important concept is **Boolean algebra** developed in the mid 19th century by George Boolean at the University College Cork in England. The logic laws of Aristotle and Boolean algebra can be used to simulate human behavior to some extent.

The very first algorithm in the world was Euclid's algorithm for finding the greatest common denominator[6]. This was the first mechanical way of calculation something without the need to think at all. It is the same principle that is used to create weak AI today[6]. Algorithms repeat a problem solution figured out by a human earlier.

2.5.2 Mathematics

The idea of logic in AI can be traced back to the ancient Greeks but mathematically the development begun with the work of George Boyle who worked out the details of Boolean logic. Gottlob Frege extended Bool's logic to include objects and relations, creating the **first-order logic** that is used today as the most basic knowledge representation[6].

Another fundamental of artificial intelligence is to make computers make the most economic decision. Meaning that AI agents should make the decision that the greatest payoff for energy spent or the most efficient choice. The idea is very simple. Make an AI agent that maximize payoff. It can be buying and selling stuff but can also be the energy use of a transportation, for example the transportation route of a mail[1]

Another aspect of making AI that is economic is to make an AI agent that makes decisions that are good enough instead of making the optimal decision every time. Comparable to realtime systems when a approximative value fast is better then a exact value later. Similar to that are decisions that are "good enough" made by an AI agent is the behavior more like a human decision than the optimal performance. Because humans do not always make the optimal decision.

2.5.3 Neuropsychology

However one problem that still is apparent with today's computers is that human brains and computers function fundamentally different. A CPU is 1,000,000 times faster then a human brain in raw switching speed but the human brain is 100,000 times faster at what it does. A human brain contains 1000 times more neurons then a CPU has logic gates and the brain uses all neurons at the same time and a CPU uses only 8-128 logic gates at a time depending on the environment. Scientists

[1] Not the electronic type.

predict that we will have computers that can match a human brain in performance and speed in 2020[6].

Reflexes on the other hand fall under a subcategory of weak AI and are much easier to make then thinking. Imagine that you accidentally put you hand on a hot plate then you will pull you hand away before you feel the pain from the burning. This kind of behavior is easier to imitate with computers then real rational thinking.

2.5.4 Control theory

The goal of **control theory** using AI is to make computers behave optimal[6]. A thermostat is an intelligent component that reads the surrounding and changes its state. But a thermostat cannot plan its actions, therefore it has no artificial intelligence, but it gets the same result that a human sitting with a switch adjusting the temperature to its liking would do.

2.6 The history of Artificial Intelligence

This section will give a summary of the history of the artificial intelligence. It will not go deep into the details of the history, for that see *"Artificial Intelligence: a modern Approach"*[6].

2.6.1 The birth

The official birthplace of AI was Dartmouth Collage in the USA[6]. There the computer scientists John McCarthy, Marvin Minsky, Claude Shannon and Nathaniel Rochester started a workshop including themselves, Trenchard More, Arthur Samuel,Ray Solomonoff, Oliver Selfridge, Allen Newell and Herbert Simon. Their purpose with the workshop was to do research about automata theory and neural

nets. The Dartmouth Workshop lasted for two months in the summer of 1956 and did not accomplish anything of significance except to adopt the term artificial intelligence[6].

During the last years of the fifties and the sixties a lot of relative big steps forward were taken[6]. One of the major once was the general problem solver, or GPS, that were first to imitate human behavior to solve simple board games and puzzles. In its limited way of solving problems it could still make predictions of possible outcomes and choose its actions[6].

2.6.2 The first AI agents

In 1958 McCarthy published a paper describing a hypothetical program, Active Talker, that used knowledge to find solutions to problems. The program was designed so that it could add more knowledge to its database during execution, which meant that it could learn and make depositions it was not programmed for[6].

Arthur Samuel wrote a checkers program that could learn as it played and soon became better at playing then its creator. The program was displayed on TV in 1956.[6] Checkers is, by the way, one of the few problems that have been absolutely solved meaning that the program will win if it is possible to win in a given situation[13].

In 1963 James Slagle made the program SAINT that could solve first order closed-form calculus. In 1967 Daniel Bobrow made the program STUDENT that could parse natural English, interpret the problem and give a correct solution. In 1968 Tom Evan made the program ANALOGY that could analyze geometrical shapes and solve visual problems. In 1973 all these techniques were combined by several other scientists to make the program SHRDLU which could play with colored blocks and solve problems at an infants level[6].

2.6.3 Setback

The early success soon proved to be a disappointment. The simple problems that had been solved in the beginning were only possible to solve in their closed systems and not usable in outside the contexts for which they were designed. One funny example was when the American military funded a project which should make a electronic translator from Russian to English. A test run of the program resulted in the sentence "the spirit is willing but the flesh is weak" translated from English to Russian and back again to "the vodka if good but the meat is rotten"[6][2]. For example, if a human is to misinterpret the expression "pissed", which means different things on different sides of the Atlantic[3], it is considered funny but when a computer does it it is considered stupid.

2.6.4 Knowledge based systems

The next real step forward came with the program DENDRAL witch separated the knowledge from the the production rules used. What DENDRAL did was to analyze spectral lines from molecules bombarded with electrons. DENDRAL was important because it used a large number of special case rules which knowledge could be added later, see Section 4.2.1. DENDRAL was actually a system implementing the principle described in McCarthy's program Active Talker. The technique was compared to a cook following the recipes in a cookbook because many courses are cooked the same way but they have different ingredients and the same ingredients can make different courses[6].

This technique was applied in the medical computer MYCIN that analyzed blood infections and gave a diagnose. The results this expert system produced was above the level of junior doctors. MYCIN used a technique called certainty

[2]After this the military funding to the project was cut.
[3]In Great Britain pissed means drunk and in north America pissed means angry.

factors for calculating which was an attempt to imitate the way the doctors make decisions[6].

The first AI system that was used in commercial industry with success was an expert system called R1. It was used for helping to configure complete computer systems. It was developed in 1980 an it saved the Digital Equipment Corporation $40 million a year[6] by 1986. In Japan in 1981 a ten year plan for creating intelligent computers was made. The project was called Fifth Generation and the purpose was to, within a period of ten years create a computer that could actually think[6]. This was of course immediately followed by similar plans from the Unites States and Great Britain.

2.6.5 The AI winter

In the period of 1980 - 1988, AI was a big part of the computer industry. But when they failed to deliver what was promised many companies went bankrupt. This period was called the **AI winter**[6].

In later years it has become more common to create real working AI systems rather then making new theories. The reason for this was partly because of the failure with strong AI during the AI winter. The effort to make weak AI became more attracting since computer systems do not need to think like a human to do whatever it was meant to do as long as they did it successfully[6].

In the middle of the 1980s, the research of neural networks begun anew and is still an area where much of the scientific effort is put[6].

2.6.6 AI becomes a science

It was in the time after the AI winter that people learned that it is time to start using scientific methods in the research. One of the reasons that so many unrealistic promises were made in the early 1980 was that the AI industry had not

analyzed their premisses in a scientific way. They had just set out to create strong AI without knowing what algorithm to use. Now hypothesizes must be analyzed, tested with rigorous empirical experiments and the data must be evaluated before the hypothesis will be accepted as scientific[6].

Speech recognition illustrates this process. It started in the 1970s as ad hoc with more or less trial and error. Speech recognition survived the AI winter and has today become common in many applications. It was because of the use of the Hidden Markow models (HMM), which is a combination of mathematics and lots of empirical data in the field of speech recognition. The related field of recognizing handwritten text has developed in a similar from ad hoc to science in a period of 20 years[6].

2.7 AI today

Today AI is a big industry. There are AI agents planning everything from train routes to space travel, monitoring everything from automatic greenhouses to chemical experiments. There for example is an AI system called ALVINN that can steer a car[6]. It used image recognition to keep the car on the road and to avoid accidents. ALVINN controlled a minivan for 98% of the time on 4590 km of road across the United States. We have lots of diagnostics systems using probabilistic calculations to help doctors make decisions about how to treat patients in hospitals. As well as expert diagnostic systems, robotics are used in hospitals to help doctors preform critical surgery. And of course in the manufacturing industry there has been robots helping to make just about everything for the last decades. In language understanding almost every text processor has built in spell checking and grammatical checking, this paper was written in one of those.

These topics are called **mainstream AI**. Which is different from **game AI**,

which this dissertation is about.

2.7.1 Robotics

Robotics is the manipulation of physical objects which not necessarily have to be guided with an AI agent but the things that come to most peoples minds when they hear the word robot is AI. In hospitals doctors are assisted i surgery with robot arms that are only partly controlled by an AI agent. The robots in car factories on the other hand are entirely controlled by an AI agent.

2.7.2 Spam filters

Many e-mail services offers a spam filter service which makes a scan of all incoming e-mail and sort the ones that are believed to be spam in a special folder. An AI agent reads all incoming mail and recognize the common used phrases in the mails marked as spam by the user. The AI agent learns more and more and becomes better and better at recognizing spam each time the user marks a letter as spam. This of course requires the user to actively teach the AI agent what mails are spam.

This technique is called Bayesian filtering because it uses an improved version of the theorem presented by the British mathematician Thomas Bayes.

2.7.3 Virus scanners

Virus scanners can use a similar technique that spam filters use to recognize the patterns of new unknown viruses. But of course the paranoia level needs to be higher with virus scans than with e-mail scans.

2.7.4 Security agents

An AI agent that handles network security could for example recognize when someone, meaning one IP address, repeatedly try to log in to an account and stop responding to that address because it could be someone that is trying to crack a password using brute-force.

A security agent could also gather statistics about what the user does when logged in. It could store the most common used application for each user[4] and react when a user does something out of the ordinary. This would be a security expert system.

2.8 Image recognition

Image recognition, **object recognition** or **computer vision** are synonyms for the way AI agent gather information about the outside world. The term perception is used when more factors then visual input is used. In the industry this is used quite a lot in the manufacturing industry to make the robots put the parts in the right place. But in this dissertation the term image recognition will be used because the AI agents in a 2D game only needs to recognize two dimensional images.

2.8.1 Introduction

The first thing needed in order use image recognition is in fact an image. When it is in robotics or industrial machines the image is gained by using a camera. The result of that is a picture made up of pixels or raster graphics, that is, a picture made up by a finite number of squares. This is the image itself that shall be interpreted.

[4]This is a little 1984 warning[18].

With, for example, a robot walking on Mars every single detail in the image is needed to be recognized and analyzed. But an industrial robot that only works within a given context in an specific environment only parts of the image needs to be scanned and analyzed, for example a robot that places the engine in a car on a conveyer belt needs only to know the position of the car in order to place the engine and the position can be gained by recognizing the corners of the car and from the position of the corners the place that the engine shall be placed in calculated from pre programmed knowledge.

2.8.2 Reading a image

Reading a pixeled image and transforming it into mathematical formulas that is usable for an AI agent is a complicated task. Begin with thinking that a computer animated frame for motion picture takes several hours to render from mathematical formulas into raster graphics. Lets call this function that transforms mathematical formulas into an image

$$image = f(math)$$

where f is the function that renders the image. Then consider the facts that a opposite operation of creating mathematical formulas of an image,

$$math = f^{-1}(image)$$

, is more complicated and in order to work well in needed to be done in realtime. Further, function $f^{-1}()$ is not really an inverse of the function $f()$ because all the depth of the image is lost. Of course this can be partly recovered with the use of stereoscopic vision, but that is a chapter of its own. Furthermore an image recognition agent cannot tell the difference between a large poster of a car and an actual car[6].

This is somewhat related to the fact that mirages in the dessert may look like lakes. It is objectively true that, under certain circumstances, a mirage in the dessert give our eyes the exact same information that an actual lake would. In that context we have to use our mind and figure out if it is a lake or an mirage. In similar ways an AI agent have to use its knowledge database calculate the probability of what is see really is the most common thing. Overall in the world the image of a lake is most likely a lake but in the dessert the image of a lake is most likely a mirage. Just like a human the AI agent have to know its context otherwise a lot of stupid decisions can de done.

2.8.3 Transforming the image into data

The technique for transforming **raster graphics** into data described in *"Artificial Intelligence: A modern Approach"*[6] is done by first stripping the image from all color into a greyscale image. Then the image read pixelline by pixelline in search for great contrast changes. When a great change in contrast is found it means that there is an edge on that position in the image and it is noted by placing a pixel on that position. By repeating this for every pixelline in the image, lines will be drawn at the edges of the objects in the image. These pixels will form the edge lines which the AI agent later will interpret.

Figure 2.1 is taken from *"Artificial Intelligence: A modern Approach"*[6] and it illustrates how an image is transformed from raster graphics to lines by finding the contrast changes and drawing edge lines at the edges of the objects in the picture. In the right picture there are black dots which should not be there. These are "noise" edges caused by disturbance in the photograph. This can be solved by a smoothing algorithm that recalculates the brightness of each pixel by the mean value of its neighboring pixels. If a smoothing algorithm had been done on the photograph before it was scanned for edges, the noise edges would disappear[6].

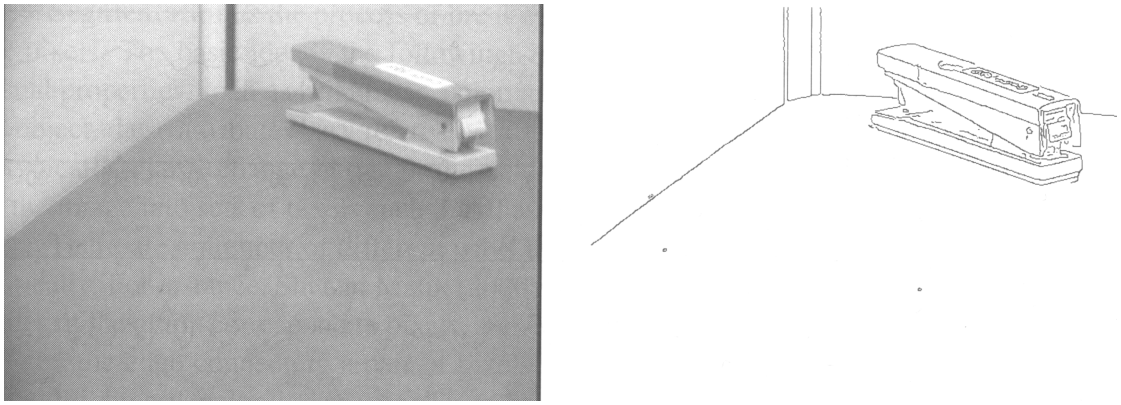

Figure 2.1: A photograph of a stapler (left) and edges computed from the photograph (right).

2.8.4 Recognizing objects

The next step is to identify the corners of the object by finding the adjacency points between the edges. Figure 2.2 shows the adjacency points from the stapler in Figure 2.1, Figure 2.2 was also taken from *"Artificial Intelligence: A Modern Approach"*[6]. The reasons for drawing edges for finding the adjacency points instead of using the pixeled image directly is one: Data reduction, there is a lot fewer lines then pixels on the image and two: Illumination indifference, the edges will with a good algorithm appear at the same coordinates on the image no matter the lighting circumstances[6].

These lines are abstractions of the image and abstractions is the computer science method of creating concepts and the purpose of concepts is: "...to reduce a vast amount of information to a minimal number of units."[14].

As soon as the adjacency points of an object in an image is known the remaining problem is to identify what it is. This is done by a form of parsing of the points of the object and comparing it to the points of the objects in the object/image recognition AI agents knowledge database. If the object was always viewed from

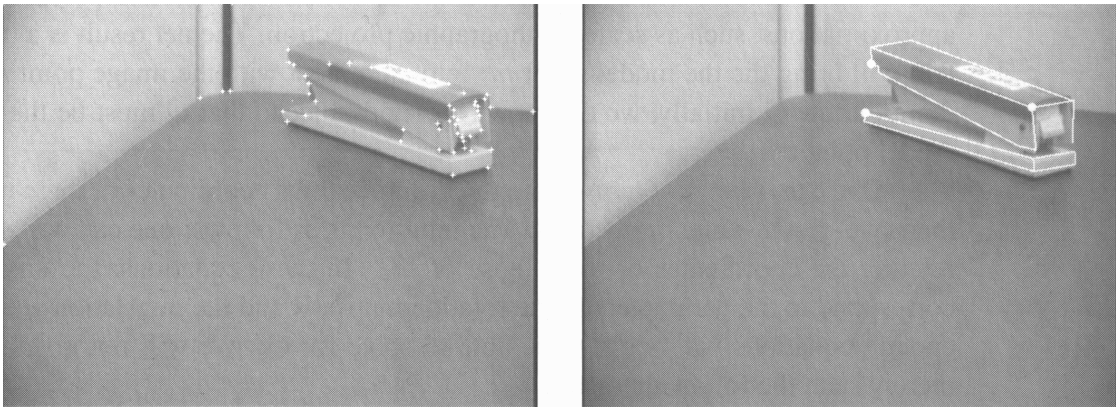

Figure 2.2: The same stapler as in Figure 2.1 but with the adjacency points between the lines (left) and the lines of the object known by the AI drawn out(right).

the same angle the search algorithm for finding the points are rather simple, for example a fingerprint search technique always use the same kind os image, a perfect print of pattern of the fingertips.

But since the AI agents sees the object from an unknown angle, in the case of a robot using a camera, and it is possible to see a object from any angle in the real world the AI agent has to test every single possible angle on the object in its database before it can identify what kind of object it is looking at. Imagine it like the AI agent is rotating the object in its memory until it find one with adjacency points exactly like the object in image. There is quite a lot of combinations possible even after the object type in known.

The worst case scenario for visual object recognition is

$$O\left(M^4 N^3 \log N\right)$$

according to an algorithm presentend in *"Artificial Intelligence: A Modern Approach"*[6]. Where M is the number of models in the knowledge database and N is

the number of points in the object.

But of course, in a 2D computer game the picture is already known and the problem with getting the image is not there. And no objets in the game are necessary to really identify because all objects are part of the game system so the object recognition recognition problem is not present to begin with in a game. What is left is knowing positions relative to the AI agent in the game. In summation; in games it is possible for the programmers and designers to cheat around real AI problems.

2.9 Deep Blue

Deep blue was the first computer ever to defeat a human world champion in chess. It won over Garry Kasparov, witch was at the time the world champion in chess, in a series of games in 1997 with the score 3.5 - 2.5 to Deep Blue. Deep Blue had actually lost the first series of games against Kasparov in 1996 but was upgraded in between the games. The interesting thing about the first series of games was that Kasparov lost the very first game but managed to learn how the computer played and defeated it. But after the upgrade the computer knew how to play against Kasparow and won. Kasparow was very angry for loosing and demanded rematch but IBM, that had made Beep Blue, declined. He accused IBM for designing Deep Blue for the sole purpose or defeating him and that it would stand very little chance against other world class chess players. IBM retired Deep Blue since the value of all IBM stock had increased by $18 Billion in total and Deep Blue had furfilled its purpose.

Deep Blue was a state of the art AI agent using a 30 node RS/6000 system with 480 additional special purpose VLSI chess processors. The program itself was written in C and run on the AIX operating system witch was a version of UNIX.

The complete system was capable of performing 100 million position evaluations per sec ond[29][5].

2.10 Summary

Artificial intelligence is a special field of computer science that actinally was one of the earliest ideas in the field but did not really gain success in the industry until recently. A lot of money was invested in projects that had unrealistic goals and the disappointments were great when the promises were not made and AI suffered great setback.

AI has its base in over two thousand year old philosophy, modern day mathematics and psychology.

AI can be used literally to anything that we want to use it as because it is such a broad and useful field of computer science. And in recent years it have exploded because so many new scientists have chosen AI as their field. It is one of the most popular fields that other scientist wished they were in if they where not already involved deeply in other fealds[6].

[5] Remember that this was in 1996.

Chapter 3

Introduction to game programming

Games have in later years become a large industry but game programming have not yet become common in academic teachings. Many academics have even missed the boom entirely. Therefore this chapter will give an introduction to computer games to a reader who has no knowledge or experience of games.

The first section is about the three different kind of game hardware platforms and the differences between them. The next section explains the difference about two dimensional graphics and three dimensional graphics. That section also includes a little history. Following is a listing of the terminology needed for programming games and some of what is used only when games are played. Next follows descriptions of the most common game genres. The description of game genres are only stereotypical, there is a lot of games that do not fit into any category or is a combination of many genres. Then follows a section about game application programming interfaces. And the last section is an introduction to how games could be programmed.

3.1 Game platforms

Computer games come in many different formats and genres that go by different names but it is in essence always a computer in one shape or another. Many different words can describe the same thing for example does the words console game, video game and most common in Swedish "TV-spel" all describe the same platform. Games on the same platform may be very different from each other. These terms may be new to some but is crystal clear to a player or game programmer. This subsection will describe these platforms from the players point of view. From a programmers point of view; all game plattforms are computer games.

3.1.1 Arcade games

Arcade games were the most common form of computer games in the 70s and first half of the 80s. These games were the big coin activated game machines that was put in shopping malls that parents dropped their kids off at, with some coins, and the parents could do the boring shopping and the kids would be happy. A arcade game is a special computer with a ROM memory containing only the game. All arcade and has a built in screen for displaying the game and a input device in form of a joystick, searing wheel or mouse ball and often one or more buttons. In order to play these games you have to insert a coin and you get a number of credits, the number of chances you get in the game before you have to insert another coin in order to continue, when you fail in the game you lose a credit and when you have no credits left you cannot play any more.

Classic arcade games include *Pac-Man*, *Donkey Kong*, *Galaxian*, *Space Invaders*, *Streep Figther II*, *Defender* and *Bombjack*.

To play arcade games could become very expensive so in the 80s arcade games became less popular because of cheeper game consoles, with comparable hardware

capacity to arcade games, become available. Since arcade games could be very big and expensive, the hardware for each game could be custom made which resulted in superior graphics and sound compared to game consoles and home computers at the time. Somtimes entire stores at malls are devotet to arcade games these special halls contanining nothing but arcade games are called **arcade halls**.

3.1.2 Console games

Console games are mostly called **video games** or in Swedish **TV-spel** but the correct term is console game. A **game console**, **video game console** or in Swedish **TV-spelskonsoll**, is a computer specially designed for playing games. The games themselves for a game console are stored on **cartridges** or **compact discs**. Cartridges were used from the start of the home console industry up to Nintendos console Nintendo^{64} which was the last console to use cartridges. The first successful console that used compact discs were Sonys Playstation. The advantages of cartridges is that the cartridges contain hardware which makes it possible to upgrade the graphics and sound of a game, which goes beyond the consoles original capacity and that it is possible to save game data on the cartridges themselves which is not possible on compact discs. The advantages of compact discs is that is is easier to make consols backwards compatible with old games and it is cheaper to make compact discs then to make cartridges.

Console games is the most common form of game platform. Mainly because consoles are purely designed for playing games which makes the consoles much cheaper than a computer. Microsoft's console X-Box costs about 1'500 SEK, when this was written, compared to a personal computer witch costs about 10'000 SEK. And if the computer shall be considered good the price rises to about 20'000.

Besides, consoles are more user friendly than ordinary computers. All that is necessary to do in order to play a console game is to insert a game in the console

Figure 3.1: Super Mario Bros, the first screen scrolling platform game.

and push the power button. No installation in necessary in order to play on a game console, all you need to do is connect the console to the television.

The leading console in the market today is Nintendo's Gamecube, Sony's Playstation 2 and Microsoft's X-Box. SEGA used to be a big competitor on the console market but since their Dreamcast console failed miserably SEGA became a **third party manufacturer**. A third party manufacturer is a company that only make the games themself, or just the **software**.

Hand held game consoles are included in the category console games. A hand held game console is a game console small enough to be hold in ones' hands. Nintendo's Gameboy is by far the leading hand held console[1].

3.1.3 Computer games

Computer games are games that are played on a personal computer. There is nothing special about game applications compared to ordinary application except that the game is for entertainment purposes. The disadvantages of computer ga-

[1]This was written in February 2005. By the time this dissertation is finished Sony's hand held console, Playstation Portable PSP, will be launched in Europe

mes is that computers is expensive compared to game consoles and that computer games need installation and sometimes special drivers for the computer hardware. But the advantage is that there is no need to pay a licence to the hardware manufacturer when you release a computer game. Since the purpose of a computer is that you is suppose to do whatever you want with it it is much easier to make your own computer games compared to what it is to make a game for a game console.

