Semantic navigation in video games

MSc thesis

L.M. van Driel

October 2008

Committee:

Prof.dr.ir. F.W. Jansen

Dr.ir. A.R. Bidarra

M.H. Kruithof, MSc

Dr. A.H.J. Oomes

Abstract

The exhibition of intelligent or challenging behavior by artificially controlled characters (AI) in computer games strongly depends on their navigational performance, often referred to as pathfinding. While recent games solve increasingly complex and specific navigation problems, the documented approaches to navigation remain primitive and isolated within the gaming software realm. We propose a semantic approach to navigation that allows AI to extract and incorporate a wide variety of navigation information into the planning of character behavior, including raw geometry, strategic objects, and abilities for locomotion. We design and implement a navigation system for Unreal Engine 3 that requires little designer intervention, has a rich interface for AI programmers, and can be extended with other types of semantic information. This subsystem proves to outperform the original navigation functionality by delivering more natural paths, fewer world annotation, and dynamic re-planning.

Acknowledgments

The MSc project, of which this thesis takes a final part, spans from March until October 2008. During this time and prior research, I have received a lot of encouragement and support, for which I would like to thank the following people.

First of all, my thanks go to Rafael Bidarra, for his forthright supervision and personal encouragement.

It has been a real pleasure working on my thesis project at Coded Illusions. My thanks and appreciation to the programmers and game designers, for all their feedback and advice. Special thanks to Thijs Kruithof, for his unconditional involvement in my progress and the project as a whole.

Thanks to my beloved housemates. I really appreciated their curiosity. Special thanks to Erik for his refreshing comments and to Bram for his design assistance.

Finally, I would like to thank Rinske, for her persistent interest in my work and all her loving support.

Leonard van Driel, October 15th 2008

Contents

Chapter 1

Introduction

1.1 Motivation

Software for video games is characterized by fast and incremental development. Because this development is mostly driven by eye candy and catchy game play, the applied techniques are evaluated by the visual beauty of their results and not their inner workings [7]. Using this approach, the game industry has produced many fantastic games and gained a professional allure, budgetary dominating the movie industry.

These successful developments have not been evenly distributed over the existing video game development areas. The ongoing trend of enhancing visual realism has released large budgets by both the consumer and the producer. Looking at the games of today with the eyes of last year, most popular video games show astonishing graphical performance.

On the contrary, there are relatively few innovative applications of artificial intelligence (AI) techniques in these video games, and we have seen only little improvement over the last decades. A small fraction of video games, including Creatures [20], Half-Life [49], and Black & White [36], have shown significant sophistication when compared to games at that time. Still, most video games today provide game challenges by means of quantity, e.g. many simplistic opponents at once, instead of quality, like the tactical sophistication found in real-life human behavior.

Here we see similarities with the development of academic AI, which, like game AI, has experienced a turbulent development over the past two decades; big leaps followed by quiet periods have made it into an unpredictable area, where development budgets where hard to justify [4]. While other game development areas pushed rapidly forward, using the most state of the art technologies, game AI was said to be at least 25 years behind the academic level of development.

The graphical focus of the gaming industry was predicted to shift to other fields, like artificial intelligence [11, 29, 37]. The rich set of academic AI techniques available would finally be used in practice and video games would be valued by their AI performance instead of their eye candy. This prediction was substantiated by the continuing hardware developments, stagnating graphical improvements, and gamer's expectations about upcoming games.

Advancing AI

Although the substantiating factors have emerged, the resulting game AI advancement remains modest. Instead, game developers prefer the illusion of intelligence with predefined scripted behaviors and very predictable and limited AI techniques. These do provide more control, allowing the designer to prescribe character behavior, instead of mastering and restricting a creative and complex AI.

The important lesson learned, was the fact that a complex AI does not necessarily make the game more fun to play [15]. A game like Half-Life, which was applauded for its tactical AI performance, is said to have used a very basic AI [32]. The team tactics, which were applied by the opposing soldiers in Half-Life, were based on just a few easily evaluated but well devised rules, which provided very challenging emergent behaviors. Still, these successes will not directly carry over and contribute to future games.

Academic AI can be fun, if it can be tamed by designers in a very open and flexible way. We must understand that this requires a more scientific approach to how a system can behave intelligently, and what structure is needed for such a system. There can no longer be one big AI, but instead a set of layers or modules, which are exchangeable and operate independently. We expect AI systems to improve, not by combining even more fancy AI techniques, but instead by designing modular AI systems that demonstrate their value over time. This way, developers will be in control over these systems, while the game AI provides better tactics, more immersion, and thereby better game play.

Semantic navigation

Navigation plays a central role at the heart of many games' AI. Most navigation tasks are rudimentary and undeviating, still they involve a considerable amount of computation and require a fail-safe design. Faster, more robust, and more diverse navigational performance is an essential support at the basis of game AI.

The importance of good navigation has led to many techniques in the area of pathfinding, motion planning, and path data extraction. Navigational performance, however, lies not only in the individual techniques applied, but also in the combination of techniques and the role of information. The weakness of existing systems is the absence of this broad internal consistency, which can be issued to the lack of semantics.

This report will study this problem and focus on the design of a navigation system based on the fundamental idea of semantic structure, in our aim to contribute a robust and extensible improvement to game AI in general.

1.2 Goals

To express our aim in a more explicit way, and to allow us to substantiate and validate our work, we set the following goals within the scope of video games:

1. To understand the importance of semantics in a navigation system for video games and to find ways to classify and represent this semantics.

2. To identify the actors, either users or systems, that interact with a navigation system.
3. To develop a semantic approach leading to the design of a navigation system that respects the requirements of these actors.
4. To evaluate such a design by means of an implementation.

In order to gain a better understanding of semantics in navigation, we issue the following questions:

- What do we understand of navigation and how is this currently provided in games?
- Which semantics is relevant to navigation in video games and how can we classify it?
- How can semantics be represented and applied in a navigation system?

We will revisit these goals in Section 7.2.

1.3 Outline

This chapter introduced the subject of navigation and semantics and the goals that form the basis for our project. The remainder of this report is structured as follows.

Chapter 2 covers general navigation in video games. It introduces the area of video games and game AI, and focuses on how navigation in video games proceeds in general. Next, Chapter 3 covers the semantics of navigation in video games. It presents our view on semantics and a categorized overview of the semantics that is relevant to our project

In Chapter 4, we take a software engineering approach to semantics and navigation. Here we introduce the main requirements for the semantic navigation system that is built during this project. Chapter 5 continues with a semantics-oriented design based on these requirements. Here we present the main structure of our system and the central semantic classes. The implementation of this design is discussed in Chapter 6, where we cover its architecture and the techniques that were applied.

We discuss our navigation approach, and the research in its totality in Chapter 7, where we return to the system requirements and the project goals, after which we elaborate on remaining and related problems. Finally we offer our conclusions in Chapter 8.

Chapter 2

Navigation

Navigation, is the act of *planning* and *directing* the course of a vessel or body through space over time. While real-life navigation is about assisting humans in finding their way around by providing a small amount of more or less implicit information, navigation in a game environment, also referred to as *pathfinding,* has many complications in different areas. This is mainly because low-level information about the game environment is ubiquitous, while global and local motion planning is far from obvious.

This chapter covers the current state of arts in games and game development, specifically in AI and navigation. We build up starting the first two sections with a short overview of video games in general and the game genres related to our project. Section 2.3 further focuses on the navigation module as part of the game's AI, followed by an overview of navigation systems that are currently used in the gaming industry.

2.1 Video games

The area of artificial intelligence in all its diversity has adopted a vast vocabulary over the past decades. Especially around navigation topics like pathfinding, action planning, and motion control, since these areas were the first to be investigated in depth. In this section we will discuss the basic nomenclature by means of an introduction to the topics that will be discussed in the upcoming chapters.

Games and gamers

A *video game* is a piece of software that offers typical game-like challenges for entertainment and training uses. The person who controls the machine that runs the game or a part of it, and thereby influences the course of the game, is called a *gamer*. A *player* is a person or system that controls a part of the game, for example a gamer, but this role can also be governed by an artificially controlled player.

A term that is often and widely used when attributing games is *game play*. Although this term has a very broad and sometimes ambiguous meaning, it refers to the experiences received by the gamer while playing a game. Game play is, among many other things, about the man-machine interaction, the amount of

control, the flow of events, and the immersion in the game. *Immersion* is the state of being deeply engaged and involved in the course of the game, perceiving as if it were real.

Development

Games are made by *game developers*. A game developers team typically consists of *game designers*, *level designers*, *artists*, and *programmers*. Game designers develop the game as a whole, level designers create individual game parts, and artists provide detailed content. Programmers create the tools and functionality, which the designers use to create the game.

To get an idea of the amount of work that goes into the creation of full size modern computer games; it takes more than 500.000 lines of program code, many gigabytes of artistic and geometric content, and a development period of 2-4 years. In order for the development process to complete in only a few short development cycles, it is very important to avoid reinventing the wheel and instead building up a strong base of extensible development tools that are available for reuse.

The combining tool in this is the *game engine*, which joins all models, geometry and dynamics into a game. A game engine's purposes include the coordination of the flow of events in the game, managing of storage, in- and output, controlling the movement of characters, and the computation of optimal paths.

Game elements

The game, as it is perceived by the gamer, is the projection of the *game environment* onto the screen. This game environment contains every virtual element that we find in a game. It is the environment in which the gamer is immersed. Depending on the game or game genre, we categorize the elements in the environment based on their function in the game.

The elements that make up the physical world are called the *geometry* or base map. While theoretically every element has some physical representation and is therefore part of the geometry, in practice the geometry consists of the floors to walk on, the walls that make up rooms and mountains that one should walk over or around.

Characters are game elements with some special function or ability, often designed to exert some kind of personality, intelligence, or free will. Depending on what parts of the game are controlled by human players, we distinguish two types of characters, namely *human characters*, which derive high-level decisions from human input; and *AI characters*, which are fully automated on every level including decision making. Note that on some levels, all characters are subjected to automated control, for example in the detailed movement of arms and legs. Some literature also refers to human character*s* as *player characters* and AI characters as *non-player characters* (NPC). 'Player characters' should not be confused with 'players', where the former are in-game characters, while the latter stand outside the game.

Figure 2.1: Screen shot from 'TA Spring', a complex strategy game, featuring a diversity of units (characters) and terrain (geometry). Navigation routes thousands of units through a terrain with gradually changing topology. (Image taken from spring.clansy.com.)

2.2 Genres

Our project focuses on games in the *action-adventure genre*, one of the main genres of games we find today. This section covers three of the most related genres from the perspective of navigation, namely strategy, action, and adventure, to sketch the context of our area and make our project scope more clear [11, 29, 48].

One of today's popular video game genres is *strategy*, which includes games that require strategic planning and careful commanding in order to be victorious. In most strategy games, the player controls an army of troops with various abilities by issuing military orders like 'move to position', 'guard object', and 'shoot at first sight'. Strategy games pose interesting navigational problems that involve evading pathfinding, approach axis, strategic formation, and group movement.

Strategy games typically feature hundreds of units, which individually manage local navigation, while moving collectively on a larger scale. Since strategy games are often played using a wide top-down view and units are positioned on a fixed grid, fast and dynamic pathfinding is of critical importance, while detailed motion planning can be dealt with in a procedural way. As the name of this genre suggests, strategy determines the outcome of the game. This also involves navigation functionality for terrain and formation analysis.

Another popular genre with a very different game play is the *action* genre. Action games are characterized by game play with emphasis on individual exploration and combat. Action games feature a three dimensional environment with objects of different sizes that accommodate the richness of the combat, like tactical positioning behind walls and moving objects like wooden crates.

Figure 2.2: Screen shot from 'Half-Life 2', an extensive action shooter game. Navigation provides tactical pathfinding; traversing cover positions, avoiding confrontations, picking up objects, and the use of vehicles. (Image taken from www.gamespot.com.)

Figure 2.3: Screen shot from 'Assassin's Creed', an action-adventure game with versatile controls and beautiful graphics. (Image taken from assassinscreed.uk.ubi.com.)

The player controls one character or a few related characters and directs them through the world, while dynamically coordinating the detailed interaction with the environment and other game characters. Action games often impose a strong feeling of immersion on the player by displaying the world from the perspective of the character under control.

Typical for action games is navigation in 3D space, well timed and positioned actions, and up to ten units at a time. Actual pathfinding has a much less dominant role than in strategy games. It is mostly the well timed and detailed motion planning, like cover finding and weapon collecting, that makes behavior in combat situations realistic. Action games, like first person shooters, however tend to cover up behavior mistakes through short time and long range combat.

While action games have a large emphasis on the handling of dynamics, like fast exploration and one-on-one combats, *adventure games* tend to focus on problem solving. They cast the player as the protagonist of a story, normally

requiring the player to cooperate with other characters, in order to solve a puzzle. With less combat and more joint exploration, navigation in adventure games further focus on the details of motion planning and correct interaction with the environment.

In each of these three genres, navigation is crucial to AI and therefore of our interest, however each one for a different reason. In strategy games, reasoning about the environment is of strategic importance and has a strong influence on paths and plans, but the scale and dimension of the environment do not fit the environments we deal with in our project. Action games do fit our scope, but have a strong tendency towards combat, while there are many other acts to our interest besides fighting. Adventure games on the contrary are much more diverse and feature many different acts in a much richer environment. The dynamics, on the other hand, are often not as time critical as in action games.

In our project, we primarily focus on action-adventure games; games that feature elements from both the action and the adventure genre. In case games from other genres are issued, this will be explicitly mentioned. Although navigation in strategy games is not our primary focus, the larger unit count and more procedural navigation shows how techniques can be made more robust and scaled to larger counts.

2.3 Game AI

Game AI covers the areas related to character and world behavior. It is referred to as artificial intelligence because it often strives for the simulation of intelligent behavior of game characters. Techniques in this area however have proven to be more widely applicable, for example to collective animal behavior and adaptive story telling.

In this section, we look at game AI from the perspective of action-adventure games. Given the variety of AI systems within this genre, we present the general layout of these systems while focusing on navigation. This layout is based on experience with the systems presented in Section 2.4 and the work of Mesdaghi et al. [35] and Yue et al. [57].

Higher AI

To understand the positioning of navigation inside a game's AI, we take a look at the functional parts of a game's AI that are involved in or related to the act of navigating and identify those that eventually make up the navigation system. A schematic representation is shown in Figure 2.4.

The motivations of a character's AI to navigate are only interesting from a navigational perspective. The higher intentions the AI might have are not relevant, thereby avoiding a circular dependency where the modules try to reason about each other's functioning. The parts that are occupied with these higher intentions, like decisions of strategic and tactical nature, are referred to as the *higher AI*. The main task of a higher AI is to make a character act, based on its perceptions, internal state, and knowledge about the world.

The collective AI in strategy games is mostly concerned with strategic decisions that are expressed by just a few basic behaviors, while a character AI in action games has almost no strategy, but survives on a complicated scheme

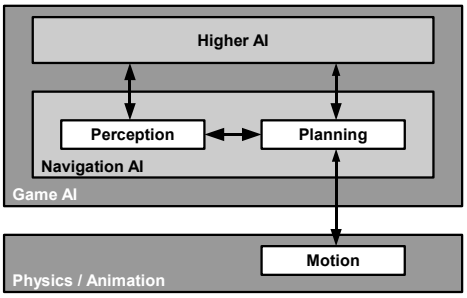

Figure 2.4: Typical architecture of the navigation AI inside the game AI. Arrows indicate flow of instructions and feedback.

of behavior types that prescribes how these are combined and applied. In both situations, we see that navigation skills are required from the moment that the character needs to be guided through the world. Guiding a character is basically telling it when and where to perform what action based on a set of requirements set by the higher AI.

Planning

Successful guidance always requires a way of looking ahead and planning the required steps. Throughout the game AI, we see different kinds of planning problems; from strategic decisions, to the way the arms and legs of the character move; using different kinds of approaches to solve, because these problems strongly differ in nature. Because of the latter, we will only consider those planning problems that are related to navigation. Solving these planning problems typically involves a series of methods depending on the level of detail. At the top, we find the topological planner, which searches a path based on the connectivity of space, for example the rooms that should be crossed. At the bottom, we have motion planning involving body movement and collision avoidance. Other types are considered to be dealt with by the higher AI.

One might argue that on top of the topological planning problem there are navigation-related planning problems, like how to plan a series of paths that involve a fight or another small plot. In fact, one might argue that all actions in the path from the begin of the game to the end, are navigation-related in nature. This is however the task of the higher AI, for the navigation AI should only be able to plan a path based on the physical state and abilities of the character, without the higher intentions and strategic considerations. Guiding the character does not mean that it has to make decisions it is not capable of making or allowed to understand.

If a higher AI is only state-based, there is no higher planning capability present. In such a case, we need a planner that is able to sequence paths and basic actions, like opening a door. If these actions are sufficiently local, we can also plan them using the topological planner .

Motion

The guiding role of a navigation system is limited, both at a high level, where the role of planning is held by the higher AI, and at a low level, where the navigation system guides the motor skills of the character in effectuating the plan. Like at the high level, there is no unique way to distinct between navigation and motion control at the low level. Daily life navigation tools typically give directional information, which is interpreted by a human at a high level. The guiding role of a navigation system often reaches much further, where problems are still solved using typical navigational search methods.

Nevertheless, at a certain level, a search is not required anymore and only basic laws, like those for velocity, rotation, and collision, govern the motion of a character. These laws are often run by physics and animation systems. The deterministic nature of the systems allows the navigation system to well understand their behavior and create detailed plans for their actions. Though there often is a certain amount of uncertainty in these systems, which makes it simply impossible to provide error free plans. Therefore, they should provide accurate feedback on their performance, and preferably deal with their instructions in a slightly intelligent way.

A typical example of this occurs when a character is guided around a corner and its shoulder slightly hits the edge. This can happen for many reasons, like the accumulation of small rounding errors, a slight unknown offset of the geometry, or poor physics simulation due to a low frame rate. As the shoulder hits the edge, a corresponding collision event bluntly stops the character from moving. Here, collision systems are very strict and unable to distinguish insignificant collisions, from relevant ones. If the navigation AI is notified immediately with accurate collision information, it can quickly anticipate and continue the original path.

Perception

For its planning tasks, the navigation system has to be well aware of the environment features that matter to navigation. While this information is mainly used by the system itself, we see a more wider application of this information in modern game AI. This role of perception can be described as providing the higher AI with a view on the environment through the eyes of the navigation system.

Instead of making decisions based only on ray-casts and level annotations, the higher AI also uses the navigation system to make decisions. Here, instructions reach the navigation system, but do not propagate to motion control. A detailed plan is returned to the higher AI, which enables it to predict the outcome of its actions.

2.4 Navigation in middleware

The modularity of an AI framework allows for the development of software that provides general AI techniques like navigation. The increasing complexity and standardizing of AI in games makes the use of AI middleware an interesting option. This section looks at five commercial AI frameworks, ranging from standard Unreal Engine 3 pathfinding to the hardware accelerated AIseek.

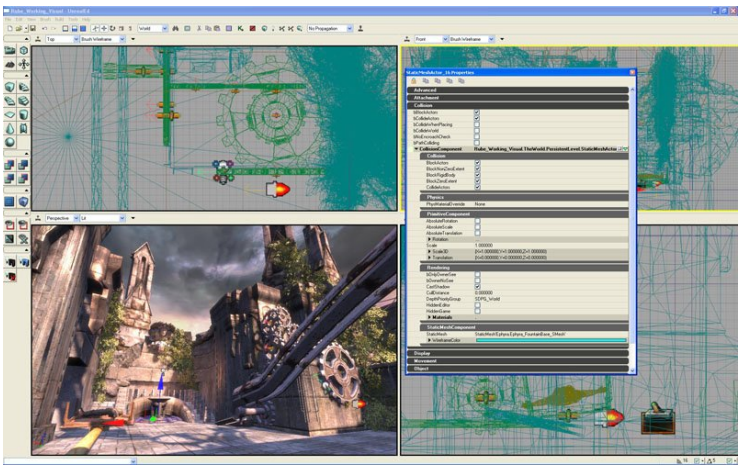

Figure 2.5: Screenshot of UnrealEd; the central game design tool of UE3. (Image taken from www.unrealtechnology.com.)

