Charles University in Prague

Faculty of Mathematics and Physics

BACHELOR THESIS

Michal Klein

Coordinated Movement of Virtual Agents in Unreal Tournament 2004

Department of Software and Computer Science Education

Supervisor of the bachelor thesis:	Jakub Gemrot
Study programme:	Computer Science
Specialization:	Theoretical Computer Science

Prague 2014

I would like to thank my supervisor Jakub Gemrot for help throughout my work on this thesis.

Also I would like to thank the whole AMIS group for regular consultations.

Last but not least I want say thank you to my family for continuous support.

Název práce: Koordinovaný pohyb virtuálnich agentů v Unreal Tournament 2004

Autor: Michal Klein

Katedra: Kabinet software a výuky informatiky

Vedoucí bakalářské práce: Mgr. Jakub Gemrot, Kabinet software a výuky informatiky

Abstrakt: Pohyb ve formacích je typ koordinovaného pohybu, který udržuje virtuální agenty v striktním, předem určeném tvaru. Tato práce se zabývá vytvořením robustního parametrizovatelného skupinového navigátoru, který je schopen navigovat skupinu distribuovaných agentů v rámci 3D prostředí počítačové hry Unreal Tournament 2004. Navigátor je schopen udržet formaci navzdory překážkám prostředí a šířky chodeb. Před psaním této práce nebyl žádný způsob, jak kontrolovat a navigovat virtuální agenty jako skupinu. Toto dává uživatelům Pogamutu platformu, která umožňuje vytváření a řízení virtuálních agentů, provozování tohoto navigátoru pro týmové úkoly nebo další experimentování s touto technologií. To jim dává možnost vytvářet své vlastní vzory formací a použít je se skupinovým navigátorem.

Klíčová slova: virtuální agenti, koordinace, pohyb ve formaci

Title: Coordinated Movement of Virtual Agents in Unreal Tournament 2004

Author: Michal Klein

Department: Department of Software and Computer Science Education

Supervisor: Mgr. Jakub Gemrot, Department of Software and Computer Science Education

Abstract: Formation movement is a type of coordinated movement that keeps virtual agents in a strict, prespecified shape. This thesis deals with creating a robust, parametrized group navigator, that is capable of navigating a group of distributed agents through a 3D environment of a computer game Unreal Tournament 2004. The navigator is capable of maintaining the formation in spite of environmental obstacles and a corridor width. Before writing this thesis, there was no way to control and navigate virtual agents as a group. This gives users of Pogamut, a platform that allows creating and controlling virtual agents, to operate this navigator for team tasks or experiment with this technology further. It gives them the ability to create their own formation patterns and use them with the group navigator.

Keywords: virtual agents, coordination, movement in formation

Contents

Introduction

In the thesis we'll be working with intelligent virtual agents, IVAs in short. Their primary goal is nothing less than to approximate behavior of living organisms (often humans), performing specific tasks.

They are used for simulations in many field, ranging from sociology through military to economics. They are used to entertain as well, whether be it in film industry or game development.

Research of IVAs is closely tied to artificial intelligence, an area of computer science, where researchers always aim to further our understanding. Nowadays there are plenty of different approaches to developing IVAs, each with its pros and cons, tailored to specific types of IVAs.

As a consequence of their heavy usage, they've been getting ever smarter, capable of more realistic and complex portrayals of behaviours.

Goals

The goal of this thesis is to investigate various options how the group of IVAs can move through the environment of Unreal Tournament 2004 maintaining a prespecified formation and to implement a robust group-navigator that will handle the movement regardless of the path constraints. The most importantly, the group-navigator will be able to handle navigation through both wide areas and narrow corridors and also through paths requiring jumping.

The group-navigator will be distributed. There will not be a master director that would move with all IVA bodies belonging to single group directly. Rather, IVAs will move for themselves, accepting commands from the group owner that will be the one dictating the formation shape. The owner will be free to alter or dismiss this formation at any given time.

1. Motivation

Artificial intelligence has always played a role in gaming industry. It used to get only a tiny bit of computing power, partly due to games designed back then being less reliant on AI, but also because of hardware limitations. With better hardware and new gaming genres (such as real time strategy games), developers realized that putting even more resources into graphics started yielding diminishing returns. It was only natural to realize that smarter opponents would give players a superior gaming experience and replay value, thus giving the developers an edge over their competition.

It didn't take long for developers to start experimenting with coordinated behavior of virtual agents, which in gaming industry are often called bots. This area encompasses a plethora of often overlapping sub-problems, such as squad tactics, coordinated movement and communication between agents. Increasingly games require agents to move in a coordinated fashion. This can happen at two levels. Characters can choose actions that compliment each other, making it resemble coordinated movement. Or they can make decisions in unison and move as a coordinated group. The latter is called *formation motion.* Here the agents move through an environment as a cohesive group, trying to keep a specific rigid shape, such as a *wedge* or *line* formations[5], both shown in Figure 1.1. It's not limited to this, though. Agents can utilize the environment, depending on the game, for tactical benefits, e.g. taking cover.

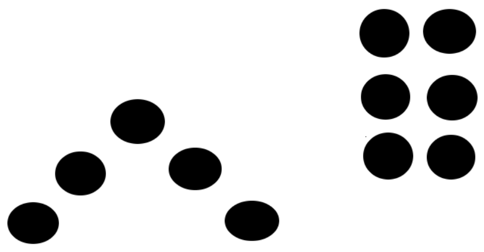

Figure 1.1: Wedge and column formations

The topic of this thesis is formation movement in its purest form. Agents navigating a rich 3D environment, trying to maintain predetermined shape. Furthermore, agents will be distributed, which means protocol for communication between the agents will have to be designed.

2. Background

2.1 Pogamut Framework

Pogamut is a Java middle-ware that enables controlling virtual agents in multiple environments, provided by game engines. Currently Unreal Tournament 2004, UnrealEngine2RuntimeDemo, Unreal Development Kit, and DEFCON games are supported [1]. Pogamut provides a Java API for spawning and controlling virtual agents and GUI that simplifies debugging of the agents. Its main objective was to simplify the "physical" part of the agent creation. Most actions in the environment (even complex ones like path finding) can be performed by one or two commands. This enables user to concentrate his efforts fully on the interesting parts of the development process.