The usual problems with viruses, spyware and other security issues are apparent with comperes but that have nothing special to do whit the games themselves. Except maybe when a trojan in built in to a freeware game, but security is not the topic of this dissertation.

3.2 2D and 3D games

In the beginning of time[2] all graphical computer games were flat two dimensional games. In **2D games** the graphics in the game is displayed in two dimensional pictures or is drawn with vectors in a two dimensional plane. The best selling 2D game ever is Nintendo's *Super Mario Bros* released in 1986[30], see Figure 3.1. This game spawned the popularity for 2D platform games. *Super Mario Bros* was not the first platform game. The very first platform game ever[19] was *Pitfall!* released in 1982, se Figure 3.2. But *Super Mario Bros* was the first 2D platform game that scrolled the screen. In *Pitfall!* the player moved between one static screen at a time, so *Super Mario Bros* is still refereed to as the grandfather of platform games in most cases. Even though it is technically wrong.

Super Mario Bros is the classical stereotype of a computer game. It used cartoonish graphics with a simple gameplay made it the the best selling game in the world for years to come. Two dimensional games were in total domination of

[2]Viewed from the perspective of computer games this is 1962 when Spacewar was created on the supercomputer PDP1 om MIT.

Figure 3.2: Pitfall!, the worlds first platform game.

the market until in the middle of the 1990s. This were because computers were much slower in the commercial market then and 3D were only a tool in architecture, simulation and in some extent motion pictures.

The characteristics of a 2D game, except the graphics, is that the main character or objects can only move in two dimensions, just like chess. All board games are 2D games, if you would make a graph of the board. But in a 2D computer game the board is the screen and it is often a game of very different characteristics than a board game.

In a **3D game**, the graphics are not in still pictures but are made up of vectors on the screen rendered in realtime while the game is running. The simplest geometric shape using straight lines i a triangle. So all 3D computer graphics are made up of triangles linked together to make bigger shapes, like squares, spheres and other polygons. The surface on the polygons can be monochrome or have a picture attached on it. The view in a 3D game is not fixed like in a 2D game but dynamic due to that the graphics is rendered in realtime during the execution of the game. Therefore it is possible to change the view in the game by moving the

Figure 3.3: Ultima Underworld, one of the first real 3D games.

camera. This feature adds simplicity in adding the feeling reality to the game. In a motion picture the effect of moving the camera add to the drama in the movie, the same principles can be applied in a 3D computer game.

The first game that used 3D graphics was *Ultima Underworld*, see Figure 3.3 released in 1991 by Looking Glass Studios. *Ultima Underworld* was the start of the **first person perspective** in computer games[3]. Later 3D games started to use **third person perspective**, which means that the view the player has is outside of the main character just like if it had a small camera was hovering behind the main character. In $Mario^{64}$, released in 1996, it was possible to see the camera in the mirrors in the game, see Figure 3.4.

There are some hybrids between 2D and 3D games. *Snake Rattle Roll* was a 2D game graphicly but was 3D in the gameplay. Which could cause some confusion in some places of the game. *Dungeon Sedge* was 3D in graphics only, the player could only move two dimensions. This was a 2D game in the **gameplay** with a

[3]Some games had used first person perspective but only with still pictures.

Figure 3.4: Mario[64], a third person perspective game. Notice the reflection of the little guy hovering on the cloud with the camera in the mirror.

3D wrapper so to say.

3.3 Terminology

This section will state and describe the terms used in this dissertation. Since this is a dissertation about 2D games only the terms used i 2D games and game programming will be mentioned.

Computer game

This term will be used for games on personal computers, video game consoles, arcade games and other. Since there is no difference from the developers side one term will be used for games for all kinds of platforms[4] Throughout this dissertation.

Game engine

The **game engine** is the core of every computer game. It handles the backend information and calculation like the graphics, physics, sound, input/output, network

[4]In this context platforms means operating system or game console

if it is a multiplayer game and all other technologies needed to execute the game. All game engines have a kernel just like an operating system.

Player

The **player** is the user of the game application. The player is the human sitting in front of the computer or TV playing the game. In most cases the game has only one player but many games have the option to have several players who compete or cooperate in the game. Another word commonly used as a synonym for player is **gamer**.

Enemy

Enemies are agents in the game that use AI in one form of another and are hostile to the player. There may be characters in the game that are not hostile to the player but enemies are by far the most common.

Hitpoints

Hitpoints is a unit used to measure the health of the characters in the game and is in some cases just called **health**. It is measured from zero to a set maximum level. It is common to represent hitpoints in the form of percent, that is, the maximum amount of hitpoints is 100

The player and the different kind of enemies often have a different amount of hitpoints to make the enemies tougher the the player. Each time a successful attack damages a character the character looses some of its hitpoints and when the characters hitpoints reaches zero the character either dies or loses consciousness.

Almost all new games use hitpoints or a variety of hitpoints. Games that do not use hitpoints today are rare.

Figure 3.5: Guybrush Threepwood from The Secret of Monkey Island™. The rectangle will disappear when it is sprited on the screen.

Bitmap

Bitmaps are the simplest form of computer graphics, **raster graphics**. It is simply a two dimensional matrix of integers representing a color.

Sprite

Sprites are the graphical objects that are visible on the screen. They may or may not move on the screen. The technique of animating a sprite is exactly the same as then one that is used in cartoons. One motionless image is quickly changed to another. Since the average frame only lasts for about 1/40 of a second the many changes of images on the sprite will animate the sprite.

One important characteristic of sprites is that some of the pixels on the bitmap are transparent when drawn on the screen so that every sprite looks like it was drawn directly on the picture used as background. This way one of the colors on the smaller bitmap disappears and the character or item is perfectly integrated on the background. In Figure 3.7 the rectangle around the character, see Figure 3.5, disappear and is not seen on the picture.

There is no theoretical limit for how many sprites can be visible on the screen at the same time except system memory.

Frame

A **frame** in computer game programming can refer to two things depending on the context.

First a frame is the bitmap image displayed on the screen for about 1/40 of a second. When it is said that the screen has a **frame rate** of 40 **fps**[5] it means that the game updates the screen 40 times per second, or has 40 Hz.

A sprite also has a frames. This is the bitmaps that is changed on the sprite when a frame on the screen is updated to make the animation. Just like the animations in a cartoon is made by the rapidly changing images the rapidly changing sprites make the objects move in a game.

Input

The input are the signals the game gets from the player using the computer hardware. This include **gamepad**, **joystick**, **keyboard**, **mouse** and all subcategories to these devices. The input is interpreted and used by the game engine once every frame. Even though there are many input signals the word input is used in singular.

Stage

A **stage** is a closed part of a game. When a player is on a stage the player have do finish a goal of some kind. It is often just to move from one point A to B. So when the player finish stage 1 the game continues on stage 2 and so on.

Level

A **level** is sometimes called map or landscape. A level is a area in the virtual world that the player can move around on. It contains all the sprites that the player can

[5]Frames Per Second

interact with, including the enemies. In programming the term is used describe the map of the game.

Another thing the word level is used for is in the context of difficulty level. The higher the difficulty level the harder it is to play the game.

Level can also mean experience level. More of this follows in the Role Playing Game section.

The word level is sometimes used as a synonym for stage.

Blocks

The **blocks** are the **terrain objects** in the level that the player can walk, jump, climb, crawl on and in other ways interact with. They can come in the chapes of platforms[6], floors, walls, ceilings, ground and many more. The term for block wary from context to context but in the data structure in the backbone implementation they are all the same type. The term used in this dissertation will wary depending on the context where it is used.

In a 2D game each block has at least one sprite or more if the block is animated. In 3D game a platform is a geometrical shape made of vectors.

Lagging

In gaming terms lagging means that the time between frames is increased resulting in that the game runs slower then it is suppose to. The causes can be a slow computer or slow network connection.

[6]In this context platform means a object in the game that the player can jump on

	Platform	FPS	Strategy	Simulator	Beat'em'up	Shoot'em up	RPG	Adventure	MOO	Puzzle	Sim
Platform		✗		✗							✗
FPS											✗
Strategy					✗	✗					
Simulator		✓	✓								
Beat'em'up	✓					✗				✗	✗
Shoot'em up				✓						✗	✗
RPG	✓	✓	✓	✓							
Adventure	✓	✓	✓	✓	✓		✓				
MOO			✓	✓			✓				
Puzzle	✓							✓			
Sim									✓		

Table 3.1: Diagram showing how different genres often and seldom are combined.

3.4 Game Genres

Most games fit into a **category** or **genres**. This section will give a introduction to some of the most common. Although most games fall in one of these genres some games combine elements from two or more genres. As seen from Table 3.1; first person shooters can have role play game elements but a first person shooters cannot be combined with a platform game, then it will no longer be a first persons shooter by definition. Table 3.1 shows how these genres can be combined. A check sign means that the genres can be combined and an X means that the cannot be combined. This picture is not an absolute truth about genres, it merely shows common combinations of genres.

3.4.1 Platform games

A 2D **platform game** is a game where the player sees the game from one side, see Figure 3.1 and Figure 3.2. The challenge in a platform game is to jump on different platforms, over gorges and get past obstacles. The control in a platform game is very simple, the player use one button each to move left and right and one

to jump[7] and in most cases one button to use a weapon of some kind. When the player pushes the move right button the main character moves right on the screen and when the player pushes the move left button the main character turns around and moves left and when the jump button is pushed the main character jumps.

Platform games were the most popular game genre for about ten years spawning a lot of games witch vary in quality from masterpieces that still sell, like *Castlevania: Symphony of the Night*, to complete crap that newer got of the shelves. The popularity of platform games in the early days of the really big industry made a lot of people associate computer games with 2D platform games during the 2D era.

When 3D became a standard in games there were few 3D platform games but 3D platform games started to arise in the later half of the 1990s with the game consoles Sony Playstation and Nintendo64.

For some reason not so many platform games for computers have been released. They have been released almost exclusive for consoles and arcade games and nearly no platform games have been realest on arcade for the latest ten years.

3.4.2 First Person Shooter games

First Person Shooter games, **FPS** for short, games are viewed from the eyes of the main character. The game is controlled by changing the direction the character is looking and walking forwards, backward and strafing from side to side. The weapons used are fired directly forward in the direction the player is looking/aming. Nowadays the most common way to look in different direction in a FPS game is to use the mouse and the keys on the keyboard or the mouse buttons are used to move in the level. The goal in the most simple FPS games is

[7]There exceptions, in some platform games there the main character cannot jump, like in *Bionic Commando*, but those are rare.

Figure 3.6: Wolfenstein 3D, the worlds first First Person Shooter.

to get from one point to another and it will be enemies in between that the player has to kill. The challenge in FPS games it to control your reaction speed and your eye-hand coordination. FPS games can be challenging since it is hard to hit moving targets when the player moves in order to avoid shoots from the enemies.

First Person Shooter games were introduced to the world with id software's[8] *Wolfenstein 3D*, see Figure 3.6. Released in 1992, *Wolfenstein 3D* had great graphics for its time and was a fast action game. Popular FPS games today are *DooM*[3], *Halo 2* and *Half Life 2*.

3.4.3 Role Playing Games

Role Playing Games, or **RPG** for short, are games where the player controls a character or a group of characters in the game and plays the roles in a story told by the game. The challenge in RPG is not the finger dexterity of the player but

[8] A computer game company. Id is the raw and brutal part of the human brain that was named by Sigmoid Freud.

rather solving problems, finding the way out of labyrinths and have a good tactics in battles. In most RPG the story is not told linear but the order and path it takes depends on the different choices the player makes during the gameplay.

One important trait of RPG is that the characters in the game grow more powerful as the game proceeds. This is represented by **experience points**. Whenever the player accomplices something in the game the character or characters in the game gain experience points. The more experience points the characters in the game have the more powerful they are. When a character earns enough experience point the characters advance in **experience level**, or simply **level**. Not all RPG systems have levels but it is very common.

In RPG the players character have certain **skills** that is represented by a integer. The higher the integer the greater the skill. During gameplay when the player uses one of the character's skills a random number is generated and depending on that random number relative to characters skill decides if the action was successful. These random numbers are meant to simulate the roll of the dices in ordinary paper RPG[17]. This is called a **skillroll**.

A typical RPG is set in a **science fiction** of **fantasy** world. This is because spells and high technology are just more suitable for gaining in power when the character gains experience.

It is quite common for RPG to be licensed by a real board role playing game, an example is **Dungeons & Dragons**. These games use the same rules and systems as the corresponding board game. A example of this is *Baldur's Gate*, a game set in *Forgotten Realms*. But most games use its own game system that has nothing to do with a board game. *Fallout* is a successful game that used its own "rules" so to say.

Hack n' slash

Hack n' slash is a sub genre of RPG. The main purpose of a hack n' slash game is to kill enemies and collect the treasures they drop when they die. Also the player gains experience for each kill. Some very simple puzzles and problems are included in most cases. The story is not that important in hack n' slash games.

The concept of just killing everything that appears on the screen sound boring in theory but it is exceptionally entertaining because it talks to a basic instinct within us, hunt and gather, when the player kills a enemy, gains experience from the killing and collect the things the enemy drops the players instinct of hunt and gather is satisfied.

The most popular hack n' slash game today is *Diablo 2*, it is a relatively old game but it is was a so big hit when it was released that it still is the most popular hack n' slash game in Sweden[20].

3.4.4 Adventure games

Like RPG the telling of a story is a important part of **adventure games**. One of the most important trait of adventure games is exploration, the player gets to explore the game freely in any order chosen just like in RPG. It is hard to draw a explicit line between adventure games and RPG because they share so many games that have the characteristics of both genres.

Since there are many sub genres of adventure games it is no standard gameplay or control system. Some adventure games have the gameplay of a platform game, some are in first person perspective, some of the earliest were entirely text based, some controlled by clicking the mouse and some are viewed from above the head of the main character.

Action adventure

Adventure games are a lot like RPG but have one important difference! There are no experience points. So the character development and statistics of the main character is not so important, and in some cases not important at all.

Action adventure games focus more on the reflexes of the player then ordinary adventure games and RPG. Although some games, like the **The Legend of Zelda** series have all the traits of a RPG except experience points[9]. Action adventure games are sometimes just like any other action game with a complex story and are hard to point to either category. Some adventure games are like a 2D or 3D platform game where the player is free to explore the the game in any order chosen rather then linear. It is discussed what genre these games are but since platform games are more of a **gameplay genre** then a game genre, those games are considered adventure games by this author.

Point and Click

Point and click games are adventure games focused entirely on puzzle solving. This is the games where the main character walks to the point of the screen that the player clicks. The most common way to perform actions in the old point and click games were to click on a action then on the object to perform the action. *The Secret of Monkey Island*^{TM}, released 1990 in see Figure 3.7, is the stereotype of point and click games with its panel of actions, inventory and mouse cursor in the lover left part of the screen.

Conversation is a important part of point and click games. When the player engage in a conversation with a **NPC**[10] a number of alternative things to say is displayed on the screen and the player can choose in which way to guide the

[9] *Zelda II: The Adventure of Link* used experience points but it was the only game in the series that did this.

[10] Non Playing Character.

Figure 3.7: The Secret of Monkey Island™, a classic point and click game.

conversation.

3.4.5 Strategy games

Strategy games are games where the player has to plan the actions in forehand. The most common strategy games are combat strategy when the player controls an army and has to collect resorters of some kind in order to get new troops and pay for the upkeep of the existing ones. In battle with enemy troops the player has to place the controlled troops in a favorable position, mostly this means on high ground or behind some kind of shelter.

Most strategy games consist of an equal amount of combat planing and resource gathering. The games are normally controlled with the mouse pointer. The player clicks a unit to select it then gives the unit an order by clicking again somewhere on the map or on another unit. The view is typically from above or slightly angled from above.

Figure 3.8: Dune II: The Battle for Akkaris, the first Realtime Strategy game.

Realtime Strategy games

Realtime Strategy games, or **RTS** for short, are strategy games everything happen in realtime and the players reflexes and quick thinking mater for the outcome of the game, contrary to turn based where only planing matter. Realtime strategy often consist of more fast combat action then the planing of collecting resource, collecting resources is often very simple since the focus of the game is on the combat.

Turn based strategy games

Turn based strategy games was the first kind of strategy games. This was because turn based strategy games do not need fast calculation by the AI agents in the game. The AI could get all the time it needed between turns. This meant that turned based gamed do not have as high hardware requires as RTS games and thus occurred earlier in history. The concept is simple, the player have an

infinitive amount of time to plan the moves and balance the production[11] and when the player feel finished the player press the next turn button.The economy in turn based strategy is much more in focus then in RTS games. This is because in turn based strategy the player often controls the entire production of recourses then rather just a small combat unit or a military base. Turn based strategy games often do not contain levels but the entire game contain of one large level. In *Masters of Orion 3* this is a galaxy.

3.4.6 Beat'em up

Fighting games or **beat'em up** games are action games where the purpose is to defeat your opponent in hand to hand combat. The first games of this kind were single player games where the player walked forward and faced hordes of enemies that were relatively easy to defeat. In later years it has become more common that these games are pure versus battle games, meaning that there are only two characters on the screen at once. These games are popular when human players play against other human player instead of only the AI agent. The combat in beat'em up games are often inspired by real marshal arts and the characters are often portrayed as masters of their marshal art.

3.4.7 Shoot'em up

Shoot'em up games are games where the the player controls a small spaceship or equitant that fly forward on a self scrolling screen swarming with enemies. The challenge is to shoot down the enemies without being shot self. In the greatest majority of shoot'em up games there is at least one kind of ammunition that is infinite so the player can focus entirely on the action. The path taken in a shoot'em

[11] When played in multiplayer the players often have a time limit.

Figure 3.9: Gradius, a side scrolling 2D shot'em up game.

up game is largely predetermined, the screen scrolls all the time and the player cannot alter i very much except sometimes in only one dimension perpendicular to the direction the screen is scrolling.

The first shoot'em up game ever, **Space Invaders**, did not, like some other of the older games, scroll the screen. Instead, the player could only move in one dimension with all the enemies on the screen at the start of the level.

New shoot'em up games scroll the screen and the player can move in all directions in the two dimensional plane. There are however exceptions, *Star Fox*[12] is a 3D shoot em up game that has not crossed the line to be a simulator because of the fixed path the planes take and that the view is not from inside of the planes but from behind.

[12] *Star Fox* were renamed to *Starwing* or *Lylat Wars* in Europe because of a Danish vacuum cleaner called Star Fox.

3.4.8 Simulator games

This is the kind of games where the player once again sees the game world from the eyes of the main character, like in FPS games, but this time it is from inside the cockpit of a plane or from the driver seat of a car. As the name of the genre, the purpose of the game is to simulate the feeling of really flying a plane or driving a car. These games are often played with a large force feedback joystick or steering wheel to add to the experience of flying or driving. Some simulators resemble cockpits of real airplanes and some fictional planes. As well as some joysticks resemble the controls in real airplanes. **Simulator** that simulate a car race is called a **racing game**, but it still falls in the category of a simulator. These games often have the option of viewing the vehicle from behind and can be played with a steering wheelgas/break pedals and gearshift stick accessory.

Simulators that overdo it with fancy equipment and large props can be found in the arcade halls. Examples are arcade games where the players sit on a natural size motorcycle and control the game with the switches and pedals resembling those of a real motorcycle. It is these kind of gimmicks that have made arcade games survive today.

3.4.9 Sim games

Sim games are special kind of simulator/strategy games that simulates something else then a real simulator. They are strategy games with a special touch. The first Sim game *SimCity* simulated a city. When *SimCity* first was released it was considered a strategy game, which it actually is, but over time Sim games has become a sub genre of its own. In *SimCity* the player was the major of a city and had to plan the economy and build the buildings in the city. The player could raise and lower taxes, build and demolish buildings, build roads and much more.

SimCity has no real goal to complete. The purpose is just to play the game and build a so large city as possible.

There are a lot more Sim games simulation different this. Sim games include *Sim Earth*, *Sim Ant*, *Sim Farm*, *Sim Town*, *Sim Copter*, *Sim Park*, *Sim Life*, *Streets of Sim City* and *The Sims*.

3.4.10 Sports games

Sport games are as the name suggests games where the player takes control of an athlete or a team and plays a sports game in the computer. Since sports games are not based on the concept of computers it attracts a lot of players that would not normally play computer games to start playing computer games. Thus making sports games one of the best selling computer game genres.

The games themselves use the rules of the sport it is trying to simulate.

3.4.11 Massive Multiplayer Online games

These are games played entirely on the internet. This is not really a game genre itself but a way of implementing the playing. The difference between a ordinary multiplayer game over a network and a **Massive Multiplayer Online Game**, **MMOG**, is that all players playing the same game can interact at the same time. Contrary to a ordinary multiplayer game where only players connected to the same private server can play together. Usually not more than at about 50 players play on the same server in an ordinary multiplayer game. In a MMOG, all players connect to the same server so thousands of players "are on the same server" and playing together in cooperative play or against each other. This of course demands a special kind of game architecture and financing. Most MMOG require a monthly fee in order to play. This is for paying for the internet server that need constant upkeep.

The gameplay of a MMOG is just like a ordinary game of the genre except the important fact that it is not possible to finish a MMOG in the same sense it is possible to finish a single player game. The game continues without end even if only one player is online and playing since the game itself only exist on the central game server or servers and not on all the different players computers.

The most common for of MMOG are **Massive Multiplayer Online Role Playing Games**, or **MMORPG**. A MMORPG is RPG that it played online on a massive server together with thousands of other players. This is the most common form of MMOG because of the character building element in RPGs i suitable for online gaming. When a player plays the players character becomes a little more powerful for each time the player plays the game. Thus the competition against other player about who has the most powerful character is a exiting element in MMORPG.

3.4.12 Puzzle games

Puzzle games are games that have a very different gameplay then the games mentioned abode. Puzzle games are, as can be figured out from the name, games where the player solves puzzles. This genre attracts other kind of players that would not play any other kind of games and only plays puzzle games. My mother, for example, only plays puzzle games and no other games. The purpose of most puzzle games is not to finish the game in the traditional way a game is "beat" but rather to collect as many points as possible or just solve the puzzle quickly.

A popular puzzle game is *Tetris*, see figure 3.10. In *Tetris* the player collect points by making a whole line from the pieces falling down.

Figure 3.10: Tetris, a simple but popular puzzle game.

3.5 Game Application Programming Interface

There are a few **Application Programming Interface**, **API** for short, available for programmers wanting to make their own games. some of them contain all the tools needed for creating games and some only contains the graphics libraries. Here the three most common API used in game programming will be described.

3.5.1 DirectX

DirectX is Microsoft's API for making computer games. DirectX was created for developing games on the Windows 95 platform. The purpose was to make games run faster and game development easier in the windows environment. The benefit of DirectX over older libraries was that all the hardware specific code was put in the backend so the game developers do not need to write a specific function for every hardware manufacturers specification. Before DirectX players needed to setup their own hardware configuration themselves. This also made DirectX compatible with all older DirectX games. The problem with 640k[13] was still there

[13]It was a common problem for DOS games that the games required very much of the commercial memory and did nor run on newer operating systems because the operating systems used a lot of the commercial memory.

in 1994, with DirectX, or any other graphics API, that problem will newer occur again with newer games.

DirectX is also used on Microsoft's game console X-Box.

3.5.2 SDL

SDL is a acronym for **Simple DirectMedia Layer**, witch is an abstraction of several platforms graphics, sound and input API. SDL is in its basic idea very similar to DirectX but the major difference is that SDL is platform independent. So when you write a SDL game all you need to do i recompile the code in on another platform in order for it to run on that platform[16].

SDL i nothing more then a wrapper outside other API:s. In Windows the SDL functions call DirectX functions.

3.5.3 OpenGL

OpenGL is short for **Open Graphics Library** and is a cross platform API developed by Silicon Graphics for the purpose of creating 3D applications. But as can be figured out from the name, OpenGL if a open source library free to be modified and used by anyone. OpenGL was originally designed as the name implies a graphics library only. The library did not contain any functions for input or audio.

The library itself contain of about 250 function which cover the basic needs for 3D rendering. These functions can be used to build the most complex 3D shapes using simple primitives.

Unlike DirectX and SDL, which is used almost purely in game programming, OpenGL is used in the industry to visualize CAD projects and in scientific simulations to visualize the results.