Unreal Engine

Unreal Engine [16], currently at the third version (UE3), is one of the most popular game engines for action and adventure games. UE3 is a game development framework for the third generation of gaming consoles and equivalent personal computer. It offers a collection of core technologies and content creation tools for rendering, models, physics, animation, audio, game scripting, user interfaces, and AI.

Navigation in UE3 mainly considers the movement of human-like characters along key locations that have been marked by the level designer. Using a so-called waypoint graph, where nodes resemble these locations, a character traverses from node to node, following fixed edges to move inbetween two fixed locations in the environment.

The advantage of Unreal's waypoint approach is its simplicity, which makes it easy to comprehend and apply. Unfortunately, because Unreal has a strong focus on design tools and graphical performance, AI development is stalling. The waypoint approach lacks many features, including dynamic editing and updating, customization to character abilities, and a good representation of space. These shortcomings will be further discussed in 5.1.

Kynapse

Kynapse [46] is middleware with a focus on navigation. Kynapse is specialized in navigation for large crowds and individual combat. Among its features are path planning, path smoothing, dynamic avoidance, hierarchical 3D pathfinding, automatic 3D perception, pathfinding data generation, run-time identification of key topological places for hiding, surrounding, and organizing opposite flank assault.

Kynogon, the developer of the Kynapse AI engine, does not provide detailed information about the inner workings of its pathfinding functions. The online documentation gives some insight on the global workings of the AI engine and

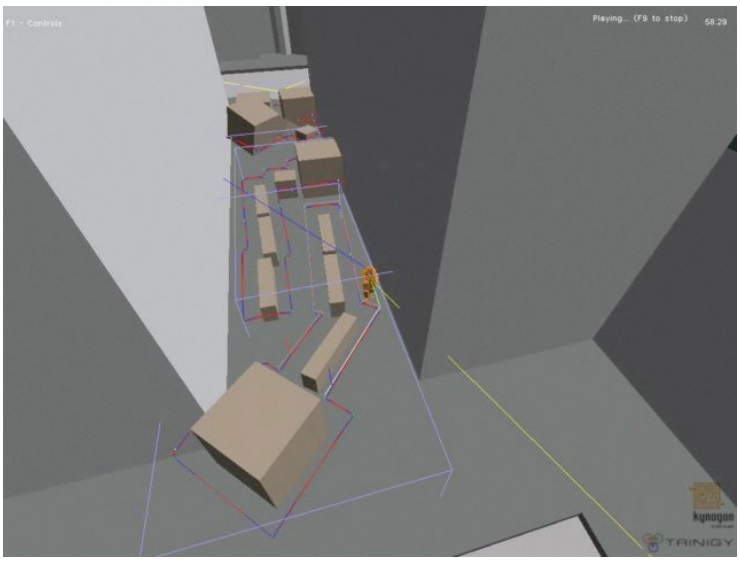

Figure 2.6: Screenshot of an environment with a character controlled by Kynapse. This images shows dynamic object bounds which Kynapse uses to steer the character around emerging objects. (Image taken from www.kynogon.com.)

the integration of pathfinding functionality, which is summarized in this section.

The AI world contains Characters, Actions, Brains, Agents, Services, and Teams.

- *Character* can be any object in the real world that is controlled by the AI engine. The synchronization of the world and the AI engine is done each frame.
- *Actions* represent basic movements or *behaviors* that can be performed by characters. During run-time, Actions are translated by the AI engine into *steering* instructions for the game engine.
- The *Brain* hosts the thinking logic of Characters. Based on perceptions and its own thinking logic, the Brain sends an Action to the Character under control. In its decision making, the Brain consults appropriate Services.
- *Agents* provide complex *behaviors* by combining Actions, therefore simplifying the brain programming. The Brain consults an Agent, which returns an action proposal, e.g. the Brain requests fleeing behavior and the Agent returns a specific move Action.
- *Services* handle all kinds of data storage, perception requests and actions. The Graph and PathWay data manager are Services that are used by the Brain to perform *pathfinding*.
- A *Team* is a special Character with the purpose of simplifying team behaviors, communications, and joint information storage.

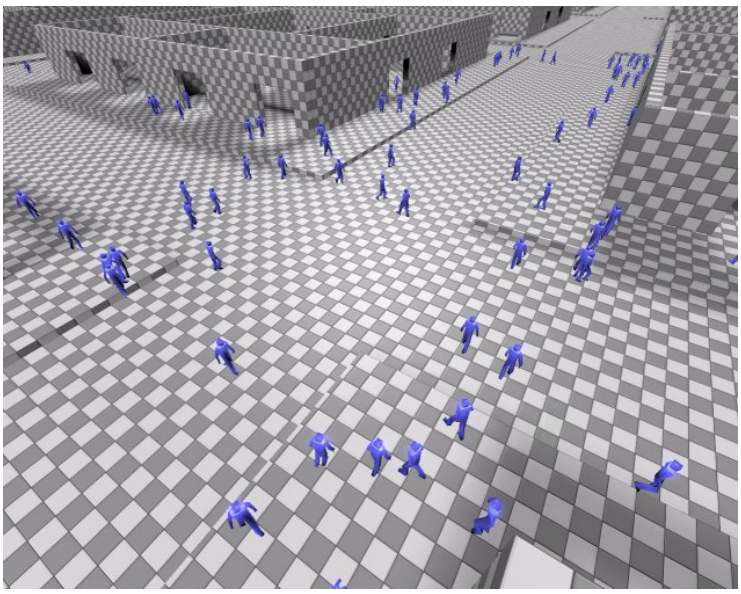

Figure 2.7: AI.implant's crowd simulation. Image shows people in a typical city environment; walking on the sidewalk and crossing the street, while avoiding bumping into one another. (Image taken from www.ai-implant.com.)

One of the core components of Kynapse is pathfinding. The pathfinder searches a spatial graph using the A* algorithm, based on the constraints and heuristics given. The resulting path can be furtive, danger avoiding or as short as possible. Heuristics are estimates for the length of the optimal path to the target, where the cost function contains the criteria distance, time, and more.

During the construction of the path, the PathObject provides edge costs based on the state of dynamic objects. During traversal, the PathObject can modify the path in order to bypass nodes, e.g. a dynamic object that blocks the path, and give the required actions.

AI.implant

Middleware provider AI.implant [1], which is currently under Pregasis Inc., describes itself as "a production-proven Artificial Intelligence software and SDK for the training and simulation market". AI.implant includes tools to create human and vehicular behaviors and inserts these in an existing simulation.

The main features of AI.implant are hierarchical and dynamic pathfinding, obstacle avoidance, and path variation. Although these techniques are presented in a general way, the demonstrated implementations show an emphasis on the dynamic simulation of large crowds. Therefore it is mainly used for movies that feature massive battles and animal flocks.

AI.implant only offers product features and no description of their workings or details about the framework. Game developers that have used AI.implant praise AI.implant for the deep integration into the Unreal engine and the Dynamic Path Refinement technology which is far better optimized than standard pathfinder support by UE3.

AIseek

As specific game functionality matures by becoming stable through different games, it can be separated into general frameworks or even a dedicated piece of hardware. An example of the latter was the successful introduction of the first 2D/3D cards in 1995. AIseek [2] announced in 2005 the introduction of the Intia Processor, a special AI processing unit (AIPU). This chip, with its central Graph Processing Core, is said to accelerate lower-level AI performance in a number of areas.

- The Intia processor provides very fast and optimal graph search, i.e. finding the shortest path between two nodes in a graph. As A* is blamed for its suboptimal pathfinding and path artifacts, the brute force approach taken by AIseek reputedly performs between 100 and 200 times faster than regular A* implementations.
- It is able to determine visibility of 512 characters against each other within 0,02 seconds. Therefor, preprocessing is not required any more, allowing all visibility tests to be performed at run-time.
- It is specialized for running the typical algorithms to analyze the environment terrain features, to identify several types of strategic positions for use in combat situations.

The shiny claims made by AIseek have not yet been supported by more detailed specifications let alone a consumer product. While physics simulation hardware is still struggling for popularity, the AIPU is received as yet another dedicated processor.

Chapter 3

Semantics

This chapter discusses the role of semantics in action-adventure video games. It covers a general comprehension of semantics, which semantics we find in games, how this can be categorized, and in what way semantics can further improve AI performance.

3.1 Role of semantics

The role and interpretation of semantics is generally understood in different ways. It is therefore important to carefully introduce the word 'semantics' and relate it to the approach taken in this report. To start off, we look at the following example, which illustrates a rather basic but fundamental semantic investigation of an in-game situation.

Imagine an in-game situation where you are being chased through a narrow corridor by an angry aggressive guard. While running for your life you enter a small room with a wooden crate. Inventive as you are, you quickly push the crate in front of the corridor opening; a simple block that might buy you some time to come up with a solid escape plan (Figure 3.1).

However, it turns out that this simple move really did the trick in the race against your vicious rival. Instead of pushing the crate aside, jumping over it,

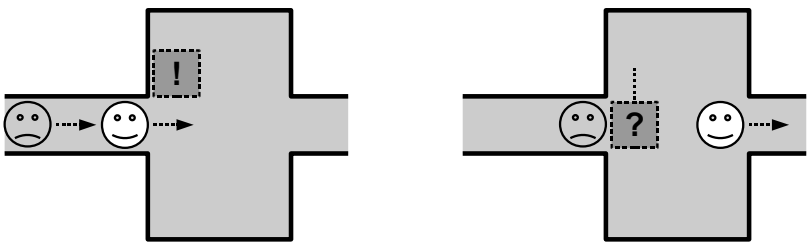

Figure 3.1: Chased by guard through a narrow corridor ending in a room. The guard fails the chase due to a lack of understanding wooden crates.

or even shooting it to shreds, the guard is completely disoriented. Inside its tiny brain, A* pathfinding is desperately looking for alternative paths, but none is popping up. The guard realizes it is still in some kind of adrenaline flooded phase and therefore decides to find cover, but where to find cover in a narrow corridor? Maybe walk back? No, that brings it too far from the prey. Maybe throw a grenade? No, that can only be done from a covered position. Maybe just a random move? No, lets just stay put and see what happens.

One can argue that playing a game is playing by its rules. In this example, the player will feel victorious at first, having captivated this brainless monster. But wasn't it supposed to be a guard, a human soldier with years of training? The illusion of challenge is starting to faint, and so is the immersion. This AI clearly was not ready to live inside the head of a human being.

Note that there are games on the market with AI that does handle these situations well; there is, however, still a majority of recently released games that gain more results from restricting designers in their freedom to place wooden crates around, than from designing an AI that has the abilities and creativity of a human player. If the AI had known why and how to push the crate aside, or to climb over, or to shoot it, the crate would have had meaning to it, the crate would have spoken more than only the words 'thou shall not pass'.

This knowledge and interpretation leads to the central question in our semantic study: Which information matters and in which way? In our context, 'semantics' does not only refer to *the study of meaning*, as in the area of linguistics, but also to *meaningful information*, addressing both the information and its meaning. An explicit definition of information or meaning is generally case-dependent, for example of a game or an AI system. In general, information refers to the states or value of memory and meaning refers to what implications this information has on processes; how the information is processed or applied.

Meaning is often associated with something to which it means. Although it is arguable whether semantics requires information to mean something to someone, we will assume that semantics is always relevant to a thing that processes or applies the information. We assume semantics to be primarily *interpreted* by a character, or more specifically, in the scope of our project by a navigation AI. Since the navigation AI consists of smaller systems and abstract concepts, interpretation always propagates further down through the navigation AI itself.

3.2 Semantics approaches

In this section, we briefly look at previous work on semantics in general and in relation to navigation.

In research conducted prior to this project [10], we introduce a general approach to semantics based on *semantic units*. The information or data inside a navigation AI is composed from smaller structures, which up to some level will be referred to as semantic units. A semantic unit can represent an ability of the character, a feature of the environment, how to fit the ability to the feature, and what this means to the state of the character in terms of for example health, position, time, or inventory.

The report further describes which specific semantics is likely to exist in a navigation system. We introduce a general representation of a path based on

metrics, a pawn based on abilities, and the use of graph search as a conservative but well fit choice to solve navigation problems while preserving sufficient freedom of design.

The main purpose of the semantic unit is to stretch out through the navigation AI, thereby relating perception, planning, and motion. This notion will be further discussed in Section 4.2 in the context of so called atomic semantics and language.

The work of Kallmann [24] introduces the notion of *smart objects*, where the interaction between a character and some object is described by eight different types of interaction features. These types provide a basic language for the object to express all its meaning to the interacting character, including physical properties, gestures, behaviors, and planning features. The smart objects approach was implemented in the successful series of 'The Sims' games [13].

Navigation is scarcely issued in this work of Kallmann, for it is only about interaction with objects, not about how to get to them, or interaction with the environment as a whole. Also semantics is not directly issued, but implicitly addressed in the eight types of interaction features. We see different objects providing information, communicating with the character, and the interpretation of objects by the character. Kallmann's language is rich enough to express how to interact with the object, also in concurrency with other characters.

Another point that is issued in this article, is the problem of generality. There are many levels on which semantics can be specified, some of which being more general and therefore more adaptable. These, however, are in turn harder to specify and implement, and do require more computation. Here, the latter is one of the major issues in games, and one should be aware of the marginal gain in generality, when providing more of both design effort and computation time. Kallmann addresses this based on the modularity of the model; when more generality is needed, split the object into smaller, more basic and independent objects.

A discussion of semantics in the context of navigation can be found in an article by Martínez and Mata [34], who propose a semantic model that provides a layer of semantic information for virtual environments. This layer does not intend to substitute application-dependent approaches, but to constitute a common lower level for all of them. It provides ten semantic elements, including object type, navigation network, minimal paths, and various object properties and relations. The layered semantic structure does provide a much clearer view on the available semantics and allows different applications to use the same common level semantics. However, the generality of these semantic types, and the lack of more explicit relations, makes general applicability questionable.

In an article by Tutenel et al. [54], three levels of semantic specification are identified: *object semantics*, which are physical and functional properties; *object relations*, e.g. by class, inclusion, proximity, coordination; and *world semantics*, like time, weather, and climate. The article further presents how constraint solving techniques are used to maintain these object relations, and how semantics can be applied during a design phase and at run-time. Their approach strongly depends on constraints management and solving techniques. Although the authors have good reasons to believe that such an approach will eventually be applicable in games, the current state of constraint solving techniques is still

too specific and computation intensive to be generally applied to navigation in games.

The notion of a local object space and a global world space can be found in many approaches to semantic structure. This observation is discussed in an article by Russell [44] where he argues that many systems define space only in terms of a single location or the environment as a whole. He shows that there is also a need for representing space larger than a single location and smaller than the environment, which he refers to as *places*.

He criticizes current AI architectures for reducing the spatial environment to a list of objects at specific points, and for using observer and task-independent features to describe these objects. In short, he issues that places are socially constructed, action-oriented, layered both concurrently and hierarchically, and can provide a vocabulary for reasoning about space. Instead of the typical sequence of global 'go there', and local 'do this' behavior, places represent a space to both 'go' and 'do'. Russell further discusses a number of existing techniques that apply this theory of places, mostly based on the expansion of the local object to a region.

3.3 Types

Research in semantics generally faces the great challenge of dealing with the complexity of the real world. Because games generally aim at providing a very realistic or detailed simulation of a similar or even greater complexity, general semantics of games also face this challenge.

Our scope does offer strong limitations to, for example, space, characters, and types of interaction. This should be beneficial to the practical successes of our semantic approach, given the fact that most successful applications are those tailored to the game or genre at hand. This trade-off is the general argument for not trying to deal with general game semantics.

Our approach to the semantics of navigation is therefore based on a structured set of typical in-game situations with a direct relation to navigation. Four main types of semantics will be addressed; geometry, objects, characters, and dynamics. These types do *not* represent mutual exclusive categories of semantics. Though, each type requires a specific way of looking at the environment in order to identify the relevant semantics.

3.3.1 Geometry

Of all things in the game environment that matter to a navigating character, geometry is the most dominant. We define *geometry* as the collection of physical bodies that make up the environment as we see and feel it. In practice, geometry is often defined by meshes with a texture and collision data. The main abstract feature of geometry is that it gives strong directions to paths of light and moving objects. We identify three sub-types of geometry: vision and visibility, cover, locomotion, and space.

Interestingly, the historic focus on basic pathfinding techniques to solve the problem of geometric navigation has remained until today. The reasons for geometric navigation are still present, i.e. walls didn't just disappear, and still dominate the typical search space. The restrictive power of geometry is however

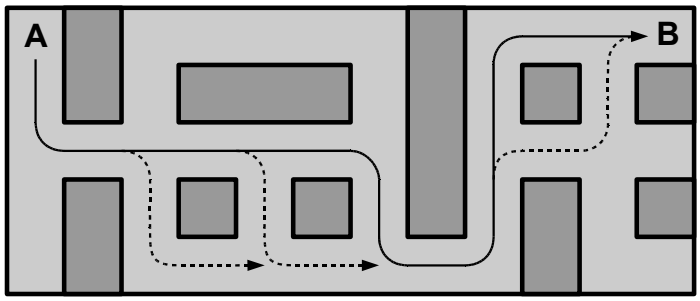

Figure 3.2: Schematic display of basic navigation. 'A' and 'B' respectively indicate the start and end point, inbetween which many feasible paths exist. The walkable floor has a light shade of gray. Non-traversable areas, e.g. walls, have a dark shade. The solid line indicate one shortest path. The dashed lines indicate alternative sub-paths.

moving to the background, because it exists only at the lowest levels of interpretation. Higher-level interpretations of geometry now appear, posing new challenges within these basic restrictions.

The remainder of this report contains a number of schematic drawings to support the discussion at hand. Figure 3.2 provides an introduction to the visual vocabulary by presenting a very basic navigation act.

Vision and visibility

One of these higher-level interpretations is the processing of visual input. Vision is the dominant type of perception in humans, especially for players of modern graphics-oriented games involving combat, shooting, or cover taking. In general, vision is the ability to perceive the environment through incoming light rays; visibility is the property of an object given a character's viewpoint, indicating that the eyes of the character register the object. In practice, the proper evaluation of visibility involves many aspects of human vision, material properties, and lighting.

Because vision is so dominant, the meaning visibility provides can be very diverse for both human and artificial human-like characters. Visibility can trigger a character to initiate action or to constantly modify its plans. At the same time, a character also has a physical body and thereby controls its visibility by intentionally hiding behind geometry or control others' visibility by a chase.

One generic property is the presence of a viewer which has a *viewing point* and additional properties like a viewing frustum (see Figure 3.3). An object is visible from a view point, if sufficient visible rays exist between the viewpoint and some point on the surface of the object. Here, 'sufficient' can have different meanings in both quality and quantity, depending on both the viewer and the object in view. In human vision, the viewer has only a limited resolution in the collection of rays and therefore requires sufficient density and angular spread of the visibility rays in the viewing frustum. This is often modeled by a fixed viewing distance. An additional property of human vision is that the ray resolution becomes higher when the object sends out different colors or light intensities

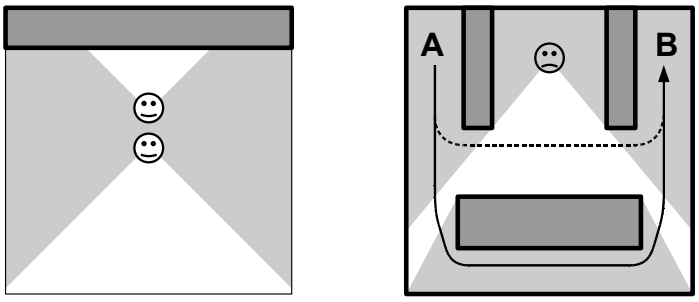

Figure 3.3: Left: A character's vision depends on many factors including the amount of space it is perceiving. The upper character looks straight into a wall, rendering bad vision. The lower character has a good overview of its surroundings and thereby good vision. Right: In order to minimize visibility, it can be wise to take a small detour when moving from A to B.

over time, e.g. when the object is in motion.

A player character controls its vision mostly by movement, head orientation, and the use of tools like night vision goggles. These are however fairly complicated factors that most AI developers avoid by providing ubiquitous environment information, combined with a good filter of ignorance to preserve the illusion that the AI is really using its eyes.

