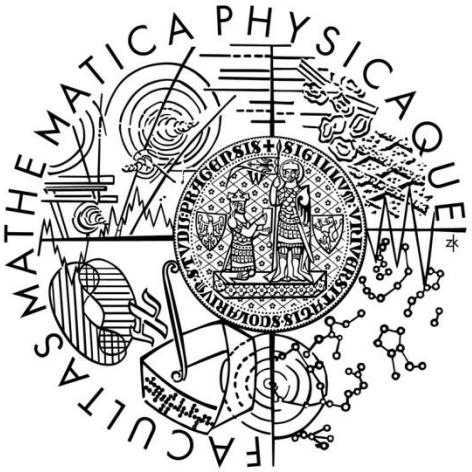
Charles University in Prague

Faculty of Mathematics and Physics

MASTER THESIS



Bohuslav Macháč

IVA Movement Quality Improvement for the Virtual Environment of Unreal Tournament 2004

Department of Software and Computer Science Education

Supervisor of the master thesis: Mgr. Jakub Gemrot

Study programme: Computer Science

Specialization: Software Systems

Prague 2014

I would like to thank my supervisor Jakub Gemrot for his supervision and counselling which guided me throughout my work.

Next, I would also like to thank all members of the AMIS group for remarks on my work.

Last, I would like to give my deepest thanks to my family and Janina for their great support and patience.

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular the fact that the Charles University in Prague has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 paragraph 1 of the Copyright Act.

In Líně on 30.07.2014 signature of the author

Název práce: Zvýšení kvality pohybu IVA ve virtuálním prostředí Unreal Tournament 2004

Autor: Bohuslav Macháč

Katedra / Ústav: Kabinet software a výuky informatiky

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

Abstrakt: PogamutUT2004 je rozšíření platformy Pogamut navržené pro vývoj inteligentních virtuálních agentů (IVA) ve hře Unreal Tournament 2004. Navigaci IVA má v Pogamutu na starosti navigační systém, který používá navigační graf jako abstrakci prostředí. Navigační mesh je novou, pokročilejší abstrakcí, ale navigační systém není schopný využívat jejích výhod. Vytvořili jsme nový navigační systém, který těží z výhod navigačního meshe a řeší několik dalších problémů starého systému. Ukazujeme, že nový navigační systém zvyšuje kvalitu navigace.

Abychom mohli demonstrovat zvýšení kvality, vytvořili jsme hodnotící framework pro porovnávání navigačních systémů. Systémy byly porovnány v absolutním počtu důležitých cest, které je systém schopen úspěšně následovat, v délce cesty a době, kterou zabere její následování.

Vybrali jsme 18 různých map pro detailní vyhodnocení a provedli základní vyhodnocení na dalších 58 mapách. Nový systém je úspěšnější na 16 detailně vyhodnocených mapách s průměrným zlepšením 6,75% v úspěšnosti navigace. Podařilo se nám zvýšit použitelnost PogamutUT2004 pro tvorbu IVA do AI soutěží a jako vzdělávací platformy.

Klíčová slova: inteligentní virtuální agenti, navigace, navigační mesh, následování cesty

Title: IVA Movement Quality Improvement for the Virtual Environment of Unreal Tournament 2004

Author: Bohuslav Macháč

Department / Institute: Department of Software and Computer Science Education

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

Abstract: PogamutUT2004 is an extension of the Pogamut platform designed for developing intelligent virtual agents (IVAs) in Unreal Tournament 2004. Navigation of IVAs in Pogamut is handled by a navigation system, which uses a navigation graph as an environment abstraction. Navigation mesh is a new, more advanced abstraction, but the existing navigation system is not capable of using its advantages. We created a new navigation system, which exploits advantages of the navigation mesh and solves several other issues of the old one. We show that the new navigation system improves the quality of navigation.

To demonstrate the quality improvement, an evaluation framework was created for the comparison of navigation systems. Systems were compared in terms of total number of significant paths on the map, which the system is able to follow, length of the path and time of the navigation.

We selected 18 different maps for thorough evaluation and we performed the basic evaluation on 58 other maps. The new system is more successful on 16 of the thoroughly evaluated maps, with moderate improvement of 6.75% in the success rate of the navigation. This improves the usability of the PogamutUT2004 as the platform for creating IVAs to several AI competitions and as an education platform.

Keywords: intelligent virtual agent, navigation, navigation mesh, path following

Contents

1. Introduction

Pogamut (1) project is a complex tool for research and development of intelligent virtual agents (IVAs) (2). It enables controlling of virtual agents in environments provided by computer game engines. One of the supported games is Unreal Tournament 2004 (UT2004). PogamutUT2004 is an extension of the Pogamut platform specifically designed for UT2004.

UT2004 is a first person shooter (FPS) developed by the Epic Games Company (3). FPS is a type of game, where the player sees the world from the eyes of the controlled avatar and uses different weapons to shoot his enemies. We concentrate on the agents controlled by the computer. Figure 1.1 shows Pogamut agent moving in the environment of UT2004.

Figure 1.1: Pogamut agent in UT2004.

There are many actions an agent can do in the environment, but most of them have one thing in common, to be able to do them, the agent has to move around the environment. Most common actions of the agent in the game are picking up items and fighting an enemy and for both of them, a movement is needed. The agent has to reach items it wants to pick up and it has to find the enemy on the map to fight him The navigation system is a very important part of controlling the agent, without it, the agent is not capable of performing most of its available actions and the quality of the navigation system directly affects the performance of the agent.