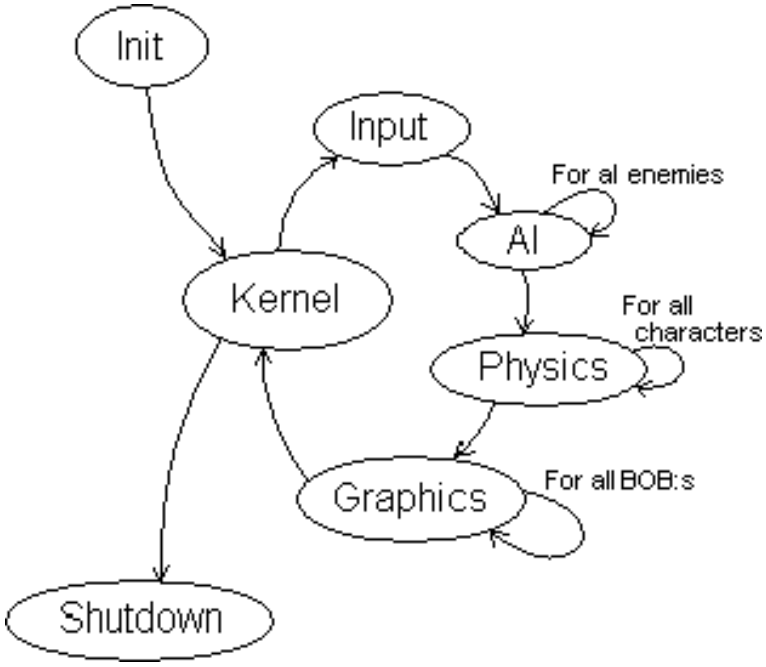

Figure 3.11: Diagram of a main event loop.

3.6 Programming games

Games can be programmed using a **main event loop**[1]. The technique is very simple, the game program consists of a big loop that is repeated once every frame. In the main event loop all input, physics, AI and everything else handled by the game engine i gone though at least once, see Figure 3.11[1]. The enemies, blocks and the player are each stored in objects in the level. In the main event loop the input the player is sending in stored, each enemy calls its AI function to give instructions about what input the enemies should send. In the physics part all the calculations for velocity and the collision detect is done and in the graphics part all the graphics is done.[1] When the game is started up a special sequence have to be run in order to load the graphics and level architecture and when the game is shut down another sequence is run that deallocate all the memory used.[1]

Since the frame rate need to be at least 40 fps in order for making the game run smoothly the code in the game engine have to be efficient and cannot be to advanced. This used to the bottleneck that kept game AI in realtime games at relatively low quality when the game industry was young. If the AI, graphics or physics were to advanced in old games the game would run to slow to be playable. Now, when processing speed and memory capacity have doubled many times since the first games AI, graphics and game physics can be much more advanced.

An alternative way is to have the graphics and physics in two different **threads**. One tread would handle the input, AI and physics and the other tread would handle the graphics. This means that the frame rate is dynamic because of that the graphic do not have to wait for a fixed time for the AI and physics to finish their processing. This software architecture is used in 3D games and can be observed when the graphics settings in a game is set to be more detailed then the computer hardware can handle. Then the game still runs in normal speed but the frame rate is so low that the game is not playable.

3.6.1 Modifications

Modifications, or **Mod** for short, is a home made upgrade to a existing game using the engine of a existing game. A mod is made using a special language that was created only for the purpose of making the game and was released to the general public alongside with the game or to be downloadable from a website later on. Making a mod to a existing game require skills in programming since the modification language often is quite advanced. A **level editor** must be included in order make the maps for the own, homemade, scenarios.

Modding, as it is called when someone modify a exiting game using the script language and tools provided with the game, have spawned a lot of big communities on the internet. The people doing the modification are called modders. The

Halflife mod *Counter Strike* was a mod that was realest as a game of its own. Developed in home by armatures just having fun and later be realest just like a commercial game.

3.7 Summary

Games is a great source of entertainment for some and a complete waste of time for others. But with all the many different kind of games genres it is almost certain that anyone with a interest in computer games will have a kind of game suitable for them. And if there is no games suitable for there taste, there is a lot of API:s designed for the purpose of game programming. So there is nothing that stops anyone with programming skill from making a game of there own.

Chapter 4

Introduction to game AI

How smart the computer controlled characters in a game is play a great part in determining how entertaining the game is. This chapter will describe techniques commonly used in game AI. It will give a description of what game AI is and then discuss the differences between game and conventional academic AI. That section will also include description of three common techniques of conventional AI that are also used in game AI. Following that is a section that describes the technique fussy logic. Finally there will be a description of the algorithm pathfinding with A*, a very common one in

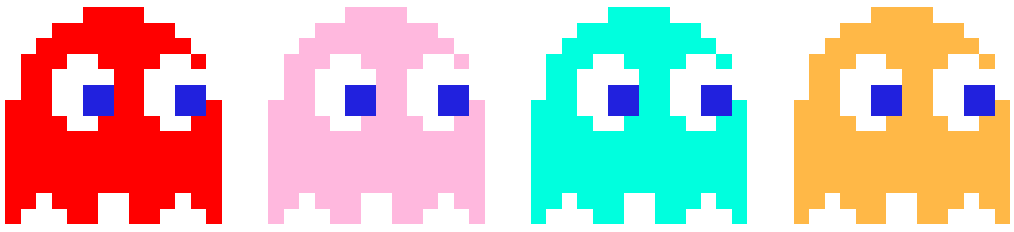

Figure 4.1: The ghosts from Pac-Man. Blinky, Pinky, Inky and Clyde.

game AI.

4.1 Introduction

The purpose of **game AI** is not to make the enemies in the game intelligent but rather to give the illusion of intelligence[4]. That is to make the enemies behave in a way that makes the player unable do decide if he or she is playing against another human player or against a AI agent.

Game AI falls under the category of weak AI[3]. Many of the techniques used in mainstream AI are also used in game AI. But actually the most simple techniques finite state machines, production systems and decision trees have proven to be most successful[4].

4.2 Mainstream AI and game AI

Mainstream AI covers the topics of AI that is researched in the academic world[4]. Since the field of game AI in only a part of all other AI basic knowledge of mainstream AI is needed for programming game AI.

Not all the techniques used in mainstream AI are used in game AI and some are used frequently. A short description of a couple common techniques used in game AI will be given.

4.2.1 Expert systems and production systems

According to the book *"AI game programming wisdom 2"*[5] **production systems** and **expert systems** are just two different terms for the same technique. In this dissertation both the terms will be used interchangeably.

Expert systems are AI agents that attempt to solve problems just like a human

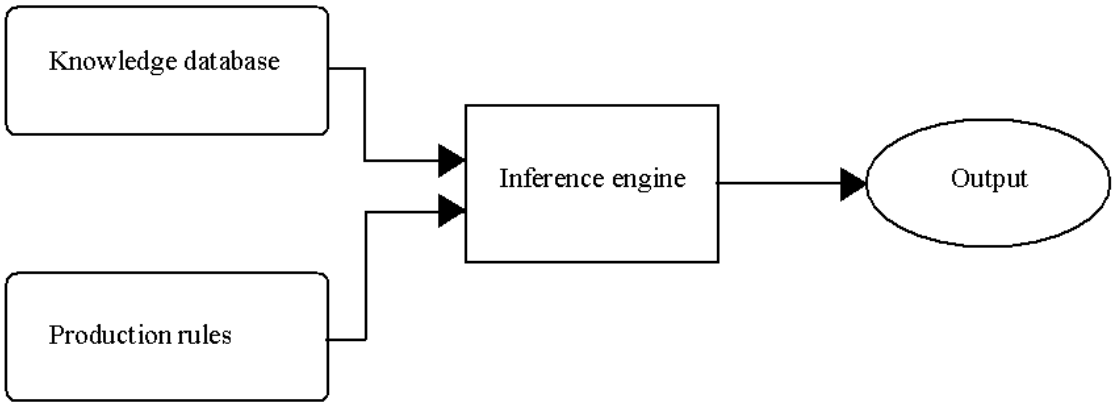

Figure 4.2: A basic production system.

expert in a given situation. Expert systems are based on knowledge and experience from human experts. The output from an expert systems shall be the same as from human experts for the same question[4].

Expert systems contain two parts which is the knowledge database and the logic rules or the reasoning[25].

The **knowledge database** and the **production rules** are implemented separately and are modular so they can be designed independent from each other. Later when they are finished they can be modified separately without disturbing the other[3].

The system's production rules are basically a large set of if-then-statements and the knowledge database is the parameters in the statements and the actions performed is the knowledge. The rules in system are often in the form of a chain. Meaning that when one decision is made one more is immediately made and possibly one more in a chain of reactions of rules. These depictions often have a certainty factor that increases by a certain amount for every step in the chain.

Table 4.1 show the knowledge database and Table 4.2 show the production rules for a given system. The knowledge database and the production rules are used

Group
Human
Animal
Plant

Skill
Fly
Jump
Climb

Attribute
Fur

Table 4.1: The knowledge database for a system expert system

Production rules	
Is	Group
Can	Skill
Have	Attribute

Table 4.2: The production rules for a system expert system

together by a **inference engine**. The production rules can take any knowledge from the knowledge database and apply it to the object of choice. The production rule 'Is' is applied to the knowledge 'Group', it checks what group a subject belong to. The production rule 'Can' is applied to the knowledge 'skill', it checks what a subject can do. The production rule 'Have' is applied to the knowledge 'Attribute' to check what properties a subject have.

The code for the inference engine may look something like this:

```
bool inference(rule,subject,info)
{
    switch(rule.type)
    {
```

```
    case Is: return(subject.type&info.type);
    case Can: return(subject.skill&info.skill);
    case Have: return(subject.attrib&info.attrib);
    default: assert(0);
    }
}
```

Where *rule* is what type of production rule that shall be used, *subject* is the object that we are gathering information about and $info$ is the knowledge we have an want to test. This inference function could be used by a recursive function thus creating a chain of tests.

In an practical example, lets say that we have one object named Fred[26]. For simplicity, lets assume that the system also has a knowledge database containing the animals Horse, Bat, Rabbit, Lizard, Monkey, Cat, Frog, Turtle, Fish, Snail, Duck and Snake. If we apply the set of rules and knowledge on Fred we can figure out what Fred is:

- Question 1: Is Fred a animal? Yes.

  Then Fred could be a horse, a bat, a rabbit, a lizard, a monkey, a cat, a frog a, a turtle, a fish, a snail, a duck or a snake.

- Question 2: Can Fred fly? No.

  Then Fred be be a horse, a rabbit, a lizard, a monkey, a cat, a frog a, a turtle, a fish, a snail or a snake.

- Question 3: Can Fred jump? Yes.

  Then Fred be be a horse, a rabbit, a monkey, a cat or a frog.

- Question 4: Can Fred climb? Yes.

  Then Fred be be a monkey, a cat or a frog.

- Question 5: Do Fred have fur? No.

  Then Fred can only be a frog.

This kind of guessing game is a example of rules using knowledge in a chain. The knowledge database used here was absolute minimum for the example, but more knowledge can be added in the form of animals, attributes and skills without affecting the rules. And more rules can be added later to get a more detailed answer. If we assume that question 5 was the last question in the chain but the answer to the question had been "Yes" it should have resulted in two possible answers, monkey and cat. Both are animals that cannot fly but can jump, climb and have fur. If we add the attribute "claws" then the system can figure out if it is a monkey or cat but in the current state it cannot. This same principle used in this simple biology example can be applied in any situation that require human expert knowledge, like combat strategy in a game or disease identification and treatment in a hospital medical system.

A large database of knowledge of the specific topic is required for a expert system to work properly. There would be no point of an expert system to exist if they did not contain more knowledge then what a human professional can keep in the head. But in game AI it is sometimes not required for a good result, a smaller database will give better performance in speed.

4.2.2 Artificial life

Artificial life, **A-life** for short, is a technique that simulates the behavior of life for animals in an artificial world or robot animals in the real world. It is commonly used on background critters in games and in simulations and robotics in mainstream AI. For example a little rabbit can run away and hide when the player comes close or a shoal of fishes keep their formation when they escape from

a player when he or she jumps into the water.

Since A-life is not really useful for enemies in a 2D platform game other then background critters and since A-life is more of an implementation specific result then a technique this will not be discussed more in the dissertation.

Flocking

Flocking is a special case of A-life where many single agents move in a simple manner but the result is a complex pattern for the entire group of animals. The technique is simply based on reacting to the movement of other members in flock by simple rules[5]. But in games there has to be a little random delay before the reaction because if the reaction was exactly afterwards it would be only one frame in between the reaction of each flock member and that would look mechanical and artificial. However, the purpose is to make it look natural.

Flocking can be applied to all kinds of animals that move in groups. Even human riots can be simulated using flocking.

In games flocking can be used combined with line of sight, see Chapter 5. It could be enough that only one enemy AI agent needs to see the player for all enemy AI agents to react on it. It could be done by enemy AI agents giving each other orders like move left, fire weapons at the shrubbery and flee if the player is considered to dangerous for the group of enemies, for how this can be decided see Section 4.3.

4.2.3 Finite state machine

A **finite state machine** is an abstract machine that can exist in one of several predefined states. A Finite state machine can also define the conditions that determine when the state should change. The states can also be triggered to change after time intervals. The actual state determines how state machine behaves[3]. A

finite state machine consist of:

1. A finite set of *states*, denoted Q.

2. A finite set of *input symbols*, denoted Σ.

3. A *translation function* that takes as arguments a state and an input symbol and returns a state. The translation function is denoted δ. If q is a state, and a is an input symbol, then $\delta(q, a)$ is a state p such that there is an acre labeled a from q to p.

4. A *start state*, one of the states in Q.

5. A set of *final* or *accepting* states F. The set F is a subset of Q.

A finite state machine is thus formally defined as a 5-tuple:

$$A = (Q, \Sigma, \delta, q_0, F)$$

where A is the name of the finite state machine, Q is its states, Σ is its input symbols ,δ it its transition functions, q_0 its start state, F its set of accepting states[9].

Finite state machines are one of the oldest forms of game AI. The ghosts from *Pac-Man*, see Figure 4.1, were finite state machines. There behavior changed depending on what Pac-Man did. For example did the ghosts change from chasing to evading when Pac-Man ate a super pill [3].

Even though finite state machines are old technology they are frequently used games because they are simple to use and give good result for enemy behavior.

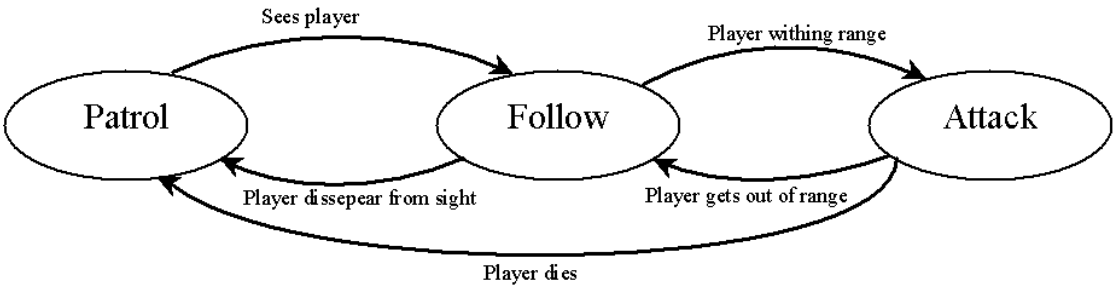

Figure 4.3: A simple finite state machine for a enemy AI agent.

A finite state machine for a castle guard

Figure 4.3 is a diagram of a simple finite state machine for an enemy. In the initial state, patrol, the enemy just patrols the closest area where is stands. If the player gets within the enemies line of sight, explained in Chapter 5, the state is changed to follow where the enemy is chasing the player. If the player gets out of the enemies line of sight the enemy starts patrolling again. But if the enemy gets within attack range of the player the state is again changed but now it is changed into the attack state. Where the enemy attacks the player until the player is either dead or gets out of attack range then the state is changed back to patrol if the player dies and to follow if the player tried to escape and succeeds.

This simple example can provide a good end result for enemies in games. Even though this state machine would not work very well as a AI agent in a finished game it would still be a enemy that reacts to what the player does in the game which was the desired result.

One fine attribute with finite state machines is that they are extremely easy to modify. All that is needed to do with the existing machine is to add a state and one single condition in one of the existing states that change state to the new one. Then the new one can have conditions that take changes the state to any of the

exiting states. There could be any number of new state change condition in the new stage.

This simple change would not do much in the given example of the guard but in a large finite state machine a new stage could result in a dramatic change in the behavior of the machine.

Finite state machines and production systems

Finite state machines is not a technique used only by artificial intelligence. In computer science is used in several other applications, like bottom up parsers and realtime systems. Since finite state machines are such a useful technique in computing in general there are a lot of weak AI applications that can get a good result with a finite state machine.

The example given above about the guard contains simple conditions that change with single if-then statements. But if each state in the finite state machine were a production system, then really complex state changes could be made. Resulting in even better artificial intelligence.

4.3 Fuzzy logic

Fussy logic vas introduced in 1965 by Lotfi Zadeh in the article *"Fuzzy Sets"*[12]. What he wanted to do was to make computers solve problems in a similar way as humans do[3].

Fuzzy logic is an extension of boolean algebra using one more value to the two original true and false. The new value is maybe, truth to a degree. If a car has one door painted red but the rest of the car is painted blue it is true to say that the car is blue but it would also be true to say that the car is red. It is all a matter of degree. A common word to describe fuzzy logic is to use the term **fuzzy set**

theory. The blue car with one red door is member of both the set of blue cars and the set of red cars. In fuzzy logic everything is a mater of degree. With the car the car is more blue then red so if a choice had to be made about the color of the car, the choice would be blue because the car is more blue then red even though is is a member of both sets.

In computer games the **fuzziness** can be the health of a enemy AI agent, see Section 3.3. If the enemy have 100 hitpoints the health if the enemy is fussily defined because there are 100 possible health levels. The fuzzy thing is to decide at what health level the enemy AI agent should consider it as having low health. Depending on how many hitpoints the enemy AI agent have left the more aggressively the enemy AI agent will act and the lower health the enemy AI agent has the more defensively the enemy AI agent will act.

Lets define three sets of health that the enemy can be part of. The sets are *unhurt*, *injured* and *wounded*. Which set the enemy is a member of is decided by the the current number of hitpoints that the enemy have.

4.3.1 Fussy sets

Fuzzy logic is based on **fuzzy set** membership. The example with the car was simple and the answer was just as simple but the colors of the car does not vary over time, but the hitpoint of a enemy can change. The fuzzy member value is a value between 0 and 1. Figure 4.5 shows a graph of how the value varies depending on how much health the AI has. As the figure shows it is possible for the enemy AI agent to be member of several sets at once but in this example it is just as possible to be member of two at once. The different sets are defined by trapezoids.

The prototype for the function for getting the set memberships is defined as:

$$float \quad fuzzyTrapezoid(value, x_1, x_2, x_3, x_4);$$

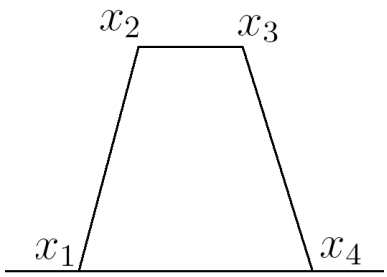

Figure 4.4: A fuzzy trapezoid set.

Where *value* is the value is the number of hitpoints in this case and x_1, x_2, x_3 and x_4 are the values that define the points of the trapezoid. x_1 is the leftmost point on the trapezoid that indicate that the membership of the set begins if the value is higher than the value of x_1, x_2 is the point on the trapezoid that indicate that the membership is now 1 if the value is greater than it, x_3 is the end of the of the full membership and x_4 indicate that the end. If *value* is somewhere between x_1 and x_2 than the membership is somewhere between 0 and 1, if *value* is between x_2 and x_3 than the membership is 1 and if *value* is x_3 and x_4 than the membership is somewhere between 0 and 1. If *value* is smaller than x_1 or larger than x_4 than the fuzzy membership is 0. Figure 4.4 show where the different points is on a trapezoid.

If the enemy AI agent is mostly a member of the wounded set in Figure 4.5 the AI will attempt to run away, if the AI is mostly a member of the injured set the AI will stand guard and if the AI is mostly a member of the unhurt set the AI will attempt to attack.

If the enemy has all hitpoints left it is in the unhurt set and if only a few hitpoints left the enemy is in the wounded set. In these examples the membership in the sets [wounded, injured, unhurt] is [0, 0, 1] and [1,0,0].

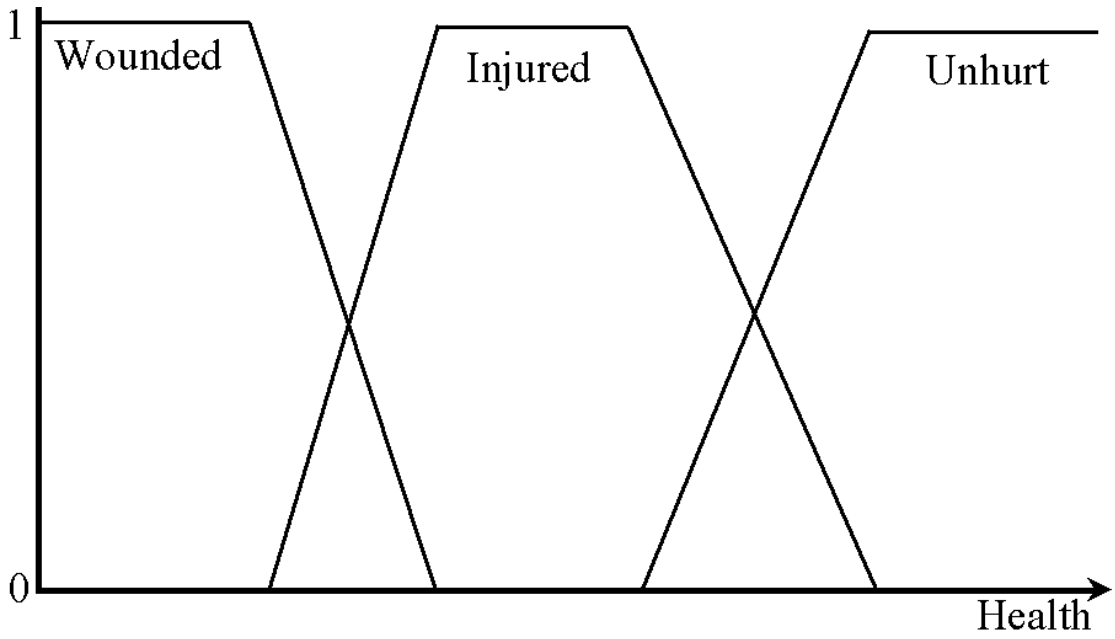

Figure 4.5: Graph displaying the membership in fuzzy sets of a enemy AI agent depending on how much health it has.

4.3.2 Defuzzification

In this example it is simple to decide what to, it is just a matter of picking the set with the greatest member value and do the corresponding action. But if several fuzzy sets are variables in the fuzzy logic mathematical formulas is needed. If the memberships are [0, 0.3, 0.7] the AI has lost a little of its hitpoints. This information was gained with the function $fuzzyTrapezoid(value, x_1, x_2, x_3, x_4)$, Factors are attached to the sets to determine how strong they are. The factor will be called x. Wounded gets $x = -10$ injured gets $x = 1$ and unhurt gets $x = 10$. The set membership value is represented by μ.

The factors have value because the different set memberships are different important to the decision making. For example the health is more important than the eventual armor of the character so set memberships strong armor and weak armor would have the factors 5 and -5 which is less powerful then the factors the

different health sets have.

The formula for the defuzzification is:

$$output = \frac{\sum_{i=1}^{n} \mu_i \cdot x_i}{\sum_{i=1}^{n} \mu_i}$$

If the output is negative then the enemy will run away. In this case the output will be:

$$output = \frac{0.3 \cdot 1 + 0.7 \cdot 10}{0.3 + .0.7} = 7.3$$

The result of 7.3 will result in the enemy AI agent attacking.

This was a simple example but if more fuzzy sets are added to the formula, like armor, ammunition, weapon type and the status of the player, there will be many variables that need an automated process. If more fuzzy sets are added to the decision making all that have to be done is add more terms to the equation.

4.4 Pathfinding with A*

A*, pronounced a-star, is a common algorithm for finding the shortest paths in game programming. The A* algorithm is a greedy algorithm for exploring. If there exists a path between two nodes in a map the A* algorithm will find it. Pathfinding with A* is much faster then ordinary shortest path algorith1m which is used in game programming[4].