It's being developed by a group of computer scientists at Faculty Of Mathematics and Physics, Charles University[1]. Pogamut toolkit integrates five main components, as seen in Figure 2.1: Unreal Tournament 2004, GameBots2004, the GaviaLib library, Pogamut agent and the IDE[2].

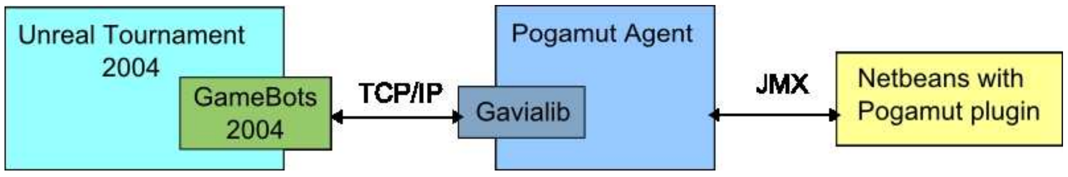

Figure 2.1: Five main components of the Pogamut project[2]

Unreal Tournament 2004 is a first person shooter video game developed by Epic Games and Digital Extremes. It features simulated 3D environment with a variety of locations, ranging from space stations and forests to small towns and meadows. More importantly, it has its own scripting language, *UnrealScript*[2]. It is a native scripting language developed by the creators of Unreal Tournament 2004. Aside from graphics and physics engines, almost everything is programmed using UnrealScript.

It's possible to write agent code in UnrealScript, however it's often prudent to use other tools for the purpose. That's where *GameBots2004* comes into play. It creates a binding to the Unreal Tournament 2004 and exports useful data from the game. Users can connect to the GameBots2004 using a TCP/IP connection and communicate using a text based protocol, where GameBots2004 is a server and user's agent is a client[2].

The *GaviaLib* library is a general purpose Java library for connecting agents to almost any virtual environment. It communicates with GameBots2004 using TCP/IP, controls agents' life cycle and allows for remote control of the agent via JMX, a standard Java protocol for remote instrumentation of programs[2].

The *Pogamut Agent* is a set of Java classes and interfaces built on top of the GaviaLib library. It adds Unreal Tournament 2004 specific features, such as its sensory motor primitives, the navigation using a system of waypoints and auxiliary classes providing information about the game rules[2].

The *IDE* is developed as a NetBeans plugin. It can communicate with running agents via JMX. It offers designer tools such as template agents and other tools for easier development and debuggin[2].

2.2 Navigating The Environment

Navigating through a complex 3D world, such as the one in Unreal Tournmanent 2004, poses a question as to how its environment should be modelled. Pogamut provides agents with a high level navigation module, *UT2004Navigation*. Merely by writing a one line of code the agent starts to navigate to the target location (2.2).

```
navigation.navigate(targetLocation);
```

Figure 2.2: Simple use of UT2004Navigation

UT2004Navigation is itself composed from multiple other modules. The most important ones are *Path planner* (with corresponding interface IPathPlanner) and *Path executor* (IPathExecutor). They cover two main steps Pogamut deals with navigation.

First, path from starting location to the target location has to be found, which is a role of path planner. It is provided with environment representation and an algorithm that computes the path. The path is a series of locations that can be traversed by going in a straight line between two locations. Second, given a path, agent is given instructions to follow the path. It is easy to realize that path planning and path executing is decoupled. Even path planner can use different algorithms and world representations.

2.2.1 Navigational Graph

There are several path planners in Pogamut using different algorithms. But we can look at two conceptually distinctive methods of path planning. One is using *navigation graphs* (Figure 2.3). The world is represented by a set of nodes connected by edges. This is a classic representation that has been used in games for a long time and has been in Pogamut since very beginning of the project. The main drawback of this method is that much of the information about the environment is lost to agents. Since the nodes of the graph, called *Navpoints*, are connected by a mere two dimensional edges, agent doesn't know how far he can deviate from the path because he lacks knowledge of the path's width. Another limitation is that while path planner will find shortest path on the navigation graph, in reality it's most likely not.

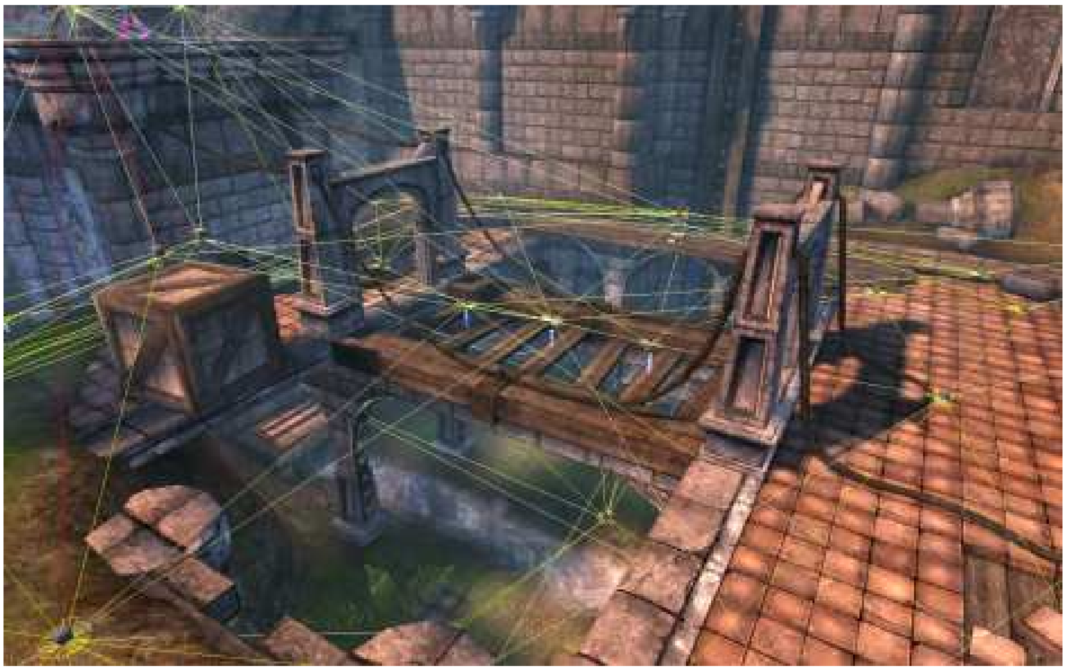

Figure 2.3: Navigation graph in Unreal Tournament 2004

2.2.2 Navigational Mesh

This brings us to the second representation of the environment, navigational mesh (Navmesh, Figure 2.4). All walkable area reachable by agents is divided into convex polygons. Agents can move freely within each polygon without any fear of hitting any environmental obstacles. You can also travel between polygons that share an edge. Special edges can connect two polygons to represent you can, for instance, just from one to the other. It's easy to realise that this approach is a qualitative improvement and solves aforementioned drawbacks.