The navigation system consists of two main parts, path planning and path following. Path planning is a process of finding a path to the desired location in the environment and to do this, the navigation system needs information about the game environment in which it should plan the path. Path following is an action of following the path created by the path planning subsystem. Again, information about the game environment is needed, this time together with current information about the agent and about possible dynamic parts of the environment, like lifts.

Game environment (or map) consists of the level geometry, its textures and items spawned in the game. Using the whole map geometry of an environment for navigation would be slow and ineffective. Typically, some abstraction of the environment is used. Until recently, a navigation graph was the only such abstraction available in PogamutUT2004.

The navigation graph represents an environment by a set of navigation points (graph vertices) and edges, over which an IVA can move safely. Sometimes, the navigation is possible in one way only (jump down from the high ground), therefore the edges are oriented. Edges can carry additional information about how to follow them, e.g. if the jump is required for passing the edge. Recently, the navigation mesh was made available as another abstraction of the UT2004 environment as a goal of master thesis of Jakub Tomek (4). Our work follows up on that thesis.

The navigation mesh is a set of convex polygons that cover all reachable places in the virtual environment and allow IVAs to move over much greater part of the virtual environment than navigation graph. But where the navigation graph contains additional information how to follow an edge (jump is needed, it is the teleport edge), navigation mesh is strictly an abstraction of environment's geometry. The navigation mesh is enriched by off-mesh connections to avoid losing information about any navigation opportunities. Off-mesh connections represent such navigation opportunities from the navigation graph, which were not captured by the basic mesh. They are typically edges from the original navigation graph, but leading from one polygon to another, instead of between vertices. This should give the navigation mesh at least the same navigation capability as with the navigation graph.

The goal of this work is to improve the movement quality of IVAs in the UT2004 environment. This should be done by adapting the navigation system to navigation mesh abstraction and by resolving some of the path following problems in the original system. Higher quality of the movement brings agents the capability of moving over larger part of the map and a higher chance to reach the desired target successfully.

A testing framework will be created for the evaluation of the quality improvement and thus demonstrating the successful completion of the goals of the thesis. It will enable a robust comparison of the original navigation system with the new one. The evaluation will be concluded on several significant maps, which will be chosen from different clusters of the available maps. The main monitored statistics will be the total number of paths the navigation system can successfully follow, the length of the path and time of the following the path.

1.1 Goals

Goals of our work can be summarized by this list:

1. Improve the movement quality of agents in PogamutUT2004.
2. Evaluation of navigation.

Successful completion of those goals would improve the usability of the PogamutUT2004 both as the teaching platform (5) as well as the platform for creating IVAs to several AI competitions, where bots made with Pogamut already participate in. (6), (7) We will complete the first goal by making quality navigation mesh available as an abstraction of environment (Goal 1.1) and by improving the current navigation subsystem (Goal 1.2). The completion of the Goal 1.1 will make the navigation mesh available also for other uses, like navigation in formations for team modes of the game. To fulfill the second goal, we will create testing framework for comparing navigation systems and their configurations, which will be also available for testing navigation systems in the future.

1.2 Structure of the thesis

In the following part, the environment of the UT2004, Pogamut, the existing navigation system, and available abstractions of the environment are described in detail. Chapter 3 summarizes the related work. Next chapter describes the analysis of the current navigation system and its problems, together with the description of our work process. (Ch. 4) The following part contains upgrades of navigation meshes (Ch. 5) and the next part contains changes made to the navigation system. (Ch. 6) Then, the evaluation is described, along with the presentation of the testing framework and its capabilities, evaluation process and metrics we concentrated. (Ch. 7) Results of the evaluation and comparison of the original and the new systems are also part of this chapter. The analysis of results and fulfillment of the goals together with the outline of possible future work form the last chapter. (Ch. 8)

2. Background

2.1 Pogamut platform

“Pogamut is a Java middleware that enables controlling virtual agents in multiple environments provided by game engines. Currently Unreal Tournament 2004, UnrealEngine2RuntimeDemo, Unreal Development Kit and DEFCON game are supported. Pogamut provides a Java API for spawning and controlling virtual agents and GUI (NetBeans plugin) that simplifies debugging of the agents.

The main objective was to simplify the "physical" part of agent creation. Most actions in the environment (even the complicated ones, like pathfinding and gathering information in agent's memory) can be performed by one or two commands. This enables user to concentrate his efforts on the interesting parts.” from official web pages of Pogamut platform. (8)

2.1.1. Pogamut components

The first version of Pogamut was released more than 10 years ago. Since then, its development continues with current version of 3.6.1. Pogamut 3 integrates 5 different components: UT2004, Gamebots2004 (GB2004), the GaviaLib library, the Pogamut bot and the IDE as displayed on Figure 2.1 from (8).

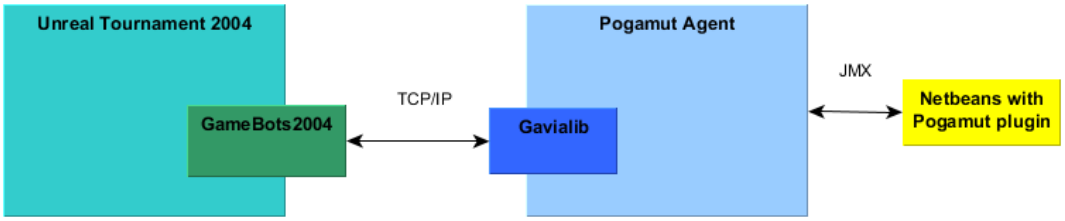


Figure 2.1: Pogamut architecture

GB2004 is responsible for exporting information about the game environment of UT2004 through a text based protocol over TCP/IP. The GaviaLib transforms the messages from GB2004 to Java classes, which are afterwards delivered to Pogamut bot listeners, registered for specific type of such messages. The IDE part serves for easier development and debugging of Pogamut bots.