The control of visibility however is something much harder to cheat with, e.g. an AI character can't just make itself disappear to its likings. Only in cases, such as character spawning, the frustum of player characters is explicitly avoided. In most cases, visibility is a matter of hiding behind objects, moving by crouching, or wearing a camouflaged suite. If the game offers dynamic lighting, the character can also influence its lighting properties like brightness, color, and position; it can turn off the light, produce an overwhelming amount of it, or plan its path through less illuminated areas.

Cover

The concept of light for visibility can be extended to fast traveling projectiles like bullets or laser beams. Because the motion of these objects can be assumed to be on a straight line, their path can be modeled with a visibility ray. The meaning of transparency changes to pierceable and the viewer would be a shooting device.

A difference with visibility here, is that being visible is often not that much of a problem compared to being exposed to gun fire. With the viewpoint of a gun being slightly displaced from the eyes, shoot and cover offer a delicate interplay between viewing the enemy while not being shot, or shooting the enemy before being seen. Good cover positions are therefore both critical and hard to identify, especially in a dynamic weapon fight. A tactical AI character should be well aware of both cover and visibility to offer opposition against player characters.

The strong tendency of the game industry towards combat simulation games has fueled the terrain analysis research around character positioning, aiming, and cover seeking [30, 31, 50]. Most of the AI systems developed so far apply

very basic and computationally fast heuristics to estimate optimal positioning that have proven to be very effective.

Liden presents strategies to evaluate visibility of regions in a square 2D environment with opaque walls [30, 31]. This strategy is based on a number of waypoints that are distributed over the square according to the distribution of viewers. The visibility of a waypoint is defined as the number of other waypoints that are visible from this point. Liden identifies good sniper character positions as high visibility regions that are next to low visibility regions.

The Half-Life [49] combat AI, which uses Liden's approach and was described by the gaming community as intelligent and well opposing, has some major drawbacks when tested more thoroughly. Most problematic were the facts that this model does not dynamically adapt itself during the game and this model does not differentiate between viewing directions of characters. These problems have been addressed by Sterren [50]. Sterren takes on the waypoint-based approach, incorporating directional information and dynamic updating, to find optimal sniper locations. This approach first identifies a number of abstract features in the scene like connectivity, directional visibility, reachability, overlook, cover and focus. By combining these features using specific weights, a good sniping location is defined.

Locomotion

Characters in games can have all kinds of *abilities* that allow them to move through space in many different ways. The *locomotion* of a character embodies these abilities by defining the way geometry influences movement. We identify two types of locomotion; movement over uniform geometry and movement through different types of geometry.

In a uniform part of space, one type of locomotion is required and applied in a uniform way. A basic example of uniform space is a flat horizontal surface. This type of geometry has been very popular in games, because a uniform locomotion type requires only basic animation and no dynamic adaptation to the geometry. Even when the surface is slightly tilted and walking up the slope will look different than walking down, the plane remains uniform.

The main motivation to consider semantics of locomotion is the transition from one type of geometry to another. Navigation on a uniform surface does not need any local information about the surface, for it is uniform. However, when a character needs to follow a shortest path that involves different types of geometry, local semantics play a role in both the animation and the navigation of the character, involving a careful timing and adaption to both uniform spaces.

Imagine for example an environment with two horizontal planes which are separated by a small chasm (Figure 3.4), and a human character standing on the first plane with the goal to navigate to a position on the second plane. If the character is not aware of the chasm, it will either be unable to navigate to the second plane, or attempt to just walk to the goal position and fall into the chasm. If the character is aware of the chasm, it will figure that, given its size, this chasm requires a perpendicular jump to be crossed. The character will then walk in the direction of the goal, turn perpendicular to the chasm when it is sufficiently approached, jump, land and walk towards the goal again. Other examples of special geometry are elevators, rims, rails, holes, ladders, and open water which all can be combined into one path.

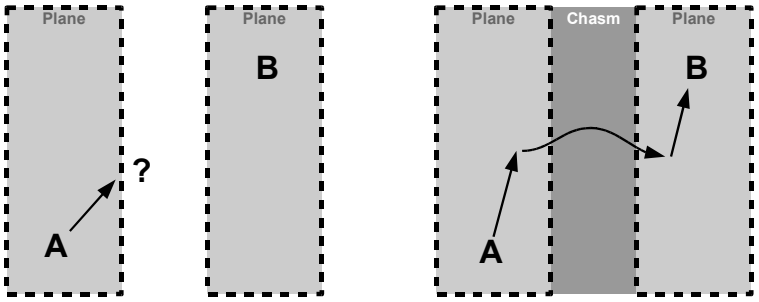

Figure 3.4: Two horizontal planes, which are separated by a small chasm. By making the character aware of this chasm, we allow the character to traverse the chasm in a similar way as the planes.

Space

Navigation is as much about the meaning of moving around as it is about the meaning of the spaces you move from, to, or through. A *space* is a subset of the world space, bound by geometry like walls and floors, and character-related based on for example visibility and tactics.

An important notion of space is the identification of rooms in buildings or areas in the open. Firstly because AI players are often limited in their navigation space, which often means that they have to stay inside a certain room or field. Secondly, we see that events propagate within a room, but not outside it, e.g. the presence of an enemy in a room makes it as a whole particularly dangerous; an explosion or fire inside a room fills its space; sounds often stay inside a room.

A second notion is *place*; a space that is socially constructed or action-oriented. Places do not need a direct relation to geometry. Instead it can be indirectly defined by geometry through for example visibility, cover, or lookout. Places are also often used by designers to limit AI behavior, or to control scripted action, for example to indicate a safe spot to land a helicopter. Another use is the definition of an influence region; to indicate the presence of a special smaller place or object, for example around an opponents' current path, to indicate high chance of being detected.

The most limited indication of space is a single *location*. Locations were the most used indicators of space, because they give very precise control. They leave however very little freedom to interpretation by the AI, for example when the AI has to walk a path from point to point, and at the same time act realistically. Therefore locations are increasingly less attractive indications of space.

Spaces are relevant to navigation in a number of ways. They can be a goal or a departure position; which is the quickest way to get into or leave the space. A space can change the character state of the AI, e.g. the chance of receiving damage when inside the space. It can also influence the character its abilities; e.g. running faster or being able to make a special jump.

Ways of dealing with space and places in particular can be found in the area of *terrain reasoning*, which is an approach originating from real-time strategy games [41]. Using basic techniques like pathfinding and influence maps, the meaning of objects or locations is translated into places. These techniques

are mostly concerned with the uniform evaluation of the terrain and to derive semantics by summing over multiple evaluation functions. They work well where the environment is relatively homogeneous and with properties that can be unified by summation.

Navigation in more maze-like environments, like office buildings or caves, also involves the connectivity of space and the spaces that can be derived from connectivity. A nice example of this is the derivation of *pinch points* and *ambush points* [14, 31, 56]. A pinch point is space that connects two rooms and is relatively narrow. For example a doorway or corridor that connects two rooms. Because enemies are certain to pass the pinch point when going from one room to another, the pinch point would be a perfect spot to block their path or to place a booby trap. The ambush point is an area close to a pinch point, but not visible from the pinch point. Ambush points are ideal positions to initiate a surprise attack.

3.3.2 Objects

An *object* is a part of the environment that has additional meaning, for example through annotation. The meaning of an object to a character and its path is not necessarily restricted by uniform laws like the laws of physics. This allows objects to be used in games for almost everything that has meaning in a special location for some purpose. We identify four types of objects: Supplies, Triggers, Equipment, and Movable geometry.

Supplies

Some of the most frequently used objects we find in games, are those that provide some kind of supply like food, health, or ammunition. In most games, the meaning of a supply object to AI is restricted to its primary function; supplying the character. In these cases, when a player interacts with the object, its health, or ammunition level increases. The meaning of a supply can be reduced to a fixed threshold, by which the AI will decide to change its objective to move to the closest supply location.

Most human players prefer not to wait until the moment that a supply becomes a main goal. Instead, they constantly monitor their levels and bias their path towards the supplies accordingly. In contrast to levels and bias, which have a floating scale, supplies are discrete and will only be gained when crossed. This demands a discrete path planning approach which decides to plan a route via a supply or to ignore the supply completely (see Figure 3.5).

When a character is in serious needs for a supply, it needs to find the closest supply of that type available. This type of search is similar to the search for the closest cover position, though cover can be given by an area, while supply is often more position based.

Triggers

Trigger objects change the state of the environment when interacted with. Instead of supply objects, trigger objects serve critical tasks in the path under search. The relation of a trigger with the state change it causes, demands for a relationship that must be understood by the character.

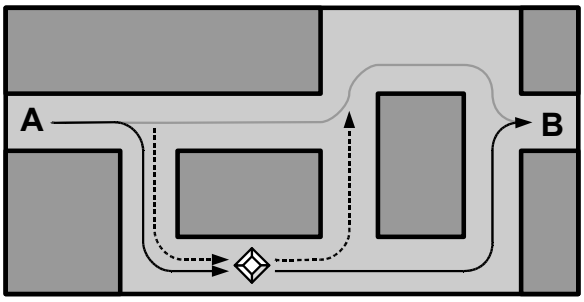

Figure 3.5: When moving from A to B, it might be preferable to detour via a supply object (diamond). One option is to first search for the shortest path from A to B (gray arrow) and then insert the shortest detour into the path (dashed arrow). A more optimal solution would be to traverse the shortest path to the supply, followed by the shortest path to B (solid arrow).

A construction we often find in games is the combination of a door and a button or key to open it. For a pathfinder, this means that the door does not block the way, provided that we visit the button before we try to pass the door. In practice, the door induces a small delay, but it can also be a dangerous route, because all properties of the path to the button, apply also to this door.

Because the triggered state change can involve practically everything, it is hard to find a general meaning for triggers other than the fact that a character needs to be able to trigger it by e.g. passing, pressing, or shooting. Triggers often do change the environment sufficiently to incorporate them into navigation. The consequence of the trigger however is better suited for understanding by the higher AI, for example in tactical planning.

Equipment

Some objects supply a new ability or special extension for a character, rather than a pair of spare bullets. By picking up this special supply, which we will refer to as *equipment*, the character changes, gaining new abilities that influence its navigation. Equipment is like triggers in a sense that it can change navigation completely by coming close to it. Equipment however stays with the character and therefore has a much more persistent consequence for future actions.

Equipment can be very diverse; it can be a weapon, allowing more effective shooting in combat situations; a door key, which like the previous door trigger can open a specific door, but only by the owner (see Figure 3.6); a flash light for better vision; a communication device to signal other characters; even a portable trigger, like a key; or a vehicle to assist with locomotion.

Like triggers, it is hard to describe a general meaning for them, forcing the understanding of equipment to be processed by the higher AI. Related to this is the fact that in most games, AI only uses basic equipment like weapons, while state is only influenced by supplies.

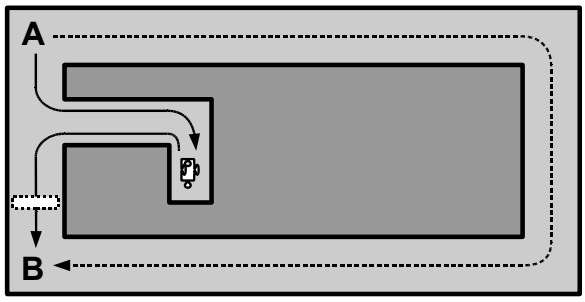

Figure 3.6: To move from A to B, one can either take a long walk, or go for the key that opens the door (white rectangle). The latter requires one to realize that the door is relevant to the path and the key opens the door.

Movable geometry

Movable geometry includes objects that represent a part of the world geometry and can be moved by a character as one object. We classify movable geometry as an object because dynamic elements in games are as a utility to perform specialized actions, like the picking up and throwing of an oil drum.

With the upcoming of games with destructible environments, movable geometry is starting to play an increasing role in navigation. Its meaning can quickly change back and forth between just some piece of dynamic geometry, to specialized equipment. Again, for most geometry holds that it requires a very specific and case-dependent meaning definition, which is more likely to exist within the higher AI. The fact, however, that it is geometry and able to move, makes paths around it less safe and predictable over time.

3.3.3 Characters

Characters are objects that have the ability to perceive the environment and act based on these perceptions. Characters are often the most challenging elements in the environment because of their diverse physical and intellectual abilities.

Self

Characters are defined by their abilities, state, and physical properties. *Abilities* are the means of a character that allow it to change its own state and the environment. A character *state* includes position, speed, orientation, and the state of its supplies and equipment. Physical properties include the dimensions and weight of a character.

The meaning of properties to the character itself has already been largely addressed in the contexts of visibility, locomotion, equipment and movable geometry. Abilities primarily have to be used in order to reach the navigation goal, keeping in mind properties, like size to avoid getting stuck. In the mean time, the state changes implied by a certain path have to be monitored to avoid for example running through a fire without having much health left.

Others

Given that characters can act upon other characters' acting and characters can perform any sort of reasoning, the cause of some action is hard to derive and therefore characters behavior is only partially predictable.

In the cases that a character is controlled by a human player, there is no way of knowing for sure what reasoning goes on. Assuming the player has spent some time in the game, its behavior should have converged to a pattern that results in survival and success. To an AI, this behavior can be very complex, hard to understand, and above all hard to predict. A solution could be to match the behavior to a set of behavior types of which the AI knows how to act upon. Laird[28] applies intentional models in game AI to predict the movement of a human player with respect to supplies.

The interpretation of another character is largely depending on how this character changes the environment and other characters. From our point of view, the meaning of a character to our navigation depends on the abilities of the character to change our local environment and character state, on the intentions of the character, and on the reactions of the character when the environment changes.

In the following sections, we briefly look at two typical ways of multi-character navigation, namely cooperation and opposition.

Cooperation

A group of *cooperative* characters behaves in a coordinated way, optimizing group-specific behavior or the sum of the successes of the individual characters. In order to achieve cooperation, some kind of alliance is required, where some characters are prepared to make a sacrifice to assist others in achieving a joint objective. This cooperation contract can be broken when other characters are not convinced of others alliance. From a navigational point of view, the joint objective should be some navigational goal.

The question is what other characters can do, to assist you in moving around. This always involves sharing information or changing the environment to each other's advantage. The case of sharing information is not particularly interesting, because we assume that a navigating AI character has access to all available knowledge. Changing the environment however can be done in many ways, depending on the abilities of both characters to do so.

In a combat situation, the optimal path is influenced by the position of enemies and the chances of being hit. An ally could change these to your advantage when delivering suppressive fire. In that case, the ally forces the enemy to temporarily draw back by shooting at them. Chances of being hit shortly decrease which allows you to move towards your goal position. Do not forget to offer your ally the same suppressive fire in order to preserve the cooperation.

Other combat tactics are to agree that each ally takes another route to confuse the enemy or to take the same route in order to avoid being seen. The first tactic requires all characters to find their individual path, while avoiding clotting together. Characters have to weigh out between the length of the path and the mutual distances. When jointly traveling the same route to deliver minimal disturbance, the allies have to agree on taking the same path or on

following one group leader. In this case, the allies or the leader have to weigh out path length and capacity.

A daily life example of navigational cooperation is flocking, which basically means that you follow a character that is traveling in the same direction as you are. When traveling through a crowd where each character has a different destination, flocking delivers far better results than just ignoring the presence of others and constantly bumping into each other.

When a crowd of people is moving in the same direction, collective flocking might lead to huge congestion, when the capacity of the path decreases. If many characters plan on taking the same path, it might be a good idea to traverse a longer path which is less congested, provided that the travel time should be minimized.

Opposition

An *opposing* character has the objective of preventing you from achieving yours, or is prepared to exploit or hinder you in achieving his, or vice verse. When confronted with such an opponent or enemy, the question is how to avoid being exploited or hindered, assuming that you are pursuing a navigational goal.

One way is to just avoid confrontation as much as possible. In combat, path cover is crucial to completely avoid confrontation. By planning a path with the most total cover or the largest least cover, one can avoid being seen at all. When the enemy is aware of your presence, cover can still prevent you from being hit, but some suppressive fire can be of great assistance. When no survivable path exists, other actions have to be taken first, like making a distraction maneuver or taking out the enemy. When there is no enemy present, it might still be a good idea to avoid certain areas that might be used by the enemy in an ambush.

3.3.4 Dynamics

Time is a degree of freedom in the navigational search space which is often not involved or even mentioned in practice. The dominant reason to omit this dimension, apart from the fact that many people forget it is actually there, is the fact that it does not dominate the outcome of a navigational search, as long as the world is sufficiently static.

From a computational point of view, the additional costs of navigation in the temporal dimension are unacceptable for real-time navigation. Therefore, most games avoid navigation in highly dynamic and unpredictable environments and, if necessary, approximate dynamics by a local event-based system. This section covers the semantics of time while keeping in mind the major computational issues.

Compared to the previous categories of semantics, which are fairly independent, semantics of time is orthogonal to the categories mentioned. Like the spatial dimensions, time is an additional degree of freedom in the state of all geometry, objects, and characters. To avoid the recapitulation of each previous example, this section only illustrates the consequences of time.

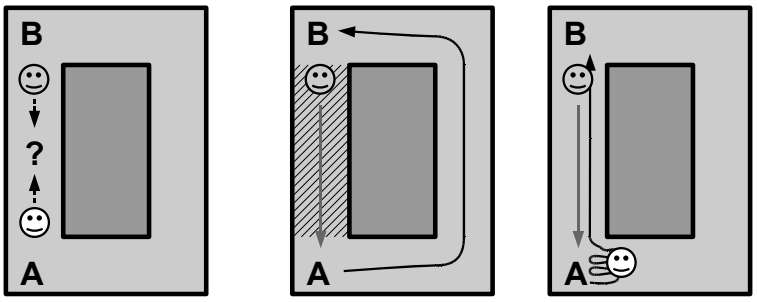

Figure 3.7: Two characters approaching a narrow corridor. To solve this situation, our character (white) must either take another route, or wait for the other character (gray) the pass. In the former case (depicted left), the character only interprets the area covered by the other's path as an impassable area (dashed). In the latter case (depicted right), the character is also aware of the influence of waiting; a better solution in this case.

Include time

The notion of space has similarities with the notion of time. For example to indicate a dangerous place, it is also important to specify the time frame in which this danger is present. Therefore spaces could also be defined as a 4D subset of the space-time continuum. In addition, the presence of objects, paths, and characters can also be restricted to a time frame.

Normally this time component is omitted and the space or object is only 'present' during the time that it exists. Because navigation requires good planning, knowledge about where *and* when features are present can be critical. For example, imagine two AI characters that are in need of ammunition and decide to walk towards a supply. Since one of them will arrive first and take it all, the other will still lack ammunition and probably receive extra damage. In another example, one planned a route across a triggered grenade. In this case, waiting a few seconds for the explosion to happen, might save a lot of health. Last, when two characters approach a narrow corridor from opposite sides, a good solution would be for one character to wait until the other has passed. This situation is illustrated in Figure 3.7.

Discrete time

A popular approach to space-time navigation is to reduce temporal changes into events. Temporal changes in the world are assumed to be short time emergent, and perceived as an event which suddenly modifies space. After the event has passed, the world has changed, but is again static, which allows the spatial navigation to continue. This discrete approach in particular delivers a low sample rate and therefore few computations.

A most basic event-based approach assumes that events cannot be predicted and therefore only need to be applied when they occur. This means navigation is not allowed to account for these events, until they occur and therefore does not need to reason about time or events at all. For example, imagine a character

which has planned a path through a corridor. When a part of the building suddenly collapses, it blocks the character its path and it becomes aware of this as soon as this happens. Now, it has to re-plan its route taking the blockade into account.

In a slightly more advanced approach, events and their consequences are known to happen in the near future at a given time. In this case, navigation can reason about events while planning a path through the initial environment, by book-keeping the elapsed time and switching to a new environment when an event occurs.

Returning to the collapsing corridor, the character will not plan a path through the corridor, because it knows that when it arrives there, the corridor will already be collapsed. This will make the character's navigation much more accurate, but might be perceived as cheating behavior.

Continuous time

If the environment changes just gradually, many small events are required for a suitable approximation of these changes. At some point, the computational burden of the large event structure calls for a continuous time approach. The discrete nature of computers still requires the sampling of space-time over time, however instead of passively waiting for events, navigation now actively predicts the environment over time.

If the changes in the world are too chaotic for prediction or the prediction is too computational intensive, we return to the above mentioned situation where navigation is unaware of future events. Assuming this is not the case, navigation can again reason about future events and book keep the elapsed time.