4.4.1 Terms

Before Pathfinding with A* can be explained in detail, a few special terms is need to be explained.

Map

The **map** is equitant to level in Section 3.3. This is simply the area at which the AI agent finds its path between two given points[4].

Node

These are the **waypoints** on the map that the AI agents use as reference points when they move in the level. They are located on the map or level depending on what word you like best. **Nodes** on the map are just like any nodes in any graph that are connected with undirected connections. But each node is connected to only the closest nodes on the map. Unlike ordinary nodes in graphs the nodes on the map have a fixed position on the map and are not allowed to be morphed around like an ordinary graph[4].

These nodes contain information critical for the A* algorithm[4].

Heuristic distance

Heuristic distance, or just distance, is how good it is to explore a particular node[4]. It is not possible to know exactly how suitable a particular node is to explore in the algorithm. Based on machine learning or preprogrammed knowledge a qualified guess have to be done. This guess is heuristic, see Section 2.2.

Cost

The **cost** of a node is the distance between the particular node and the staring node[4]. This is totaly application specific and cannot be formally defined.

4.4.2 The algorithm

The A* algorithm needs a starting node and a destination node in order to work and it finds the shortest path between them. The nodes used in the algorithm do not only hold the position on the map but also three attributes called f,g and h refereed to as fitness, goal and heuristic. The following definitions is directly stolen from ‘‘`AI Game Programming Wisdom`’’[4].

- ***g*** is the cost to get from the starting node to this node. Many different paths go from the start node to to this map location, but this cost represent a single path to it.

- ***h*** is the estimated cost to get from this node to the goal. In this setting h stands for heuristic and means educated guess, since we do not really know the cost (that's why we're looking for a path).

- ***f*** is the sum of g and h. f represents our best guess for the cost of its path going through this node. The lower the value of f, the better we *think* the path is.

The only problem with these attributes is h, which cannot be known for certain at this time. g is known absolutely and f is uncertain due to that h is uncertain.

The A* algorithm keeps two lists, these are called the open list and the closed list. The open list contains all the nodes that have not been explored yet and the closed list contains all nodes that have explored. A node is explored if the g,h and f values have been calculated for all nodes connected to it and added to the open list for future exploration. The open and closed lists are needed because if it were not for them it would be possible to move back the way to a node which is not good for performance purposes.

4.4.3 Pseudo code for the algorithm

This is a listing of the pseudo code for the A* algorithm[4].

```
1. Let P = the staring point.
2. Assign f,g,an h values to P.
3. Add P to the open list. At this point P is the only
   node in the open list.
4. Let B = the best node from the the open list (best
   node have the lowest f value).
   a. If B is the goal node, then quit - a path have
      been found.
   b. If the open list is empty, then quit - a path
      cannot be found.
5. Let C = a valid node connected to B.
   a. Assign f,g, and h values to C.
   b. Check whether C is on the open or closed list.
      i.  If so, check whether the new path is more
          efficient (lower f-value).
           1. If so, update the path.
      ii. Else, add C to the open list.
   c. Repeat step 5 for all valid children of B.
6. Move B from the open list to the closed list and
   repeat from step 4.
```

No example will be given since this dissertation will give an alternative solution to pathfinding. See the literature for more details.

4.5 A complete enemy AI agent

A complete AI agent in a game would be implemented using the three techniques described above. The base of the agent could be a finite state machine. Each state in the finite state machine would be a independent production system that is used to make the decisions in the current state or decide what state transition should be done if a change of state is needed. The result gained from the fuzzy logic set membership equation could be used as parameters in the production system or simply decide when a stage transition should be done.

4.6 Summary

Good game AI will always be praised by players when it give a balanced and natural behavior of the enemies, cursed when the enemies have unnaturally good skills and laughed at when it is stupidly predictable. In any case it is a challenging and important part of all modern games.

Game AI is a subset to mainstream AI and a few of the techniques used in mainstream AI is used equally much in game AI. Some techniques have found special usage in game AI.

Part II

Experiment

Chapter 5

Line of sight

Determining what the enemies in a game can see is important for making them act naturally. This chapter will describe the theory and techniques used for determining what is within the line of sight of an enemy AI agent in a 2D computer game.

First there will be an introduction that will explain the problem and how it was like in older 2D games. After the introduction a section about theory that will describe the three techniques used in making a 2D line of sight algorithm for AI agents within the game and give a brief explanation about the differences between 2D and 3D line of sight. Finally there will be a section tying the things together making a complete line of sight technique for AI agents in 2D computer games.

See Appendix A for the implementation of this technique.

5.1 Introduction

The area within the enemies **line of sight** is the part of the level the enemies can see. It is used in game AI to determine if the enemy AI agent knows where the player is and if the enemy AI agent can see the player or other objects in the game.

In really old 2D platform games the enemies always saw the player, even if the player and the enemy AI agent had a wall between them or if they were too far apart that there were several screen widths between them. Some games had line of sight triggered to rooms or large areas, which meant that the enemies saw the player when the player entered the room that the enemy AI agent was in. Some games had the enemies totaly inanimated until they become visible on the players screen and then they know where the player was until the player left the level.

This is of course unacceptable in new games and since this dissertation is about 2D games the line of sight for enemies will be the same one as the player plus the limitation of the blocks if there is any between the enemy and the player or any other object that can be seen by the enemy AI agent.

5.2 Theory

The techniques for line of sight in 2D games is actually quite simple but it has had low priority in old computer games therefore there were no 2D games with complex line of sight techniques. In newer games, which have been mostly 3D games, designers and programmers have put effort into more complex line of sight algorithms. But in 3D games there are so many points to handle that an exact line of sight have been almost impossible to implement so they have had to make approximations and the line of sight have not been exact. But in 2D games with today's computers we can afford to have an exact line of sight algorithm without loosing performance. Even the later 2D games that had more complex line of sight still used approximations.

Blackthorne, see Figure 5.1, is a 2D game that used advanced line of sight for its time. But the enemy AI agents could not see diagonally on the screen, they could only see the player then they were parallel in the x-axis. However, it was

Figure 5.1: Blackthorne, released in 1994, a 2D platform game where the enemies only reacted on what they saw.

still an advanced AI for its time in a 2D platform game.

5.2.1 Visual limit

The theoretical limit of what the enemy can see in a 2D game is what is inside the screen of the enemy if the enemy was a player. When the player moves around the level the only things that the player can see is what is on the screen. Now imagine that the enemy was a player and the screen showed the objects that was on the enemy's screen. This means a limit to how much the enemy can see in both dimensions. The player, or any other object for that matter, cannot be seen by the enemy if it is further away from the enemy then half a screen width or height. The distance is half a screen width or height because the player is almost always in the center of the screen and the border of what is on the screen is half the distance

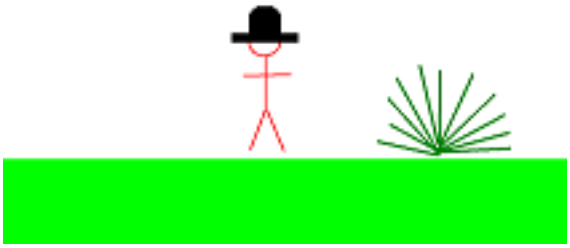

Figure 5.2: This picture shows a screenshot of a level with the a character in the center.

from the center of the screen. This is the **visual limit**.

Since the screen is limited in size the entire level cannot be displayed at the same time. It is a limit to what the player can see when the game is played. Figure 5.2 shows the limit of what a character can see. The image has the character in the center of the screen. This means that this picture shows what is shown on the screen when the game is running.

Figure 5.3 shows a screenshot of the same level but everything has been panned a little to the right so the character with the hat is no longer on the screen. But the bush on the ground still is but it have moved a bit to the left on the screen and another character is visible. The limit for how far the character with the hat can see is marked with a dotted line in the picture. The bush is visible to both the characters but they cannot see each other because they are too far apart.

The reason for this limit is so that the enemy AI agents would not get an unfair advantage on the player or players if there were no limit for how far the AI agents could see. The player can only sees what is on the screen and since the AI agents have access to the position of the player all the time, since they are part of the computer, this limit is needed otherwise the enemy AI agents and the player would

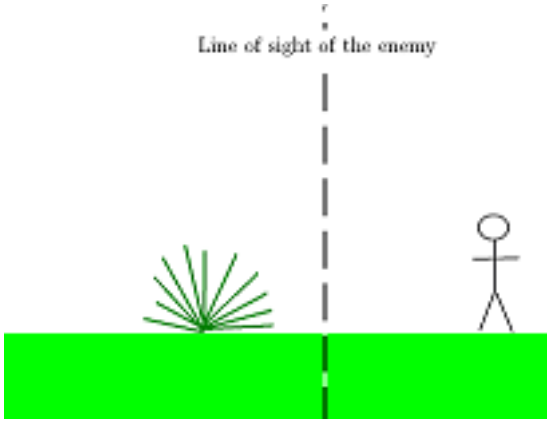

Figure 5.3: This picture shows the same level as Figure 5.2 but is shows what would be on the screen if everything were moved a bit to the left.

not play on equal terms.

See Appendix A.3 for the implementation of this technique.

5.2.2 Free sight

The sight between two objects is free if there is no other object in between the objects, then it is **free sight** between the objects. Just like that you cannot see what is on the other side of the wall in front of you[1], an AI agent in the game cannot see the player if there is a wall, floor or any other object in between them.

To be absolutely certain that there are no opaque objects each single pixel between the enemy AI agent and the player have to be checked. This is done from a single point, preferably the eyes for realism, of the enemy and the player forming a cone, see Section 5.3.

Figure 5.4 shows two characters, it can be the player and one enemy AI agent or two enemy AI agents, it does not matter, on the same screen with a terrain block between them. The fact that there is a block between them means that the

[1] Assuming that you are inside and the walls are not transparent.

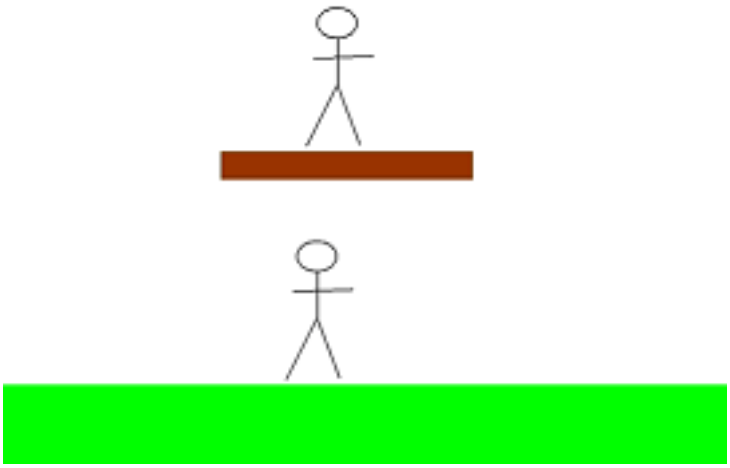

Figure 5.4: Free sight example 1. Two characters on the same screen.

two characters cannot see each other. Figure 5.5 shows the area of the screen that the lower character can see, the visible area is highlighted and the non visible area is plain white. As it can be seen on the picture it is possible do draw a straight line from all points in the highlighted area and a point in the lower characters head. However, in the non highlighted area the platform is in the way blocking the line of sight.

If it is possible to draw to just one single line from the enemy AI agent to the player then it means that the enemy can see the player. It is this kind of details that has never been done in 2D games.

See appendix A.5 for the implementation of the technique.

5.2.3 Bresenham's algorithm

To have the description of an ordinary algorithm in a section about the theory about AI might seem odd to begin with. But this algorithm is very necessary for implementing line of sight in games, which will be explained in Section 5.3, that it deserves to be part of the theory of this chapter.

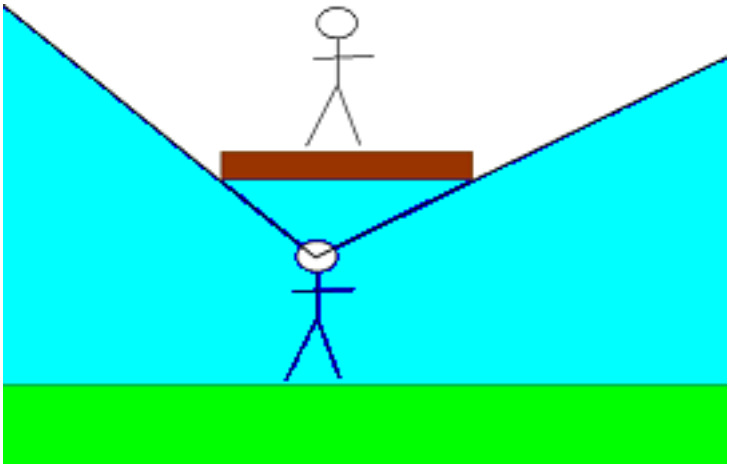

Figure 5.5: Free sight example 2. Highlighting the area of the screen that is within the line of sight of the character below the block.

The algorithm is called **Bresenham's line scan algorithm** or just **Bresenham's algorithm** and was originally published in 1965 in one of IBM internal documents[11].

Drawing exact lines on a computer is impossible, since lines are defined as an infinite number of zero-area points which lie between two end points. The smallest unit on a computer screen is the pixel, and its area is quite a lot more than zero. Approximations on the other hand, are quite easy to draw if you use floating point operations:

```
void draw_line(int x1,int y1,int x2,int y2)
{
        int dx = x2-x1;
        int dy = y2-y1;
        float m = dy/dx;
        for(int x = x1 ; x<x2 ; x++)
```

```
        {
                int y = m * x + y 1 + 0.5;
                putpixel(x,y);
        }
}
```

This, however is too slow to be acceptable in a game where speed is of the essence. The solution is Bresenham's algorithm that will compute the point coordinates correctly, using only integer math.

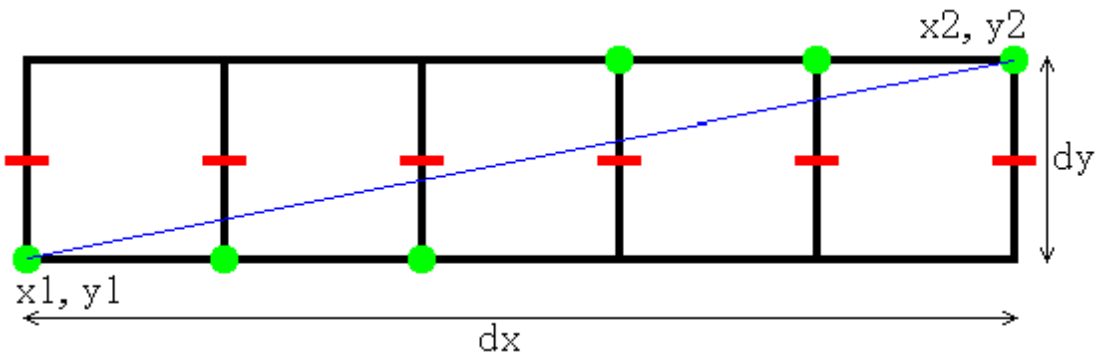

Figure 5.6: Line Approximations[1].

Figure 5.6 shows how pixel positions (the dots in the corners of the grid) are chosen depending on the true line's (the line between the bottom left corner and the top right corner of the grid) position relative to a midpoint (short horizontal lines). If the line is below the current midline, we plot the next pixel to the right. If, however, the blue line is above the midline, we should plot above and to the right:

```
if(BlueLine < Midpoint)
        Plot_Right_Pixel();
else
        Plot_AboveRight_Pixel();
```

This is in essence the basic idea of the Bresenham's algorithm. See [11] or [1] for more details. A example for how a Bresenham line function is implemented is given in appendix A.1.

5.2.4 Efficiency

The efficiency of the line of sight technique depends on two variables, the length of the Bresenham line and the number of object forming the terrain. But also the number of enemies in the level, although this is not caused by the technique itself but rather a variable depending on the architecture of the level. In total it is $O(NML)$ where N it is the number of objects in the terrain, M is the length of the line and L is the number of enemies that have the player on its screen.

See Appendix A for details about the efficiency of the different part of the implemented technique.

5.3 Implementation

The theories of line of sight in 2D games, visual limit, free sight and Bresenham's algorithm, is used together to give the enemy AI agents a natural behavior in the game. To check if the player is seen by the enemy AI agent the first check is if the player is on the enemies "screen", explained in Section 5.2.1.

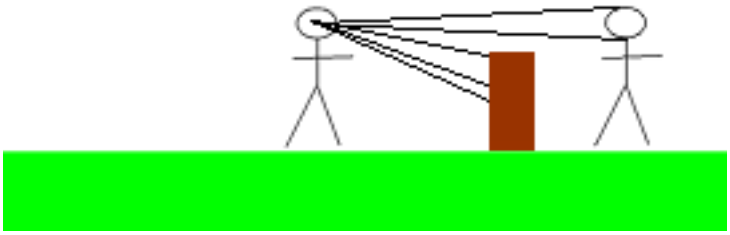

Figure 5.7: Illustration of that several lines can be drawn between two characters in a game.

If the player, or any other object for that matter, is on the enemy AI agents screen a free sight check, explained in Section 5.2.2, to the object is done.

The line drawn between the characters is done using Bresenham's algorithm, described in Section 5.2.3. But instead of drawing a pixel on every point between the enemy AI agent and the player a collision detection is done on every point between the AI agent and the player. If there is a collision with a block in the terrain on a single point between the player and the AI agent it means that there is not a free sight between the player and the AI agent on that particular line.

But several lines can, most of the time, be drawn between the enemy AI agent and the player as shown in Figure 5.7. It is possible for several more lines to be drawn between the characters to blocked by the terrain or other objects and the enemy AI agent might still see the player. Because it is enough that only one single of all possible lines between the enemy AI agent to have no obstacles in its path for the player, or any other object for that matter, to be within the enemy AI agents line of sight. As seen in the picture in Figure 5.7 three of the five drawn lines are blocked by the tree stub between them but two of the drawn lines have

Lines	56	560	20	4
Frames	2500	1500	5000	2500
Time	228	2079	88	31

Table 5.1: Table over the statistics for average time it took to complete a frame with visible debug lines.

no objects disrupting them resulting in that the left character can see the right character.

Appendix A gives a detailed description of how this technique was implemented.

5.3.1 One problem

One problem with this brute-force line of sight technique is which of all the possible lines between the characters should be chosen to check for free sight. If the player was large, say 253 pixels wide and high, then there would be a lot of possible lines to draw between the player and the enemy AI agent, $253 + 253 - 1 = 505$ different lines in the worst case scenario. 505 lines are too many to check every frame even with a computer for a 1 GHz processor, it runs too slow, so obviously all possible lines cannot be checked.

5.3.2 Statistics

Table 5.1 shows statistics of the average time it took for a certain number of frames to complete on a small level. The desired time was 30, the time is measured in milliseconds. The test computer had a 1 GHz Pentium III processor with 256 MB RAM and a ATI Radeon 7500 graphics card. The screen resolution was 1600×1200. The statistics for this table was gathered with the line of sight lines actually visible on the screen during gameplay. The purpose for this test was to

Lines	20	56
Frames	3000	2500
Time	30	30

Table 5.2: Table over the statistics for the same level as in Table 5.1 but no lines are drawn.

Lines	24	40	1
Frames	5000	5000	5000
Time	30	31	35
Enemies	1	1	40

Table 5.3: Table of the average time between frames with the number of object in the level greatly increased.

demonstrate the time difference with and without the lines visible. Without the lines the time between the frames is much shorter.

Table 5.2 shows statistics for the same level as the statistics in Table 5.1 but this time the lines were not drawn on the screen and as a result the time between the frames is much shorter. Even with 56 lines, there were no notable lagging in the game.

In the statistics of Table 5.3 the level was changed into a level with more than 100 terrain blocks. This means that the collisions checks in the collision detection described in Section 5.2.2 will increase. As far as up to this test there have only been one single enemy AI agent and one player in the level. But for the test for Table 5.3 the enemies were increased to 40 but only one line per enemy. With 100 blocks it is possible to build very big levels if the blocks themselves are large.

5.3.3 Solution

From the statistics it is possible to read that about 40 free sight checks is the maximum amount that can be checked per second. That means that one enemy AI agent can perform 40 checks or that 40 enemy AI agents can perform one check, or any combination in between. As shown in Figure 5.7 some lines, between the player and enemy AI agent, have free line of sight and some may not have free line of sight. As mentioned in Section 5.3.1 many more lines than 40 can often be drawn between the player and the enemy. If all the lines chosen to be checked are blocked by the terrain, but there exist lines that are free, the result will be that the enemy AI agent will not see the player but the player is actually within the line of sight of the enemy.

If we assume that only one line per enemy will be checked per enemy and frame, the problem is to choose what line to check. Since one frame only lasts for 30 milliseconds it means that if one line is checked every frame then 33 checks are preformed every second. The solution is to check one random line between the enemy AI agent.

The fact that line of sight is not one hundred percent accurate gives a more human behavior because a human will not notice everything exactly when it appears all the time. A test run with the chosen point being a random point within the player the game gave a very natural result.

5.4 Summary

Line of sight is an important part of any game and is very simple when it is the player we think about. The player's line of sight is what is shown on the screen. But when it is about the enemy AI agents line of sight the problem is to limit what they cannot see. The first limit is to limit the distance how far the AI can

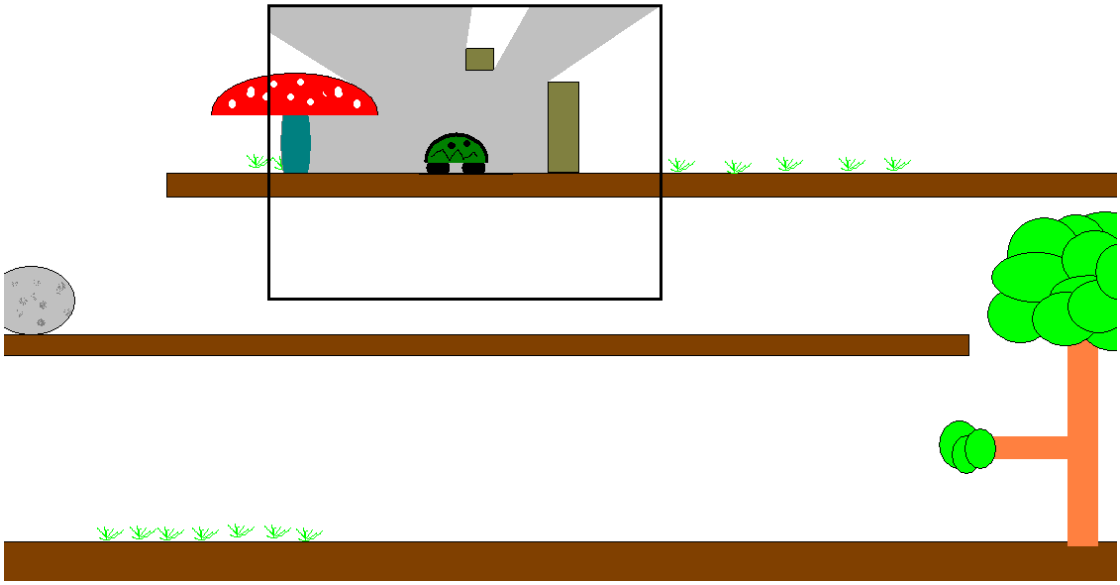

Figure 5.8: Summary of line of sight, the shaded area within the rectangle is the area of the level that the enemy AI agent can see.

see then it is a matter to limit what is blocked by the objects in the terrain and then to speed things up by an algorithm for checking if the line of sight is free.

Figure 5.8 pretty much summarizes this entire chapter. The entire picture is the level, the funny looking thing right of the mushroom is the enemy AI agent, the rectangle around the enemy is the enemies screen and the shaded area is what the enemy can see, or the enemy's line of sight.

Chapter 6

Image recognition of the level

Image recognition is the technique used for identifying what the AI agent can see. This chapter describes how AI agents within a game can use a form of image recognition to "see" the terrain in the level.

After the introduction a general description of how collision detection can be used to to know what is in the AI agents closest surroundings. The next section describes how an enemy AI agent knows that it shall jump when it reaches a gorge or a cliff. The section after that one describes how an AI agent predicts if the jump will be successful. After that follows a section about jumping over high objects.

What is described in this chapter is only a part of all possible combinations that can be done with this technique but they cover a lot of behavior in a 2D platform game.