Gamebots2004. GB2004 consists of several classes written in UnrealScript and XML definitions of messages it sends. UnrealScript classes extend the UT2004 server with wrappers of selected game types, observer connection and wrappers of other UT2004 features. Messages, in the direction from GB2004 to PogamutUT2004, are sent through a text based protocol periodically, in synchronous batches with frequency of 4 Hz. A new feature is the `LocationUpdate` message, which is a cornerstone of faster navigation logic cycle presented in our work. This message is

sent 5 times more often than synchronous batch, with frequency of 20 Hz. The same text based protocol is used for sending commands from PogamutUT2004 to GB2004. Commands represent direct actions of the agent, like jump or shoot, as well as requests for additional information from the game engine, like ray cast or path computation through UT2004's internal path planner.

Pogamut bot. A Pogamut bot represents the brain of the UT2004 agent thus directing its body. It analyses the situation of the agent and gives commands to fulfill desired goals. The Pogamut bot consists of several classes, implementing interfaces or extending base classes from Pogamut module. The main concern is the logic cycle, invoked by receiving of the synchronous batch message from GB2004. Its frequency is therefore the same as the one of the batch message, 4 Hz. In the logic cycle, bot reacts on current information from the game engine and responds with actions. These actions are then translated to commands by the PogamutUT2004 and sent to GB2004. Apart from the main logic cycle, the bot can also react to specific events through listeners, which are registered to concrete class of messages and invoked only upon receiving a message of such type. In these listeners, the bot can react with direct actions or by updating corresponding data in its memory and postponing the reaction to the main logic cycle, where the data can be considered together with other information from the batch message.

2.2 Unreal Tournament 2004

UT2004 is a FPS based on fighting in arenas and is highly oriented on multiplayer game. The player stands against other players or bots. They have the same capabilities and properties as the player and only differ in the equipment collected on the map. The equipment consists of weapons and ammunition, armor and special double damage bonus, which duration is time-limited. A player also earns adrenaline, as a reward for besting its enemies or by picking it up in form of capsules. The adrenaline can be used to gain other time-limited bonuses as higher speed, invisibility and others.

To win a game in UT2004, a player must have the highest score of all at the end of the game. The game is ended either by reaching the preset score target or by time running out. A score is gained for different actions, based on the game mode. UT2004 has both individual and team modes.

In team modes, multiple bots often move together in a formation. For quality movement in a formation, a navigation graph is insufficient because it contains too few information about the environment. This is partially solved by defining a safe corridor around each edge, where bots can move in, which UT2004 provides. However, this information is not always precise and thus it cannot be relied on. Navigation mesh could solve this problem as it provides information

about all reachable places in environment; however, the key again is to create reliable mesh that would fit well for a particular model of UT2004 bot's body.

UT2004 has total of 11 game modes, but PogamutUT2004 supports only subset of them. Supported modes are Deathmatch (DM), Team Deathmatch (TDM), Capture the Flag (CTF), Bombing Run (BR) and Double Domination (DOM).

There are vehicles in UT2004, but they only occur in unsupported modes. This is important for the radius of the bot, which differs only when driving a vehicle, but since there are no vehicles in the supported modes, we can work with constant radius, as described in the navigation mesh generating process.

UT2004 has more than 100 maps. Each map is designed for one or more of the game modes. We concentrate only on the maps for the supported modes, which leaves out only maps for Assault and Onslaught modes, because other unsupported modes are played on DM or CTF maps and there are no special maps for them. This makes 91 maps for Pogamut supported modes and therefore interesting for us.

A map in UT2004 consists of the level geometry and items, which are present in the environment. The level geometry can contain terrain definitions and static meshes with textures composing parts of structures or other objects. Working with complete level geometry for navigation would be ineffective and slow. The game works with the navigation graph as an abstraction of the environment.

2.3 Environment abstractions

In this section, we will introduce abstractions of environment available in PogamutUT2004, their properties, advantages and disadvantages.

2.3.1. Level geometry

Let us consider that the level geometry would be used for navigation directly. Main advantage would be that no manual input would be needed in such case. It seems that it could work if the bot would have the right set of steerings defined and used enough ray-casts to have sufficient information about the environment for the steerings to work correctly.

There are two main problems with this approach. First is the complexity of the map, so if we would simply steer the bot in the direction of its target, it would not work on more complex maps. The solution could be to divide the map into areas where the steerings could work correctly and to define transitions between such areas. A path would be found on such abstraction and the bot would be gradually steered between individual transitions. Second problem would be the computational requirements of ray casts needed for the navigation. It would not have to be that many for simple running, but when we would need to compute if

some jump is possible, we would need many ray casts to rule out the possibility of some (rather small) obstacle which would make the jump impossible. So we could add the jumps manually, as we would have to do that for lifts already, but then we would lose the main advantage of this approach. Figure 2.2 is the example of a level geometry visualized by Recast (9) program and Figure 2.3 shows level geometry directly in the game.