There is however a significant difference compared to the discrete time approach. Assuming that environmental changes can happen at any speed, sample rates need to vary over space and time. In the collapsing corridor example, navigation through the corridor requires only a low sample rate, until the computed moment that the corridor starts to collapse. Now, at a high sample rate, each piece of debris needs to be modeled to allow the character to plan a sophisticated move like a quick dive and roll to evade the falling bricks and pass the corridor safely at the other end.

Predictability

Clearly time plays a critical role in navigation. In practice one can deal with it quite easily without even defining it explicitly. But from the moment that the future cannot be modeled accurately enough using the present, we need a way to predict the future. Human players still have a significant advantage, having learned all kinds of ways to deal with time in daily life.

For the AI to know the future, it has to simulate it using models of human players and the laws of physics. These are both very hard to learn and to compute. In practice, predictability is limited to tens of seconds in the future. Added to that is the fact that one can never be sure and therefore should use predictions as limitations on already feasible plans.

3.4 Discussion

In this chapter, we have looked at a wide variety of semantics for navigation in games. Because semantics is only limited by the creativity of the game designer and player, this study has only summarized some important semantics in today's action-adventure games. Semantics and games are related to each other in complicated ways and can have overlap with multiple areas like general AI, strategy, planning, and navigation.

When identifying semantics, it is important to classify the interpretation that it embodies. A piece of information is interpreted differently by a designer, programmer, navigation AI, or motion control routine. For example, a path can be visualized to the designer by a curved line describing the expected movement of the traversing character. To a programmer, it means the path to a health pack, because the character's health value passed a predefined threshold. To the navigation AI, it simply means the current path it is traversing, not knowing what the words 'health pack' mean. Finally, to motion control, it is only an array of points through which it has to steer the character.

Designing the behavior of an AI character should proceed in a way that is natural to designers, while the routines that execute the navigation require instructions that are natural to them. We do *not* want to query an instruction using terms like graphs, nodes, curves, and accelerations. Instead, we want to query a safe path to a safe location, having defined safe earlier in terms that were understood by the navigation AI.

The semantics of navigation for low-level routines are normally well developed. Primarily because the source code that defines these routines need to run on a computer, which has led programmers to think the way computers do. In the following chapters, we will therefore refer to semantics to indicate the interpretation by a human or high-level system.

Semantic navigation is easily associated with semantic pathfinding. However, semantics of navigation also includes tactics, behaviors, and detailed physics. When searching a path, tactical information can come to light, which should not be ignored. The same holds for the influence of certain paths on behavior and the detailed movement of the navigating character. Although this information should not be handled by the navigation AI itself, the actively sharing of information with other AI components is crucial.

On the other hand, semantic navigation is restricted to navigation. This strongly restricts the meaning to what is relevant to navigation, excluding higher AI topics like planning and strategy. Many game-related semantics is specific and case-dependent to such a degree that their definition needs to reside in higher AI. For the navigation AI, this semantics is reduced to a rudimentary form like a generic object or space definition.

Our semantic approach should deal with both the broadness of navigation compared to pathfinding and the limitations compared to higher AI. Add to this the fact that we should avoid over-complication, which is generally unwanted in games, while remaining rich enough for the higher AI to be able to express its needs.

Because there is no single set of semantics that provides this richness and limitation, we have to distinguish semantics by how well this fits to certain

games. The semantic types we discussed do provide good views, but do not make semantics comparable in this sense. In this case, a more layered view would be better; having the most fundamental semantics at the bottom, and building up by subdividing this into a set of atomic semantics at the top. This more practical approach will be used in Chapter 5.

Chapter 4

System requirements

One of the main goals of this project is to design a navigation system to demonstrate how navigation and semantics *can* meet each other in practice. In this chapter we present the requirements and specifications that shape our system. Here, we distinguish four distinct groups, namely navigation, semantics, users, and game engine. The former two groups have been introduced in respectively Chapter 2 and 3. The latter two groups are more directly related to the implementation of our system, and are therefore introduced in this chapter.

This project has been supported by Coded Illusions, a game developer working on an action-adventure game using Unreal Engine 3. The requirements presented in this chapter have been established in discussion with developers at Coded Illusions, in order for the navigation system to fit in their line of development.

Note that we will discuss both general navigation systems, and the system that we design following these requirements. When discussing *a* system, we will be talking about a general navigation system, like the system that is already present in Unreal Engine 3, while *our* system refers specifically to the system that is to be designed.

4.1 Navigation

The principal goal of a navigation system is to navigate characters, as to guide them across the environment based on a set of instructions by some higher AI. In this section, we specify requirements for this principal goal, covering perception, guiding, and a set of non-functional requirements.

Perception

Our navigation system has to *provide the higher AI with navigation-related information for decision making.* Given a character, our system should be able to:

- Detect key locations in the environment, like safe locations or pick-ups.
- Predict the existence of a path for this character.
- Estimate travel distance, time, and danger.

- Provide progress information on path search and traversal.

These perceptions are also necessary for the navigation system itself, in order to find and traverse paths. This requirement therefore mainly aims at the interface that is provided to the higher AI to issue this information.

Guiding

Our navigation system has to *assist the higher AI in navigating by guiding a character based on a high-level path and character description.* Here *guiding* means planning motion instructions for the character up to the level where other systems like physics and animation are able to interpret (Section 4.4).

The path is described by:

- A starting location.
- Route properties like: stay covered, maintaining line-of-fire, or just the quickest route.
- The path's end, which can be one or more locations, places, or specific objects or characters.

A character is described by its:

- Physical state: location, orientation.
- Abilities: walking, running, crouching, and jumping. These are further specified in how they apply to the environment and what is required from the state of the character in order to use this ability.

Dynamics and autonomy

Our system should be able to *autonomously interpret a dynamic environment at run-time for the tasks of guiding and perceiving.* This means that:

- The system should determine available routes and actions at run-time.
- The higher AI does not need to be aware of these actions, as long as the given instructions are properly executed.

These requirements are less accurate as they can easily limit the design to the extent that any system would become unfeasible, given the available hardware resources. Depending on the available resources, more effort should be invested in improving the independence of the system, and its run-time performance to deal with dynamic environments.

4.2 Semantics

While fast and memory-efficient pathfinding techniques are essential to navigation in games, it is primarily the richness and applicability of a semantic structure that should embody the power of our navigation system. This requires the system to be able or extensible to include every aspect of navigation.

In this section we present requirements on the language that we use to represent the semantics of our system.

The perceiving and guiding role of the navigation system requires it to communicate to neighboring systems, like the higher AI and motion control. For this, we need a language that can be understood by these neighbors, and used internally by the navigation system itself. This language should be rich enough to serve the intent and allow sufficient disambiguation. Still the language should remain minimalistic to avoid over-specification and too computation intensive interpretation. Additionally the language should allow for extensions in the form of variation on the previously mentioned requirements. We will now assess different perspectives within the game AI to derive requirements for our languages, using the semantics types addressed earlier in Chapter 3.

Our first perspective will be that of the higher AI. This part has the intention to send one or more characters to a specified location taking into account a number of additional demands. For such a request, we need to indicate the set of *characters* that require navigation, followed by a specification of the *path*. These characters are defined by their current *state* and available *abilities*. The path, instead, can be specified by anything between something explicit, like an exact goal location using the shortest path, or implicit, like the safest route out of a dangerous zone. For both types of paths, we need a way to specify the kind of *place* to go, and the *route* to go there.

The higher AI also needs semantics for its strategic and tactical considerations. This includes hypothetical navigation, like the expected outcome of a travel across a dangerous area. This requires the path to contain semantic information about the impact it has on the character. The navigation system also has the knowledge to find safe places, which have to be implicitly defined by the higher AI.

Secondly, we take the animation and physics perspective. In order to render the moving character on the screen, it has to know which basic movements it needs to run, in order to evaluate laws of physics and to render animated moves with that.

The navigation system is our third perspective. One of its tasks is to transform paths from implicit to explicit. Therefore it has to understand implicit path specifications, by planning a correct and detailed motion description for animation and physics to understand. The correctness of a plan is defined by how the higher AI had intended it to be up to where it was specified. In order to plan, it has to understand what a character can do based on its state and ability set.

Apart from the imperative communication discussed above, there is also the need for feedback. When an implicit path or place has no explicit specification, the higher AI needs to be informed. This can be a direct response of the navigation system, or an indirect response originating from the physics system. Another reason for feedback is to be aware of progression. During path traversal, the higher AI might be interested in how much time is left. Because the navigation system is well aware of the surroundings, the higher AI might also be interested in special objects along the way, which the navigation AI does not understand, but can be aware of and report their existence to the higher AI. Motion control can also provide feedback, like 'move finished' or 'failed', to which the system has to react, or further propagate to the higher AI.

Taking these perspectives into account, we need a language that has repre-

sentations for characters, paths, places, and objects, in both implicit and explicit manners. These representations can be further specified using states and abilities for characters, and goal and route for paths. Here, it is important to identify *atomic* information, for which further specification provides no additional meaning. Optionally, we find atomic representation in equipment, optimality, failure, event, danger, distance, time, or connectivity.

Note that in general the creation of classes for each meaningful unit of information is unwanted. Information that is sufficiently meaningful by itself does not need additional annotation as it would only contribute to a jungle of classes, instead of a purposeful set of classes that is full of meaning.

4.3 Users

While the navigation system plays a central role in the functioning of a game's AI, it is only a low-level component of the AI and an even smaller part of the game as a whole. This subordinate role of the navigation system, does not allow it to dictate the way game developers work with it. Instead, the development methods should be respected in order for our system to integrate, not only into the game AI, but also into an entire game.

There are two principal groups of people that directly interact with the navigation system: *level designers* and *AI programmers*. In order to elicit their requirements, we interviewed both designers and programmers that work with Unreal Engine 3. They were asked about their opinion on this engine, and how they use it. Next a number of improvements based on earlier research were proposed [10], and again they were asked their opinion on these improvements and how they would like to use them. These include:

- *To make the AI use multiple abilities to move around, like jumping over chasms and climbing up hangs.*
- *Make the AI aware of more than only distance, for example the amount of cover.*
- *Extension of capabilities into the dynamic domain, e.g. dynamic obstacle avoidance.*
- *Get more information out of a graph search, e.g. tagging pick-ups.*

Designers

Most of the time designers currently spend on AI consists of the construction of a waypoint graph. Here, the graph nodes are manually placed, followed by the running of the path building routine, which automatically builds a waypoint graph out of these nodes. In order to create a suitable graph, this sequence of node placement and path building has to be repeated several times until the graph allows for the navigation that is required.

There are four main problems with this approach:

- A significant amount of level design time is spent on creating an acceptable waypoint graph.

- Feedback is slow, since one path building run can take up to several minutes.
- Graph edges that indicate to be accessible, can at run-time turn out to be impassable, therefore requiring in-game testing.
- When the level design changes, the nodes need to be updated, requiring the same process to be repeated.

Designers' general response to an extension of the navigation system is that it is way more important to have this done in a reliable automated fashion, than to extend it with new features. Above all, the system should:

- Place nodes automatically, and directly provide feedback about connectivity, so designers can take this into account while building and modifying the level.
- Provide a reliable graph, showing only edges where a character is really able to walk.

Added to that, of the proposed improvements the following where preferred:

- *To make the AI use multiple abilities to move around.* The character does not need to equal the human player abilities, but should at least be able to jump.
- *Extension of capabilities into the dynamic domain, e.g. dynamic obstacle avoidance.* This allows the designer to introduce more dynamic objects around AI, providing a much richer game play.

Note that these improvements mostly appeal to the designer because they are closely related to the limitations that designers currently experience. Designers were not easily convinced on the benefits of a technical improvement like the tagging of pickups in the search graph.

Programmers

The creation of an AI character is done by the AI programmer, who is mainly occupied with the programming of a scripted state-based AI system. Here, the navigation system is used for moving to specific spots that originate from a scripted storyline, or during combat situations, to move in-between covered positions.

The current navigation system, which will be discussed in Section 4.4, lacks a good interface between the navigation AI and higher AI, having the following drawbacks:

- The navigation interface resides in the character, providing a poor overview.
- There is a very limited set of instructions. One can only specify the goal location and a look-at direction.
- Dealing with navigation on a lower level is very difficult since navigation is spread through the character's source code, badly documented, and inconsistently extended in many places and many ways.

Other problems the programmer faces are related to path traversal, e.g.:

- Traversing a path requires many independent path search operations, requiring a lot of redundant search time.
- Although the character gets stuck from time to time, there is no standard functionality that deals with that.
- To have the character deviate from the unnatural path along the waypoints, several tricks have to be applied that do not guarantee a successful path traversal.

Like level designers, AI programmers also indicate that many problems originate from the unreliability of the system. This, however, would not be a major problem, if there were easy ways of dealing with this. AI programmers prefer the following improvements:

- Easier and safer ways of making the character traverse naturally looking paths.
- Better ways of integrating jumping and cover finding in the path search.

Of the proposed changes, these were preferred:

- *To make the AI use multiple abilities to move around, like jumping over chasms and climbing up hangs.*
- *Make graph search aware of more than only distance.*

Discussion

Interviews with these groups of developers revealed a strong requirement for feature improvement over expansion. First of all, a navigation system should be robust and reliable; it should be able to deal with a wide variety of cases. Here, failure is tolerated, provided that problems are detected and reported to the higher AI. This brings us to the second issue of feedback. The relevant inner workings of the navigation system should be observable for both designers and programmers. This includes a set of debugging tools that allow the working of the system to be presented visually, and on different levels.

Returning to the proposed features, we have assembled a list based on the feedback from the interviews, ordered by descending preference:

1. *To make the AI use multiple abilities to move around, like jumping over chasms and climbing up walls.*
   Another typical 'away from scripting' issue. This time with a strong challenge in motion planning. For this, we need a framework that can handle movement in a more general way and provide fast motion synthesis at run-time.
2. *Extension of capabilities into the dynamic domain, e.g. dynamic obstacle avoidance.*
   It would be nice if a path would reroute itself to avoid obstacles or follow a moving character. This is quite challenging since this grants automated

initiative to the AI. On the other hand, the complication associated with dynamics mostly concerns optimization of low-level routines to allow fast real-time updating.

3. *Make graph search aware of more than only distance.*
   This can be realized by the introduction of a more advanced path metric. This feature is fairly easy to implement because it emerges only in specific key places and has little meaning beyond those.

4. *Get more information out of a graph search.*
   Because this is quite a vague and abstract feature, designers do not seem to be aware of the fact that by annotating the search graph and taking a few more variables into consideration during the search process, a lot of free information becomes available. For example, when searching for a distant goal position, without heuristics, the algorithm finds distances to all locations that are closer than the goal location, which can be quite valuable when planning for multiple goals.

This ordering of preference will not be respected in the design phase, which is covered in Chapter 5. From the previous sections, we learn that the features 1 and 3 are required to be in the design, while 2 and 4 are only preferred. Still, the preferred improvements will be used in the graphical and class interface design.

4.4 Game engine

A navigation system is part of the game AI, which in turn is part of the game engine. The engine dictates mainly what information is available for navigation, how motion instructions are specified, and which hardware resources are available. On the other hand, an engine does also provide a well developed and rich set of tools. This way, a game engine both restricts and supports our system.

We will be using the *Unreal Engine 3*, discussed earlier in Section 2.4. This section presents in what ways Unreal Engine 3 poses requirements for a navigation system in general. Non-functional requirements will be discussed in Chapter 5, for they are issued from a design perspective. In the remainder of this report, we will use 'Unreal' to refer to Unreal Engine 3.

Physics simulation

Generally, game physics embodies the simulation of physics like laws to provide realistic motion of objects and characters, ranging from collision detection up to world state prediction. Unreal is equipped with powerful physics simulation functionality, which unfortunately only runs in-game and according to game time, i.e. it simulates only the current game state. In fact, the only physics tool available for navigation is collision detection, operating on the current world state.

This obviously contradicts with navigation system's task to plan motion, which requires some insight in future world states. Our navigation system should be able to deal with this dominant shortcoming, mainly by a more primitive and inaccurate simulation of physics. This, of course, leads to an increasing amount of discrepancy over time, forcing the system to regularly reappraise its plans as the world state changes constantly.

Luckily, designers tend to keep AI away from highly dynamic areas. It is therefore *not* a requirement for the navigation system to deal with these environments. Most of the geometry will be static and only a limited set of objects and characters will move around. Nevertheless, there is a demand for systems that handle dynamics in a generic way, allowing it to handle dynamic environments as more hardware resources become available.

Motion instruction

Our navigation system depends on Unreal to execute the motion plan that results from for example a 'move to' instruction. Unreal provides a set of basic motion types, including walking, flying, and moving on a ladder. In combination with the character orientation and speed, these motion types result in the physical movement and animation of the character. For example, to fly, one switches to the 'flying' state, and adds an upwards acceleration. When the character has landed, the motion type is switched back to 'walking' and we know we have landed.

This means that a navigation plan should eventually come down to a series of type and velocity pairs, which are fed to Unreal one by one on the right moment. Because Unreal is not able to detect when a character fails a specific motion step, the system must always have an expectation of the motion, which it can compare to the real outcome, and detect these failures from their difference.

Editor and game

The Unreal engine has two types of run-time; editor and game. The editor is where the level is designed; where everything can be dynamically modified, while the physical time stands still. The game is where the game runs, where physics are simulated, and no persistent modifications to the level can be made. This distinction should be respected in a way that the navigation system should handle both editor dynamics as game dynamics.

In the editor, which is mainly used by level designers, our navigation system is required to provide feedback on how a static layout of geometry and objects allows characters to move around. Although there is no way of showing a moving character, the designer should be able to predict where paths will exist during the game. For example, when creating a chasm, the designer should be able to see directly whether a character will be able to jump it.

In game, one is more interested in the specific cases at hand, for example: why a character takes a certain path, what its plans are, how these are realized, and which problems arise during traversal. Because this is where the AI programmer gets most of his feedback, the game should be equipped with more detailed information, also in combination with the running of script code. Adding the fact that time control is only limited to pausing or slowing down, this information should also be accessible after a testing session.

Scripting

One of Unreal's important features, is that is has its own scripting language in which basically all the game programming takes place. Unreal Script is an object-oriented language with high-level constructs like state machines, component models, latent execution, and language tools like a profiler. It allows

for fast and iterative development, at the cost of a significant performance decrease. When performance is critical, one must move from script to native `C++` programming. At the cost of losing many of the scripting features, one gains up to 20 times faster execution.

For this reason, much time-critical functionality of the navigation system has to be implemented in the native environment. This functionality will not be as accessible as it would be in script, and therefore should not require modification in daily use. Instead, it should be possible to monitor these processes from the script environment up to the level where the AI programmer is aware of the processes running behind the scenes. This is not only necessary for debugging purposes, but also to allow script programmers to understand the navigation system, thereby making the system more predictable and stimulating its diverse application.

Chapter 5

System design

Based on the requirements, and our knowledge on general navigation and semantics, we now propose a system design for semantic navigation by presenting four different views on the design. First we take a closer look at the Unreal engine to set the basis for our system. Secondly, the system is discussed on a semantic level, where we introduce its main semantic components. Thirdly, we review the architecture while focusing on the modularity of the system. Finally, a number of key techniques are discussed that make up the core of the navigation AI.

Note that in the remainder of this chapter, we will refer to an object class by using a capitalized name, e.g. Path to refer to the class that represents a path.

5.1 Unreal Engine 3

As a preliminary step toward the design of our system, we first have a look at some important features of Unreal. We present an overview of the design of Unreal Engine 3, and discuss how navigation is currently performed. In the mean time, we compare this engine with our requirements and decide where the navigation system fits, how the engine can be used, and which parts need to be discarded.

Architecture

The central object in an Unreal game is one instance of the *UnWorld* class. Note the use of the 'Un' prefix to indicate a native Unreal class. A single instance of UnWorld contains all objects that are related to the current level or scene that the game is in. Using UnWorld, we have access to all geometry, objects, and characters, and we can request ray casts and collision checks.

A character is represented by an instance of the *UnPawn* class. The pawn is the physical body of a character containing its geometry, animations, and equipment. The behavior of a pawn is governed by a *UnController* instance. The controller is the brain of character, as it runs the AI routines that plan the moves of a pawn and steer it around in the environment.