Navmesh navigation was implemented into Pogamut only recently[3]. It is worth noting that the current implementation uses both navigation graphs and navmesh. That is because of imperfect navmesh generation. Some areas lack polygons and navpoints are used instead. This is something called *navmesh degeneration.*

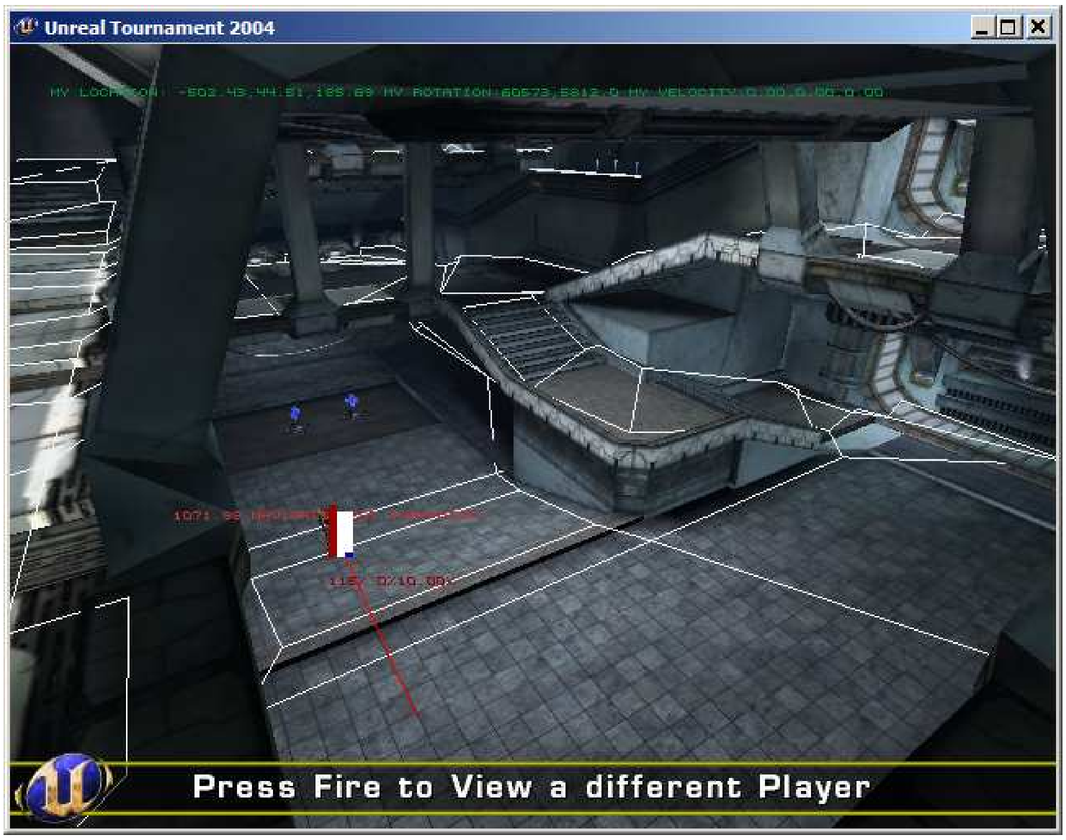

Figure 2.4: Caption

In order to use navmesh on a map (e.g CTF-LostFaith.ut2) we first need a file containing the navmesh data (CTF-LostFaith.navmesh). Creating this file is a three step process. First we extract geometry from the map using *UShock* tool. Then, using *UShochToRecast* we convert the geometry to a Recast friendly format. *Recast* is a program for generating navigational meshes from supplied level geometry. It creates the needed file for navmesh navigation to work in Pogamut.

2.3 Related Work

This chapter surveys previous work in the field of formation movement. Most importantly, how the formations should be represented and work internally.

2.3.1 Decentralized Approach

One way is to have decentralized formation[4]. There's no leader figure, meaning all formation members are equals. Everyone chooses his position in the formation based on where other agents are located. For example, if the formation has a V shape, every member would choose someone to be their target and try to follow him. If that target is already taken, he needs to find someone else.

There's no strict geometric pattern. The pattern itself emerges (also called emergent formations) from individual decisions, without outside coordination.

It is very difficult to define rules for keeping formations and it can lead to an unwanted outcome.

As with all emergent behavior, debugging can be very difficult.

Because they are all equal, they need to be capable to navigate themselves in the environment. This may not always be cheap, however it is very robust and doesn't need to deal specifically with obstacles in the environments, as they react to them individually.

This approach allows for easily scalable formations with easy adding and removing of members. Also, as they don't require supervision, this method can work with distributed agents.

Most importantly for this thesis, though, is that outcome often looks like organised disorder. When precise and strict group movement is required this approach is not a very fortunate choice.

2.3.2 Centralized Approach

Here, there's a leader figure, that navigates the whole formation. Other members, followers, are very simple, with limited information about the environment and primitive movement. That is the reason why this approach saves CPU power, but it makes it hard to use with decentralized agents[4].

Every formation member has a fixed position in the formation. They use vectors - offsets - to mark their position with respect to the leader's location. This guarantees strict and organized formation shape.

Leader should move more slowly than other members, so they can catch up when they fall behind. Another solution is to regulate speeds of all formation members[5].

The requirement of fixed positions brings a difficulty with scalability of the formation. For certain shapes it can be very difficult to add new members.

Strict fixed shape makes it also difficult to deal with obstacles and navigating through narrow areas.