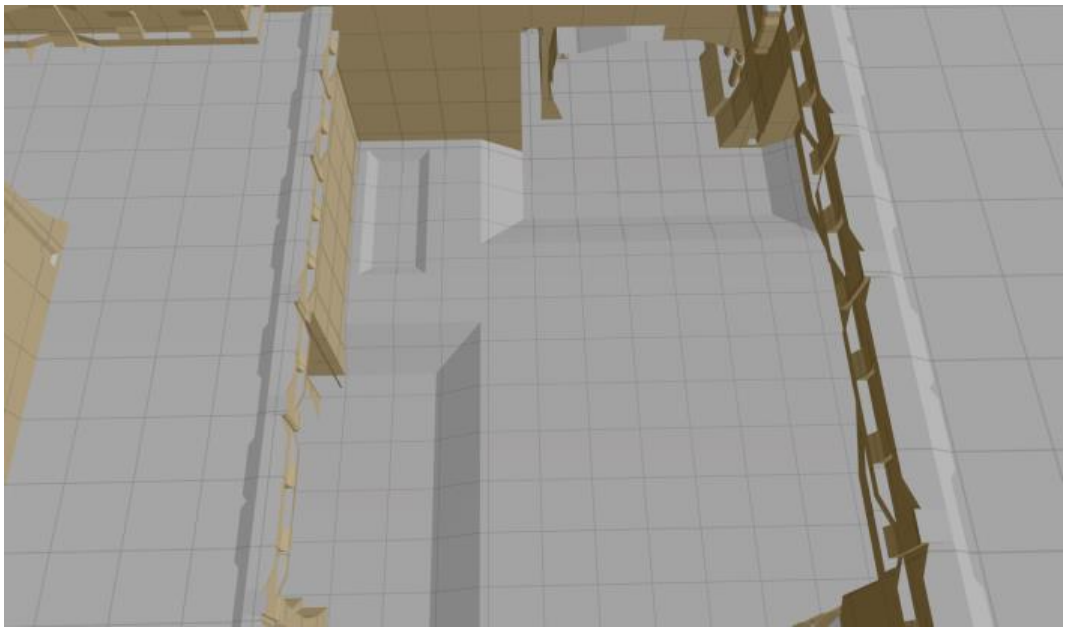


Figure 2.2: Level geometry of part of the DM-1on1-Trite map.

Figure 2.3: Level geometry in the game.

2.3.2. Navigation graph

The navigation graph is a set of navigation points (or NavPoints) and oriented edges between them. The navigation graph in UT2004 was created manually by the map designers when they created the map and it is extracted by the GB2004 module to PogamutUT2004. There is an example of the navigation graph in UT2004 from the part of the DM-1on1-Trite map on Figure 2.4.

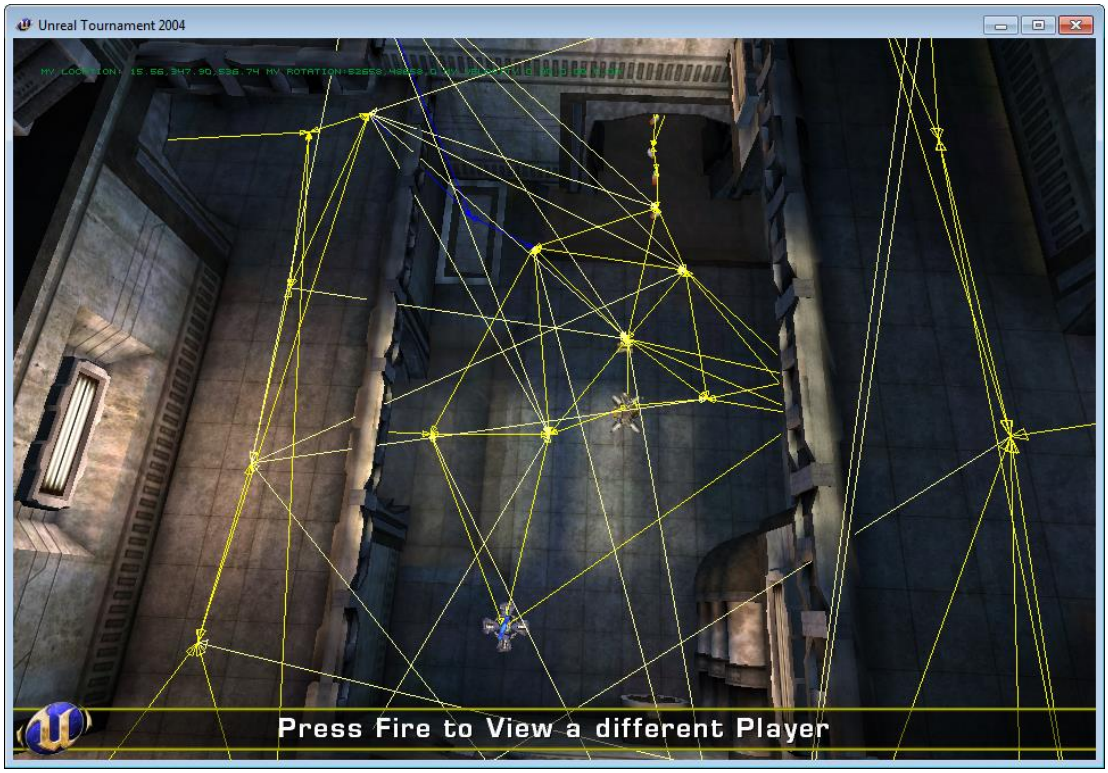


Figure 2.4: Navigation graph of the part of DM-1on1-Trite map.