Geometry is represented by *UnPrimitiveComponent* instances, which contain a set of meshes of a fixed relative orientation. A primitive component is an

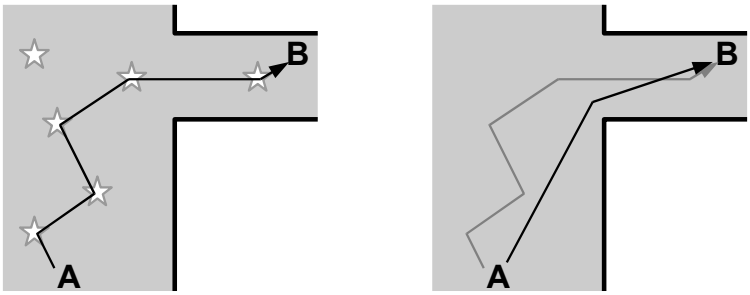

Figure 5.1: Waypoint zig-zag versus the shortest or natural looking path.

atomic part of the environment's geometry, and is stored in a list inside the world.

Waypoints

Navigation in Unreal is based on the waypoint graph concept, where a character travels between two places by first going to the nearest waypoint, then traversing a series of waypoints in the graph, to end at the waypoint nearest to its goal location. Waypoint navigation works under the assumption that from every reachable position, there exists a waypoint that can be reached by walking in a straight line. Edges of the waypoint graph represent walkable lines over which one safely travels from waypoint to waypoint.

One major advantage of the waypoint graph is that it can be manually put into place by the level designer, leaving the designer in control of where the character will navigate. The Unreal Editor is equipped with routines that connect waypoints in a sensible way.

Waypoint graphs are still in use by modern games with a less demanding AI. There are however a number of problems that cannot easily be overcome by the extension of waypoint graphs. The origin of these problems is the fact that waypoint graphs poorly represent space. Although graphs preserve topological information, they do not indicate how much space there is to navigate. The result is that it is very hard to program AI behavior for walking the shortest path, instead of a typical zig-zag motion along waypoints (Figure 5.1). Another problem appears when dynamic obstacles or other characters block waypoints or edges; there is no knowledge about the surroundings, and therefore no way to find a way around the obstacle.

Unreal offers a series of tricks that attempt to leverage these problems. These include the addition of a radius around a waypoint, a lane width along an edge, and a dynamic sub-graph around dynamic obstacle. These solutions, however, prove to be difficult to configure and remain error prone.

Because the semantics of space has a prominent role in our system, waypoints are hard to fit to our system and should therefore be discarded.

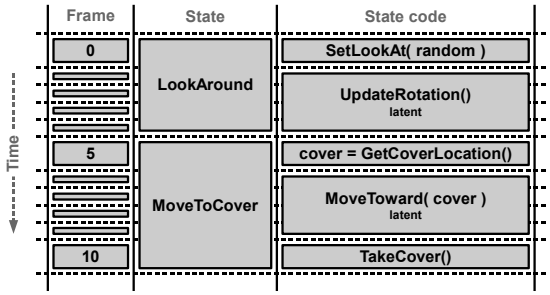

Figure 5.2: Running of state code inside a frames-based game engine. Each frame is one game tick; the passing of one time unit of about 30 milliseconds. Latent functions can take up to hundreds of frames, thereby blocking the execution of state code for several seconds.

AI programming

A pawn's behavior is created primarily by programming in Unreal script, as we discussed earlier in Section 4.4. The Unreal approach to AI is the classical state machine. States have been integrated deeply into the language and currently dictate the way AI is programmed in Unreal.

Each pawn's controller has a set of states of which one can be active. The script code of a controller is spread over these states and a state's code runs continuously while the controller is in that state. This way, the AI programmer can focus on behavior per state, instead of deciding each time frame what to do. To cope with the fact that the central game loop iterates about 30 times per second, the state code is sliced over multiple frames. A state change occurs only when the current state code requests a different state to be activated.

Because Unreal script runs on a `C++` programmed engine, most script instructions result in a native `C++` call. Because some of these calls can easily take more than one frame, latent execution is introduced. For example to move the pawn to a given location, the latent `MoveTo` function is called, which pauses the execution of script code until the pawn has reached this location.

Certainly the state-based AI in Unreal has its limitations, especially when compared to goal-oriented AI systems. Still, we are bound to shape our navigation system to it, since rewriting the fundamentals of Unreal AI would be out of our scope. Typical limitations are the following:

- There is no clear model of what behaviors are available and what rules are used to discern between them.
- Since there can be only one state active at a time, a controller can only exhibit one behavior type at a time.
- Latent execution blocks state code, theoretically disabling the AI.

Given these limitations, in order to make a pawn both move to a certain location and aim at another enemy, the controller can either emulate a second state in a function that is invoked each frame, or adopt a second so called *UnComponent*, which has its own state machine.

For the controller to quickly make a decision that involves a navigation query that spans multiple time frames, we have to provide decision information without the use of latent functions. Therefore, we need a way to latently provide information to the AI using either active querying, where the controller frequently checks to see if the information has been found; or passive events, where the controller waits for the navigation AI to send a message when the information has been found.

Semantics

In order to match our semantic navigation system to Unreal, there also needs to be semantics from Unreal's side to match with. This unfortunately is currently a dominant problem in the area of integration. We have seen that Unreal offers classes to represent a character and its brain, but when further decomposing the character from an AI point of view, we find only 'hidden' semantics, defined implicitly in language-oriented constructs and implementation.

There is for example, no explicit, complete, and compact model that describes character movement. Instead, character movement is a combination of physics simulation, character basing on the ground, forced translations, and a set of obscure functions that handle specific movements related to AI or special player character moves. Since this implicit definition is only programmed to forward execute a move provided that the complex character state is well configured, the explicit description will probably only be an approximation and require quite some tailoring to attain an acceptable exception rate.

The solution to this is to create semantic versions of the functionality that surrounds our system. These *semantic wrappings* only provide a translation using derived information and augments the meaning of information in general. This information can be derived from the limited set of *location*, *velocity*, and *collision*, which are very accurate and generally available in Unreal. In Chapter 7, we will further discuss the problems of semantics in Unreal's AI and engine as a whole.

It is important to note that a wrapping represents one concept that exists in two systems, namely the Unreal engine and the navigation system. While such classes are clearly separated on the design level, they can overlap on a conceptual level where different classes represent the same concept or entity. The problem that arises here, is that this strong conceptual link does not result in a link on the implementation level, which we will refer to as the *alignment* problem. For example, the navigation space representation in the navigation system is conceptually linked to the environment representation in Unreal's UnLevel class. This will require us to closely link their implementation and to put effort in synchronizing their state.

5.2 Semantic classes

In our semantic requirements (Section 4.2), we presented the need for a semantic language to represent specific semantics for navigation. While most of the information for navigation and its meaning is already present in Unreal in a hidden or implicit way, we provide a set of classes that allows us to specify semantics explicitly and add functionality for richer and more powerful navigation.

We assume that object-oriented programming provides a sufficiently rich language to build navigation-specific semantics directly on top. An intermediate layer of pure abstract semantic classes is not necessary, and not preferred for the practical purposes of fitting into the Unreal engine and the existing programming strategies. The value of our semantic navigation system lies in the design of these semantic classes; their interfaces, their relations, and the distribution of functions. Most important here is that we have to design classes and functions based on what they represent and mean, and *not* based on how we solve typical problems on our specific hardware.

This section presents the central semantic classes that constitute our design of a semantic navigation system in Unreal. Here we approach our design by first introducing its components, followed by the general architecture in Section 5.3.

Pawn

The *Pawn* class represents a character that can navigate. The Pawn class inherits the character-related semantics, like visual appearance, from the UnPawn class, while new navigation semantics is added.

A pawn is represented by its state and abilities. For navigation, this *state* consists only of physical properties, including position and velocity. Because state variables are modified very frequently and unpredictably, the pawn state will *not* be wrapped by a class thereby avoiding incorrect alignment of the state definition in Unreal and our wrapping.

The abilities of a character, on the other hand, are versatile in type and meaning and are therefore defined in the *Ability* class. Each Pawn instance has a set of Ability instances that represent walking, jumping, or crouching. Abilities can be shared by different pawns, and can be sequenced along a path.

A pawn interface provides the following:

- Its state; position, velocity, and collision volume.
- Its abilities; a set of Ability instances.
- Tests to determine if the pawn can stand on or traverse some part of the environment.

This offline test should accurately simulate the online motion that will be performed in real-time. Because this can never be perfectly aligned, the simulation prefers false negatives over false positives, avoiding to mark an area walkable while it is not.

An Ability interface provides the following:

- Space test; to see if this ability can be applied for traversal of a space, for example a piece of terrain surface.
- Link test; to see if this ability can be used to traverse from one space to another, for example by jumping.
- Anchor test; to see if a pawn can position itself at a given location using this ability. The anchor will be further discussed in Subsection 6.2.6.
- Executing this action to move from one location to another.

- Axis aligned bounding boxes (AABB) used for pruning search space, for example to prune calls to computation intensive functions like space and link testing.

The larger part of these interfaces is dedicated to predicting whether the pawn and its abilities will be able to make a certain move in real-time. Here prediction and execution should be aligned and provide identical results based on different information. The same differentiation we see in the way we control our (human) body, where we constantly predict what would happen if we made a certain move, sometimes followed by actually making this move, keeping in mind the same aim. Luckily, our daily thoughts are not dominated by this low-level matching between planning and execution and both are generally dealt with as if they are perfectly identical.

At a lower level however, there is a big difference between building a good prediction and making a smooth move. When dealing with all the possible outcomes one knows, both a thorough pruning as well as accounting for misses and unknowns are a necessity. Prediction requires a model that contains only the relevant information for the actions that can be executed, such that one can safely and quickly search though this sparse space, while carefully collecting all the additional information about the necessary precautions to be made.

Space and Place

In order to navigate, or to issue navigation orders, one needs a notion of space. The representation of space is required for two main purposes, namely the representation of those environment features that are relevant to navigation, and the representation of general space for semantic use.

The former type of space is contained in the *Mesh* class. A mesh contains, or in fact is, a representation of the environment for a navigation perspective. It mainly consists of a graph of *Space* nodes and *Link* edges. A Space instance represents any set of locations in some space, for example a piece of 3D volume or 2D surface, or a specific location. A Link instance represents a connection between two spaces, for example a trajectory of a jump between two separated surfaces, or a set of actions that lead from one side of a door to the other. Spaces and links are a translation of the environment in navigation terms, allowing the navigation AI to understand the environment.

The latter type of space is represented by the *Place* class. A place is, like a space, a set of locations in some space, but as a space is aligned to the environment, a place is a conceptual space that is designed by the higher AI. A place can be explicit, like a set of locations, a set of spaces, a moving character, or a place can be implicit, like a covered location, or a spot with good line of sight to the enemy.

To keep the mesh aligned with the environment, we introduce a layer of *PrimitiveComponent* and *Geom* classes. A primitive component is a semantic wrapping around the UnPrimitiveComponent (Section 5.1), and represents one piece of solid geometry. Each flat piece of surface of a primitive component is represented by a Geom instance. The purpose of a geom is to provide a well formed polygon that can be used to place our Space instances on, to provide clear feedback to the designer on how the geometry is interpreted at a basic level, and to provide a structure for controlled propagation of geometry modifications.

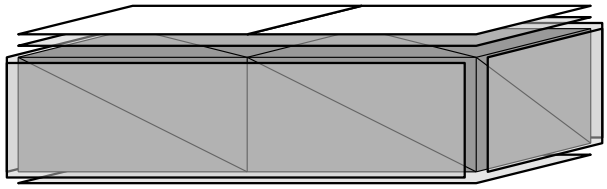

Figure 5.3: A rectangular primitive component consisting of 20 triangles (thin lines, dark gray). Each of the six sides is covered with one geom (transparent light gray). Since only the top geom can be walked on, this geom is patched with two spaces (white). Note that a geom can be patched with any number of spaces. The geoms and spaces have been drawn with a small offset for visualization purposes; spaces, geoms, and mesh triangles are coplanar for each of the six sides.

Note that both classes serve the purpose of gluing the mesh to the Unreal geometry representation.

Spaces come in two flavors, namely from *annotations* made by the designer or based on *surface patching* of geoms. In the former case, the designer adds spaces that cannot be derived from geometry features or go beyond the navigation AI's understanding, like a special hide out or a rope that can be walked on. Most spaces however, will be derived from geoms and the abilities of the pawns that traverse these geoms. Given an ability and a geom, we construct a set of convex surface patches on this geom where this ability can be practiced. In reference to Subsection 3.3.1, surfaces are the uniform subsets of space.

Links are the edges in the mesh's space-link graph. A link connects a space based on a given ability. For example, when traversing from a walking space to a crouching space, the link instructs the pawn to duck, or to traverse a jump, the corresponding link provides hints on the jump speed and direction.

The mesh primarily provides information to other parts of the navigation AI. This also holds for primitive components, geoms, spaces, and links. Their interface mainly provides access to their attributes and to functionality that maintains alignment when geometry in Unreal is modified. The PrimitiveComponent class contains key features of its wrapped UnPrimitiveComponent to detect modifications, and the set of Geom instances it supports. In turn, geoms contain a set of spaces and spaces a set of links that go from them and a set that link to them. Because spaces and links are ability-specific, they also contain a reference to this ability.

Note that in our design the ability has a dominant role in the creation of spaces and links. Other approaches often create ability-independent spaces and links, to later on match abilities to them. From a semantic point of view, this leads to further complication, because initially these spaces and links need some criteria to derive them from geometry, which will be somehow ability-related. This information is however discarded and spaces are reinterpreted during path search and traversal, which results in different models that represent the same

navigation space. In our case, both spaces and links have a rich meaning allowing the navigation AI to quickly identify all suitable spaces and links given a pawn and its abilities.

Spaces and links also have a set of attributes dedicated to the graph search process, further discussed in Subsection 6.2.1. Although important to the search algorithm in use, these attributes have no further semantic value, i.e. they mean only something to the search algorithm. This also holds for the navigation mesh (not to be confused with the Mesh class) building algorithm, which resides in the Mesh class. This algorithm, although important to the navigation AI, has no semantic value up the level where it combines space and link test functionality from an Ability instance to trace spaces inside a geom. Such an algorithm, as discussed in Subsection 6.2.2, is replaceable without interfering with the semantics of spaces, links, or abilities.

Path

The Path class represents a path inside the environment. It can be defined both *explicitly* by a series of instructions that lead a pawn from one place to another, or *implicitly* by a start and end place and a metric that defines the optimal set of locations. A *Metric* instance is able to measure path optimality, by weighing the spaces and links that make up the path. Assuming that we are always looking for optimal paths, we can define a path based on a metric, e.g. the safest route, and a pair of places. A path is not just a set of locations, because without the relation to a pawn or ability, locations are meaningless.

The Path is a central class in the communication with the navigation AI. The higher AI explains the system which path it is interested in, using an implicit definition that only specifies a character type, place, and metric. Next, the navigation system searches for an explicit representation, which is then valued by the higher AI. If the path exists, and if it provides what the higher AI needs, then the higher AI instructs the navigation AI to make some character follow that path. If the action sequence somehow does not lead to the goal location, the navigation system will inform the higher AI and search for a new explicit path using the same implicit definition.

Making a path explicit when a search is initiated results in a clear path search procedure, which always results in either a fixed or no path. A disadvantage is that once the path is made explicit, it will not match its implicit description under the changes of a dynamic environment. For example, when chasing a moving character, we can plan a path to its current position, but once we are at the end of the path, our goal character is long gone. Because an explicit path representation is always required for a pawn to traverse the path, these outdated paths are inevitable. To avoid the need for a complicated interface that is able to deal with dynamics, we moved dynamic path planning to the Navigator class which is discussed later on.

After the aforementioned search, the path is explicitly defined by a series of *Action* instances, which describe what action on what space or link should be performed. An action has attributes that refer to:

- The ability needed for executing this action.
- The start and end space.

- The exact start and end location.
- Some ability-dependent speed and directions hints.

Navigator

The navigation AI is a fast and predictable instrument to find and traverse paths, and to inquire navigation-related information. Finding paths in a dynamic environment does not quite fit this description, as it has many approaches which should be fully tailored to the requirements of the higher AI. If, for example, we have fixed geometry, we can walk around safely only avoiding collision with other characters, while a highly dynamic environment requires a character to move around carefully, planning many redundant paths, and sometimes taking time to re-plan its current path.

While the navigation AI is not fit to do specialized jobs like dynamic path planning, the programmer of the higher AI should not be bothered with it either. Therefore, we introduce the *Navigator* class as a layer in between the navigation AI and the higher AI. A navigator provides game-specific functionality by combining general functionalities of the navigation AI.

For example a basic navigator for a steadily changing environment can be built as follows. The higher AI tells the navigator to move a pawn to a place given a metric. The navigator requests an initial path using this place and metric. Now, every two seconds, the navigator stops the pawn, requests a new path from the current position, and reinstructs the pawn to follow the new path.

Clearly, this example navigator makes the pawn stop too often, delivering unnatural motion for the pawn. However, by programming its behavior in Unreal Script, it can be easily customized, to plan paths in advance. At the cost of a few redundant path searches, we can build a navigator which provides temporal smooth paths, as will be shown in Chapter 6.

5.3 Architecture

The classes that were introduced in the previous section provide the foundations of the semantic language that we will use for building our navigation system. To provide additional structure to the design of this language, we introduce a few additional specifications of our design, beyond the functional class level.

Layers

We first introduce a layered system architecture. This should result in a more clear class dependency, where a class has only meaning to classes in its own layer, and those above it.

The lowest and most fundamental layer is formed by the Path, Mesh, Pawn, and Controller classes (Figure 5.4). This layer is mostly conceptual, based on the central role of the path and the mesh. The mesh is a representation of the environment through which we navigate. It forms the set of all solutions to those navigation tasks we can solve. A path is one of these solutions and solves a specific navigation task. The idea is that a path can be used by a controller to communicate with the navigation AI and its pawn.

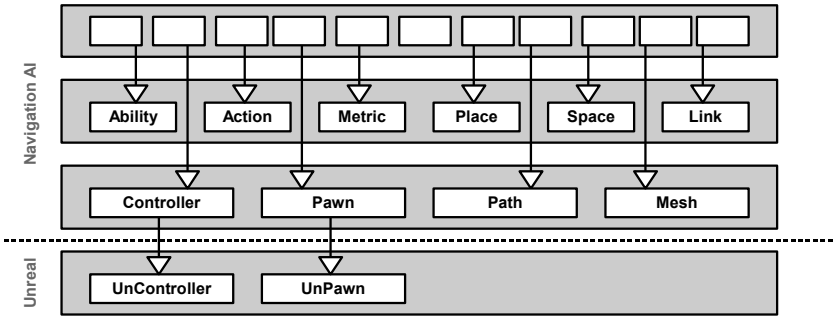

Figure 5.4: Layered semantics. Inside the navigation AI, the bottom layer contains Controller, Pawn, Path, and Mesh. The middle layer contains Ability, Action, Metric, Place, Space, and Link. The arrows indicate inheritance. The top layer will be defined in Chapter 6.

On top of this path layer, we introduce a middle layer with Ability, Action, Metric, Place, Space and Link. In contrast to the fundamental role of the previous layer, these classes form a more practical tool set to both the programmer and the navigation AI. It allows the programmer to define pawns using abilities, and paths using metrics and places. On the other hand, these classes also exist for implementation-related reasons, namely the action to instruct Unreal's motion control, and spaces and links to define a search graph, which will be further discussed in Chapter 6. This mixing of semantics and implementation will be even more prevalent in higher layers.

The top layer consists of classes that will not be specified in this design. They will be subclasses of the classes we have seen in the two previous layers, thereby remaining meaningful inside these lower layers. However, the specific meaning they have is only understood by classes in that layer and of course the programmer.

In order to implement this design, one has to fill out the top layer with classes that inherit the meaning of their superclasses, and implement specific functionality, for example to solve path searches, allow motion control, or draw a visualization on the screen. These classes provide additional meaning, like a metric to measure distance using the vocabulary of the middle and bottom layer. It is important to realize that the meaning it has to the AI programmer is much richer than the meaning it has to the bottom two layers of the navigation AI. On a lower level, the system will never really understand what distance is, it can only be instructed to use the distance metric as a metric of some kind. This inherently limits our navigation AI to what lower layers provide. However, if we are able to specify distance as a metric subclass, we can use this new distance concept in the top layer, in a way it really means distance.