2.3.3 Two-Level Approach

Two-level approach[4] brings both strict geometry and flexibility to formation movement. Everyone has an offset defining his position, however they are allowed to react to the environment, if necessary.

Like in centralized approach, there still is a leader who is responsible for steering the formation as a whole. Here, though, the followers are smarter and have their own navigation. It is still simpler than leader's navigation and thus saves some CPU power, but it can avoid collisions with the environment. Followers have their fixed positions as a target for their navigation.

Adapting this approach for distributed agents is much easier than with pure centralized approach, although, unlike with emergent formations, some changes have to be made.

3. Chosen Approach

3.1 Steering

Considering decentralized agents lack rigid coordination, their emergent steering isn't particularly good model for formation movement, which requires a strict adherence to positioning. Thus, a centralized solution seems to be the most plausible choice, with an arbiter controlling the shape of the formation.

The character of Pogamut agents needs to be taken into consideration. Every agent is autonomous and can be run from a different virtual machines. Their nature is in principle distributed, therefore classic centralized solutions with a puppet master controlling all agents' behaviors is not possible. Every agent needs to be able to think and act for himself.

Two level formation steering gives us an advantage of centralized control and emergent behavior. It can also be intuitively adapted for working with distributed agents, as already the model allows all agents to coordinate through communication, with each of them having control of what they're doing.

One of the ideas behind two level formation steering is that expensive path planning is done only once and bots use steering behaviors for arriving to their designated locations, computed using assigned off-sets. This saves some computing power. This thesis will try to hold to it, however, there are certain exceptions which will be explained later. Suffice it to say, we need every formation member to be capable of path planning. It's also because the agent should be generally indistinguishable from others, aside from the role he's currently playing in the formation.

As with all centralized approaches, a decision has to be made about the nature of the command structure. More lenient approach is to have a sort of coaching commands that server more as a recommendation than a directive. In this case, each agent would ultimately decide on his own what the best course of action is. This isn't suitable solution for formation movement as free agent movement would defeat the whole purpose of the formation. Instead, stronger, more authoritative command is a better fit for this task. Agents are given certain lenience in how they execute the commands but they must follow in the end.

3.2 Roles

As has been mentioned previously, all agents are equal in their capabilities. They can assume different roles, though. This chapter will try to explain further how typical leader and follower roles function in this work and mention some of their capabilities and responsibilities.

Leader

Leader is in control of navmesh navigation module. When a formation is told to go to a specified location, leader uses his location as a source location to compute a path to the destination. While on the move he sends update messages to other formation members containing his current location, velocity and other information useful for keeping the formation in its shape. He also handles messages received from followers and can adjust his movement, e.g. speed, to make it easier for them to keep up. This is just the core idea behind the leader's role. Specifics will be explained later in the paper.

Follower

Follower tries to stay in his position relative to the leader. He is provided with an offset which is used with information retrieved from leader's updates to calculate the relative position. He isn't using complex path planners as his path is determined solely by leader's positions. (There is an exception to this rule and will be explained later). Leader and Follower are called Formation Roles.

There is one more role an agent can play in the formation movement, albeit more indirect. For easier use of formations in the Pogamut environment it's good to have someone who is controlling the formation. It could be the leader but that would require the controlling agent be a part of the formation himself. It is a likely scenario that user would want to create and control the formation with one agent and have him do something else entirely, irrespective of the formation. For this purpose *Formation Commander* role was created. The commander can be a leader or a follower or neither. He can manipulate the formation and is aware of its state. This role is here as much for more convenient use as for decoupling responsibilities.

4. Design

In this chapter internal design will be examined. The purpose behind modules, how they communicate in Pogamut and how they can be brought all together to make agents be a part of formation will be shown here as well.

4.1 Modules

The project is composed of three main modules: *Formation Commander*, *Formation Manager* and *Formation Navigator*. The first two of them exist as tools for a Pogamut agent.

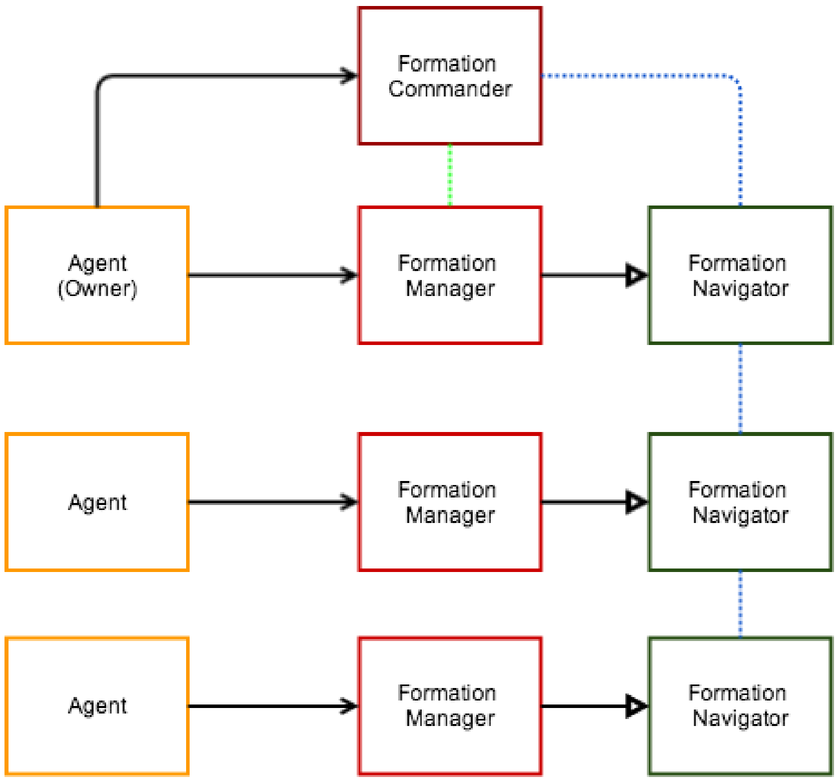

Figure 4.1: Agents, modules and their relations between each other. Full arrows mean ownership of a class instance, empty arrows mean that it can also be null. Green dotted line shows exchange of managing messages and blue dotted line express communication for navigational purposes.