Types of navigation points:

- InventorySpot - represents a place when some item is periodically spawned,
- PlayerStart – a place where player can be spawned at the beginning of the game or at the respawn after his death,
- PathNode – a point with no special meaning, serves only for navigation purposes,
- JumpSpot - special kind of a PathNode, typically placed at the edge, from where jump edges lead,
- AIMarker – has special meaning for the game AI only,
- SnipingSpot – a place marked as ideal for sniping, usually has good outlook on significant part of the map,
- LiftExit - marks enter and exit place for the lifts,
- LiftCenter – a point at the center of the lift, moves with the lift,

- Teleport - marks a teleport, which instantly translocates the player to the other end of the teleport (indicated by an edge between those points),
- Jumppad - gives a player significant boost in the up direction.

Not all of these types are important for navigation and have to be treated specially, we work differently only with LiftCenters, LiftExits, Teleports and Jumppads, other types of navigation points are treated the same.

Types of navigation edges:

- Standard - no special meaning,
- Jump - indicates that the jump is needed to follow it,
- Lift - edge between LiftExit and LiftCenter points,
- Special - some special action is needed to pass this edge.

The main deficit of the navigation graph is the lack of information about the environment it describes, as it consist only of volumeless points and lines between them. When a bot detours from the navigation graph, it has no information about its surroundings and has to return to the graph before the navigation can meaningfully continue. Even when moving on the graph, the bot does not have all the data it needs. It applies mainly to jump edges. The bot knows that it has to jump somewhere between those points, but often the jump directly from the start point to the end would be too long and the bot needs to jump later and/or land earlier in order to safely reach the other end of an edge.

Both problems could be solved by making the navigation graph dense enough and by responsibly placing navigation points at the start and at the end of each jump but; on the other hand, it would increase the memory and computational requirements of the navigation, which would defy the navigation graph purpose.

2.3.3. Navigation mesh

The navigation mesh is a set of convex polygons that cover all reachable places in the virtual environment. The convexity of the polygon provides the certainty, that the shortest path from any point in the polygon to any other point in the polygon leads fully through the polygon and therefore if the polygon covers only reachable place, such path is possible to traverse directly. Navigation is possible between connected polygons. But not all areas on the map have to be connected together. Or even if they are, some shorter path can exist, if we have ability to jump or to use lifts or even teleports. To allow such features, the mesh is expanded by off-mesh points and edges.

An off-mesh point is a point inside the mesh, from which or to which some off-mesh edge leads or a point completely out of the mesh, connected with it only by an off-mesh edge. An off-mesh edge (or path of edges, with internal points being off-mesh points completely out of the mesh) connects two off-mesh points and usually represents a shorter path between two points on the mesh, or even such path, which did not exist in the original mesh. A navigation mesh can be generated (semi-)automatically[1] from the level geometry, if the conditions on the quality and the format of the level geometry are redeemed, but the off-mesh capabilities must be added manually.

Figure 2.5 pictures example of the navigation mesh with off-mesh connections from PogamutUT2004, it is the same part of the DM-1on1-Trite map as it was in the navigation graph example.

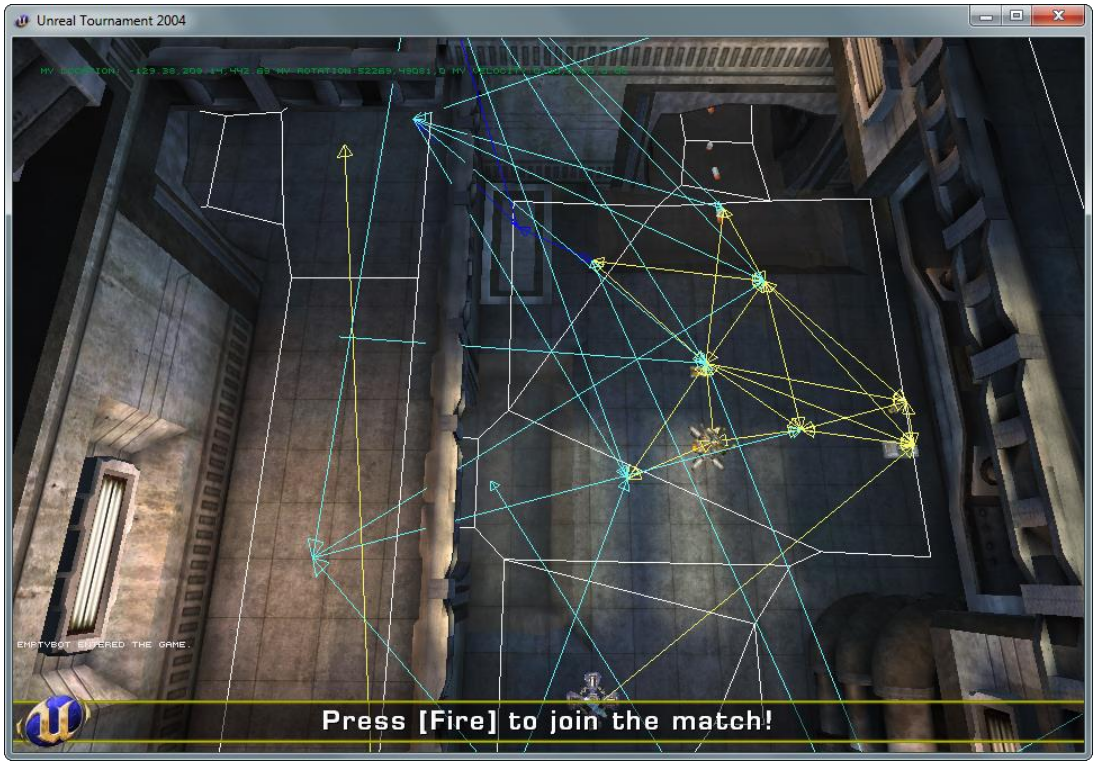


Figure 2.5: Navigation mesh with off-mesh connections.