See Appendig B for the implementation of this technique.

6.1 Introduction

Making the enemy AI agents see their closest surroundings is related to image recognition in mainstream AI. The purpose is to make the enemy AI agents act

like a human player would when things appear on the screen. Chapter 5 discussed when the enemy AI agent sees the player but the topic image recognition of the level is about seeing the terrain.

When the enemy AI agent shall move between two points on the level and there are terrain blocks in the way blocking the straight path a technique for knowing when to jump, duck and do all other kinds of evasion maneuvers that are needed. It is possible to know if the enemy AI agents can visually analyze the level of their own without cheating triggers that tell the enemy AI agents what todo, which was common in old games, but then the enemies movement became too predictable.

6.2 Collision detection for making AI agents "see"

Collision detection is an important feature in game AI because it is the only technique an enemy can use to know where the objects are in the virtual world. The technique itself is very simple, it is just a matter of checking if an object is in a certain area defined by the enemy.

Some argue that collision detection does not fall under the category of game AI[4][5] but if that was the case then image recognition would not be a part of AI either because collision detection is the technique used by enemies to visually recognize the virtual world. The player use human eyes to see what happens on the screen in order to visually analyze the virtual world.

6.3 Jump over gorges

The first and most simple form of collision detection in 2D platform games is to detect when the enemy AI agent has reached a position where it is necessary to either jump or change direction in order not to fall down, see Figure 6.1.

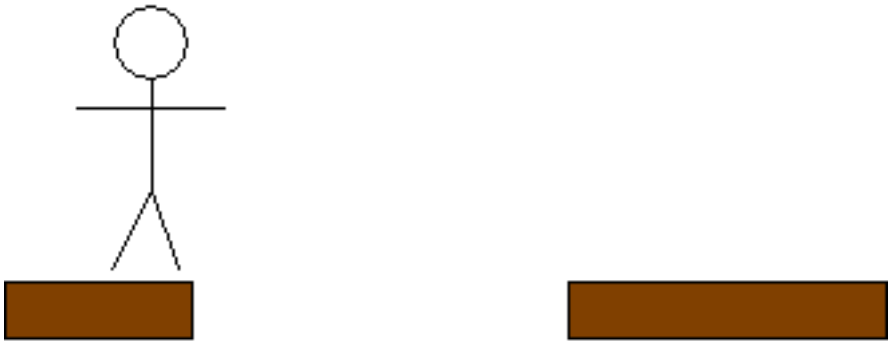

Figure 6.1: The enemy AI agent standing in front of a gorge.

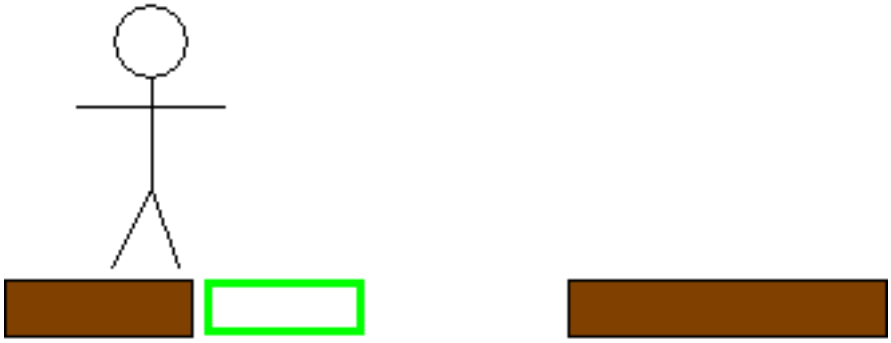

Figure 6.2: The enemy AI agent checking the ground in front of it.

6.3.1 To know when to jump

This can be done with collision detection in front of the character all of the time as Figure 6.2 shows. The area that is checked is represented by a rectangle in the picture. If the AI agent reaches a gorge that is too wide for it to step over it must either jump or turn around. The technique is simply about checking if there are any terrain blocks or platforms in the area directly in front of the AI agent. If there is no collision at that particular check then it means that there is nothing to walk on before the feet of the AI agent. Figure 6.2 illustrates what area that an AI agent moving right in the screen would check. Note that this rectangle is not

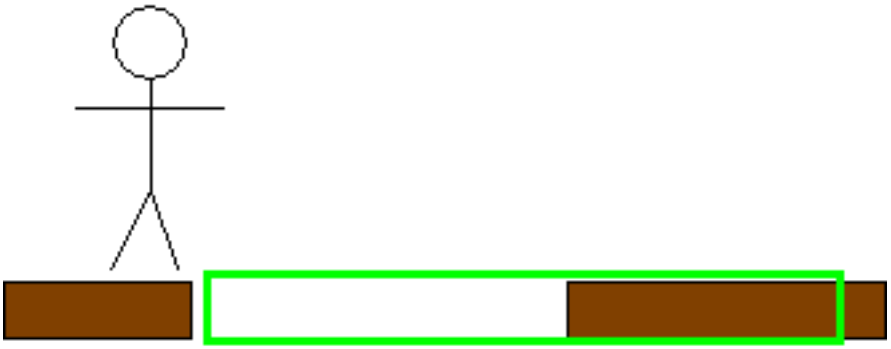

Figure 6.3: The enemy AI agent checking if there is something to stand on within the distance of how far the enemy AI agent can jump.

the exact area that will be checked, the rectangle is only drawn there to illustrate an approximate area that will be checked. In this case there is no collision because there is no terrain block of any kind within the small rectangle in front the AI agent. When there is no collision in the rectangle in front and below of the AI agent it will fall down if it continues without doing anything so it has to jump.

See Appendix B.1 and B.2 for the implementation of this technique.

6.3.2 To know if it is possible to jump

After the AI agent knows that it has to jump in order to continue in its path. This is done by a second collision detection with the size equal of the distance the AI agent can jump. Figure 6.3 shows this by the rectangle covering the gorge and some of the platform to the eight side of the picture. If it is a platform or other kind of terrain block that it is possible to stand on within the area checked, it is possible for the AI agent to jump over the gorge and land on the other side.

See Appendix B.3 for the implementation of this technique.

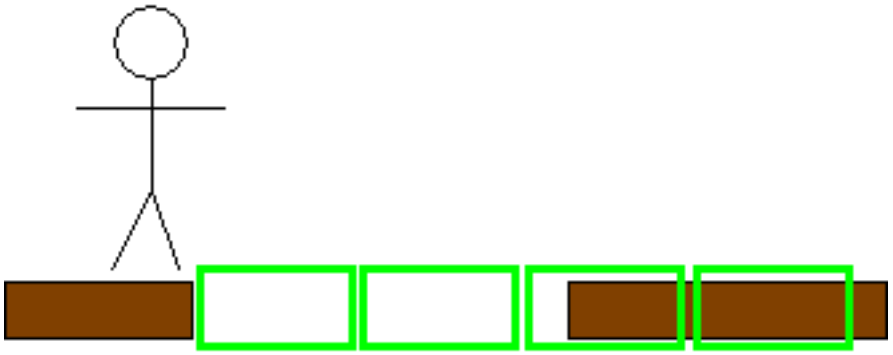

Figure 6.4: The enemy AI agent checking how far it have to jump in order land on a platform.

6.3.3 To know how far to jump

The next problem is to know how far to jump. This is done with several checks with width equal to the maximum width that the AI agents just walk over. This way the minimal number of checks will be done, but the position of the platform it is possible to stand on will still be found. Figure 6.4 illustrates how an enemy AI agent can check the gorge in order to know how far it has to jump to land on the other side. Compared with Figure 6.3, the rightmost rectangles right side in Figure 6.4 is at the same position as the rectangles right side in Figure 6.3.

See Appendix B.4 for the implementation of this technique.

6.3.4 Summary

Figure 6.5 shows a summary of how an AI agent in a 2D platform game can check the ground in front of it. The small horizontal rectangle is checked every frame. If there is no collision in that rectangle it means that there is a hole in the ground that is to large to step over and the AI agent have to jump. If the AI agent have to jump a second check is done, the large rectangle with curved corners, if there

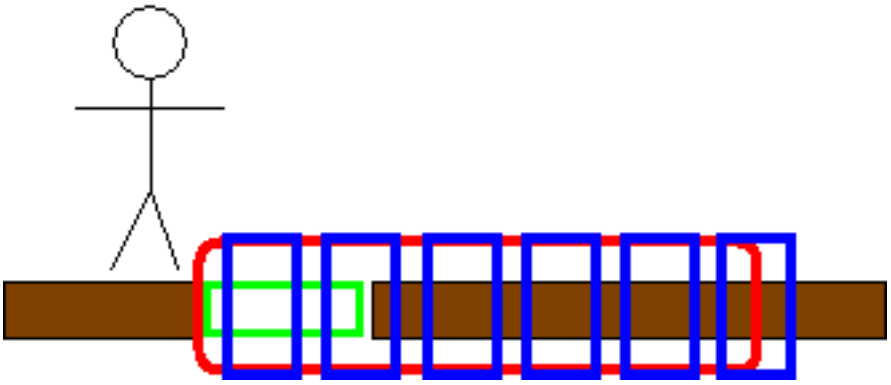

Figure 6.5: Collision detect used by a AI agent in a game.

is a collision with the ground or an other platform it means that it is possible for the AI agent to jump to the platform. Then additional checks is done in order to check how long the distance is between the platforms, illustrated by the small vertical rectangles in Figure6.5.

6.4 Free jump trajectory

The scenario in Section 6.3 assumes that there is nothing above the gorge hindering the AI agent in the jump trajectory. But what if there is a terrain block that will stop the character in mid air and then fall down the gorge? In the technique described above the enemy AI agent will jump, hit the block and fall down the gorge. In most cases this is not desirable and a technique used to check if the path the AI agent will take in the jump is unblocked by anything. In order to know if the trajectory i free the AI agent have to know many things.

6.4.1 To know if there is jump area is free

If there is no block in the way of the trajectory the AI agent can jump without any problem. But if the situation is as Figure 6.6 illustrates the trajectory is not free

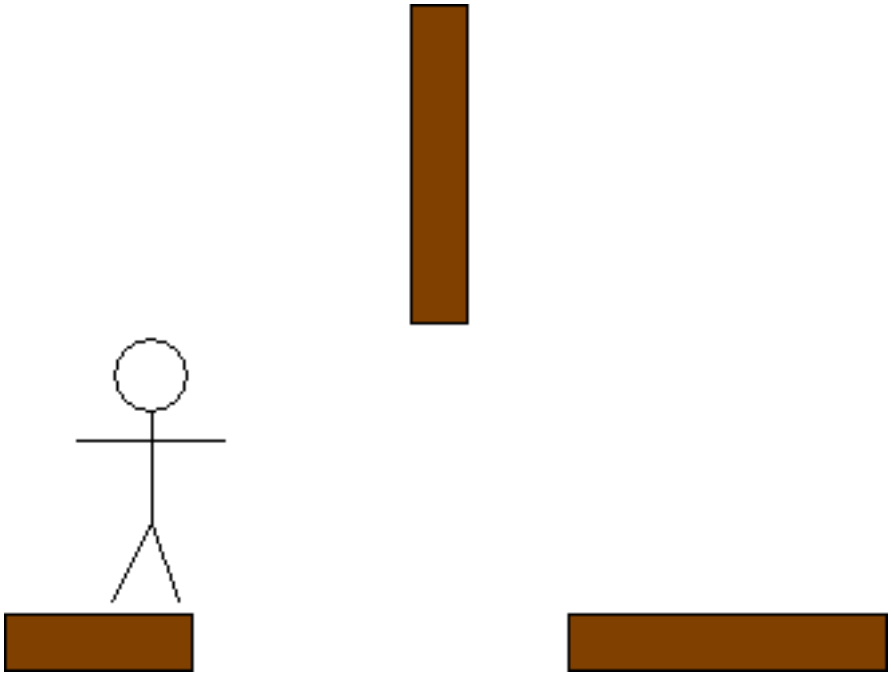

Figure 6.6: A scenario where a bock i blocking the jump trajectory of the AI agent.

and it is not possible to jump over the gorge even though the technique described in Section 6.3 will think that it is possible to jump over the gorge.

First of all a large area is checked to see if there is any terrain block within the area of the trajectory, see Figure 6.7. If there is no collision then it means that the trajectory is free and no further checks are needed. But if there is a block within that area it means that the the trajectory maybe is blocked and the jump cannot be done but it may still be possible that the AI agent does not know this yet so more checks are needed before it is possible to know.

See Appendix B.5 for the implementation of this technique.

6.4.2 To know if the trajectory path is free

The large rectangle in Figure 6.7 covers more then the actual area that the AI agent will cover in the actual jump trajectory. So in this case more checks have

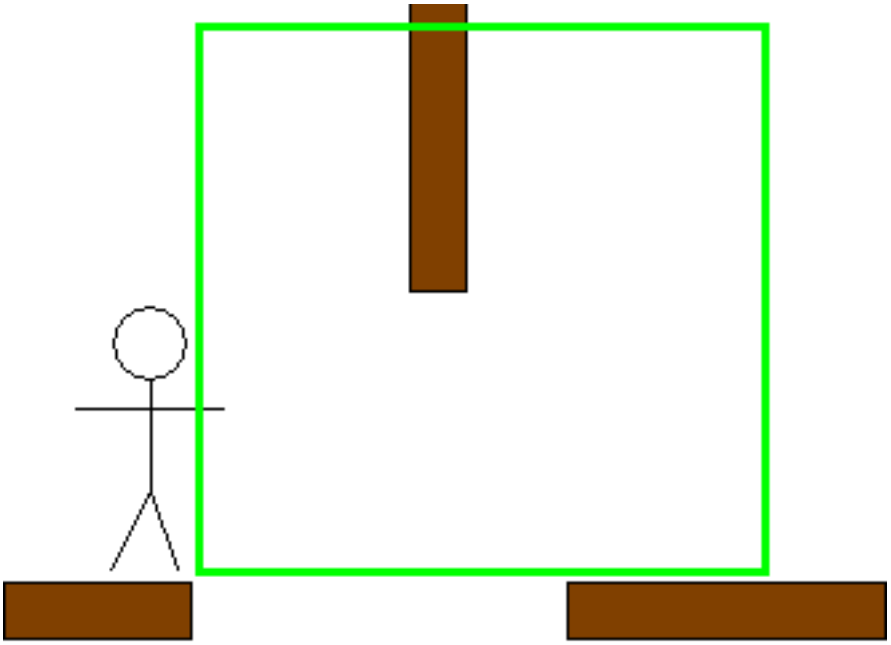

Figure 6.7: The AI agent checking the area that it will jump through over the gorge.

to be done before the AI agent can know if it is possible to jump over the gorge without hitting an obstacle and fall down. Since the jump trajectory that the AI agent, as well as all characters in the game when they jump, will take is a parabola the small checks will be made in somewhat a parabola.

Figure 6.8 shows how the enemy AI agent checks the path it will take in the jump over the gorge. The exact position of the checks varies by some pixels each time to not make the AI agent to predictable. This performs many more then the large check in Figure 6.7 but it covers less area. These checks overlap each other and do not cover the exact path the AI agent will take in the jump, because an exact prediction of the path would take too much time to complete and the game would run slow. If there had been no collision with the checks in Figure 6.8 the jump might be successful and it might fail due to the truncation of float number in the formula used to calculate the distance the enemy can jump. This

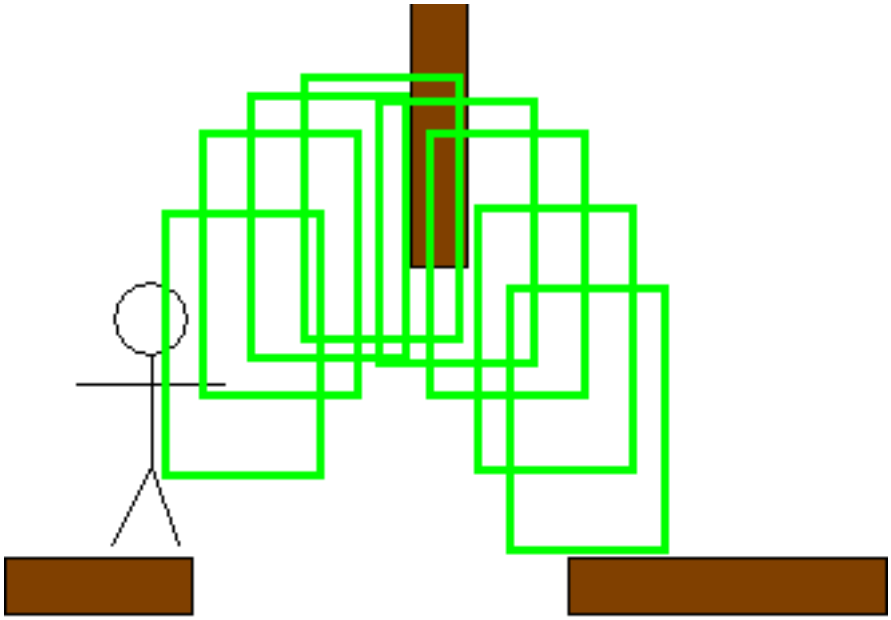

Figure 6.8: The AI agent checking the approximate path it will take in the jump trajectory.

particular problem can be made so it never occur with smart level design, this of course requires that the level designers know what they are doing.

In the scenario in Figure 6.9 where the blocks are inside the large rectangle but outside all the small checks. All the checks are drawn as rectangles in this picture. The small blocks will make the AI perform the second set of checks but in this scenario the second set of checks will tell the AI agent that there are blocks in the way in the second test and it will be possible to jump over the gorge.

See Appendix B.6 for the implementation of this technique.

6.5 Jump over objects

Another kind of obstacle in a 2D platform game are blocks in the path of the AI that have hight and are needed to be jumped over. Figure 6.10 shows an AI agent

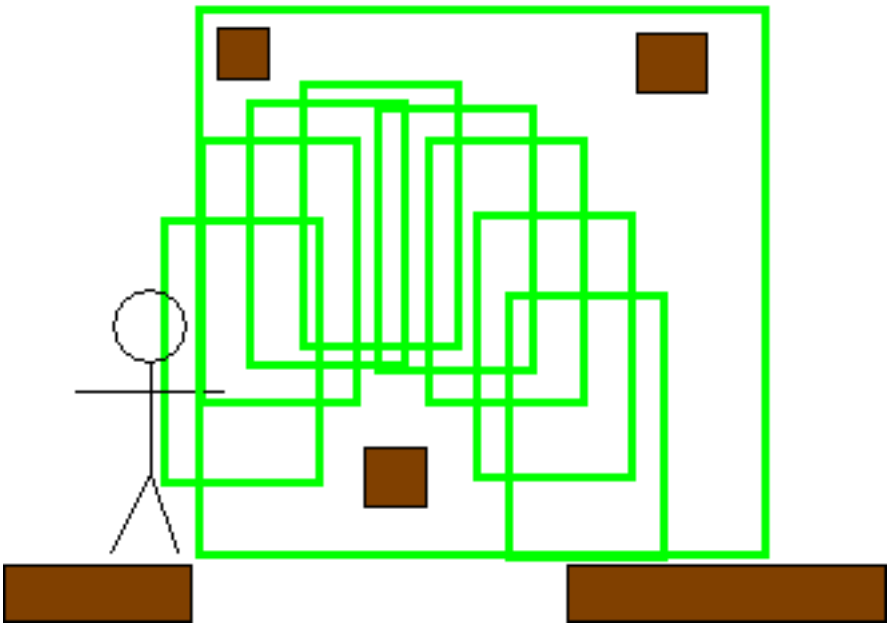

Figure 6.9: A gorge where it is possible for the enemy AI agent jump over it even though there it blocks in the large check.

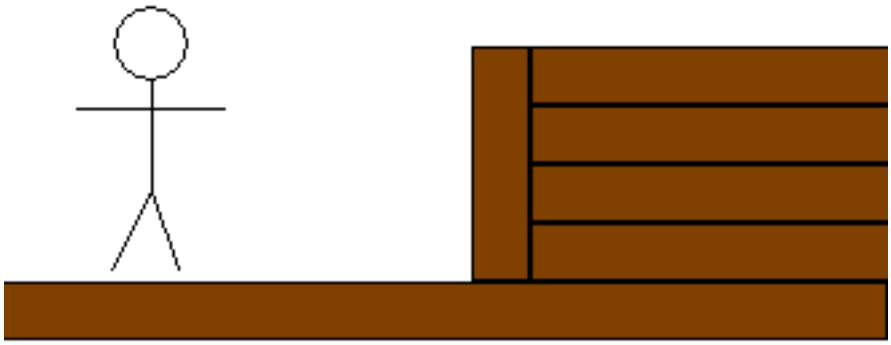

Figure 6.10: The AI agent and a obstacle with height.

with a block of which it has to predict the height of before it knows if it can jump over it.

6.5.1 To know the obstacle is not too high

To get the most natural look, the AI agent has to predict that it will have to jump before it actually reaches the block. Otherwise the AI agent will not jump until it stands too close to the block and the trajectory will not be good.

Figure 6.11 shows what path the AI agent will take if it does not make the jump to get over the obstacle before it reaches it. The solution is to make a check a bit in front of the AI agent. The check have to be made at the position in front of the character at which the character will be at its maximum height if it would jump. Figure 6.12 show where the collision detection should be made. This check is done a little above the ground because it is possible to just walk over small

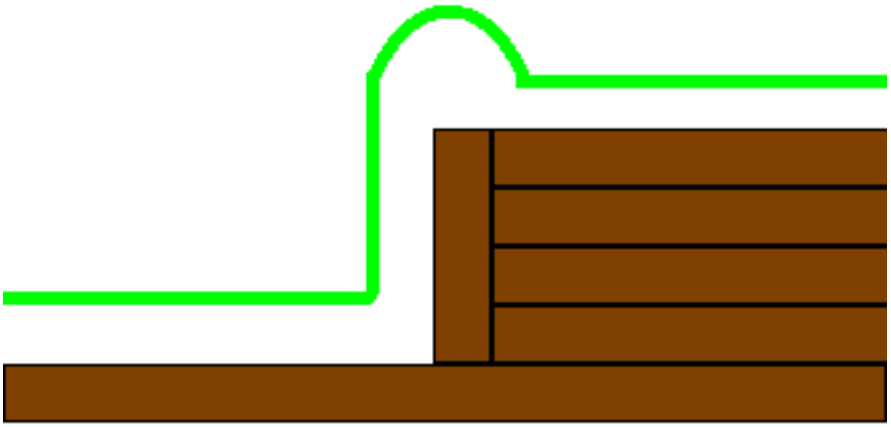

Figure 6.11: Displaying the path the AI agent will take if it do not predict when to jump.

obstacles[1]. The width of the area checked has to be equal or greater than the horizontal velocity[2] of the character otherwise a problem similar to the **Running through walls problem**[1] would occur. The running through walls problem occurs when a character in a game moves faster in pixels per frame then its width and is i simply moved by addition or substraction its velocity to its positions. Then is is possible to pass though a wall if no check is done within the area it is moved. In this case the area checked in front of the character must cover every single pixel that the character will pass. If not it is possible to miss an obstacle and the AI agent will look stupid and the players will laugh at the programmers, which it a very bad thing.

See Appendix B.7 for the implementation of the technique.

When the AI agent in Figure 6.12 gets a true returned from the collision detection it has in front of itself, then it will know that it has to jump in order to continue its path. Figure 6.13 shows the trajectory path that the AI agent will take if it jumps before it reaches the obstacle, compare with the Figure 6.11 when

[1]Or at least that is the goal.
[2]Measured in pixels per frame.

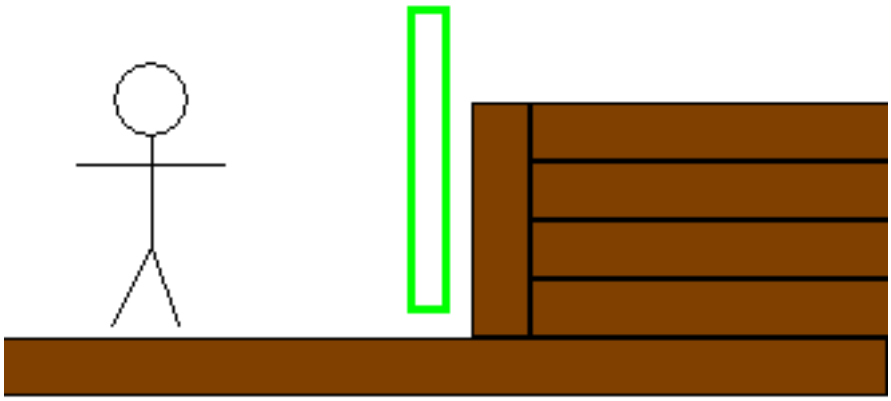

Figure 6.12: The AI agent is checking the path in front of it in case it is a obstacle in the way.

the AI did not jump until it actually reached the obstacle. The path in Figure 6.13 does not only look better, it makes the enemy move faster to because in Figure 6.11 the horizontal movement stops when the AI agent jumps and starts to move vertical and does not start to move horizontal until it gets above the obstacle. In Figure 6.13 the horizontal movement never stops but is constant throughout the entire jump.