Formation Commander, represented by *IFormationCommander* interface, corresponds to the equally named role. If the agent wants to create and control a formation, he does it by creating an instance of a class that implements *IFormationCommander* and by using its API. It allows the agent to invite bots to the formation and control it by sending move commands. Instance of this classed is owned only by someone who is commanding the formation this class represents. Thus the agent is sometimes called formation owner.

Formation Manager is represented by *UT2004FormationManager* class. Unlike formation commander, every single agent with formation movement capability owns an instance of this class. Its API allows agents know whether they were invited to a formation, which they can choose to join. Agents can use it to find out their formation role in the formation, if any. Basically they can manage their involvement with a formation from a client point of view.

Formation Navigator, represented by *UT2004Formation* class, is the very core of this project. Only an agent that is a member of a formation has an instance of this class. Note that it's hidden from user and is contained within UT2004FormationManager class. It knows all important data about the agent (e.g. his role) to navigate him as a part of a bigger formation group.

All these modules are interconnected. Formation Commander and Formation Manager exchange communication about joining and leaving the formation. Formation Commander controls the pattern and can adjust it per request form Formation Navigator. It also gets state updates from all members' Formation Navigators so it can relate to the owner if formation is executing or not.

4.2 Communication

For communication between agents we use UT2004TeamComm project that is a part of Pogamut framework. It is a server built over the PogamutUT2004 and Apache Mina project that allows bots to exchange arbitrary Java objects that are Serializable. Its heart is UT2004TCServer class that uses GameBots2004's Control Messages feature to let all connected bots know where the server is listening and defines protocol to exchange messages over global, team, custom team and private channels between bots.

All messages inherit *TCMessageData* class. For instance *TCInvite* class is used to send formation invites. Its constructor takes two arguments, UnrealId of the formation owner and channel ID, integer that identifies team channel in TeamComm server.

UT2004TCClient can then send the invite message to any agent connected to the server simply by one method call (Figure 4.2).

```
tcClient.sendToBot(receivingBotID, inviteMessage);
```

Figure 4.2: Sending invite message

Messages can then be received using event listeners, as shown in Figure 4.3.

```
@EventListener(eventClass = TCInvite.class)
public void handleInvite(TCInvite message) {
    // Handle Invite
}
```

Figure 4.3: Catching formation invite

4.3 Classes

This chapter shines some light on more important classe and their notable methods.

UT2004FormationManager

Life Cycle

Formation manager must be created before any formation invitations could come and should be alive until the bot leaves the game.

Responsibilities

- Allows agents to assess their role in a formation, if any
- Stores formation invites it received from UT2004FormationCommander
- Provides API for leaving a formation or accepting/rejecting an invite

Notable Methods

- *public FormationRole getRole()*

  Retrieves a bot's role in a formation

- *@EventListener(eventClass=TCInvite.class)*
  public void handleInvite(TCInvite inviteMessage)

  Stores an invite as last pending invite. User can choose to join, if he want to.

- *@EventListener(eventClass=TCKick.class)*
  public void handleKick(TCKick kickMessage)

  Bot was kicked from the formation and this method makes sure to reinitialize the class

UT2004FormationCommander

implements *IFormationCommander*

Life Cylce

Is created when the user wants to create a formation. It can be thrown away when not needed, however *disband()* method needs to be called or other agents won't know the formation was destroyed.

Responsibilities

- Gives full control over a formation.
- Sends invites and kicks to formation members.
- Manages formation pattern and allows for his change.
- Communicates with formation navigators of formation members.

Notable Methods

- *public void disband()*

  Disbands the formation and lets everyone in it know it's over.
- *public void moveTo(Location location)*

  Sends an order to move to a location.
- *public void invite(UnrealId bot)*

  Sends an invite to a bot.
- *public void formAt(Location location)*

  Sends an order to group up the formation at a location.
- *public void setPattern(IFormationPattern pattern)*

  Changes the pattern of the formation. Sends all necessary information to formation members.

UT2004Formation

Life Cycle

When bot joins a formation, new instance of this class is created and when he leaves or gets kicked it's thrown away.

Responsibilities

- Takes care of the whole of bot's navigation within the formation regardless of his role
- Communicates with the formation owner, letting him know the formation's state

Notable Methods

- *@EventListener(eventClass=EndMessage.class)*
  public void logickTick(EndMessage endMessage)

  Catches a notification message from the server every 250ms. Takes care of navigation, whether as a leader or a follower. If the bot is a leader it calls *SpeedRegulator*'s method for computing speed.

SpeedRegulator

Responsibilities

- Calculates and manages speed for every formation depending on how far behind they are.

Notable Methods

- *public void computeAndUpdateSpeeds(double distanceToNextLeaderLoc)*

  Uses data for delay of every follower (stored) and of the leader (in argument) to calculate and send out new speeds to all formation members.

RayCasting

Responsibilities

- Calculates opposing force for a supplied vector and origin.

NavMeshExtension

Responsibilities

- Contains methods that improve work with navmesh.

Notable Methods

- *public static boolean isOnSameLineWith(*
  NavMesh navmesh, ILocated loc1, ILocated loc2)

  Checks whether two locations lie on the same line on the navmesh.

- *public static Location getClosestPointOnNavMesh(*
  NavMesh navmesh, Location from, Location to,
  double minimalDistanceFromEdge)

  Finds closest point on the navmesh from *from* to *to*.

5. Formation Life Cycles

This chapter will briefly explain how bots get into a group called a formation, how they group up into prespecified shape and how it all can end.

5.1 Creating And Disbanding Formation

Let's look at the process of creating a formation. First an agent has to create a class instance of IFormationCommander interface.

Then he sends invites to others he wants have inside his formation using IFormationCommander's *invite(UnrealId)* method. During the creation of the Formation Commander, a team channel is created on a TeamComm server. By invoking an invite method a *TCInvite* message is created with formation owner's UnrealId and the channel ID.

Later the invite messages are caught by invited bots' formation managers, using event listeners, as was showed in the previous example. The last invite is stored and the bot can decide to accept last pending invite using *acceptInvite()* method of UT2004FormationManager which then sends back *TCInviteResponse* message with a flag saying whether it was accepted or rejected, the sender's UnrealId and the formation channel Id, in case formation owner owns other formations as well.