The problem with insufficient information when using navigation graph and computational requirements when using the level geometry are the main reason why the navigational mesh is so widely used and why we will upgrade the PogamutUT2004 navigation to use it as well.

[1] Process of generating of the mesh is only semi-automatic because we have to set a range of parameters for it to work correctly.

2.4 Navigation system in PogamutUT2004

In this section, we will introduce the problem of navigation and describe the structure of typical navigation system and its implementation in PogamutUT2004.

Navigation of the agent in a game environment has to face several difficulties, some of them has source in the used abstraction of environment and other emerge from the mechanics of the agent's control. Abstractions of the environment contain only limited amount of data about the actual game environment as it is simplified to be possible to work with fast enough. These simplifications can cause trouble in navigation or mean that we simply do not have enough information to identify the correct navigation action at the right time. Navigation system does not receive information from the game directly, but through GB2004 connected via text based protocol, which results in delays of the information. There is also a delay when sending commands to the game and before the avatar body reacts to them, which creates the need to account for such delay and predict the bot's situation at the time of the actual execution of the command.

Typical architecture of the navigation system has two main parts: a path planning and a path following. The path planning part of navigation is represented by a path planner, which consists of some abstraction of the environment (or an access to it) and selected path finding algorithm for creating a path in the abstraction. The path following part typically has 3 levels, a runner on the lowest level, which is responsible for direct movement from one point to the other, a middle level represented by a navigator, determining points for the runner and a top-level executor, wrapping the functionality for the user and including stuck detectors. This architecture is independent to the game environment and even to the game's type.

Main function of the navigation system in PogamutUT2004 is the same as for any other navigation system and it is to navigate the bot where the programmer wants to get it. The basic approach to do that, is for the programmer to choose a target location, typically through methods available on the `WorldView` object, which allows him to access the NavPoints and filter them by their properties, to select the desired target, and then call the `navigate` method. The navigation system will take care of the rest.

2.4.1. Navigation process

The navigation process starts when the agent specifies the target of navigation by calling the `navigate` method of `UT2004Navigation`. There are 3 possible types of targets:

1. Location – The navigation system receives location where the agent wants to go to, checks if it is not already navigating to this location and asks the path planner for the path.
2. Path – The navigation system gets the whole path which should be followed.
3. Player –The agent marks the player which it wants to follow, the navigation system asks the path planner for the path to the last known target player's location.

The path is passed to `UT2004PathExecutor`. There runs the periodic navigation logic cycle. In each run of the cycle, following actions take place:

- In case of following the player, the path can be updated in case the player's location is changing and our agent can see it. The change of the location is detected by the navigation system and the new path is requested. Such path is merged with the existing one to allow for continuous movement.
- `UT2004PathNavigator` can switch to the next node if it is near enough to the current one.
- `UT2004PathNavigator` handles two special cases:
  - Teleport – Has different conditions for switching to the next node.
  - Mover – There is a possibility of waiting on the mover to return to the desired position, detecting if the agent failed in boarding the mover and is underneath it, has special conditions for switching to the next node.
- `UT2004PathRunner` navigates the current link and can invoke a jump if it is needed.

2.4.2. Components of navigation system

The navigation system consists of several interfaces and their implementing classes as shown in Figure 2.6.

We will describe individual components of it in detail, their interesting methods and responsibilities.

`IUT2004Navigation` is a top-level interface which provides access to high level methods for navigation. Its methods can be called with same parameters repeatedly and asynchronously without penalty. The most important is the overloaded `navigate` method, which allows the bot to move anywhere on the map, follow given path or follow given player, each in a single call. It also provides access to `IPathExecutor`, which it internally uses for navigation.