Modularity

An important aspect of our design is its modular architecture where classes are semantically independent, and communicate via a clearly defined interface for sharing their data and functionality. Modularity meets with our requirements

to provide independence of classes inside our navigation system, resulting in an extensible, reusable, robust, and predictable design.

The independence of classes and their clearly defined interface facilitates *extensibility*, as complication of class design does not propagate to other classes. Especially with respect to subclassing, our design allows new classes to be added without modifying the underlying structure of superclasses.

Another important aspect of modularity is the *reusability* of classes. This proceeds in two ways, as classes individually can be used elsewhere, or other implementations can readily be integrated. By clearly splitting the semantic structure from the lower-level algorithms, we allow new algorithms to be introduced with a minimum impact on the semantic structure above.

A *robust* class is able to produce reasonable output from most of the input it can receive. This kind of stability provides a less restricted use for designers and programmers, e.g. they can even feed the system with random input. Also, an anomaly inside a robust system does not expand and can more easily be contained. By making individual classes and the entire navigation system more robust, we meet our initial aim of making a solid contribution to game AI in general.

A *predictable* class produces similar output for similar input over time. This mathematical continuity property primarily benefits designers and programmers, because it allows them to foresee the outcome of their actions and thereby plan game development in an efficient way, which was a major requirement on the users side.

When designing for modularity, it is particularly important to take notice of the *alignment problem* as discussed at the end of Section 5.1. Our aim is to avoid alignment primarily by making single representations for concepts, which is stimulated by our semantic approach. Otherwise, programmers should be well aware of this conceptual relation outside the class interface and deal with that, without passing this responsibility to the designer. In that case, our approach would be to detect, quantify, and minimize alignment problems.

Mixing

Earlier this section, we shortly discussed the *mixing* of semantic concepts and implementation in classes in the higher layers of our system. Because the system's implementation, and specifically its algorithms, provide the semantics we need, mixing of both is inevitable.

Still we should be aware of this process and separate both on a class level to make a clear distinction between semantic structure and the underlying algorithms and programming techniques. For example the interface of an ability provides semantic functions to allow space testing. Although this function is very important to the implementation of space creation algorithms and graph search, it primarily functions as a way to find out where this ability can be applied. In contrast, the axis aligned bounding boxes that are also provided by this ability, have little meaning outside the algorithm that prunes its search tree using these bounding boxes.

By emphasizing the semantic importance of functions in the interface of a class, we can separate the implementation from the semantics. This allows us to better understand the importance of these functions with respect to performance and extensibility.

Chapter 6

Implementation

A large share of our project time has been dedicated to the implementation of a system for semantic navigation. This implementation serves the purpose of being both a proof-of-concept for the design proposed in Chapter 5, and an extension to the game engine used at Coded Illusions.

In this chapter, we present the design and the most relevant techniques of this implementation. We show how we go from the proposed design to this actual implementation, without fully specifying the latter. Here we illustrate how to build a navigation system that is based on a semantic design and fit for serving in game development.

6.1 Design

The design as presented in Chapter 5 mainly describes our navigation system from a semantic point of view. In order to have the navigation system perform the actual navigation tasks described in the requirements, we have to extend this design by providing semantic classes for specific tasks, and the implementation of techniques that provide these semantics.

Section 5.3 introduces the top layer of the navigation system, which is where we extend our design to implement specific semantics. Figure 6.1 shows the layered design including the classes that make up the top layer in our implementation. The purpose of each class will be discussed in Section 6.2.

We cover the subclasses of *Ability*, *Controller*, and *Pawn* in Subsection 6.2.8; subclasses of *Metric* in Subsection 6.2.5; subclasses of *Place* and the *Anchor* class in Subsection 6.2.6; the *TileSpace, SampledLink*, *PrimitiveComponent,* and *Geom* class in Subsection 6.2.2; the *SmoothPath* and *Action* class in Subsection 6.2.7; and the *Mesh* and *Task* class in Subsection 6.2.9.

The diagram in Figure 6.1 offers a view on the final design from a semantic perspective. Additional specification of the design has been omitted to emphasize that these classes are the important building blocks of our implementation. Many other classes were added to provide:

- a better integration into Unreal,
- a basic user interface in the editor,
- optimization of data structures,

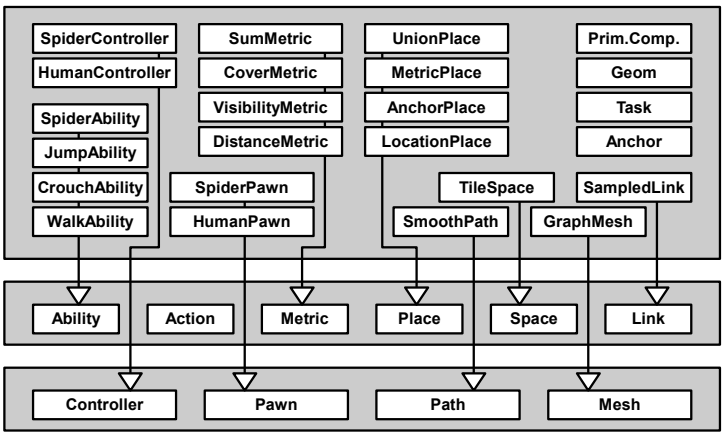

Figure 6.1: Class diagram, introducing the content of the top semantic layer (see Section 5.3). The arrows indicate inheritance. The PrimitiveComponent, Geom, Task, and Anchor classes were introduced for implementation purposes, like the connection between the Unreal engine and our navigation system.

- general drawing facilities,
- assistance with debugging and profiling.

A basic idea of the setup of our implementation is provided by the composition structure depicted in Figure 6.2. This figure shows two root containers, namely the pawn and the mesh, which represent respectively the character and the environment.

The *pawn* owns its abilities and a controller to assist it in using these abilities. The controller creates places and metrics to specify navigation plans, and uses its navigator to issue these plans. The navigator searches for paths, which consist of a series of actions.

The *mesh* represents all that is not related to a character, which is mainly the environment. The mesh is composed of the primitive components that make up the environment's geometry. This is the start of the composition chain where a primitive component is composed of geoms, a geom is composed of spaces, and a space owns its links. The mesh also contains the queue of tasks it runs latently.

6.2 Techniques

Besides a semantic structure, our navigation system requires a set of specialized techniques to deal with the real challenges of navigation in games. These do not have specific value to our semantic approach and could be replaced by another technique without much interference with this semantic structure. Still, they are interesting because they form the heart of the system and because they were selected for matching non-semantic requirements.

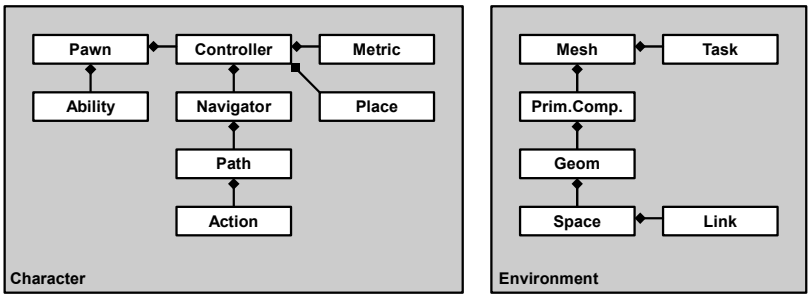

Figure 6.2: Diagram showing composition relations between the classes that make up a character and the classes that represent the entire environment.

In this section, we present implementation details from a number of different perspectives. Our aim here, is *not* to cover the complete implementation, but instead to explain how semantics, algorithms, and programming techniques meet to deliver the core of our implementation.

6.2.1 Pathfinding

Inherently related to navigation in games is the field of pathfinding. Pathfinding is the act of finding a sequence of locations, a route or path, which fits a predefined goal. The field of pathfinding has developed a rich set of techniques for pathfinding under various constraints. This chapter takes a quick look at some relevant pathfinding techniques for our navigation system.

The field of pathfinding has significantly matured over the last decades. Specific task requirements allow a definite selection from the wide variety of algorithms available [23, 43]. Still the implementation of a pathfinder requires significant customization, which makes every implementation a technique on its own. What can be learned from a brief investigation is the type of search method and the type of search space that matches our navigational needs.

Topology

In order to classify the search spaces we are faced with in our navigation, we study the topological features of the environment. The topology describes the structure of space without specifying the exact shapes and positions of objects, holes, and edges.

The environment of action-adventure games is typically a maze-like space; many boundaries and holes make up a large set of partially connected uniform areas. This mix of boundaries and uniform space makes pathfinding a diverse task [42]. The continuous topology of uniform space calls for a continuous or physics-based approach, while the discrete topology imposed by the boundaries can be better represented by a finite graph.

Continuous methods for pathfinding are mostly based on potential fields like force fields [3, 21, 25, 53]. These methods can be well described by mathematical

constructs that support path existence and optimality theorems. The discrete nature of our search space troubles these methods as they become unstable and get trapped in local minima.

Attempts to tackle general pathfinding problems have used the more recent optimization techniques from the fields of neural networks [18, 33] and genetic algorithms [19, 17]. Apart from the fact that they require quite some tailoring, these techniques can perform well in pathfinding. Still, successful neural and genetic applications in games are scarce, as they prove to be computationally intensive and deliver unpredictable results.

In order to perform a pathfinding search within computation and memory constraints, we have to reduce the detailed complexity of the environment to a representation that sufficiently preserves its topology. Here the search complexity should be reduced to an acceptable level, without violating the completeness of the search too much [8].

The most favored topology reduction in games has been the spatial sampling of the environment into a graph of nodes and edges. In this reduction, uniform spaces are captured in nodes, while discrete features are captured in edges. Pathfinding is now reduced to graph search, for which very efficient and specialized algorithms exist.

Graph search

The sampling of continuous space into a search graph is a far from trivial task and significantly influences the performance of the pathfinder that searches it [40, 45, 55]. Sampling can be done using techniques like regular grid sampling, probabilistic road maps, or triangulations [39, 47]. While these methods can be well tailored for specific environment types, graph node positioning or position fine tuning is often done by hand.

After the topology of the environment is reduced to a graph structure, all other environment features can be stored in the graph nodes and edges. Semantics of the environment is now reduced to semantics of nodes and edges. Features of a position or area, like food or cover, are stored in the related nodes. Relations between areas, like distance and connectivity, are stored in the related edges.

Most graph search algorithms found in games, are based on the Dijkstra algorithm [9]. This algorithm performs optimal assuming that the search is uninformed, and no memory or computational restrictions, or search history exist. The optimal informed variant of this algorithm is the A* algorithm.

In practice both Dijkstra and A* suffer from a number of problems related to the system that performs the search [6, 38, 51]; they can consume large amounts of memory and return either the optimal path or no path at all. Another drawback is related to data reuse. In games with dynamic environments where paths constantly need to be updated, both algorithms need to run all over again.

As the shortcomings pile up, more alternatives based on A* emerge, each performing well for a specific set of constraints [5, 22, 26, 27]. The trade-offs made here are very much problem and machine-specific. We will therefore leave this discussion for optimization research.

A path that is constructed by the search algorithm, consists of a list of nodes. From this discrete representation, a continuous path needs to be reconstructed. In this postprocessing step, we try to undo the reduction imposed by graph

sampling, by smoothing node and edge features along the path. Because a good path postprocessing allows a low sample rate and therefore fast graph search, the balancing of both is critical in the fast computation of realistic and continuous paths.

6.2.2 Navigation mesh

The use of graph search to solve pathfinding problems is a logical step in solving general pathfinding problems, as discussed in the previous section. When a graph has been constructed, we apply a suitable variant of the Dijkstra search algorithm to find optimal or near optimal paths. For the generation of the graph, however, there is no general approach, since it significantly depends on what data is available to base the graph on, and how the node and edge paths are further processed in motion control.

One technique to create these search graphs is the waypoint graph, which is currently used in Unreal (see Section 5.1). The waypoint graph primarily aims at topological representation, which can be described as the environment without the space; only the shape. The crucial lack of this technique is that when the graph path has been found, there is no information available to augment our topological path with spatial information.

Around the year 2002, a new search graph technique slowly emerged in games: the navigation mesh [52]. Instead of only representing topology, the navigation mesh splits the environment locally into topology and space. The navigation mesh provides both a search graph, to apply Dijkstra search, and spatial data to expand the node-edge path to a full motion path. The idea is that characters navigate only on the surface they walk on, so we need only the topology of the surface of our 3D environment. By patching this surface with convex polygons, and connecting them when they are adjacent, we create a graph of nodes representing surface, and edges for connectivity.

Based on this idea, many variations of the navigation mesh appeared, each with their own way of patching the surface and connecting the patches [12]. Techniques mainly differ in the ways they:

- build up from geometry triangles, partition the whole surface, or grow random polygon seeds;
- patch the whole surface, or only where one can walk;
- fit exactly to the surface, or provide only a coarse indication;
- use triangle, square, polygon; axis aligned or not.

In our implementation we use a new mix of these existing ingredients that best match our requirements. Because we mostly navigate characters of the same type, with the same abilities and physical properties, we can build a navigation mesh exactly matching this character. This means that patches, to which we refer to as *tiles*, will match with where the ground center of the character can be positioned, as illustrated in Figure 6.3. This way, we do most computations in advance, allowing effortless path search and traversal.

We would like the navigation mesh to be generated without the need for a designer to annotate the level. This means that we can only use basic collision

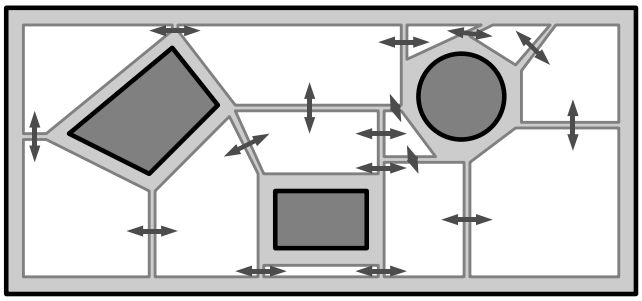

Figure 6.3: Impression of how geometry can be patched and connected. Dark gray areas are obstacles. Black lines indicate the collision mesh. White areas are the tiles (patches) that make up the navigation mesh. Arrows indicate a link between two tiles. Note that a link has no location, it only *connects* two tiles. The space between a tile and the collision mesh is a result of the collision radius of the pawn; a tile indicates where the *center* of the pawn can walk.

tests to populate the navigation mesh. To also accomplish this in real time, we want to use a minimum of tests and apply them on a controlled level of detail. This is done by character sampling, which is further discussed in the next section.

6.2.3 Sampling

Many of the geometry-related problems navigation has to deal with include some kind of continuous optimization problem. This section explains how most of these have been solved using a sampling-based discretization.

Analytic approach

For example, in order to extract walkable areas from geometry meshes, we can take an analytic approach, which deals with each polygon individually, using projections, and operations like difference and intersection. These techniques do provide very precisely carved pieces of geometry, with a running time that depends on the complexity of the geometry. There are, however, two main problems that trouble this approach.

First, one has to take care of a large number of exceptional cases, like slivers and rounding errors, that disturb the general approach. This has to do with the complexity of the geometry and to what extent the designer and programmer are involved. Here the programmer has to work hard to handle all exceptions, while the designer is limited to designs with feasible geometry that preferably is not too complex and has a homogeneous level of detail. This contradicts our requirement for an autonomous system with little designer involvement.

Secondly, an analytic approach has to be tailored to the problem at hand, requiring the programmer to reissue the details of an algorithm when new semantics is added, for example, a different character shape or a character with a different type of locomotion. The intermingling of algorithm and semantics on

an implementation level, does not match our requirement to be able to easily introduce new variations of some semantics.

Sampling approach

A solution to both these problems is the sampling of the continuous space into discrete samples, which do sufficiently represent the original space. Note that the word space is used here in an abstract mathematical sense. By creating our own samples, we control the density and accuracy of the data that will be used for optimizations.

We avoid rounding problems by sampling using a sample rate that is far from the accuracy of the floating point numbers that are used. Sampling does require a considerable amount of memory to store these samples, but again the sample rate is controlled resulting in a very predictable memory consumption compared to analytic techniques. In short, good quality control is often more important than minimizing resource consumption at the risk of incidental consumption peaks.

The second problem is also leveraged by the general use of sampling. Of course, it also requires tailoring, but on a more application invariant level, where general experience is sufficient to find the right sampling strategy and sample rate. Instead of the full integration of semantics into an analytic algorithms, we now only have to provide a basic sample providing function, which can be used for a wide variety of sampling strategies.

Application

Our system uses sampling in navigation mesh generation, to shape the space patches and to find connections between these patches. For geometry patching, we sample small instances of a locomotion type, using a recursive subdivision to rasterize selected subsets of the geometry. This requires to carefully weigh out the precision we want to capture, and the amount of resources available. Precision is mainly determined by the size of the character and the steps it makes. Clearly, the designer is free to add as much detail as desired, without influencing this precision.

To sample the connections between these patches, we use a short off-line simulation of a locomotion type on a series of locations on the edge. For each sample, the configuration is stored, to allow it to be executed in the same way during path traversal. Here, the sample rate depends on how smooth the paths need to be; more samples results in less zig-zag movement.

Although with samples we are not able to fully represent our continuous optimization problems, we can perfectly optimize within the set of samples. If we are able to sufficiently match a limited set of samples to the environment, we can quickly optimize using straightforward dynamic programming, for example, by using Dijkstra graph search.

6.2.4 Dynamics

The techniques discussed above provide good quality control, where resources are exchanged for better results on a smooth scale. But AI resource budgets can be very limited, forcing algorithms to run slow in order to come to the

right quality level. On the other hand, we want our system to handle dynamic changes, which demands real-time response to changes in the world. To serve both requirements, we need to provide a sufficient initial response to changes, followed by a smooth and slow recovery phase.

For example, when a path is blocked while a character is traversing it, we immediately need to prevent the character from walking into the wall, while the search for a new path might take a few frames to complete. To deal with the immediate change, we need real-time detection and an event system to report for example to motion control. Immediately we mark '*dirty*' the information that might require re-evaluation. Other processes can now avoid the use of this dirty information, or postpone their activities until the updating process has removed the dirty mark. In the case of our aforementioned example, the path can be marked partially dirty, allowing the character to decide whether it is going to wait a second, or to traverse the part that is still 'clean'.

Other uses of marking can be found in navigation mesh updating. After a part of the geometry has changed, we mark dirty those geoms, spaces, and links that might have been affected by it. In turn, all paths that route these spaces and links are also partially marked. Note that this locking of information is vulnerable to a dead lock, which can be avoided using a strict ordering of dirt propagation. We choose to always fully align all primitive components, which therefore are never marked dirty. A geom can only be marked dirty if all its spaces are too. This further propagates in the order of geoms, spaces, links, paths, and optionally higher plans.

6.2.5 Metrics

Metrics are the eyes of the navigation AI, for they are able to observe and value parts of the environment from a navigation perspective. Because our navigation requirements mainly aim at visibility-based navigation, we need a set of metrics that allows us to express path qualities in terms of how well it hides a character from the eyes of others, how much cover is provided, and how well the character can see other characters.

A chief metric in this set of metrics in represented by the *VisiblityMetric* class. The visibility metric has a set of observer pawns and knows the position of their eyes, which can also be interpreted as their gun's position. We now evaluate visibility by sampling both a pawn position inside the space and a location on its body, and casting a ray from one of the observers' eyes to this point. By repeating this test, we gain a ratio of hits over the number of casts, which are averaged to gain visibility. We weight these casts assuming that visibility decays quadratically over distance.

Using only the visibility metric for a path search, we obtain very strangely shaped paths, because distance is not involved in the search. To minimize both distance and visibility in one path, we use the *SumMetric* class, which combines a set of metrics using their weighted average. Now, the equally weighted average of a *DistanceMetric* instance, which measure the euclidean length of links, and a visibility metric results in both well hidden but short paths.