Formation Commander catches TCInviteResponse messages and in response sends *TCOffset* message containing bot's assigned offset which is now received by his UT2004Formation module.

After the invitation faze the owner has to pick pick a leader, which will send *TCSetLeader* message to formation members' UT2004Formation.

Destroying formation is done simply by calling *disband()* method of IFormationCommander. It should always be done before throwing away the object's reference.

5.2 Forming

Once formation is created agents can be spread out all over the map. The owner of the formation needs to use *formAt(Location)* method. It will send *TCForm* message requests to formation members that will tell formation navigator to move the bot to the specified location, adjusted by his offset. For this purpose regular navmesh path finding is used.

After the bot arrives to the destination he will send *TCReady* message, informing formation commander that he is ready to execute new order. When all bots are in their position formation commander will know that formation is no longer executing and can issue new orders.

6. Navigation

Formation navigation is accomplished in two steps: *Path preparation* and *Path following*. Both are implemented inside UT2004Formation module and will be discussed here briefly. Dealing with obstacles and degenerated navmesh will be omitted and will be discussed later in the thesis.

6.1 Path Preparation

Path preparation is done differently depending on the role of the agent performing it.

Leader is responsible for finding path to the destination. He uses navmesh module to compute path (Figure 6.1).

```
bot.getNavMesh().computePath(from, to);
```

Figure 6.1: Using navmesh path planner

The navigation system then proceeds to tweak the computed path to make it more friendly to pass with a formation that is wider than navmesh path planning takes into account. That's accomplished in two ways.

1. Every location in the path it uses ray casting to calculate the distance of the location form the wall, normal to the direction to the previous location. Then it pushes the location farther away, stronger the closer it is. It is a linear dependency.

2. Between two of each succeeding location it tries to look for very narrow, choke corridors. It does it by moving along the direction between them in small steps. Yet again it uses ray casting to calculate area width and if it's too narrow it will create an *ChokeNP* at the center of the corridor.

   ChokeNP is a class implementing interface *ILocated* that lets the agent know that there's a narrow area in the path. ChokeNP takes two arguments: the width of the area and its location. This is called *offline choke point detection.*

Follower prepares path very differently. For one, he does it all live and continuously, not just at the beginning.

Periodically he uses leader's last known position and velocity (obtained through *TCUpdate* messages sent by leader) with his offset to calculate new location. He then uses ray casting to move the location farther from nearby edges, same way the leader does it.

Sometimes the location isn't on navmesh. Then the farthest location on the navmesh from leader's location in the direction towards the offset location is found (6.2).

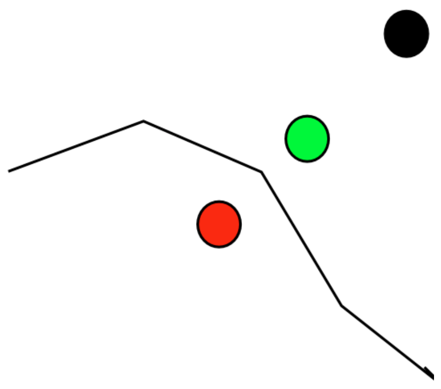

Figure 6.2: Red location is outside of navmesh. Instead, green location, closest to the edge of navmesh, is chosen

It can take time to reach these locations. Sometimes follower can get held up. For this reason they are stored on a queue which represents follower's path.

Another problem that can happen on rare occasions is that two consecutive locations do not lie on the same line on navmesh. Mostly this happens because offset locations are interrupted by a corner, unlike the leader's locations. Connecting point is added here so the agent can pass. The connecting point is created by adding inverted offset once or twice to the middle point between these locations. If neither helps it is left for path following to take care of the situation.

6.2 Path Following

Unsurprisingly both Leader and Follower behave differently in path following as well.

Leader merely sends update messages, *TCUpdate*, containing his position, velocity and rotation. Checks for degenerated navmesh and handling close quarters will be explained in specially designated chapters.

Follower, while navigating to the next location, checks if he can skip locations in its path queue, method called *location skipping*. He looks for the last location in his path queue which can be connected by a line with the bot's location on navmesh. All skipped locations in the queue are thrown out.

While on the move he also uses steering behavior. One force is pulling him towards the current target location. Other forces repel him from nearby edges. To get these he sends three rays from his location. One straight in front of him, other two 45 degrees left and right from his direction respectively.

After reaching his current target location he chooses the next one. He tries to skip locations here as well so he doesn't start moving recklessly to a location he would skip anyway later. That would cause him to freeze his movement for a while, which is unwanted behavior.

7. Holding Formation

Many difficulties can arise while navigating as a group. The terrain is complex and bots' actions don't always yield expected outcome. Followers can get delayed or environment can be impassable in a group or perhaps the formation is required to change with the passing terrain, either to give better look or to make it less likely followers get stuck.

7.1 Formation Width

Terrain in the game is very versatile and so is the width of the area bot has to travel. It is a good thing to have the formation take that into account and react accordingly.

It's leader's responsibility to check how wide is the area he's passing (7.1). One thing, already mention, is that during the path preparation phase he checks for close corridors and creates *ChokeNP* navpoints in their center. When he's on the move he can detect when he's navigating towards ChokeNP and he knows the width already.

Another thing is he does is that he scans area ahead of him, casting rays normal to his direction. The rays have a constant maximum range set inside the program. The width retrieved is the length the rays went without leaving navmesh.

If the choke is coming up, the width needs to be change right away in order for formation movement to work. However, it isn't necessary to make formation larger right away. For that leader stores the last formation width and then makes sure the formation gets wider gradually, in small steps.

Both methods are necessary in order to enhance precision and minimize risk of not detecting very narrow areas, chokes, as they are pose a threat to the stability of the navigation system.

Now that the width of the upcoming area is calculated, we can use it to change the current width of the formation. Leader creates an instance of *TCSetWidth* message, with the width of the area and sends it the owner of the formation.

The owner's Formation commander module catches the message. It then tries to the width of the formation pattern to the value obtained by TCSetWidth message (Figure 7.2).

```
pattern.setWidth(setWidthMessage.getWidth());
```

Figure 7.2: Using IFormationPattern's API to change pattern's width

WedgePattern, an implementation of *IFormationPattern*, does it by calculating the factor by which the new formation width would change and then scales all offset vectors by that factor.