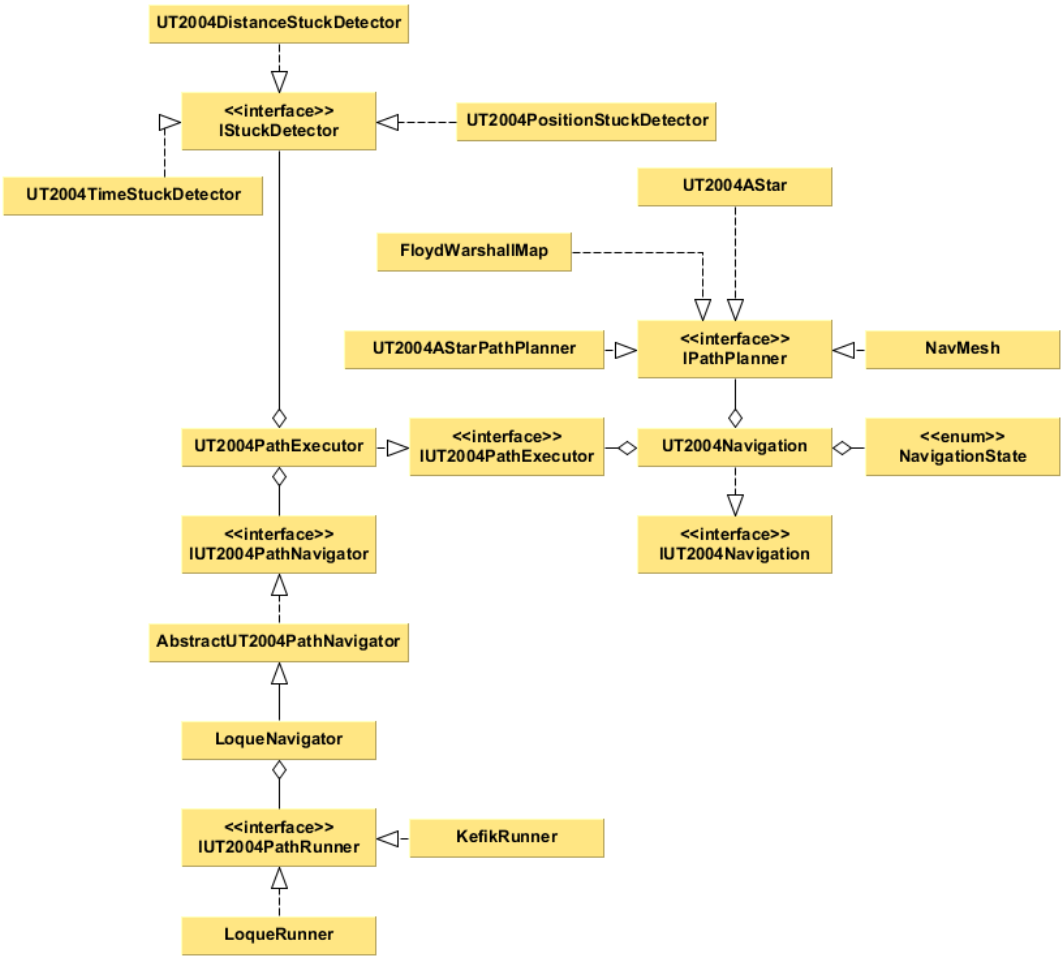


Figure 2.6: Structure of navigation system in PogamutUT2004. It contains only parts relevant to our work.

`IUT2004PathExecutor` executes given path periodically, as it has its own navigation logic cycle. This cycle is invoked upon receiving the `EndMessage` which is a part of the synchronous batch GB2004 send to the PogamutUT2004 each 250ms as described in 2.1.1. Calling the `followPath` method repeatedly causes navigation to restart each time. It also allows adding of the `IStuckDetector`s.

Purpose of `IStuckDetector` is to control that navigation did not stop working. There are several stuck detectors used internally, so the navigation system informs the bot that it failed in its task without any special care from the programmer. Available stuck detectors are:

1. `TimeStuckDetector` - Detects how long the bot has not moved (by checking its velocity) and if it reaches the threshold value, fires the stuck event. It has different threshold values for normal navigation and for the case that the bot is awaiting a mover.

2. `DistanceStuckDetector` - Evaluates whether the bot is closing to or distancing from the current target. If it is distancing for the second time while it was closing in between on same edge, the stuck is signaled.
3. `PositionStuckDetector` - Checks the change in position of the bot. If the position does not differ enough from some position in the history, the stuck is signaled.

`IUT2004PathNavigator` is responsible for navigating along given path and is called periodically from the navigation logic of the path executor. Each tick of logic it evaluates whether the bot is close enough to the current target in its path, and if so, it switches the navigation to the next node, or informs about reaching the destination, if the current node is the last in the path. It uses different conditions for evaluating special situations like going through a teleport or riding a mover.

`IUT2004PathRunner` is responsible for navigating along the current edge. It determines whether a jump is needed to pass it. If it is, it decides if the jump should start right now and computes its parameters. It sends move and jump commands to the GB2004 module.

`IPathPlanner` computes a path between given points. It can be called directly from the bot or from `UT2004Navigation`. There are 4 path planners available in the PogamutUT2004. These are FloydWarshallMap (FWMap), NavMesh, UT2004AStarPathPlanner and UT2004AStar path planner.

FWMap uses Floyd-Warshall algorithm to compute the shortest paths between all NavPoints. If bot wants to navigate from a point that is not a NavPoint, the nearest NavPoint is found, a path is computed from it and a direct line from the original point to this NavPoint is added to the beginning of the path. Same principle applies for the destination point. Because the algorithm has $O(n^3)$ computational complexity, shortest paths between all NavPoints are computed at the start of the bot, and stored in memory, so when the bot asks for some path, it is only retrieved from the pre-computed matrix and it is available instantly.

Second path planner is NavMesh. It uses a navigation mesh to compute the path between the given points. A path is computed using the A* algorithm. The resulting path does not have to be the shortest possible. There are two main reasons for it. First is, that the centers of polygons are used as the location of polygon, so the resulting path leads through centers of polygons. This path is then shortened by the funneling algorithm (10), but it is limited to use the polygons in the original path. If there is some large polygon, through which the shortest path would lead, but only through the brink of it, and some other small polygon is near, its center could be nearer than the center of the large polygon and it could be chosen instead. It is illustrated in Figure 2.7. Second reason is the ignorance of

teleports. Euclidian distance is used as a metric for A* algorithm and teleports are not treated specially, so if the shortest path leads through a teleport, but the teleport is not situated on the shortest path without the use of teleports, it could be easily ignored, as is shown in Figure 2.8.