Like the sum metric, we can use metrics to derive new metrics from. For example, we can create the *CoverMetric* class to measure the amount of cover inside a space. A space with good cover has low visibility, but is next to a space that has good visibility to an enemy character (see Figure 6.4). A cover position

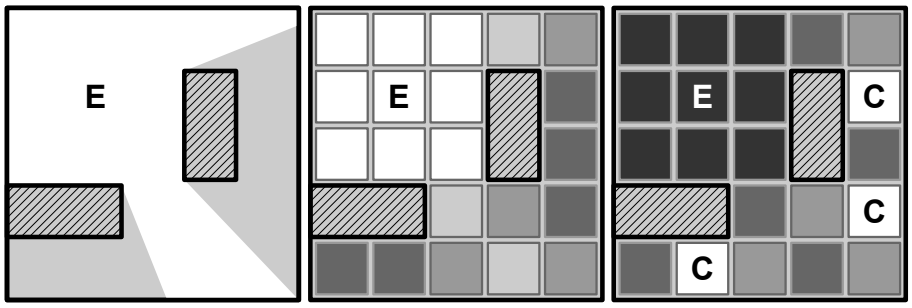

Figure 6.4: Cover detection using visibility metric. Left image shows visible area of enemy (E). Center image shows space measurement by visibility metric (white = high visibility). Right image shows space measurement by cover metric (white = good cover location (C)). Good cover locations have low visibility but are near locations of high visibility.

can be used to safely reload a gun and quickly return to action. To measure a space, the cover metric measures the visibility of that space and its neighboring spaces. If that space has low visibility, e.g. below 10, then the cover of that tile is the normalized difference of the maximum visible neighbor and that space.

6.2.6 Places and anchors

There are basically four types of places a path can start from or end at. First we have a set of fixed single point locations. Secondly, we have a set of variable locations, like moving pawns. Thirdly, we have socially constructed areas, like the visible area of a pawn. And lastly, places can be constructed by the designer to assist in the game's storyline, e.g. an airplane's drop zone.

The spaces that a place refers to can be implicitly defined, for example when we define a place based on the visibility metric. Because these implicit definitions have no further meaning to our system than being of the Place or Metric class type, there is no semantic structure to reason about them, except for the explicitly defined part of space they represent at a given moment. For this reason, we have to anchor a place before we can use it to, for example, search for a path between two of these places.

Anchoring is converting a location in space to an explicit location in navigation mesh space. For example, the anchoring of an arbitrary 3D location is finding the location inside the navigation mesh surface that best represents this 3D location. The way anchoring proceeds depends on what is using this anchor. To anchor for a human pawn, we need to project the 3D location vertically, because a human pawn walks upright. If the 3D location is too far above the surface, there exists no explicit anchor location.

6.2.7 Smooth paths

One of the major drawbacks of the waypoint graph that is currently used in Unreal, is the typical zig-zag motion along a waypoint path. The newly introduced

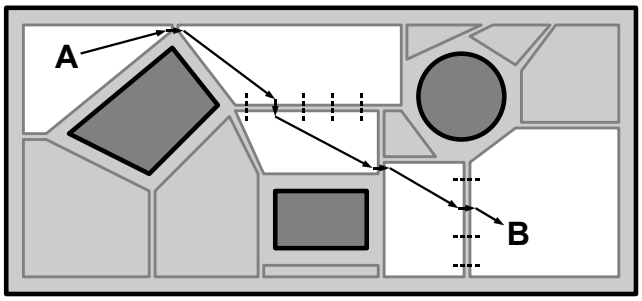

Figure 6.5: A smooth path consists of a series of actions in tile spaces and sampled links. The white tiles are the spaces in the path found by graph search. The dashed lines represent the action samples on links in this path. The arrows depict the action instances that make up the smooth path.

navigation mesh approach should provide us with more spatial information to create smooth paths. Therefore, we introduce *the SmoothPath* class, which uses *SampledLink* instances.

A sampled link is a collection of samples of how a link between two spaces could be traversed. When a link connects two tile spaces, as depicted in Figure 6.5, there are theoretically infinitely many ways this link could be traversed. By applying the sampling approach (Subsection 6.2.3), we generate a set of *Action* instances that explicitly describe how to traverse from one space to the other.

A smooth path uses these sampled links, by searching for the optimal sequence of action instances. Based on the space-link sequence that has been constructed during graph search, we use dynamic programming to select one action instance from each sampled link and find the shortest path in distance. Here we also take into account the distance traveled across each space. Note that we do not use the path metric to determine the smoothest path, because visual path smoothness is primarily distance-related, and because a metric applies to a space-link sequence, not to a set of action instances.

The quality of the resulting paths is directly related to the sample rate of action instances per sampled link. If we want the path to locally deviate less than a give distance, the sample distance should always be less than this distance. The major drawbacks of high sample rates is the large amount of memory that is required to store these samples, and the computation time. Especially the latter becomes dominant when sampling involves collision checks to see if the sample can be traversed without bumping into geometry. For example the jump action requires a series of collision checks along the parabolic trajectory of the jump. Therefore, the sampling of jumps cannot be performed on a high rate, reducing the smoothness of paths that involve a jump.

Note that smoothness mainly concerns the shortness of the path and not the continuity of its first or second order derivative. The volumes that determine the environment's collision are most often rather coarsely defined involving large flat polygons, which also makes optimal paths consist of straight edges of a size proportional to the size of collision polygons. Our implementation does not yield paths with a continuous derivative.

To traverse a smooth path, the pawn must execute each of the actions in the smooth path. Each game tick, the path checks the pawn's progress and provides new instructions to continue the current action. One action typically takes a few seconds to traverse, which equals tens up to hundreds of frames. Although the path was feasible at the moment we found it, it might become infeasible during path traversal, for example because another pawn is blocking it or due to a slight alignment problem between Unreal and our mesh. In order to remain independent of Unreal's many and unpredictable ways of recording such blocking events, we provide a time-based approach to detect problems during path traversal. Since we have accurate knowledge on the pawn's abilities, we can predict the time needed for one action. If after this time, the starting location of the next action has not been reached, we assume something went wrong and report to the navigator that this path is failing. The navigator then starts assembling a backup plan, which involves the same path search with a new starting point.

The implementation of the navigator is mostly concerned with the dynamic assembling of static paths. As discussed in Section 5.2, the navigation AI only performs static path searches. The navigator monitors the path by measuring its length on regular intervals. If this length passes a limit imposed by the controller, the navigator instructs the navigation AI to shorten the path such that only one second of path traversal time remains. Next it issues a new path search with the same goal and metric as the current path, except from a new starting position. As soon as the pawn finished the old path, it waits for the navigation AI to return the new path, and starts traversing it. The one second margin here, is to keep the pawn walking while the next path is being searched for.

If the path traversal fails, for example because an unforeseen collision occurs, the navigator makes another attempt to search for the same path. If this path fails again, the higher AI is informed of the fact that its pawn got stuck and cannot get out.

6.2.8 Human and Spider

To show the capabilities of the generic pawn, we implemented two subtypes of pawns, namely the human pawn and the spider pawn, depicted in Figure 6.6. These pawns only differ in the way they look, their size, and their abilities. Still, this results in two completely different kinds of pawns.

The *human* pawn has a cylindrical collision shape and the ability to walk, jump, and crouch. The walk ability can be performed on all surfaces that are less steep than a given maximum angle. Jumping can be performed using a maximum vertical and horizontal speed, and a maximum fall speed to limit the distance a jump can cover. Crouching is the same as walking, except for a collision shape of half the height and a lower speed.

The *spider* pawn has a cubical collision shape and the ability to walk on any surface using the spider ability. This allows it to go anywhere the surface extends, including walls an ceilings.

Figure 6.6: Left: a human pawn, which is able to walk upright, to crouch, and to jump. Right: a spider pawn, which is able to walk on surfaces of every angle or steepness.

6.2.9 Mesh

The Mesh class represents the environment with respect to navigation, and it is the central class of our navigation system. For representing the environment, we introduce the *GraphMesh* subclass, which' name refers to the fact that it uses a graph of spaces and links to represent the environment. A graph mesh provides functionality to coordinate the construction of the geoms, spaces, and links, and it provides search routines for searching paths. It also keeps references to these classes for drawing and serialization purposes.

The Mesh class is the central class in our navigation system, because it is the only class that has been declared inside Unreal to allow the engine to communicate with our system. It provides functions that allow deep integration into the engine, for example to tick frames, parse commands, update geometry, and log performance information.

6.3 Design aspects

Our implementation is required to provide room for extensibility and perform on a level that approaches the standards of today's game development. This section illustrates how this has been realized on the implementation level.

Extensibility and reusability

Games are mostly developed in a series of short development cycles, allowing designs to be adapted according to new ideas and changes in the market. To be able to keep up with changing requirements, software modules should be exchangeable and extensible, allowing new modules to be added quickly and existing ones to be extended over time.

When providing extensibility, it is important to provide a good mix of freedom and guiding restrictions. Extensions in our system always proceed through *subclassing*, avoiding the modification of existing code, which the existing architecture relies on. For performance reasons, most of our navigation system is written in `C++`. To extend these functionalities, one should overload a well documented set of virtual functions, which provide the semantic interface to the new subclass. From a semantic perspective, virtual functions already have

meaning in the navigation AI, while the subclass provides a new and specialized type of semantic information.

In the current implementation, we have been able to create all necessary semantics in subclasses. The tree-like shape of this inheritance structure might become too crowded at some point. In that case, it would be wise to use *interfaces* for semantics that applies to different classes.

The extensibility of the system was mainly tested during the implementation of the current system. This proved that the superclasses available are sufficient to build a human combat pawn and a spider pawn. Also the implementation of the navigator provides a flexible way of customizing the way our navigation AI is used in practice.

Performance and optimizations

Performance and semantics do not naturally go well together. Where performance-centered design mostly leads to fast algorithms and resources-centered programming, a semantic design creates concepts that have no relation to the algorithms that run them or the hardware they run on. Our implementation is based on a semantics-centered design, but applies a large number of optimizations, mostly based on pruning redundant computations. Simply put: we try to do only those computations that are necessary, independently of the fact that we follow a semantic design.

In this section we look at a number of performance bottlenecks and describe how we optimized our implementation to leverage these. The first problem is general *spatial lookup*, the relation between the 3D environment and our mesh of places, spaces, and links. Typical lookups are the following:

- What is the space a given 3D position should be anchored to?
- When a piece of geometry is moved, what spaces could become invalid?
- Given a space, to what spaces should it link given a pawn's abilities?

To perform these lookups in $O(\log(\#candidates))$ time, we use an octree of spaces and axis aligned bounding boxes. The octree uses a standard axis aligned model with leafs that split as soon as more than six spaces are added. Returning to the first of the above mentioned lookups, we first get a pawn-dependent anchor bounding box. By translating the box to the position of the 3D position, we get the maximum volume that should be checked for space candidates. Our octree returns a selection of spaces, from which we have to select the optimal anchoring space. Abilities, geoms, spaces, and links have virtual functions that provide bounding boxes for these lookup purposes. When subclassing, these functions should also be overridden.

A second performance issue is the AI's typical irregular consumption of resources over different frames. For example, path search and preprocessing requires more CPU cycles in less time than the traversal of the path. To spread this load over multiple frames, we adopted a task system where tasks are scheduled and sequentially executed in a time sliced fashion, covering multiple frames.

A major drawback of using tasks is that these algorithms run latently, forcing the creator of the task to pause or postpone its actions, possibly becoming latent too. A drawback of time slicing is a little less efficient CPU, memory, and

cache use. For these reasons, algorithms should run in time sliced tasks only if necessary. Currently, A* graph search and the creation of spaces and links is implemented using tasks. Here we have a significant profit as A* search can use complex metrics in game, and spaces and links can be generated while designers work in the editor. Note that it is important for the higher AI to be aware of the amount of tasks that are scheduled for execution. If the navigation AI does not have enough time to finish a task within one or two seconds, the AI starts to react slow and eventually stops. Therefore, less important path queries should only be added to a small task queue.

A third optimization is the caching of data using a time stamping and an expiration span. Metrics, like visibility, are computationally intensive, as they use many ray casts. To avoid redundant measuring, we implement a caching mechanism that adds a time stamp to the measurement. Metrics are only re-measured when they are outdated, which is metric-dependent. For example the distance metric is evaluated only once, while visibility out-dates after one second. Caching provides a controlled way to exchange performance and quality.

Chapter 7

Discussion

This report covered the work on our semantic navigation project. In this chapter, we return to the system requirements, our goals, and possible directions for continuing our work in the future. Our aim is to reflect on our work by reissuing our intentions and identifying remaining and related problems in our current design and implementation.

7.1 Requirements

Our design and implementation is based on the requirements presented in Chapter 4. This section relates our work to these requirements and determines to what extent these requirements have been met.

Navigation

Navigation requirements were presented in the areas of perception, guiding, dynamics, and autonomy. Most of these requirements have been fully met in our final implementation.

For *perception*, the requirements state that our system should be able to detect key locations in the environment, to predict the existence of a path, to estimate travel distance, and to provide progress information on path search. Key locations can be detected by a metric-based place. The cover metric identifies key cover locations of reasonable quality using a custom visibility metric. The existence of a path is predicted by performing a path search. Because the smooth path uses action instances that have been tested for collision, the paths that are found are indeed feasible at that moment. Path properties like travel distance can also be easily evaluated using a specific metric. Lastly, progress information on path traversal is available in the form of the time spend and remaining. Path search proceeds in only a few frames, therefore only task queue information is provided.

For *guiding*, the system should be able to guide a pawn using low-level motion control, based on a path described by start and end location and route properties, and a character described by its state and abilities. This proceeds in a similar fashion as we have seen with perception, except that the path is now latently traversed by the pawn. This is done by sequentially executing the actions of the path, while checking the pawn's progress on these actions.

The system is able to autonomously interpret the environment if it is provided with a pawn's ability. It is able to perform this in real time in the Unreal Editor. In game time, the system is able to use this interpretation to construct paths.

Semantics

The semantic language as described in Section 4.2 has been realized in our implementation. New derived semantic types can easily be added to the top layer of the navigation system, requiring only the implementation of a few virtual functions.

The Place class was added in a late stage of development. Because the Place class was added to the system's middle layer, it had a much greater impact on the system than adding for example a new type of metric. Introducing the concept of place, required modifications to many classes. Remarkable though, the implementation became way more manageable; interfaces became smaller, and the controller and navigator's code more transparent, while the rest of the implementation practically remained the same. This showed the value of separating implementation from semantics with respect to extensibility.

User

Both the level designers and the AI programmer at Coded-Illusions were asked several times to comment on the development of the navigation system.

Designers were literally amazed by the fact that geometry can also be understood by a computer, in contrast to their tedious efforts to place path nodes manually. Adding the fact that this process can run while they edit the level, only that feature was said to be worth the project.

Another preferred requirement was dynamic obstacle avoidance, which was not fully implemented in our implementation, but functional nonetheless. From a semantic standpoint, this feature was discarded given the time frame for this project. Dynamic obstacle avoidance is further discussed in Section 7.3.

The AI programmer particularly appreciated the fact that navigation had become independent of the surface orientation. Before, creating a spider-like pawn required a lot of low-level modifications, to make the spider only slightly aware of its surroundings. Getting a spider to also find short paths has never been possible. In our implementation, creating a spider requires the same amount of effort as the creation of a human pawn.

The fact that paths are not searched for instantly, requires a programmer to think in an unusual new way. Instead of deciding things at the moment they appear, one now has to assemble a plan based on the information provided by the navigation AI. Although this forces a slight restructuring of the current AI, it allows much more complex path searches and dynamic response, which is recognized as a fair trade-off.

Unreal

The navigation AI has been built into Unreal with a minimum of modifications on Unreal's side. Apart from the fact that modifications in the engine can have unpredictable consequences, it also allows the system to operate more

independently, which improves its applicability to other game engines or AI systems.

Our navigation AI requires the following game engine functionality:

- A list of primitive components that make up geometry.
- Collision testing for volumes and rays.
- Low-level pawn motion control.
- Tick function, allowing a fixed amount of processing time each frame.
- Serialization of one Mesh instance per game level.
- Drawing facilities.

The Unreal-related problems presented in Section 5.1, like general alignment, offline physics simulation, and non-transparent semantics, have been solved to the extent of the requirements of our design. New features might result in more semantic wrappings around Unreal data. This problem is practically impossible to overcome, because it would require a complete redesign of the engine's semantic structure.

Another problem that remains in our implementation is the semantic distance between our system and the Unreal engine. The fact that our system has only a few connections to Unreal, as discussed in the first paragraph of this section, results in a remote cooperation between the two. The semantic distance refers to the fact that both systems have a very poor mutual understanding, which makes it hard to communicate in an efficient way. For example, when a pawn gets stuck while traversing a path, the navigation AI has no better option than to inform the navigator of this event, which in turn has no better option than to reattempt a path search and traversal. Better understanding might have let the navigation AI to step back, and continue the path in a slightly different way.

7.2 Project

During the project, the understanding of semantics and navigation have evolved into the content of the foregoing chapters. This section focuses on the project itself, on the process of development, and on the extent to which we met our goals.

Phases

In advance to this project, research was conducted on semantic navigation in video games [10]. This research founded the first ideas on how to represent semantics in software and which design aspects should be taken into account when designing and implementing a semantic navigation system.

The first phase of this project was dedicated to investigating the functioning of Unreal's own navigation system. The structure of this system was perceived as modest, both from a semantic and implementation perspective, which resulted in the decision to build a new navigation system, instead of modifying the current system.

In the second phase, we identified the fundamental set of semantic classes, namely the Space, Link, Metric, and Ability classes. We built a prototype system inside Unreal, with these classes to become familiar with the engine, and test the value of a semantic approach based on classes. This prototype included functionality for navigation mesh generation, A* graph search, and rudimentary motion control. This prototype immediately proved the value of rich spatial semantics and the positive effect of a semantic approach on the structure of the system.

In the final phase, we designed the top layer classes (see Section 5.3) and made the current implementation. This phase started with the design of a semantic language by specifying the classes and their interfaces. The implementation shows that one can quite accurately design the class structure of the system in advance. Their interfaces, on the other hand, do not only depend on the semantic design, but also on the algorithms that are used in the implementation, which stresses the importance of making a system implementation.

Unreal

Programming a system inside Unreal Engine 3 has proven to be difficult without much experience or knowledge about its functioning below the surface. About half of the time that was spent on implementing the prototype and the final system implementation, was dedicated to solving problems that had no direct relation to the techniques discussed in Chapter 6. Because documentation on Unreal and especially its AI is scarce, one is required to actively study the engine's source code in order to understand the purpose of variables and functions, to investigate a program crash, or to find out how to implement a system inside Unreal.

In our experience, Unreal proves to be not really suitable for academic work, because it has a steep learning curve for using basic techniques like motion control, scene management, and custom drawing, while most of the advanced techniques are irrelevant to the strong focus academic work generally has. The size of the engine results in long compile and startup times, and slow debugging, which also makes it less suitable for prototyping in general.

For designing games and for developing game software, the Unreal Engine 3 is very suitable. It offers a broad set of tools and bears a lot of experience in its design and implementation. Although developing for Unreal is quite a challenge, it provides a very diverse and applicable environment for designing game software.

Goals

In Section 1.2, we presented a list of project goals to substantiate and validate our work. We will shortly revisit these goals in order to assess to what extent these have been met during this project.

1. We wanted to understand the importance of semantics in a navigation system and to find ways to classify and represent semantics. We have defined navigation in many contexts and presented a number of approaches in various software distributions. An extensive classification of semantics in video games has been provided. A selection of this semantics has been built into our system.

We can now summarize the importance of semantics of navigation stating that it provides a way to structure navigation information and to understand how this information is interpreted. This results in a better understanding by programmers of concepts that reside in navigation software. It makes the systems more independent of the algorithms and programming techniques that are fashionable at the moment.

2. We identified the actors that interact with our navigation system. The users, which are level designers and AI programmers, were interviewed to find out how they currently deal with navigation and what functionality they would like a navigation system to provide. Many of these requirements have been satisfied in our design as discussed in Section 7.1. Others will be discussed in Section 7.3.
   Unreal was carefully studied, both the way it currently provides navigation and the constraints that is poses on a new navigation system. The interface that connects our system with Unreal has remained basic, aiming for minimal involvement. This makes our system more robust and independent of the engine, although at the cost of excluding some performance optimization and having to deal with semantic alignment

3. We devised a semantic approach leading to the design of a navigation system given the requirements. This approach includes a class-based language, a layered architecture, and class interfaces that distinguishes different semantics.
   The design has an open top layer to easily incorporate new semantics based on existing types. These existing types mostly restrict the language, making it unambiguous and comprehensible. Still it is rich enough to issue the navigation instructions described in the requirements.