All it remains is to inform the members of the formation of the change. Formation commander then sends new offsets via *TCOffset* message to all followers, who

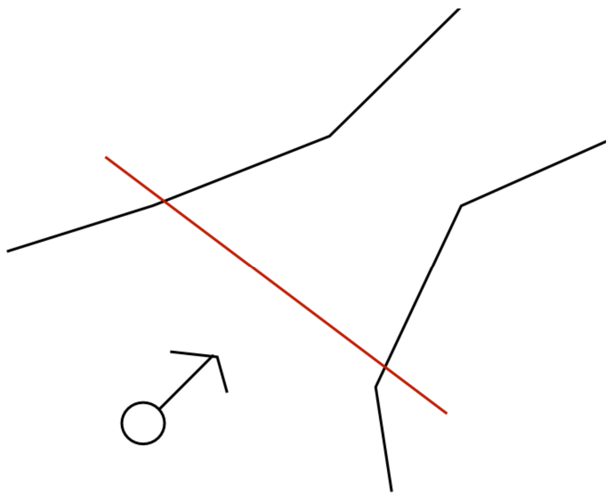

Figure 7.1: Leader checks width of an area ahead of him

then simply update them, making them move closer to the leader. *TCWidthUpdate* message with formation width is sent to the leader to let him know current width.

7.2 Obstacles

There are some problems that may arise while moving as a group. They can break the regular navigation, thus are called obstacles to the navigation system. They can be natural part of the environment as well as shortcomings of available tools.

They are dealt with in a two-step process. First, they need to be detected. That's the leader's job. Then it's up to the follower to resolve the situation after receiving warning messages from the leader.

7.2.1 Detection

There are four main types of obstacles that the leader needs to detect.

Degenerated Navigation Mesh

Degenerated Navigation Mesh scenario happens when the group has to walk through an area where navigation mesh is missing, even though it's a walkable area (7.3).

For detecting it a trick is used. It is known the bot navigates on navigation points when not using navmesh navigation (7.3). Thus we can concentrate on controlling navpoints only.

Two things can happen. One, we go through two navigation points, both on navmesh but their connecting line isn't on navmesh. Two, the next navpoint

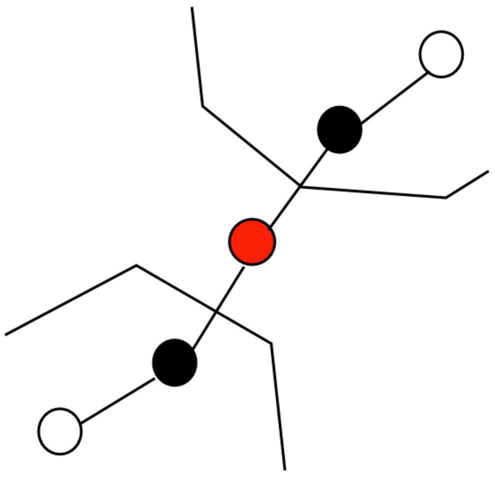

Figure 7.3: Path crossing through an area without navigation mesh. Empty circles are Locations, black are NavPoints and the red is a NavPoint outside of navigation mesh

is not on navmesh. These two cases account for all instances of degenerated navmesh.

The start of the degenerated navmesh is now detected. To make it easier to followers it's useful to send them also the first navpoint after degenerated area. The leader looks ahead and find it sends both as contents of message *TCDegenerated.*

Followers use current leader's position periodically to calculate new locations to add to his path. This doesn't work when the leader isn't on navmesh. To let them know, a flag isDegenerated is a part of the TCDegenerated message. Then, after the leader has come back to a navmesh, he sends yet another TCDegenerated message, this time only with isDegenerated flag set to false.

Jumping Edges

Jumping edges are a natural part of the environment and as such can be detected easilly. They are always marked as a flag on an edge between two navpoints. Edges are represented by *NavPointNeghbourLink* class. *LinkFlag* enum contains all the flags.

```
boolean isJumpingEdge =
    (link.getFlags() & LinkFlag.JUMP.get()) != 0;
```

Figure 7.4: Testing if NavPointNeighbourLink is a jump edge

If it is a jumping edge the information is yet again set inside TCDegenerated message. We overload its meaning for all obstacles because follower can react to them in the same way, as will be shown soon. Starting and ending locations are the ones connected by the jumping edge. When the leader arrives at the ending, he sends the message with a flag marking the end of the degenerated path.

Lift

Lifts are very difficult to deal with even when navigating a single bot in this environment. Because of that it was decided to simply try to skip them and hope for the best. That's achieved using regular navigation one bot would use himself and he navigates to a location after the lift. They all try to group up again.

Other

Other obstacles are all different kinds of navpoints. They can mark a teleporter or a jump node. Starting location argument in TCDegenerated is a navpoint before the special location and ending is the first one after.

Yet again, after reaching the ending location, the leader sends the false flag marking the end of the degenerated path.

7.2.2 Resolution

Resolution is about follower's point of view. He looks at all obstacles as degenerated path, where not only he behaves differently both in path following and reacting to leader's updates. Even though with some obstacles the leader isn't outside of navmesh the outcome is the same. Let's then asume that the leader is passing degenerated navmesh.

So the follower catches TCDegenerated message with flag set to true and *start* and *end* locations. He stores the locations for future use and stops adding new locations to his path, because the flag tells him that leader is moving outside of navmesh.

Or he catches TCDegenerated with a false flag. This means leader's returned to the navmesh and the follower can start appending new destinations at the end of his path.

The follower has to check whether *start* location of the degenerated path is coming up. IF it is, he shuts down his usual behavior. No more ray casting and following stored path, as the works only on navmesh. Instead, he uses path planner supplied with navmesh module and computes the path to the *end* location, marking the end of degenerated path and start of navmesh. Then he proceeds to use the Pogamut bot's IPathExecutor to follow the computed path. After path executor brings the follower to the *end* location, he restarts his regular behavior, knowing he is back on navmesh.