6.5.2 To know if it is possible to jump over a obstacle

Sometimes obstacles are too high for the character to jump over and the height of the obstacle has to be measured. If for example the AI agent reaches a wall that it is impossible to jump over then a prediction that the object is too high has to be done. Since the AI agent is part of the game, the computer system, and knows

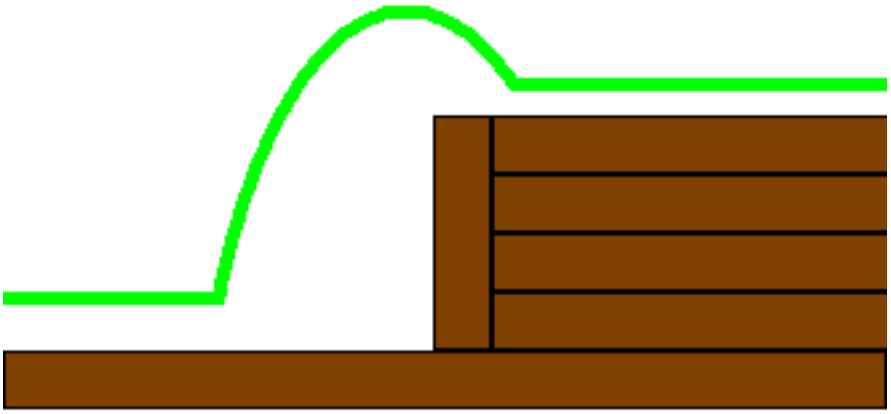

Figure 6.13: Path of the AI agent if the jump hade been predicted before it reached the wall.

everything about every object in the game the size of the object is known. If the block does not reach higher than the character can jump, then the character can jump over it.

See Appendix B.8 for the implentation of this technique.

6.5.3 To know if the opening is big enough

In some cases there might be an opening in a wall that the AI agent has to jump up to in order to get through it with a terrain block that is above the one found with the first check. Then the distance between the blocks may be too small so that the character cannot get through it.

Since this time it is a matter of not hitting objects compared to Section 6.3.3 where the goal was to land on platforms, the purpose here is pass freely over it. If there is a block that stops the enemy AI agent from pass through the opening in the wall the technique described in Section 6.5.2 will not find that the block is there. Figure 6.14 illustrates the AI agent standing in front of a wall with a opening in it. To know if the opening is wide enough the AI has to check an

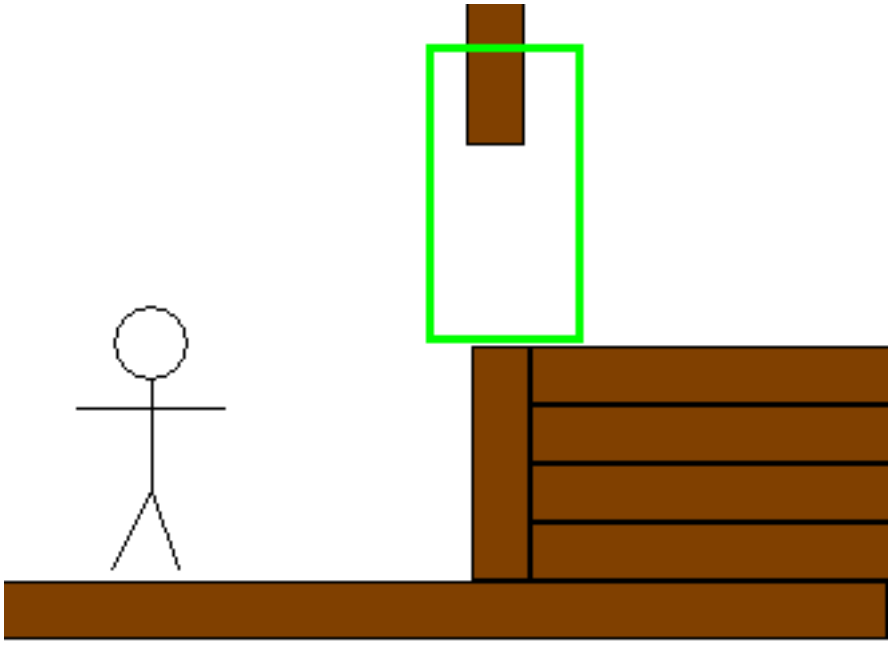

Figure 6.14: A enemy character standing in front of a wall with a opening in it.

area with the size equal to the size of the AI character. This is illustrated by the rectangle in Figure 6.14 where the check is made exactly over the block that the AI agent shall jump over. In this case it is not possible for the AI agent to jump into the opening. If the upper edge of the block that is blocking the way in is below the upper limit of how high the character can jump a second check to see if it is possible to jump over the upper block.

See Appendix B.8 for the implementation of this technique.

6.6 Triggers on the map

Another more common way of making the enemies jump when they are supposed to is to put special triggers events on the map telling the enemy AI agents exactly how to act. This is often done by putting the artificial intelligence itself in the

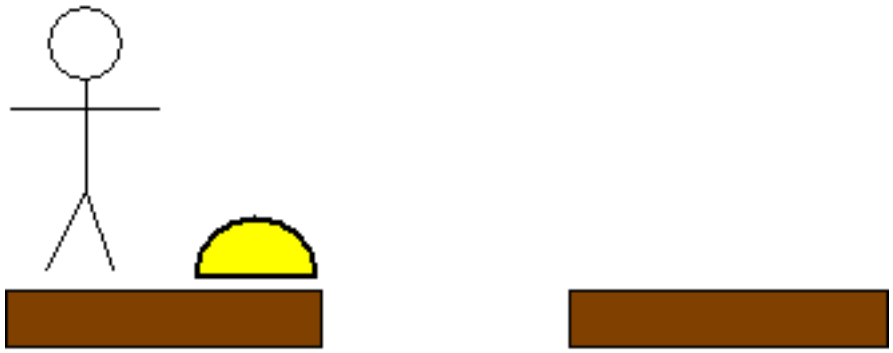

Figure 6.15: An AI agent and a jump trigger.

objects in the game[4], *The Sims* used this technique. Figure 6.15 shows an enemy AI agent in front of a gorge with a trigger, placed at the edge of the gorge, telling the AI agent to jump. In Figure 6.16 it is not possible for a character to jump because there is an object in the air blocking the jump trajectory. So the trigger will give the AI agent the instruction to turn left when it reaches the edge of the gorge. In these two cases the AI agent does no calculation itself, it just does what the triggers tell it to do.

The advocates of this technique point out that it makes the AI agents more flexible when it comes to upgrades. In some cases this is desirable but in 2D platform games that would mean a lot more objects, the triggers, on the map since every little instruction for the enemies would require a trigger. But this would mean that the enemies would move in predictable patterns which is not good in a platform game.

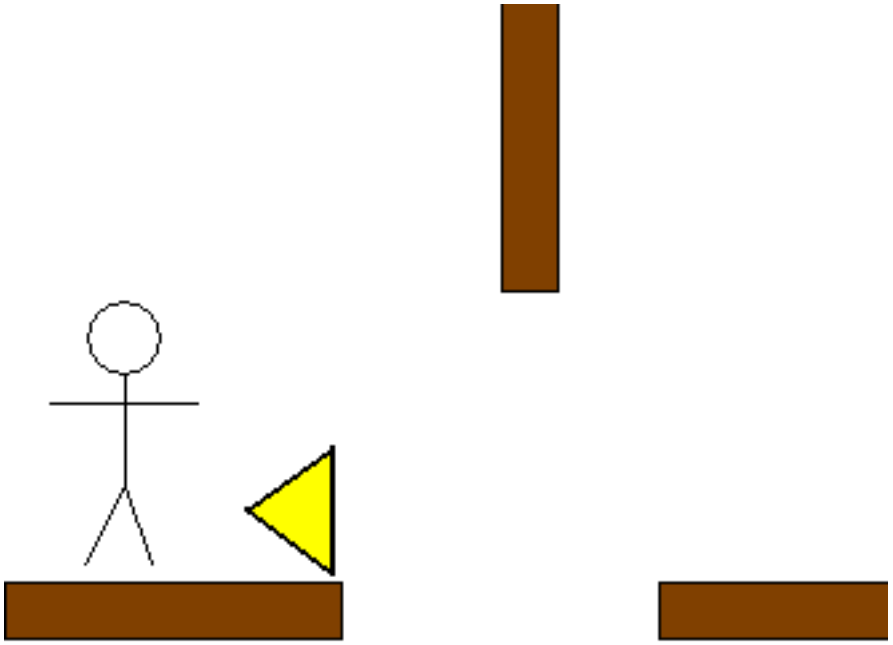

Figure 6.16: An AI agent and a turn left trigger.

It is better if every enemy AI agent acts independently in any part of the map on any map and this can be done with the special form of image recognition described below. In 2D platform games the AI agents have only a couple of simple moves that it is programmed to do. And enemies in games are in most of the time not in need of upgrades. Enemies are what they are and when it is time for the player to meet tougher resistance then new kind of enemies, with different AI techniques, are introduced to make things harder.

6.6.1 Fixed points on the map

In finite state machines when the enemy AI agent needs a few triggers or adjacency points then it is in the state patrol because it needs to know what to patrol. In most cases only one point on the map and a value of how big area should be patrolled is needed. These points can easily be a node used in pathfinding so no

extra points are needed.

6.7 Summary

These techniques described in this section are really simple when given thought but they cover the most common scenarios in 2D platform games. These techniques are not really image recognition, as stated in the chapter title, because they do not really visually analyze the graphics in the game like the player does in the human brain, but they "see" the abstract objects that is the platforms in the game. These general techniques of recognizing the level works with only the terrain itself but it is also possible to cheat even more by placing triggers on the level telling the AI agents what to do when they get there[4].

If these techniques were to be combined at the same time a quite good behavior in a 2D platform game could be archived.

Level design can and should be made to help the AI agents but it should not be too obvious for the player. Good level design can really help the AI agents. But on the other hand crappy level design can make any AI look stupid.

Chapter 7

Pathfinding

Pathfinding is the technique used by the AI agents when they move in non straight lines on a large scale. This chapter will describe a technique for pathfinding that is more efficient and accurate in runtime than the pathfinding with A* algorithm for an 2D platform games. First the basic theory of pathfinding will be described. After that the reason that the architecture of 2D platform games can be the way it is from a pathfinding perspective is discussed. Then the technique itself is described.

7.1 Introduction

Pathfinding with A* is an established pathfinding algorithm in game programming. It is good for games where the enemy AI agents can move forewards, backwards, left and right. But in 2D platform games the characters really move in only one dimension, left and right. Up and down are dimensions that the enemies no not move so much and neither does the player. With the techniques for avoiding obstacles described in Chapter 6 the number of nodes can be greatly reduced.

In reality a search algorithm for pathfinding is only useful for graphs with

very many nodes, but in a 2D platform game where there are relatively few nodes another approach can be made without loosing performance.

7.2 Theory

The nodes in pathfinding with A* know only the cost to pass them or the cost to get between the node and its connected nodes. In a system with few nodes the nodes can contain enough data to know which way is the closest way to all nodes on in the map. In other words, each node knows which one of its connected nodes it should choose next in order get to a specific point on the map the fastest way.

This require that each node has an array containing the index for all other nodes on the map and which of its connected nodes the AI agent should go to next to go the shortest path.

7.2.1 The graph of nodes

The structure of the nodes on a map would be a directed graph just like any other directed graph. The graph would most likely be cyclic but do not have to.

The way the list determines which node is the best path to any given node is calculated can be done with a complete shortest path algorithm when the level is loaded or can be preprogrammed in the file containing all the map data. In either way the CPU time during gameplay is minimal at the expense of memory usage.

7.3 2D platform games need only few nodes

As mentioned earlier 2D platform games need only a few nodes, or waypoints, in order to work. This is due to that the orientation in 2D platform games is so simple. Characters move only left and right, which is just one dimension, and the

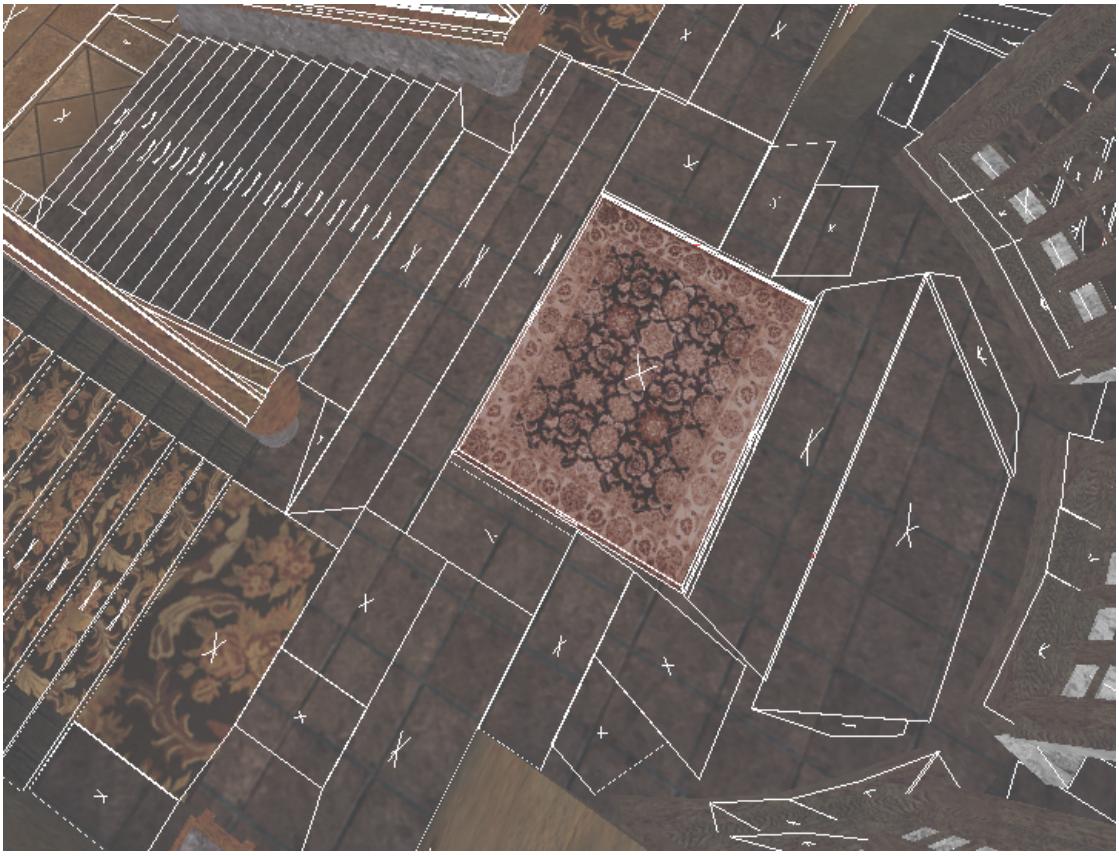

Figure 7.1: A part of a navigation mesh covering a staircase. Each convex polygon (with an 'X' in the center) is a node of the navigation mesh. Courtesy of Paul Tozour and Ion Storm Austin[4].

movement between nodes is in itself extremely simple compared to games with a birds view or a 3D game, where all the movement is in two dimensions and in some cases three dimensions. In 3D games the characters mostly move on a two dimensional surface resembling a floor or something equivalent and thus most of the movement is in two dimensions.

Figure 7.1 shows part of a large system of navigation nodes. A game of this type can contain 20000 nodes[4] Each one of these four cornered red polygon with a 'X' in the middle work as a node for the AI agents pathfinding. That is very

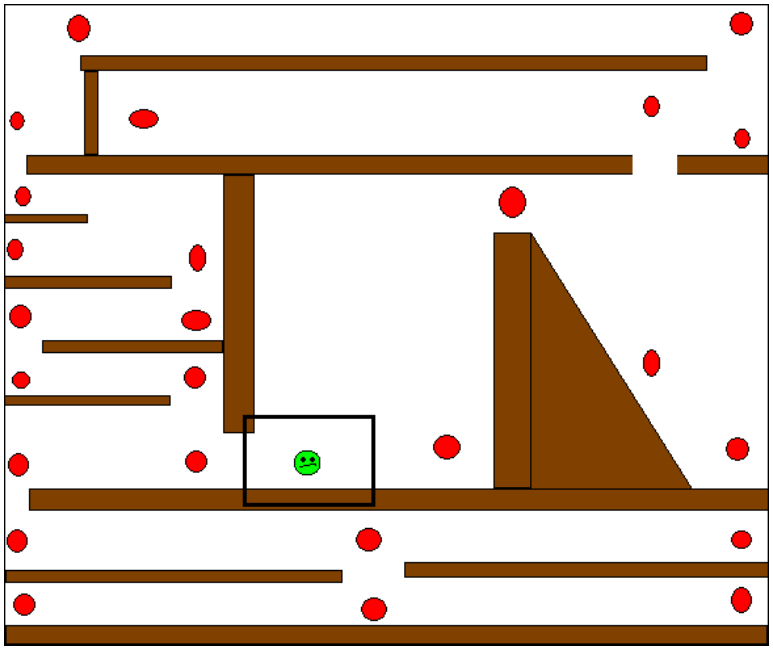

Figure 7.2: A map in a 2D platform game with the pathfinding nodes marked as ellipses.

many nodes on one map which would take up too much of the internal memory if every node knew about all nodes so a pathfinding algorithm like A* is suitable.

In a 2D platform game the map could look like Figure 7.2. The connections are not shown in this picture. The little thing within the rectangle is a AI agent, the rectangle is the AI agent's screen and the entire picture is the map.

The small, differently shaped ellipses are the nodes used for pathfinding. This is a pretty small sized map in a 2D platform game and the number of waypoint nodes, 25, is a lot less then 20,000. If the size of the map were to increase by a factor of 10 then the number of nodes would still only be less than a fraction of 20,000.

The reason that a 2D platform game can contain so few nodes is first of all that the distance between them is great. The distance can be so big because the AI agents can move around the obstacles in its path independently, described in

Chapter 6.

7.4 Implementation

The structure of the nodes could look something like this:

```
struct pathNode
{
    // the position of the node on the map
    int x,y;

    // the global unique identifier of the node
    int index;

    // the list contaning all the other nodes
    // and what of the non connected nodes is the
    // shortest path to each node
    int *list[2];

    // the connected nodes
    pathNode *nodes[];
};
```

The connected nodes and the list are dynamic because they are explicit for each unique map.

The memory usage is formally defined as $O(N^2)$ because each node have information about every other node. The table for each node in the case of the map in

Figure 7.2 with 25 nodes would be 25 lists with 24 rows in each list. If each row in the list is two integers large, each list would be $4 \cdot 4 \cdot 24 = 192$ bytes if integears are 4 bytes large. And all the lists would take up $25 \cdot 192 = 4800$ bytes. With the case of 20,000 nodes the same formula would be $4 \cdot 4 \cdot 19,999 \cdot 20,000 = 6,399,680,000$ bytes and few personal computers have 5.96 GB of internal memory, but almost every computer has more than 4.68 KB memory. The technique described here can for natural reasons not be implemented in a game system with 20,000 nodes but it is fully possible to be used in a 25 node system.

Figure 7.3 shows the same system as in Figure 7.2 but it has the connections drawn out as lines and the nodes have index numbers. All connections are undirected unless there is an arrow in the line indicating the direction. Those are directed because of the height differences, the characters cannot jump high enough to reach the edges so there is only possible to go in one direction between those nodes.

The list for the node with index 22 would be Table 7.1. It contains all nodes on the map in Figure 7.3 except itself and which one of its connected nodes is the closest path to every node on the map.

7.5 Summary

The technique described here, with the nodes having information about all other nodes, is more efficient when the number of nodes are small because there is no realtime searching but would be inefficient with a large number of nodes because the memory usage would be too great.

The technique described in this chapter may have some similarities with a certain form of technique used for finding routes for digital packets of information in a form of ethereal network[10]. It is strange that none of the literature mentioned

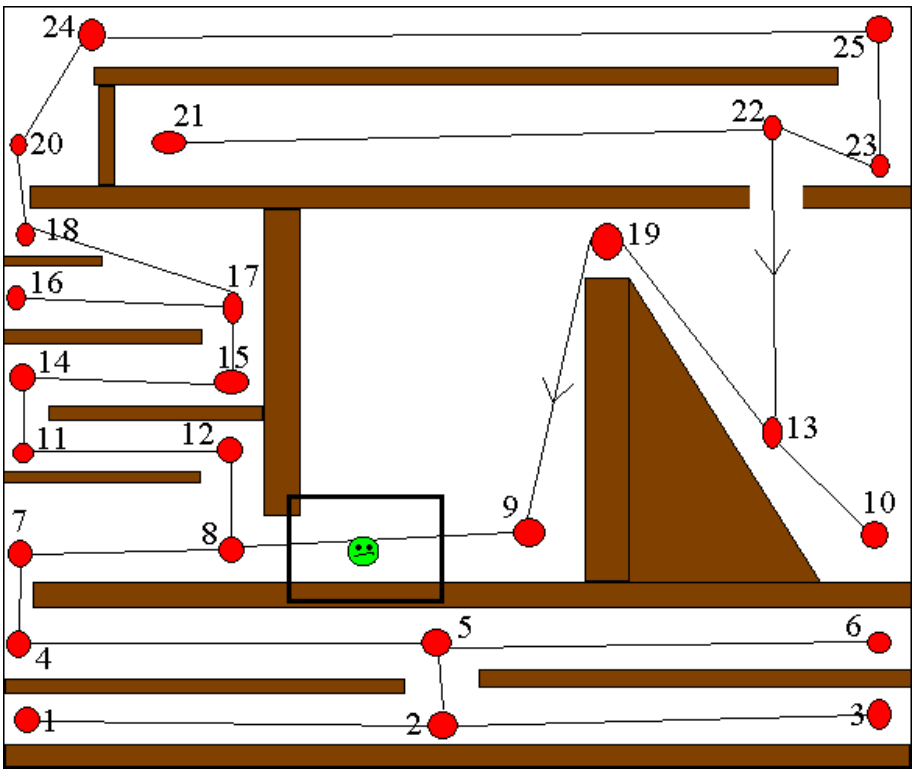

Figure 7.3: The same map as in Figure 7.2 but with the node indexes and connections.

Destination node	Next node
1	13
2	13
3	13
4	13
5	13
6	13
7	13
8	13
9	13
10	13
11	13
12	13
13	13
14	23
15	23
16	23
17	23
18	23
19	13
20	23
21	21
23	23
24	23
25	23

Table 7.1: The list for node 22 in Figure 7.3.

the simple technique that is described in this chapter.

This pathfinding technique was not implementid in this dissertation. But look up any serious book about computer communication and look up link-state routing for a description about how it can be implemented.

Part III

Discussion

Chapter 8

Evaluation

This chapter describe the evaluation of the three techniques line of sight, image recognition and pathfinding. After the introduction there is a section describing the main results. The next section is an evaluation of the implemented techniques compared to the expectations. Then there is a section about cell phones and palm pilots since most of the games for them are in 2D, which makes them worth mentioning.

8.1 Introduction

This dissertation aimed to improve what have stood still for a long time. Hopefully this will change in the future and 2D platform games will become as popular as it was in the middle of the 90:s.

The evaluation of the topics in this dissertation compares the result with the goals that were set when the work started. The goals was simply to make the techniques fulfill the specifications for each technique. The overall results of the techniques have been satisfying. The line of sight technique makes the enemies see the player under the same conditions that the players can see what is on the

screen. The image recognition of the levels makes the enemies act without any pre-programmed movement patterns. Pathfinding was the one technique that was not implemented but in theory it will work since link-state routing works and it uses the same algorithm and link-state routing works.