4. We built an implementation to evaluate our design. This has put in evidence the benefits and drawbacks of a semantic approach. We benefit with how it provides structure on where and how to interpret information. This leads to sustainable architecture that is invariant to the problem solving routines that it contains.
   The drawbacks are mainly related to the way we were used to a performance- and hardware-oriented design. Fundamental semantics and good performance do not go well together in all areas. However, our implementation has similar performance as the original navigation system, being able to route the same amount of character in similar environments.

7.3 Future work

Although our work on the design and implementation of a navigation AI has satisfied most requirements, there remains a large set of problems that cannot be solved yet or could be solved in a more efficient way by extending the current system. This section presents a set of problems and shortcomings in the current system design and implementation and suggests possible solutions.

7.3.1 Semantics

Driving and flying

While most games within our scope are designed around walking characters, the ability to drive a vehicle or fly are often used to enrich gameplay. It would be interesting to explore how we can enable AI characters to intelligently make use of the ability to drive vehicles or fly.

The current implementation supports characters that move over ground, which includes vehicles. There is however a number of aspects that should be taken into account. First of all, vehicles are much larger than humans, and therefore require a completely different mesh to navigate on. Because vehicles can cover large areas in a short time, many tiles are required to support a short drive. Therefore we have to generate large tiles, or introduce a new kind of space that represents long driving lanes. If the same collision meshes are used for vehicles and humans, large tiles are probably difficult to create. Adding the fact that designers often desire more control over vehicle paths, the latter solution seems more appropriate.

A second vehicle-related problem is the fact that it cannot strafe, i.e. move sidewards. Human path search does not need to take into account the orientation of the body, since a human can change its orientation at any location. Vehicles, instead, require space to turn and should prepare for this turn in advance. Either the vehicle should only traverse paths that are wide enough to turn, or the graph search requires an additional orientation dimension besides its two dimensional position. In large open terrains, the former solution can be applied successfully, especially if the before mentioned lane spaces are connected in a smooth way. The latter solution would have a very large impact on our implementation because it involves the rewriting of the graph search algorithm [38].

In order to fly along near-optimal paths, one cannot rely on a surface representation like tile spaces. Especially in environments with many ground obstacles, the distance on the ground is not a good measure for the distance through the air. Therefore flying requires a graph of volume spaces which fill the volume of the air like tiles tessellate the surface. The main difference with tiles is the way these volume spaces are created. While tiles require only a 2D rasterization and basic tile contour smoothing, 3D rasterization requires more samples and the analysis of 3D shapes is much more complex. On the other hand, most flying characters do not need a detailed path, and therefore require only coarse tessellation of 3D space.

Our system does support versatile characters that drive or fly very well. The concept of spaces and links is widely applicable and does not require fundamentally new semantics. Introducing military vehicles and flying dragons does mainly involve a lot of implementation work, filling in the new interpretation of the semantics that is already present.

Time and prediction

The prediction of future events is more the art of guessing than the exact foreseeing of future events. Because it involves a lot of pattern matching and case-specific learning, human beings are able to accurately predict detailed object

motion a few seconds ahead or coarse motion of an enemy based on earlier gaming sessions. An AI character, however, is mostly unable to even predict one frame ahead, forcing it to constantly anticipate on events that have passed, being unable to dynamically prune improbable future events.

Human characters will always be unpredictable, but also the prediction of game physics is far too costly to be applied in games any soon. The more unpredictable the elements are in the environment, the less accurate its current state is for prediction of the future. For these reasons, dynamics will always limit the AI's performance, forcing AI characters to operate outside the influence sphere of dynamic elements.

One thing the AI is able to predict very well is the motion of the characters that it controls. Especially in the area of cooperation, this inclusion of time into path search can be very useful. We could, for example, create a measure that weighs spaces that are probable to be occupied soon, like tiles on a path, or tiles in front of a character. But because this does not take into account the time a character is expected to be at that space, such a metric can only be of limited use, and possibly even confuse the AI. A much better solution would be to make a metric also time-dependent. Because we have accurate knowledge of the times AI characters will be at a certain spot, we can measure space occupancy. This will require the graph search algorithm to be extended to accumulate time as it searches the graph, which is relatively easy since it already accumulates the path metric measurements. An example involving time-dependent metrics was already presented in Subsection 3.3.4.

7.3.2 Performance and optimization

Path sampling

Our system precomputes a significant amount of information in the sense that much processing is done before we are sure that it will be used. An advantage of this preprocessing, is that we can schedule computations for periods when resources are cheap, such that the final assembly of information can proceed on demand and very quickly. A typical example of this is the use of sampling for creating tile spaces and sampled links, which extensively prepare a very fast path creation. When a path is requested, we only need to select the spaces that form the topological path and one action per link to create a fully smoothed path. Because spaces, links, and actions are up to date with the current state of the environment, we can immediately assume that this path is also feasible.

However, this very fast and detailed path search has its costs. First of all, not all precomputed data will be used often or even once, therefore some computations are wasted. Secondly, this data has to be stored over a long period of time. And thirdly, in a dynamic environment it holds that lots of preprocessed data require also lots of maintenance. We could say that our way of sampling is a rather luxurious technique. It is for these reasons that extensive preprocessing of samples has not been used often in existing AI systems, which are mostly restricted by a tight budget.

To avoid unnecessary preprocessing, we can postpone the sampling of spaces, links, and actions until they are required. For example, we can sample actions on a link at the moment the link is contained in a path or even wait until the pawn is near enough that samples are needed for local path smoothing. Spaces

could be sampled at the moment a graph search visits a neighboring space. The drawback is that the first paths can take a lot of time to compute.

To have a smaller memory consumption, we can discard sample information after use. In case of a sampled link, we keep the link, but remove the sample data. The problem here is that we have to do all processing at run-time, thereby proscribing certain algorithms that simply are not fast enough. In case of a jumping action, we cannot search for clever jumps when a path should be found within a second of time. Note that completely avoiding sampling is not an option, since we have to find an explicit path at some point, which is in fact a sample out of the set of all feasible paths.

Our approach allows us to apply complex geometry analysis and optimal action sequencing resulting in fast and high quality paths. It also allows us to accurately control path quality by adjusting sample rates. It is most important to sample in a purposeful and efficient way and avoid the sampling of tile spaces and sampled links that remain unused.

Multi-core

Most modern personal computers and game consoles have been equipped with a multi-core system. While game developers are working hard to rewrite their software for parallel processing, there is still a lot of processing power left unused by recent games. Although our implementation is not designed for a multi-core architecture, the techniques we apply are fit for such an approach.

When sampling, many independent samples are drawn from a fixed sample space. This independence of samples makes sampling very well suitable for a parallelized approach. Sampling locations in a tile space or actions on a sampled link for a human pawn (1,8 meter tall) requires about one hundred samples per square meter, which calculations can be distributed over different processing units. This will require the current sampling algorithm to be modified, but only on a low level.

The latent processing of geometry analysis and path search has been a first step towards parallel execution of tasks. Since the higher AI is not able to instantly retrieve information from the navigation AI, the decision making of the higher AI has become more independent of the processing of the navigation AI. In our implementation, there is no guarantee about when a task will be completed, and in what order this will occur. Therefore, we can distribute tasks to different cores, without requiring the higher AI to further adapt.

Both geometry analysis and path search, which are currently the most CPU consuming activities, can largely benefit from a multi-core approach. This will significantly improve the scalability of our system, as CPU resources are currently the main performance bottleneck.

Tile density

Our implementation is built around the idea that we have units of space. For example, to define a metric place we measure all tiles and select those that have the properties we seek. Here we assume that the environment inside one tile is homogeneous with respect to the metric. In practice, however, no part of the environment is guaranteed to be homogeneous, therefore we patch the environment with tiles that are small enough to allow accurate measurements.

The result is that we create more separate tiles than is required for the given geometry complexity, because we also want to measure cover for each tile, which requires them to be small in order to be accurate. For indoor environments, this approach works reasonably well, however for large open terrains, this could result in a too large amount of tiles.

One solution to this problem is to keep the idea that tiles are units, but to create small tiles only in places that require them. Returning to the open terrain, we can patch the open parts with large tiles, because it is also homogeneous to cover. Only around obstacles, like trees and rocks, we create a more dense set of tiles.

Another solution would be to discard the idea that a tile is a unit of space. Instead, we could make tiles purely geometry-based and further subdivide them to support more detailed measurement, e.g. for cover. This way, we can dynamically create smaller units that locally provide a more accurate spatial representation.

Obstacle avoidance

Our system has no special facility that handles collision avoidance with obstacles in the environment. If the obstacle is a piece of static geometry, the navigation mesh should be built around it and therefore the path will go around it too. Although the avoidance of dynamic geometry is currently not supported, it can be easily implemented based on the local rebuilding of the navigation mesh to account for a new obstacle. Currently, however, this process is relatively slow, as it might take up to five seconds. In addition, it temporarily blocks all paths in the proximity, because spaces, links, and actions are marked dirty during that time.

If we want to deal with obstacles that move often, we should not rely on the navigation mesh. This process will remain slow in the order of seconds, because it uses complex collision traces to provide high quality surface patching. Avoiding such objects requires a local dynamic routing system that moves with the obstacle. Such a system can be based on the same space-link concept as we used to create a navigation mesh, but without the overhead of latent updating.

One could for example introduce a waypoint-based system which builds a small circular waypoint network around the obstacle, allowing passing pawns to shortly insert a small path on this network to move around the obstacle. If multiple obstacles overlap, these individual networks should merge into one to avoid looping, i.e. avoiding obstacles without passing them. The highly dynamic nature of such a system has its impact on the slower approaches found in our current implementation. When avoiding collision, we still expect the character to move along a smooth path, which can only be computed after we started traversing the path. Still we can use sampled links to smooth the path, and avoid additional collision checks.

Chapter 8

Conclusion

Our navigation system proves to be a useful addition to navigation in Unreal. The semantic approach has led to a rich and clear semantic structure with respect to designers, programmers, Unreal, and system resources. A strong relation between semantic structure and system architecture provides an extensible design and unambiguous class interfaces. The focus on the interpretation of information has led to a more natural interaction between the navigation system and the user. A layered architecture separates general semantics from implementation-dependent semantics, which allows the system to use other problem solving techniques, without the need to redesign this general semantics.

The system allows users to deal with navigation in a more natural and intuitive way. It relieves designers from the task of providing semantically redundant navigation hints. This enables them to focus on their task of designing a game environment, while they are provided with direct and predictable navigation feedback on their creations.

Navigation has been separated from the higher AI, which improves robustness and overview of the system, where the higher AI is less dependent of the low-level routines that run the navigation. High-level AI behavior can be designed using high-level concepts, which allows programmers to have a natural understanding of the concepts that constitute a character's behavior, and design for higher-level behaviors without being distracted by low-level issues.

A semantic approach is not directly related to a good system performance. From a semantic perspective, pure performance optimization is unbalanced as it only applies to the semantics of low-level routines. Our system offers a number of optimizations that make it suitable for application in games, however not at the cost of losing essential semantics for the higher-level AI and level designers.

Well-defined semantics is often application-specific, making it hard to devise one model that is widely applicable. Often, more detail means less application, which demands a system that can be tailored to the game at hand. Our system has an open ended structure, where game-specific semantics can easily be added.

The gaming industry has been dominated by technology-centered development with a strong focus on hardware and performance. In this, a semantic approach to system design is rather unusual. A large share of the interpretation that is performed in our navigation system, can be of a greater value when

performed inside the Unreal engine. As a final speculation, I pose that to experience the benefits of a semantic approach to the full extent, one should consider a semantic update for the Unreal engine as a whole.

Bibliography

[1] AI.implant. AI middleware for high-end game development www.ai-implant.com, 2007.

[2] AIseek. Intelligence for new worlds. Technical report, AIseek Ltd. www.aiseek.com, 2005.

[3] S. Massoud Amin, Emin Y. Rodin, Michael K. Meusey, Travis W. Cusick, and Asdrubal Garcia-Ortiz. Evasive adaptive navigation and control against multiple pursuers. In *American Control Conference, Proceedings of the 1997*, volume 3, pages 1453–1457, 1997.

[4] Christian Baekkelund. Academic AI research and relations with the games industry. In *AI Game Programming Wisdom 3*. Charles River Media, 2006.

[5] Yngvi Björnsson, Markus Enzenberger, Robert Holte, and Jonathan Schaeffer. Fringe search: Beating A* at pathfinding on computer game maps. In *Proceedings of the IEEE Symposium on Computational Intelligence in Games (CIG'05)*, pages 125–132, 2005.

[6] Mark Brockington. Pawn captures wyvern: How computer chess can improve your pathfinding. In *In Proceedings of the 2000 Game Developer's Conference*, 2000.

[7] Michael Buro and Timothy Furtak. RTS games as test-bed for real-time research. In *Invited Paper at the Workshop on Game AI*, pages 481–484, 2003.

[8] John F. Canny. *The Complexity of Robot Motion Planning*. MIT Press, 1988.

[9] Edsger W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, pages 269–271, 1959.

[10] Leonard van Driel. Towards semantic navigation in video games. Master research assignment report, 2008.

[11] Chris Fairclough, Michael Fagan, Brian Namee, and Pádraig Cunningham. Research directions for AI in computer games. In *Proceedings of the 12th Irish Conference on AI and Cognitive Science*, 2001.

[12] Fredrik Farnstrom. Improving on near-optimality: More techniques for building navigation meshes. In *AI Game Programming Wisdom 3*, pages 113–128. Charles River Media, 2006.

[13] Kenneth D. Forbus. Some notes on programming objects in The Sims. 2001.

[14] Kenneth D. Forbus, James V. Mahoney, and Kevin Dill. How qualitative spatial reasoning can improve strategy game AIs. *IEEE Intelligent Systems*, 17:25–30, 2002.

[15] Malcolm R. Forster. How do simple rules 'fit to reality' in a complex world? *Journal Minds and Machines*, 9(4):543–564, 1999.

[16] Epic Games. Unreal engine 3, www.unrealtechnology.com, 2005.

[17] Ross Graham, Hugh McCabe, and Stephen Sheridan. Pathfinding in computer games. *ITB Journal*, 2003.

[18] Ross Graham, Hugh McCabe, and Stephen Sheridan. Neural networks for real-time pathfinding in computer games. In *Proceedings of ITB Research Conference*, pages 22–23, 2004.

[19] Ross Graham, Hugh McCabe, and Stephen Sheridan. Realistic agent movement in dynamic game environments. In *Proceedings of DIGRA 2005 Conference: Changing Views – Worlds in Play*, pages 249–259, 2005.

[20] Millennium Interactive (Steve Grand). Creatures, 1996.

[21] Roger J. Hubbold and M. J. Keates. Real-time simulation of a stretcher evacuation in a large-scale virtual environment. *Computer Graphics Forum*, 19(2):123–134, 2000.

[22] Tng Cheun Hou John, Edmond C. Prakash, and Narendra S. Chaudhari. Team AI: probabilistic pathfinding. In *CyberGames '06: Proceedings of the 2006 international conference on Game research and development*, pages 191–198, 2006.

[23] F. Markus Jönsson. An optimal pathfinder for vehicles in real-world digital terrain maps. Master's thesis, The Department of Numerical Analysis and Computing Science, The Royal Institute of Science, Stockholm, Sweden, 1997.

[24] Marcelo Kallmann. Interaction with 3D objects. In *Handbook of Virtual Humans*. John Wiley & Sons, 2004.

[25] Oussama Khatib. Real-time obstacle avoidance for manipulators and mobile robots. *Int. J. Rob. Res.*, 5(1):90–98, 1986.

[26] Sven Koenig and Maxim Likhachev. Real-time adaptive A*. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 281–288, 2006.

[27] Richard E. Korf. Depth-first iterative-deepening: an optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.

[28] John E. Laird. It knows what you're going to do: adding anticipation to a Quakebot. In *AGENTS '01: Proceedings of the fifth international conference on Autonomous agents*, pages 385–392, 2001.

[29] John E. Laird and Michael van Lent. Human-level AI's killer application: Interactive computer games. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, pages 1171–1178, 2000.

[30] Lars Lidén. Using nodes to develop strategies for combat with multiple enemies. In *In Artificial Intelligence and Interactive Entertainment: Papers from the 2001 AAAI Symposium*, pages 59–63, 2001.

[31] Lars Lidén. Strategic and tactical reasoning with waypoints. In *AI Game Programming Wisdom*. Charles River Media, 2002.

[32] Lars Lidén. Artificial stupidity: The art of intentional mistakes. In *AI Game Programming Wisdom 2*. Charles River Media, 2003.

[33] M. Lozano and J. Molina. A neural approach to an attentive navigation for 3D intelligent virtual agents. In *Systems, Man and Cybernetics, 2002 IEEE International Conference on*, volume 6, page 5, 2002.

[34] Jesús Ibáñez Martínez and Carlos Delgado Mata. A basic semantic common level for virtual environments. In *The International Journal of Virtual Reality*, volume 5, pages 25–32, 2006.

[35] Syrus Mesdaghi and Noel Stephens. The 2005 report of the IGDA's artificial intelligence interface standards committee. Technical report, IGDA www.igda.org/ai/report-2005/pathfinding.html, 2005.

[36] Lionhead Studios (Peter Molyneux). Black & White, 2001.

[37] Alexander Nareyek. Computer games - boon or bane for AI research. *Künstliche Intelligenz*, 2:43–44, 2004.

[38] Marco Pinter. Toward more realistic pathfinding. *Gamasutra*, 2001.

[39] Charles Pisula, Kenneth E. Hoff, Ming Lin, and Dinesh Manocha. Randomized path planning for a rigid body based on hardware accelerated voronoi sampling. In *Workshop on Algorithmic Foundations of Robotics*, 2000.

[40] Pierre Pontevia and Alain Baur. Kynapse 4.0 large scale AI. Technical report, Kynogon, 2006.

[41] Dave C. Pottinger. Terrain analysis in realtime strategy games. In *Game Developers Conference Proceedings*, 2000.

[42] Douglas A. Reece, Matt Kraus, and Paul Dumanoir. Tactical movement planning for individual combatants. In *Proceedings of the 9th Conference on Computer Generated Forces and Behavioral Representation*, 2000.

[43] Bjørn Reese and Bryan Stout. Finding a pathfinder. In *Proceedings of the AAAI Spring Symposium on Artificial Intelligence and Computer Games*, 1999.

[44] Adam Russell. Turning spaces into places. In *AI Game Programming Wisdom 4*. Charles River Media, 2008.

[45] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[46] Kynogon S.A. Kynapse SDK release 3.8, 2006.

[47] Brian Salomon, Maxim Garber, Ming C. Lin, and Dinesh Manocha. Interactive navigation in complex environments using path planning. In *I3D '03: Proceedings of the 2003 symposium on Interactive 3D graphics*, pages 41–50, 2003.

[48] Jonathan Schaeffer and H. Jaap van den Herik. Games, computers and artificial intelligence. *Artificial Intelligence*, 134(1-2):1–8, 2002.

[49] Valve Software. Half-Life, 1998.

[50] William van der Sterren. Terrain reasoning for 3D action games. In *Game Developers Conference Proceedings*, 2001.

[51] Bryan Stout. Smart moves: Intelligent pathfinding. *Game Developer*, 1996.

[52] Paul Tozour. Building a near-optimal navigation mesh. In *AI Game Programming Wisdom*, pages 171–185. Charles River Media, 2002.

[53] Adrien Treuille, Seth Cooper, and Zoran Popović. Continuum crowds. *ACM Trans. Graph.*, 25(3):1160–1168, 2006.

[54] Tim Tutenel, Rafael Bidarra, Ruben M. Smelik, and Klaas Jan de Kraker. The role of semantics in games and simulations. *ACM Computers in Entertainment (TBP)*, 2008.

[55] Jean-Paul van Waveren and Leon J. M. Rothkrantz. Automated path and route finding through arbitrary complex 3D polygonal worlds. In *Elsevier, Robotics and Autonomous Systems*, volume 54, pages 442–452, 2006.

[56] Dustin White. Clarifications and extensions to tactical waypoint graph algorithms for video games. In *ACM-SE 45: Proceedings of the 45th annual southeast regional conference*, pages 316–320, 2007.

[57] Billy Yue and Penny de Byl. The state of the art in game AI standardisation. In *CyberGames '06: Proceedings of the 2006 international conference on Game research and development*, pages 41–46, 2006.