8. Speed Regulation

With all the obstacles and terrain irregularities, it's only expected that followers start falling behind the leader. Formation movement, to be effective, requires agents to regulate bots' speeds. *SpeedRegulator* class is here to achieve just that. Leader owns the instance of this class within UT2004Formation.

Followers calculate the length of the stored path (which ends at at their offset location near the leader) and to better estimate it they try to smooth the path first by using the *location skipping* technique mentioned earlier. In short, they don't add distances to locations that can be skipped because the next location in the path can be connected by a line to the current location on a navmesh.

They proceed to send the computed length within *TCLagging* to the formation leader.

The leader catches the message and updates the path length of the sender in SpeedRegulator.

Every game update the leader calls *update()* method of the SpeedRegulator. It calculates speeds for all formation members and sends each of them *TCSetSpeed* message with the resulting speed. Speed in Pogamut is represented by the factor of max speed. Allowed values range from 0.1 to 1. The message is then intercepted by each member and they update their speed to the new value.

Anyway, let's look at how the speed is actually calculated. SpeedRegulator stores how much behind every bot is. To calculate speed he checks how far every bot, including leader, is from the leader's current target. That means he adds the leader's distance to the target to all stored distances.

At first the speed in Unreal units per time unit is computed, where time unit is a time between two calls of *logic* method, which is not constant, but it's approximately 250ms. Formation members should arrive at the same time, so let's say it'll take them t time units. t is really an arbitrary number because it will all be normalized so that the faster member will be going with max speed.

$$s_i = d_i/t$$

Figure 8.1: The equation for computing speed

Using standard equation form Figure 8.1 we calculate unadjusted speeds for members. Then, the maximum speed of the bots s_{max} and *fullSpeed* constant containing highest possible speed are used to get the constant c (Figure 8.2) to normalize speeds so that fastest bot is at full speed.

Then all speeds are normalized by the factor c (Figure 8.2) and changed so that maximum value is 1 and minimum is 0.1.

$$\begin{aligned} c &= fullSpeed/s_{max} \\ final\_s_i &= s_i * c \end{aligned}$$

Figure 8.2: The equations for calculating normalizing constant c and final speeds

The calculated speeds are then send to all formation members.

9. Evaluation

Life Cycles

Creating, controlling and disbanding formations works as intended. Formation commander really achieves to give user an easy control over the formation.

Normal Navmesh

Formation movement proved to be most effective on areas with continuous navigational mesh that covered the terrain very well.

In some areas where navmesh didn't cover the whole of walkable terrain, it could cause bots to detect a narrow path even though there was none or they were more cramped than necessary.

Turning also wasn't much of a difficulty to the navigator. In some areas, though, ray-casting could at times fail to handle sharp corners, making follower stuck for a moment. Fortunately the follower would eventually catch up as he contains a stuck resolution routine.

Area Width

Formation is capable of appropriately changing its size in response to outside conditions, like area width. In case of very narrow areas, formation navigates through the center of the corridor.

Degenerated Navmesh

Imperfections in navmesh that caused frequent navmesh degeneration on certain maps, however, were too difficult to fully overcome. At times, while bots were trying to navigate through it, the broke apart and some of them got stuck. In the end they would end up getting back together again, but on some maps they go from one degenerated navmesh to other.

Jumping Edges

Formation can work around jumping edges. Chosen implementation was sufficient. On rare occasions some bots don't make the jump. Note that this can happen to regular bots using built-in path planning and path executing. As formation bots use built-in systems, they can fail as well.

Lifts

As was expected even before writing the thesis, lifts in UT2004 are too difficult to use with bots. This implementation merely tells bots to navigate

Conclusion

Different formation movement representations were discussed at the beginning of the thesis. Adapting Two-Level steering approach for distributed agents proved to have been a good decision.

The problems with navigating the formation mostly came up due to available navmesh representation and navigation. Upcoming year a huge update is coming to Pogamut with aim to improve both generation of navmesh and navmesh navigation module.

Future Work

There are several ways how future work can build on this thesis.

Tactical Movement

Navigation of the environment could take tactical movement into consideration. Taking cover whilst navigating through a dangerous area, taking up positions that could provide cover fire, should the need arise, et cetera.

Team AI

Using team navigation to move a group of bots for an in-game objective, like attacking enemy base and capturing the flag or sieging an enemy position in any other team game mode in UT2004.

Bibliography

[1] Artificial Minds for Intelligent Systems, Research Group: The Pogamut platform Charles University in Prague, `http:\pogamut.cuni.cz` (20.7.2014)

[2] J. Gemrot, J. Kadlec, M. Bída, O. Burkert, R. Píbil, J. Havlíček, L. Zemčák, J. Šimlovič, R. Vansa, M. Štobla, T. Plch, and C. Brom *Pogamut 3 Can Assist Developers in Building AI (Not Only) for Their Videogame Agents*, LNCS 5920, Springer (2009)pp. 1-15.

[3] TOMEK, Jakub. *Navigation Mesh for the Virtual Environment of Unreal Tournament 2004*. Master's Thesis

[4] I. Millington, and J. Fungu *Artificitial Intelligence for Games*, 2nd edition. ISBN-13: 978-0123747310

[5] RABIN, Steve. *AI Game Programming Wisdom*. ISBN 1584500778

List of Abbreviations

API - application programming interface - set of methods and classes available to a programmer

GUI - graphical user interface - program's interface for visual communication with a user

IVA - intelligent virtual agents - virtual simulation of a living being, often human

AI - artificial intelligence - academic field of study in computer science with a goal of making intelligent machines

IDE - integrated development environment - software that provides comprehensive facilities for software development

UT2004 - Unreal Tournament 2004 - a first person shooter computer game

CPU - central processing unit - hardware within a computer that carries out instructions of a computer program

JMX - java management extensions - java technology that supplies tools for managing and monitoring applications, system objects, devices and service oriented networks

navmesh - navigation mesh - representation of a virtual environment using convex polygons

navpoint - navigation point - a node in a navigational graph

A. Contents of the accompanying CD

- /Installation - compiled project, GameBots2004 (with test map), JDK
- /Documents
  - /Thesis - pdf file containing the Bachelor thesis and its LaTeX source code
  - /Documentation - user manual (documentation) and software documentation
- /Sources - contains source to the program
- /Videos - videos of formation movement