8.2 Achievements

There have been mechanisms implemented in this dissertation. It is for genre 2D platform games that have not been improved for a long time.

The expectations were to have the foundation of an AI system for a 2D platform game and this has been achieved. The enemy AI agents act independently on the level and can be placed on any 2D platform game map. The pathfinding waypoints still need to be placed on the map but that cannot be avoided with today's computers.

This section will discuss the results compared to the expectations. In line of sight the expectations were to have a working technique that made the enemies only notice the player when it were nothing blocking the way between the player and the enemy. In the image recognition of the level technique the expectation was to have a technique that made the enemies adapt to there surroundings and do not need to be pre programmed for each situation. In pathfinding the expectation was just to make the enemies find their way on the level.

8.2.1 Line of sight

The line of sight technique is detailed, exact and gives a good result during gameplay. It makes the enemies only notice the player when the player is on the enemy's screen. If the player hides behind big blocks the enemy AI agent cannot see the player. It is possible for 40 enemies at a time to be on the screen and

perform a line of sight check each without any lagging. In fact, even more enemies can be on the screen simultaneously if the line of sight check is turned of when the enemies have seen the player. If the enemies then can communicate through an **observer** system[15] then no checks will be needed for all the close by enemies. Overall the line of sight technique works satisfyingly.

The line of sight technique is much more detailed than the line of sight techniques used in the old 2D games from the 1980:s and 90:s. Once the algorithm was done all that is needed is a function call in the AI function call during runtime. The only requirement is that the AI agent has knowledge of the terrain, just like the requirement for the physics within the game.

8.2.2 Image recognition of the level

This was not a set goal when this dissertation started but it turned out to be such an important compliment to making pathfinding work that it became a topic of itself. The goal was to make the enemy AI agents move independently on the map across the level. The AI agents can move in both directions and find their ways around simple obstacles and change their direction when they cannot pass an obstacle.

This technique is both simple and rather complicated at the same time. It is simple because in theory it is just a matter of checking a certain area and see if there is nothing within that area. It is complicated when it comes to calculating where exactly these areas are.

The checks needed in image recognition of the level are less then a fraction of what is needed in line of sight even though these checks are done even by the enemies that are not on the player's screen. So all enemies on the level need to do the checks all the time but there are so few checks that there is no lagging.

Compared to triggers image recognition of the level makes the enemy AI agents

more independent. It is possible for the level designers to place any enemy on any map without spending work on placing the triggers on the map. And when it comes to home made maps the amateur level designers will not be pleased when they have to do some semi-programming when they place the instructions on the map. Of course this is a matter of taste among the different modding communities but in general it is popular when as little as possible is needed to be done.

The examples in this dissertation cover only moves when jumping is involved. But the technique can easily be extended to involve other kind of moves.

8.2.3 Pathfinding

The pathfinding technique is a simple brute-force technique that saves processing power at the cost of memory. This technique for finding paths or routes has been applied for a long time but not in games. It is surprising that none of the literature about game programming and game design mentions that the algorithm for link-state routing[10] could be used in game pathfinding to increase the frame rate.

In games with few nodes this technique is useful because it saves processing during runtime and give the same result as a search algorithm. But in games with many nodes the memory usage weights over the search time for the algorithm and it is better to search then to keep record of what is the best way. There is no exact number set when it is better to keep record of a path like with the technique presented in this dissertation or when it is better to search each for a path time it is needed likw with pathfinding with A*. The technique of choice have to be decided by the programmers in each case.

It is much simpler then the pathfinding with A* algorithm. And it does not require more work of placing the nodes on the map then the pathfinding with A* needs because a shortest path algorithm can fill in the table needed for each node

when the map is designed by the level designers.

The fact that this kind of pathfinding always chooses the best path in the map can by some people be compared with triggers on the map and give the enemy AI agents a preprogrammed perfect path to follow, just like triggers do, but on a larger scale. But remember that these are enemies in a game on a map and that they are supposed to have knowledge, meaning that the enemies have already explored the level before the player got there.

8.3 Cell phones and palm pilots

2D games have gained some popularity again after almost ten years of silence in the form of games for cell phones and palm pilots, with Nokia's N-Gage as one of the most specialized for playing games. The hardware in cell phones and palm pilots is a lot less powerful then the hardware on computers and game consoles. This of course means a limitation on how advanced the games can be compared to computers and game consoles.

8.3.1 Line of sight

The main issue with the line of sight technique is that it requires so many collision detection checks, one for each line between the enemy AI agent doing the check and the character or object that it is trying to see. But the screen resolution on cell phones and palm pilots are a lot less then the screen resolution on a computer. The line of sight algorithm was test run with 1600×1200 which is much compared to games of today standards, the PAL system use 800×600 which means the console games are limited to that resolution. Computers on the other hand do not have that resolution restriction but since most of the 2D games were released on consoles that did not have that great resolution.

Cell phones and palm pilots have only a couple of hundred pixels resolution in each dimension which means a lot less collision detection so the checks will not drain so much of the CPU power.

8.3.2 Image recognition of the level

The number of checks needed for this technique will not decrease if the screen resolution decrease. The area that is checked will be smaller but that fact does not matter, it is the number of checks that drain the CPU power. But the number of checks is only a couple of dozen in worst-case scenarios when several enemies AI agents reach an obstacle in the exact same frame. So the problem of less powerful hardware will not occur with this technique.

8.3.3 Pathfinding

Since this technique has nothing to do with what is shown on the screen it will not be affected by that the screen is smaller on cell phones and palm pilots. But since the hardware for cell phones and palm pilots is less powerful than on computers and game consoles, the pathfinding with A* algorithm will probably drain so much CPU power that it might be better to use the technique described in this dissertation.

8.4 Summary

The line of site technique does not give the enemy any unfair advantage on the player. The image recognition of the level technique do not make the enemies move in pre programmed patterns but it still gets them across the level to the given destination. The pathfinding technique makes the enemies move around on the level like it was their home field and saves CPU power.

The line of sight technique can handle about 40 enemies on the screen at the same time which is more then most there will be in most games. The image recognition have fulfilled the goal of making the AI agents find its way around simple obstacles but only covers movement in left and right and jumping. The pathfinding technique gets the job done even though the path is pre programmed but the path the AI agent would take with pathfinding with A* is also pre programmed so it will give no more mechanical feeling to the enemies than pathfinding with A* would.

The techniques are assumed to work well on games on cell phones and palm pilots because they are so simple and do not drain the CPU unnecessary much and in the case of line of sight the limited visual limit cuts down the cost for calculating it.

Chapter 9

Conclusions

This chapter provides the conclusions for all techniques in the experiment part. First there will be a discussion about whether the result of each technique is good or not. Then the different problems with each technique will be mentioned and how they were solved.

9.1 Conclusion

AI techniques needed in game AI do not need to be perfect all the time. An enemy AI agent that is too good will be impossible to beat and not fun to play against. The results of the three techniques affect the backend behaviour of the AI agents. The line of sight is used to collect information about hostile characters. Image recognition of the level is used to check if a obstacle is passable and pathfinding is used to choose what way to go.

9.1.1 Line of sight

The line of sight techniques mentioned in the literature were designed for 3D games where approximation is needed for not slowing down the game. In 3D games there

is no visual limit to the free sight checks so there is no theoretical length limit so all enemies have to check for free sight all the time. That would take drain too much of the CPU power and would cause lagging.

The technique is good in 2D games but not so good in 3D games.

9.1.2 Image recognition of the level

This is a topic of game AI that never can be proven to be perfect. But as long as it gets the job done it will be considered good. And no players wants the AI agent to make the perfect decision all the time. The feeling of being better than the enemies when an enemy gets stuck at an obstacle when it is chasing the player and the player have low health is something all game makers want their players to experience.

9.1.3 Pathfinding

Compared to pathfinding with A* algorithm the technique described in this dissertation will give the best choice of path each time which the A* algorithm will not.

Pathfinding with A* is preferable when there are a lot of nodes but in systems with few nodes it is faster to have a technique when all nodes know the best path to all other nodes, because it will not drain CPU power. But with a system with too many nodes the nodes will need too much of the internal memory and the system will run slow.

9.2 Problems

Since the techniques described use mainly brute-force in one form or the other the main problem is hardware restriction which can be solved with more memory and

faster processors.

9.2.1 Line of sight

The main problem with this technique is that it requires so many collision detection checks which makes the game lag if they are to numerous. A perfect line of sight would need to check every single possible line between the AI agent and the player. If the sprite of the player is 255×255 the worst-case scenario would require 509 checks. If the player is as far away as possible from the enemy this would require $255 \times (800 - 255) + 255 \times (600 - 255) = 226950$ single collision detection checks which are too much to today's processors to handle in 30 milliseconds. Even though there may exist a line that is free between the AI agent and the player there is no way to know if the line has free sight unless it is checked. The problem was to choose which one of the lines to check.

To pick a random line of all possibilities is not perfect but it still means that there will be 33 checks per second, which means that statistically every line will be checked in 16 seconds. Which means that when only one pixel of the player is visible to the enemy it will take some time for the enemy to notice the player. This actually turned out to be a good thing because it means that if the player hides behind an object, the enemy will not see the player immediately. Just like in reality it takes some time to find objects that are hidden.

Compared to approximation techniques where is it possible to be within the enemy's line of sight for an infinite amount of time without being seen[4], this gives a natural feeling to the enemies at the same time as it is more detailed.

9.2.2 Image recognition of the level

There were really no problems with this technique unless the check is expected to be perfect. Which would mean checking every possible path that the enemy AI

agent could take in the jump trajectory in the case with jumping. The problem with approximations makes the enemies sometimes do clumsy things, which is funny for the players to watch unless it is predictable. It will seldom do the clumsy things, unlike when a trigger was misplaced on the map which would make the AI agent do the same mistake every time.

9.2.3 Pathfinding

Since an existing pathfinding technique was used no problem existed at all. The technique is the same as the one used for finding routes but it works well for paths as well.

Someone who is into computer communications might feel tempted to try the pathfinding with A* algorithm on routing and see how well that works. I leave it as an unanswered question.

Chapter 10

Plans for Future work

Future work is a very open subject with the techniques described in this dissertation. The techniques described cover far from all that is needed in a complete 2D platform game and there is a lot of room left for improvement. This dissertation covered only some basic techniques that, if they were to be the only techniques in the AI of a game, would be very much of a challenge to a player. What is needed is a complex finite state machine that handles combat with the player.

10.1 Line of sight

This technique is more or less complete. If the player is only partly visible to the enemy then it will take some time before the enemy sees the player because only one single line is checked every frame and that line is picked at random. In the future when the computers are fast enough to check every single line between the enemy AI agent and the player the technique will be perfect and the enemies will see the player immediately when it is possible for the enemies to see the player. Then it is possible to make the perception of the enemies into a skill just like any

other skill of the characters in the game. The enemies have to do a skillroll every tenth second or see if they have noticed the player just like in ordinary paper RPG.

10.2 Image recognition of the level

This is the topic that is in most need of improvement and also the topic that has the greatest possibilities to be improved.

One of the level analyzation with image recognition and, corresponding move that can be added is crouching. The enemy can if it reaches a wall check if there is an opening big enough for it to crawl through. Then if there is an opening in the wall the AI agent can crawl, crouch or stand on its knees. This of course requires a more complex finite state machine for the enemy that have the state *crouching*.

Additionally special moves like grabbing the edges of cliffs and rooftops. This check needed for this would just be a simple collision detection check in the reachable area to see if there is an edge of something that it climbs up on.

There is a lot more things that can be measured with image recognition of the level. Just like in mainstream AI there is no theoretical limit for what information can be gained with image recognition there is no limit for what information can be gained with image recognition of the level in game AI.

When the computer hardware is powerful enough to give a perfect simulation of the jump trajectory and make several simulations with different jump trajectories at different speeds and heights it would be good to add the skill jump to the enemy AI agent's skills just like it could be done with perception. If the skillroll is successful the enemy AI agent will jump, if the roll is unsuccessful the enemy AI agent will not jump and if the roll is a critical failure the enemy will fall down. This way there will be unpredictable clumsiness among the enemies, which is something that will add to the gaming experience for the player.

10.3 Pathfinding

There is really no possible way for improving this technique itself. But adding more nodes to the level would make the technique more precise, but that would consume more memory. When the computers are fast enough and have enough memory a perfect shortest path algorithm will replace this technique because the computer could calculate it without any lagging and pathfinding with A* would stop being used because it is not perfect.

Like the two other techniques, line of sight and image recognition of the level, an intentional stupid decision can make the AI less mechanical. If the enemy is in a stressful situation, like when it has low health, low ammunition and is being chased by the player, it might take the wrong path in the level resulting in not escaping. In this scenario fuzzy logic can be useful to determine if it is a stressful situation or not.

References

[1] Daniel Lindsäth and Martin Persson, *Implementation of a 2D Game Engine Using DirectX 8.1*, Karlstad Univerisy, Bachelor's Project 2004:24, 2004.

[2] Robert L. Glass, *Facts and Fallacies about Software Engineering*, Addison Wesley, ISBN 0321117425, 2004.

[3] David M. Bourg, *AI for Game Developers*, O'Reilly & Associates, ISBN 0596005555, 2004.

[4] Steve Rabin, *AI Game Programming Wisdom*, Charles River Media, ISBN 1584500778, 2002.

[5] Steve Rabin, *AI Game Programming Wisdom 2*, Charles River Media, ISBN 1584502894, 2003.

[6] Stuart Russell and Peter Norvig, *Artificial Intelligence: A Modern Approach*, Prentice Hall, 2st Edition, ISBN 0130803022, 2003.

[7] André LaMothe *Tricks of the Windows Game Programming GURUS*, Sams, 2nd Edition, ISBN 0-672-32369-9, 2002.

[8] André LaMothe *Windows Spelprogrammering för DUMMIES*, IDG AB, ISBN 91-7241-006-X, 1999.

[9] John E. Hopcroft, Rajeev Motwani and Jeffery D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, 2st Edition, 2000.

[10] James F. Kurose and Keith W. Ross, *Computer Networking: A Top-Down Approach Featuring the Internet*, Addison Wesley, 2st Edition, ISBN 0201976994. 2002.

[11] Jack Bresenham, *Algorithm for Computer Control of a Digital Plotter*, IBM Systems Journal Volume 4 Number 1 pp 25-30, 1965.

[12] Lotfi A. Zadeh, *Fuzzy sets*, Inf. Control 8, 338-353, 1965.

[13] Julian Gold, *Object-Oriented Game Development*, Addison Wesley, ISBN 032117660X, 2004.

[14] Ayn Rand, *Introduction to Objectivist Epistemology*, Meridian, ISBN 0452010306, 1990.

[15] Erich Gamma, Richard Helm, Ralph Johnson and John Vilssides, *Design Patterns*, Addison Wesley, ISBN 0201633612, 2004.

[16] Ernest Parza, *Focus on SDL*, Premier Press, ISBN 1592000304, 2003.

[17] Bill Slavicsek, Rich Baker and Kim Mohan, *Dungeons & Dragons For Dummies*, Pagina, ISBN 0764584596, 2005.

[18] George Orwell, *Nineteen eighty-four*.

[19] *Retro Gamer*, Live Publishing, Issue 12 p 28.

[20] *Svenska PC Gamer*, Hjemmet Mortensen AB, Issue 98 pp 44-53.

[21] American Heritage Dictionarie, *The American Heritage Dictionary of the English Language*, Houghton Mifflin, 4th Edition, ISBN 0395825172, 2000.

[22] Alan Turing, *Computing machinery and intelligence*, Mind, vol. LIX, no. 236, pp. 433-460, October 1950.

[23] Kurt Gödel, *Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme*, I. Monatshefte für Mathematik und Physik 38, (1931), pp. 173-198. Translated by Harvard University Press, 1971.

[24] *www.dictionary.com*, http://dictionary.reference.com/search?q=agent, 2005-02-21.

[25] *The American Assosiation of Artificial Intelligence*, http://www.aaai.org, article *Expert Systems And Artificial Intelligence*.

[26] *www.wikipedia.org*, http://en.wikipedia.org/wiki/Expert_system, 2005-05-10.

[27] *www.wikipedia.org*, http://en.wikipedia.org/wiki/Turing_test, 2005-04-04.

[28] *Google Scholar*, http://scholar.google.com/.

[29] *Deep Blue*, http://en.wikipedia.org/wiki/Deep_Blue, 2005-05-05.

[30] *Guinness World Records* http://www.guinnessworldrecords.com, 2005-12-12.

Part IV

Appendixes

Appendix A

Line of sight

This appendix will describe the functions used to implement the technique described in Chapter 5. Bresenham's algorithm is such an important part for optimizing the implementation of the technique that it is included in this appendix.

The functions *Free_Sight*() and *Simple_Ground_Collision*() are the bottlenecks of the line of sight technique. Because *Simple_Ground_Collision*() is dependent of the number of objects in the terrain the efficiency is $O(N)$ where N is the number of objects in the terrain. *Free_Sight*() uses the Bresenham algorithm and is dependent of the size of enemies screen, se Section 5.2.1 for details about this, so the efficiency for *Free_Sight*() is $O(M)$ where M is the absolute value of the distance between the player and the enemy, this is half the diagonal of the screen in the worst case scenario. So the efficiency for the line of sight technique is $O(MN)$. But for the entire AI system, which also have a number of enemies that use the line of sight technique, the efficiency is $O(NML)$ where L is the number of enemies on the screen.

All the variables and constants that are used in the code but not are in the arguments or is defined in the functions are protected data members of the object

Enemy.

See Chapter 5 for the theory this technique.

A.1 Bresenham's algorithm

This function is an example of how Bresenham's line algorithm can be implemented. This function only draws a line between two points but a modified version of this can be used for any type of line scanning, like detailed movement in game physics and line of sight checks in game AI. This code is simply a C++ implementation of the algorithm described in Section 5.2.3. The function *putpixel*() simply draws a pixel on the screen at the coordinates $x0$ and $y0$ the 8-bit color *color*.

The theory for this function is described in Section 5.2.3.

A.1.1 Code

```
void Bresenham(int x0,int y0,int x1,int y1,int color)
{
    if(x0==x1&&y0==y1)
    {
        putpixel(x0,y0,0);
        return;
    }

    int steep = 1;
    int sx, sy;
    int dx, dy;
    int e;

// inline swap for optimation purposes
int tmpswap;
#define SWAP(a,b) tmpswap=a; a=b; b=tmpswap;

    // optimize for vertical and horizontal lines here
    dx = abs(x1 - x0);
    sx = ((x1 - x0) > 0) ? 1 : -1;
```

```
    dy = abs(y1 - y0);
    sy = ((y1 - y0) > 0) ? 1 : -1;

    if(dy > dx)
    {
        steep = 0;
        SWAP(x0, y0);
        SWAP(dx, dy);
        SWAP(sx, sy);
    }

    e=(dy<<1)-dx;

    for(int i = 0; i <= dx; i++)
    {
        if (steep)
        {
            putpixel(x0,y0,color);
        }
        else
        {
            putpixel(y0,x0,color);
        }

        while(e >= 0)
        {
            y0 += sy;
            e -= (dx << 1);
        }
        x0 += sx;
        e += (dy << 1);
    }
}
```

A.2 Line of sight

It is this function that is called in the enemies AI function. The first values calculated in the function is the virtual screen of the enemy. After that it calculates the coordinates of the rectangle of the character it shall check if it can

see. The coordinates are not the coordinates of a rectangle in the normal sense, that it, the x and y coordinates and a height and a width, but the position of the four one dimensional straight lines for drawing the rectangle. Then the function *Character_On_Screen*() is called to se if the player is inside the enemy's screen. If so there are four independent checks to see if any part of the player's rectangle is outside the screen. If so the rectangle is recalculated so that only the part of the player's rectangle that is inside the screen of the enemy will be used for the next part of the function. Then it is time to check if it is possible to draw a line between the enemy and the player without hitting any object in the terrain. The position of the target point of the line is picked at random within the player because of the result of Section 5.3.3.

A.2.1 Code

```
bool Enemy::Line_Of_Sight(Character *character_arg)
{
    // calculate the corners of the enemies screen
    int left,right,up,down;
    left=x-SCREEN_WIDTH/2;
    right=x+SCREEN_WIDTH/2;
    up=y-SCREEN_HEIGHT/2;
    down=y+SCREEN_HEIGHT/2;

    // calculate the players rectangle
    int vansterkant=character_arg->x,
        hogerkant=character_arg->x+character_arg->width-3,
        bredd=character_arg->width-2,
        overkant=character_arg->y,
        underkant=character_arg->y+character_arg->height-3,
        hojd=character_arg->height-2;

    // check if the player is on the enemies "screen"
    if(Character_On_Screen(character_arg))
    {
        // first, correct the checkabel area to that it
```

```
        // only is the part of the enemy that is inside
        // the enemies "screen"

        // check if there is any part of the player that is
        // outside of the enemies "screen"
        if(right<hogerkant)
        {
            // the players right edge is outside
            // the enemies "screen"
            bredd-=hogerkant-right;
        }
        if(vansterkant<left)
        {
            // the players left edge is outside
            // the enemies "screen"
            bredd-=left-vansterkant;
            vansterkant=left;
        }
        if(down<underkant)
        {
            // the players lower edge is outside
            // the enemies "screen"
            hojd-=underkant-down;
        }
        if(overkant<up)
        {
            // the players upper edge is outside
            // the enemies "screen"
            hojd-=up-overkant;
            overkant=up;
        }
        // check if there is free sight between the enemy
        // and the area of the player that is on the
        // enemies "screen"
        if(Free_Sight(vansterkant+rand()%bredd,
                      overkant+rand()%hojd))
        {
            return(true);
        }
    }
```

```
    return(false);
}
```

A.3 Character on screen

This function checks if the character in the argument is on the enemies virtual screen. It simply checks if there is a collision between the rectangle of the character in the argument and the rectangle formed by the visual limit, se Section 5.2.1, of the enemy using the function $Collision\_Test2()$ with the lines of the rectangle of the player and the lines of the rectangle of the enemy's screen as arguments. If there is a collision the function returns $true$ if not it returns $false$.

The theory for this function is described in Section 5.2.1.

A.3.1 Code

```
bool Enemy::Character_On_Screen(Character *character_arg)
{
    return(Collision_Test2(character_arg->x,
            character_arg->y,
            character_arg->x+character_arg->width-3,
            character_arg->y+character_arg->height-3,
            x-SCREEN_WIDTH/2,
            y-SCREEN_HEIGHT/2,
            x+SCREEN_WIDTH/2,
            y+SCREEN_HEIGHT/2));
}
```

A.4 Collision test

This function is named $Collision\_Test2()$ instead of just $Collision\_Test()$ because it is version 2 of the collision function used in the game engine. This function takes eight parameters, these are the one dimensional lines used to draw the rectangles it is checking for collision of. It starts by subtracting two from coordinates of

the left and lower lines of the both rectangles. This is due to the implementation of the graphics engine. So in another graphics engine this subtraction should perhaps not be done. But with the engine written by André LaMothe in *Tricks of the Windows Game Programming GURUS*[7] and *Windows Spelprogrammering för DUMMIES*[8] used for the experiment in this dissertation this subtraction is necessary. The collision test itself it just a large number of comparisons and boolean algebra.

A.4.1 Code

```
int Collision_Test2(int x1, int y1, int w1, int h1,
                    int x2, int y2, int w2, int h2)
{
    w1-=2;
    h1-=2;
    w2-=2;
    h2-=2;

    return( ((x1<=x2 && x2<=w1) && (y1<=y2 && y2<=h1)) ||
            ((x1<=w2 && w2<=w1) && (y1<=h2 && h2<=h1)) ||
            ((x1<=x2 && x2<=w1) && (y1<=h2 && h2<=h1)) ||
            ((y1<=y2 && y2<=h1) && (x1<=w2 && w2<=w1)) ||
            ((x2<=x1 && x1<=w2) && (y2<=y1 && y1<=h2)) ||
            ((x2<=w1 && w1<=w2) && (y2<=h1 && h1<=h2)) ||
            ((x2<=x1 && x1<=w2) && (y2<=h1 && h1<=h2)) ||
            ((y2<=y1 && y1<=h2) && (x2<=w1 && w1<=w2)) );
}
```

A.5 Free sight

This function is an adaption of Bresenham's algorithm. It checks if it is possible to draw a line from the enemy and a given point. If there is no terrain objects in between the sight is free and *true* is returned, but if there is $false$ is returned. The code is exactly like the code in Bresenham's algorithm with the exception that it

checks to se if the point is colliding with the ground instead of drawing a pixel. It uses *Simple_Ground_Collision*() to check if there is no ground object on the line.

The theory for this function is described in Section 5.2.2.

A.5.1 Code

```
bool Enemy::Free_Sight(int x1,int y1)
{
    int x0=x,y0=y;

    if(x0==x1&&y0==y1)
    {
        return(true);
    }

    int steep = 1;
    int sx, sy;
    int dx, dy;
    int e;

// inline swap for optimation purposes
int tmpswap;
#define SWAP(a,b) tmpswap=a; a=b; b=tmpswap;

    // optimize for vertical and horizontal lines here
    dx = abs(x1 - x0);
    sx = ((x1 - x0) > 0) ? 1 : -1;
    dy = abs(y1 - y0);
    sy = ((y1 - y0) > 0) ? 1 : -1;

    if(dy > dx)
    {
        steep = 0;
        SWAP(x0, y0);
        SWAP(dx, dy);
        SWAP(sx, sy);
    }

    e=(dy<<1)-dx;
```

```
    for(int i = 0; i <= dx; i++)
    {
        if(steep)
        {
            if(Simple_Ground_Collision(x0,y0))
            {
                return(false);
            }
        }
        else
        {
            if(Simple_Ground_Collision(y0,x0))
            {
                return(false);
            }
        }

        while(e >= 0)
        {
            y0 += sy;
            e -= (dx << 1);
        }
        x0 += sx;
        e += (dy << 1);
    }

    return(true);
}
```

A.6 Simple ground collision

The purpose of this function is very simple, it checks if the given points in the argument list is colliding any of the terrain objects. It is a for-loop that goes through every terrain object and checks if the position of the given point in the argument is inside the rectangle formed by the coordinates and size of the terrain object. If the given point is within any of the terrain blocks the function will return $true$ otherwise $false$.

A.6.1 Code

```
bool Enemy::Simple_Ground_Collision(int x_arg,int y_arg)
{
    // every single terrain block have to be checked
    for(int i=0;i<Terrain_Size[GROUND];i++)
    {
        if((Ground[GROUND][i]->x<=x_arg &&
            x_arg<=Ground[GROUND][i]->x + Ground[GROUND][i]->width-2) &&
            (Ground[GROUND][i]->y<=y_arg &&
            y_arg<=Ground[GROUND][i]->y + Ground[GROUND][i]->height-2))
        {
            return(true);
        }
    }
    return(false);
}
```

Appendix B

Image recognition of the level

As las appendix to this dissertation this appendix will be the implementation of the image recognition of the level technique. the functins desvribed here are the ones used in the AI function in the object Enemy in the game.

All the variables and constants that are used in the code but not are in the arguments or is defined in the functions are protected data members of the object Enemy.

The theory for this technique is described in Chapter 6.

B.1 Reached left edge

This function checks if the enemy AI agent have reached an edge on the left side in the virtual world. It checks a area with the with equal to the width of the enemy since it can just walk over areas smaller then its width. If there is no collision with any of the terrain blocks within that area it means that the enemy will fall if it continue without jumping or changing its direction. The $Collision\_Test2()$ used is the same $Collision\_Test2()$ that was described in A.4.

The theory for this function is described in Section 6.3.1.

B.1.1 Code

```
bool Enemy::Reached_Left_Edge(void)
{
    // check every terrain block
    for(int i=0;i<Terrain_Size[GROUND];i++)
    {
        if(Collision_Test2(x-width, y+height,
            x, y+height+10,
            Ground[GROUND][i]->x,
            Ground[GROUND][i]->y,
            Ground[GROUND][i]->x+Ground[GROUND][i]->width,
            Ground[GROUND][i]->y+Ground[GROUND][i]->height))
        {
            return(false);
        }
    }

    return(true);
}
```

B.2 Reached right edge

This function works exactly the same way as *Reached_Left_Edge*() except that it checks the edge on the right side instead of the left.

The theory for this function is described in Section 6.3.1.

B.2.1 Code

```
bool Enemy::Reached_Right_Edge(void)
{
    // check every terrain block
    for(int i=0;i<Terrain_Size[GROUND];i++)
    {
        if(Collision_Test2(x+width, y+height,
```

```
                x+width*2, y+height+10,
                Ground[GROUND][i]->x,
                Ground[GROUND][i]->y,
                Ground[GROUND][i]->x+Ground[GROUND][i]->width,
                Ground[GROUND][i]->y+Ground[GROUND][i]->height))
            {
                return(false);
            }
        }

        // return no collision
        return(true);
}
```

B.3 Block within jumprange

This function calculates the distance that the enemy can jump horizontally using the jumpforce and the velocity if the enemy and the gravity of the level. Then checks if there is a terrain block within that distance using $Collision\_Test2()$. It returns *true* if there is a terrain block within the distance it can jump and $false$ if there is not.

The theory for this function is described in Section 6.3.2.

B.3.1 Code

```
bool Enemy::Block_Within_Jumprange(void)
{
    // check what direction the enemy is moveing
    if(xv>0)
    {
        float alpha=atan(hoppkraft/xv);
        float V_0=hoppkraft/sin(alpha);
        int dx=V_0*V_0*sin(2*alpha)/gravity;
        int distance=x+width+dx;

        // check every terrain block
        for(int i=0;i<Terrain_Size[GROUND];i++)
```

```
        {
            if(Collision_Test2(x+width, y+height,
                distance, y+height+100,
                Ground[GROUND][i]->x,
                Ground[GROUND][i]->y,
                Ground[GROUND][i]->x+Ground[GROUND][i]->width,
                Ground[GROUND][i]->y+Ground[GROUND][i]->height))
            {
            return(true);
            }
        }
    }

    if(xv<0)
    {
        float alpha=atan(hoppkraft/-xv);
        float V_0=hoppkraft/sin(alpha);
        int dx=V_0*V_0*sin(2*alpha)/gravity;
        int distance=x-dx;

        // check every terrain block
        for(int i=0;i<Terrain_Size[GROUND];i++)
        {
            if(Collision_Test2(distance, y+height,
                x, y+height+100,
                Ground[GROUND][i]->x,
                Ground[GROUND][i]->y,
                Ground[GROUND][i]->x+Ground[GROUND][i]->width,
                Ground[GROUND][i]->y+Ground[GROUND][i]->height))
            {
            return(true);
            }
        }
    }
    return(false);
}
```

B.4 Measure gap width

This function measures the width of the gorge in front of the enemy. It simply does a check one pixel at a time on the area in front of the enemy and sees if there is a collision with any terrain block. If their is a collision it returns the distance from the enemy to the block. If there is no collision it returns zero to indicate false.

The theory for this function is described in Section 6.3.3.

B.4.1 Code

```
int Enemy::Measure_Gap_Width(void)
{
    if(Direction==FACEING_RIGHT)
    {
        float alpha=atan(hoppkraft/xv);
        float V_0=hoppkraft/sin(alpha);
        int dx=V_0*V_0*sin(2*alpha)/gravity;

        // check the ground one pixel at a time
        for(int i=0;i<dx;i++)
        {
            // check every block
            for(int j=0;j<Terrain_Size[GROUND];j++)
            {
                if(Collision_Test2(x+width+i, y+height,
                    x+width+1+i, y+height+10,
                    Ground[GROUND][j]->x,
                    Ground[GROUND][j]->y,
                    Ground[GROUND][j]->x+Ground[GROUND][j]->width,
                    Ground[GROUND][j]->y+Ground[GROUND][j]->height))
                {
                    // return the width of the gap
                    return(i);
                }
            }
        }
```

```
    }

    if(Direction==FACEING_LEFT)
    {
        float alpha=atan(hoppkraft/-xv);
        float V_0=hoppkraft/sin(alpha);
        int dx=V_0*V_0*sin(2*alpha)/gravity;

        // check the ground one pixel at a time
        for(int i=0;i<dx;i++)
        {
            // check every block
            for(int j=0;j<Terrain_Size[GROUND];j++)
            {
                if(Collision_Test2(x-2-i, y+height,
                    x-1-i, y+height+10,
                    Ground[GROUND][j]->x,
                    Ground[GROUND][j]->y,
                    Ground[GROUND][j]->x+Ground[GROUND][j]->width,
                    Ground[GROUND][j]->y+Ground[GROUND][j]->height))
                {
                    // return the width of the gap
                    return(i);
                }
            }
        }
    }
    return(0);
}
```

B.5 Free jump area

This function checks a large area in front of the enemy to see if there is any blocks within that area. The possition of the area is right infront of the enemy and the size the jumpheight and jumplent of enemy, which is calculated from velosoty and jumpforce. If there is no collosion with any terrain block the function retuns *true*.

The theory for this function is described in Section 6.4.1.

B.5.1 Code

```
bool Enemy::Free_Jump_Area(void)
{
    if(xv>0)
    {
        float alpha=atan(hoppkraft/xv);
        float V_0=hoppkraft/sin(alpha);
        int hopphojd=V_0*V_0*sin(alpha)*sin(alpha)/(2*gravity);
        int dx=(V_0*V_0*sin(2*alpha)/gravity);

        // check the large jump area
        for(int i=0;i<Terrain_Size[GROUND];i++)
        {
            // is there a colission with any block
            if(Collision_Test2(x, y-hopphojd,
                x+width+dx, y+height,
                Ground[GROUND][i]->x,
                Ground[GROUND][i]->y,
                Ground[GROUND][i]->x+Ground[GROUND][i]->width,
                Ground[GROUND][i]->y+Ground[GROUND][i]->height))
            {
                return(false);
            }
        }
    }
    else
    {
        float alpha=atan(hoppkraft/-xv);
        float V_0=hoppkraft/sin(alpha);
        int hopphojd=V_0*V_0*sin(alpha)*sin(alpha)/(2*gravity);
        int dx=(V_0*V_0*sin(2*alpha)/gravity);

        // check the large jump area
        for(int i=0;i<Terrain_Size[GROUND];i++)
        {
            // is there a colission with any block
            if(Collision_Test2(x-dx, y-hopphojd,
                x+width, y+height,
                Ground[GROUND][i]->x,
                Ground[GROUND][i]->y,
```

```
                Ground[GROUND][i]->x+Ground[GROUND][i]->width,
                Ground[GROUND][i]->y+Ground[GROUND][i]->height))
            {
                return(false);
            }
        }
    }
    return(true);
}
```

B.6 Free jump trajectory

This functions checks the predicted trajectory the enemy will take when it jump. The function is a more detailed version of *Free_Jump_Area*() since it checks the actual path the enemy will take in the jump. The functions main block i a loop but instead of increasing a counter it changes the possition of area to be checked.

The theory for this function is described in Section 6.4.2.

B.6.1 Code

```
bool Enemy::Free_Jump_Trajectory(void)
{
    int xt=x;
    int yt=y;
    int xvt=xv;
    int yvt=-hoppkraft;

    // while yt is smaller then y
    while(yt<=y)
    {
        for(int i=0;i<Terrain_Size[GROUND];i++)
        {
            if(Collision_Test2(xt, yt,
                xt+width, yt+height,
                Ground[GROUND][i]->x,
                Ground[GROUND][i]->y,
                Ground[GROUND][i]->x+Ground[GROUND][i]->width,
```

```
                Ground[GROUND][i]->y+Ground[GROUND][i]->height))
            {
            return(false);
            }
        }
        xt+=xvt;
        yt+=yvt;
        yvt+=gravity;
    }
    return(true);
}
```

B.7 Block in path

This function checks the path infront of the enemy AI agent to see if there is a terrain block that is blocking its path. It is simpley a collision detect, at halv the distande the enemy can jump horizontally, infront of the enemy.

The theory for this function is described in Section 6.5.1.

B.7.1 Code

```
bool Enemy::Block_In_Path(void)
{
    if(Direction==FACEING_RIGHT&&xv>0)
    {
        float alpha=atan(hoppkraft/xv);
        float V_0=hoppkraft/sin(alpha);
        int dx=(V_0*V_0*sin(2*alpha)/gravity)/2;
        for(int i=0;i<Terrain_Size[GROUND];i++)
        {
            if(Collision_Test2(x+width+dx-xv, y,
                x+width+dx, y+height,
                Ground[GROUND][i]->x,
                Ground[GROUND][i]->y,
                Ground[GROUND][i]->x+Ground[GROUND][i]->width,
                Ground[GROUND][i]->y+Ground[GROUND][i]->height))
            {
                return(true);
```

```
            }
        }
    }
    else
    {
        float alpha=atan(hoppkraft/-xv);
        float V_0=hoppkraft/sin(alpha);
        int dx=(V_0*V_0*sin(2*alpha)/gravity)/2;
        for(int i=0;i<Terrain_Size[GROUND];i++)
        {
            if(Collision_Test2(x-dx, y,
                x-dx-xv, y+height,
                Ground[GROUND][i]->x,
                Ground[GROUND][i]->y,
                Ground[GROUND][i]->x+Ground[GROUND][i]->width,
                Ground[GROUND][i]->y+Ground[GROUND][i]->height))
            {
                return(true);
            }
        }
    }
    return(false);
}
```

B.8 Obstacle low enough

This function checks if the obstacle in front of the enemy AI agent is low enough for the enemy to jump over. The height of the area is equal to the height of the enemy of if it returns true is also means that it is no terrain object in the air blocking the path the enemy have to take over the obstacle.

This function covers two techniques at ones unlike the rest of the functions that only have one assignment. The theory for the techniques are described in Sections 6.5.2 and 6.5.3.

B.8.1 Code

```
bool Enemy::Obstackle_Low_Enought(void)
{
    if(xv>0)
    {
        float alpha=atan(hoppkraft/xv);
        float V_0=hoppkraft/sin(alpha);
        int hopphojd=(V_0*V_0*sin(alpha)*sin(alpha))/(2*gravity);
        int dx=(V_0*V_0*sin(2*alpha)/gravity)/2;
        for(int i=0;i<=hopphojd;i++)
        {
            bool flag=false;
            for(int j=0;j<Terrain_Size[GROUND];j++)
            {
                if(Collision_Test2(x+width+dx-xv, y-i,
                    x+width+dx, y+height-i,
                    Ground[GROUND][j]->x,
                    Ground[GROUND][j]->y,
                    Ground[GROUND][j]->x+Ground[GROUND][j]->width,
                    Ground[GROUND][j]->y+Ground[GROUND][j]->height))
                {
                    flag=true;
                }
            }
                if(flag==false)
                {
                    return(true);
                }
            }
        }
        else
        {
            float alpha=atan(hoppkraft/-xv);
            float V_0=hoppkraft/sin(alpha);
            int hopphojd=(V_0*V_0*sin(alpha)*sin(alpha))/(2*gravity);
            int dx=(V_0*V_0*sin(2*alpha)/gravity)/2;
            for(int i=0;i<=hopphojd;i++)
            {
                bool flag=false;
                for(int j=0;j<Terrain_Size[GROUND];j++)
                {
                    if(Collision_Test2(x-dx,y-i,x-dx-xv,y+height-i,
```

```
                        Ground[GROUND][j]->x,
                        Ground[GROUND][j]->y,
                        Ground[GROUND][j]->x+Ground[GROUND][j]->width,
                        Ground[GROUND][j]->y+Ground[GROUND][j]->height))
                    {
                        flag=true;
                    }
                }
                if(flag==false)
                {
                    return(true);
                }
            }
        }
    return(false);
}
```

Index

#

2D, 1, 35–37, 42, 134, 137

3D, 1, 35–37, 42, 45, 133, 134

A

A-life, 66

Abstraction, 26

Achievements, 126

Action adventure, 48

Active Talker, 18

Adventure games, 47

Agent, 10

AI, 9

- deffinition of, 9
- game, 61, 134
- strong, 11
- weak, 11

AI agent, 10, 18

AI winter, 20

Alan Turing, 13

Algorithm

- Bresenham's, 146
- pathfinding with A*, 76, 77

ANALOGY, 18

API, 56

Application Programming Interface, 56

Arcade games, 32, 38

Arcade hall, 33

Arcage games, 33

Aristotle, 15

Artificial Intelligence, 1

Artificial intelligence, 9

Artificial life, 66

B

Baldur's Gate, 46

Bayesian filtering, 22

Beat'em up, 51

Bitmap, 40

Blackthorne, 82

Blocks, 42

Board game, 36

Bombjack, 32

Boolean algebra, 15, 70

Bresenham's algorithm, 86, 146

Brute-force, 91

C

Cartridge, 33
Castlevania: Aria of Sorrow, 3
Castlevania: Symphony of the Night, 44
Category, 43
CD, 33
Cell phone, 129, 130
Collision detect, 96
Compact disc, 33
Computer game, 32, 34, 38
Computer vision, 23
Concept, 26
Conclusions, 133
Console, 32, 33, 129
Console games, 33, 38
Control theory, 17
Counter Strike, 60

D

Dartmouth Collage, 17
Dartmouth Workshop, 18
Deep blue, 28
Defender, 32
Deffinition of AI, 9
Defuzzification, 73
    formula, 74
DENDRAL, 19
Diablo 2, 47
DirectX, 56, 57
Disposition, 5
Donkey Kong, 32
DooM^3, 45
Dreamcast, 34
Dungeon Sedge, 37
Dungeons & Dragons, 46

E

Efficiency
    line of sight, 89, 145
Enemy, 39
Euclid's algorithm, 15
Evaluation, 125
Event, 109
Experience level, 46
Experience points, 46
Expert systems, 62

F

Fallout, 46
Fantasy, 46
Fighting games, 51
Finite state machine, 67, 111
First person perspective, 37
First Person Shooter, 44

First-order logic, 16
Flocking, 67
Forgotten Realms, 46
FPS, 44
fps, 41
Frame, 41
Frame rate, 41, 59, 128
Free sight, 85, 134
Fussy logic, 70, 139
Fussy sets, 71
Fuzziness, 71
Fuzzy set theory, 71

G

Galaxian, 32
Game AI, 21, 61, 62, 134
Game cartridge, 33
Game console, 44
Game Consoles, 33
Game engine, 38, 59
Game genre, 43
  adventure, 47
    action, 48
    point and click, 48
  beat'em up, 51
  fighting games, 51
  first person shooter, 44
  massive multiplayer online, 54
  Massive Multiplayer Online Role Playing Games, 55
  MMOG, 54
  MMORPG, 55
  platform, 43
  puzzle, 55
  racing game, 53
  role playing games, 45
    hack n' slash, 47
  shoot'em up, 51
  sim, 53
  simulator, 53
  sports, 54
  strategy, 49
    realtime, 50
    turn based, 50
Game platforms, 32
Game programming, 58
Gameboy, 34
Gamecube, 34
Gamepad, 41
Gameplay, 37
Gameplay genre, 48
Gamer, 39
Games, 53
  Baldur's Gate, 46

Blackthorne, 82
Bombjack, 32
Castlevania: Aria of Sorrow, 3
Castlevania: Symphony of the Night, 44
Counter Strike, 60
Defender, 32
Diablo 2, 47
Donkey Kong, 32
DooM^3, 45
Dungeon Sedge, 37
Fallout, 46
Galaxian, 32
Half Life 2, 45
Halflife, 60
Halo 2, 45
Lylat Wars, 52
Mario^{64}, 37
Masters of Orion 3, 51
Pac-Man, 32, 68
Pitfall!, 35, 36
Sim Ant, 54
Sim Copter, 54
Sim Earth, 54
Sim Farm, 54
Sim Life, 54
Sim Park, 54
Sim Town, 54
Snake Rattle Roll, 37
Space Invaders, 32, 52
Star Fox, 52
Starwing, 52
Street fighter II, 32
Streets of Sim City, 54
Super Mario Bros, 35
Tetris, 55
The Legend of Zelda, 48
The Secret of Monkey Island^{TM}, 48
The Sims, 54
Ultima Underworld, 37
Wolfenstein 3D, 45
Zelda II: The Adventure of Link, 48
Genres, 43
George Boolean, 15
Goal, 5
Graph, 75, 114

H

Hack n' slash, 47
Half Life 2, 45
Halflife, 60
Halo 2, 45
Health, 39, 71
Heuristic, 11

Heuristic distance, 75
History, 17
Hitpoints, 39, 71

I

Image recognition, 4, 23, 96, 125, 127, 130, 134, 135, 138
Inference engine, 64
Input, 41
Introduction, 1
Introduction to game programming, 31

J

Joystick, 41

K

Kernel, 39
Keyboard, 41
Knowledge database, 63

L

Lagging, 42, 127
Landscape, 41
Level, 41, 46
Level editor, 59
Limitation, 3
Line of sight, 4, 81, 125–127, 129, 133, 135, 137, 145
    efficiency, 89, 145
    statistics, 91
Link-state routing, 128
Logic
    first-order, 16
Lylat Wars, 52

M

Main event loop, 58
Mainstream AI, 21, 62
Map, 41, 128
Mario^{64}, 37
Massive Multiplayer Online games, 54
Massive Multiplayer Online Role Playing Games, 55
Masters of Orion 3, 51
Mathematics, 16
MMOG, 54
MMORPG, 55
Mod, 59
Modders, 59
Modding, 59
Modifications, 59
Motivation, 2
Mouse, 41
MYCIN, 19

N

N-Gage, 129
Neuropsychology, 16
New techniques, 4
Nintendo, 34, 35
Nintendo^{64}, 33, 44
Node, 75, 111, 114, 128
Nokia, 129
Non Playing Character, 48
NPC, 48

O

Object recognition, 23
Observer, 127
Open Graphics Library, 57
OpenGL, 57
Operating system, 39

P

Pac-Man, 32, 68
Palm pilot, 129, 130
Pathfinding, 5, 77, 111, 113, 125, 128, 130, 134, 136, 139
Pathfinding with A*, 61, 114, 128
    algorithm, 76
Perception, 23
Philosophy, 15
Pitfall!, 35, 36
Pixel, 23
Plans for Future work, 137
Platform game, 43, 137
Player, 39
Playstation, 33, 44
Point and Click, 48
Polygon, 36
Production rules, 19, 63, 64
Production systems, 62
Programming games, 58
Purpose, 1
Puzzle games, 55

R

R1, 20
Racing game, 53
Raster graphics, 23, 25, 40
Realtime Strategy games, 50
Realtime systems, 16
Riots, 67
Robotics, 22
Role Playing Games, 45
Routing, 136
RPG, 45, 47, 138
RTS, 50
Running through walls problem, 106

S

SAINT, 18
Science fiction, 46
SDL, 57
Security agent, 23
SEGA, 34
Self reference
    see Self reference
Shoot'em up, 51, 52
SHRDLU, 18
Sim Ant, 54
Sim Copter, 54
Sim Earth, 54
Sim Farm, 54
Sim games, 53
Sim Life, 54
Sim Park, 54
Sim Town, 54
Simple DirectMedia Layer, 57
Simulator games, 53
Skill, 46, 138
Skillroll, 46, 138
Smoothing algorithm, 25
Snake Rattle Roll, 37
Software, 34
Sony, 44
Sony Playstation, 33
Space Invaders, 32, 52
Spam, 22
Sports games, 54
Sprite, 40
Spyware, 35
Stage, 41
Star Fox, 52
Starwing, 52
Strategy games, 49, 53
Streep Figther II, 32
Streets of Sim City, 54
Strong AI, 11
STUDENT, 18
Summary
    evaluation, 130
    image recognition of the level, 112
    introductino to AI, 29
    introduction to game AI, 78
    introduction to game programming, 60
    line of sight, 93
    pathfinding, 118
Super Mario Bros, 35

T

Terminology, 38
    bitmap, 40

blocks, 42

computer game, 38

enemy, 39

frame, 41

game engine, 38

hitpoints, 39

input, 41

lagging, 42

level, 41

player, 39

sprite, 40

stage, 41

Terrain objects, 42

Tetris, 55

Text based games, 47

The Legend of Zelda, 48

The Secret of Monkey Island™, 48

The Sims, 54, 110

The Turing test, 13–15

Third party manufacturer, 34

Third person perspective, 37

Thread, 59

Trigger, 109, 127

Trojan, 35

Turn based strategy games, 50

TV-spel, 32, 33

TV-spelskonsoll, 33

U

Ultima Underworld, 37

V

Video game, 33

Virus, 22, 35

Visual limit, 83

W

Weak AI, 11

Wolfenstein 3D, 45

X

X-Box, 33, 34, 57

Z

Zelda II: The Adventure of Link, 48