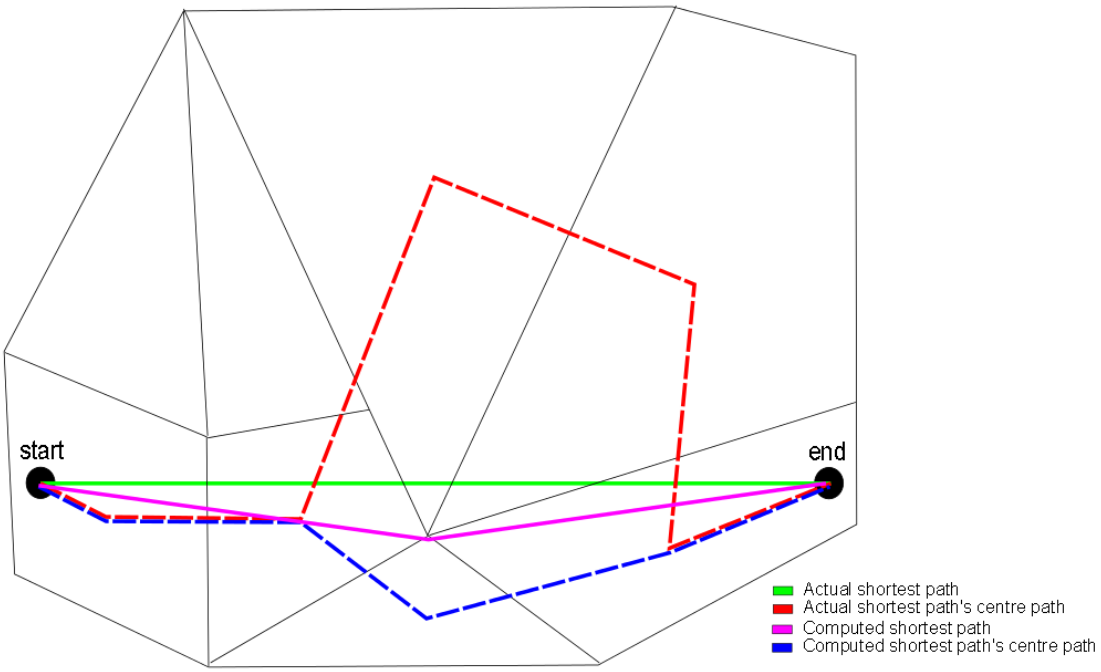


Figure 2.7: Not optimal path.

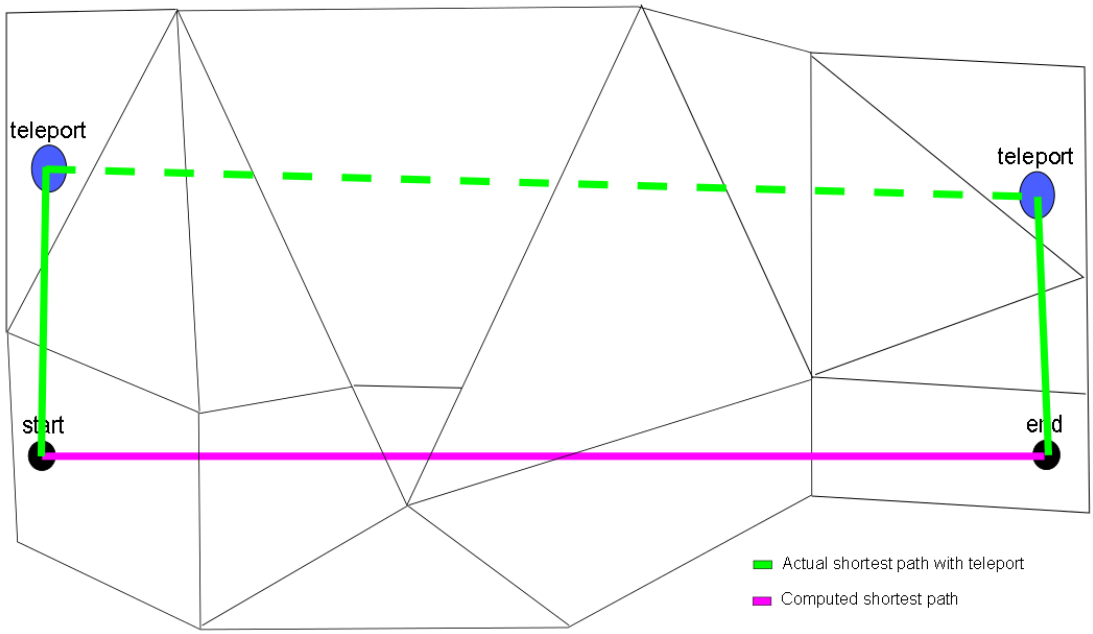


Figure 2.8: Not optimal path without teleport.

Third path planner is UT2004AStarPathPlanner which uses internal A* algorithm in the UT2004. A request for a path is sent to the GB2004 and the result is received via message from GB2004 as well. This implementation returns only paths of maximal length of 16. For longer paths, repeated requests from nodes further along the path are required.

Last available path planner is UT2004AStar, it also uses A* algorithm, but this time it is an extension of the general implementation available in Pogamut, so it is computed in the PogamutUT2004 without using the GB2004 or the UT2004. There is the same problem with teleports as in NavMesh path planner.

In the existing navigation system, FWMap is the internally used path planner, but paths from other path planners can be passed to it to navigate along them.

3. Related work

3.1 Area Awareness System

An area awareness system (AAS) was created by J. M. P. van Waveren in his thesis The Quake III Arena Bot. (11) It is a special 3D representation of the game environment. It uses 3D bounded hulls, called areas, with special property: navigational complexity from any reachable point in the area to any other reachable point in the same area is minimal. This is similar to properties of convex polygons in navigation meshes. Areas are constructed with the usage of BSP tree[2], which partitions the space on faces of obstacles.

To be able to compute reachability between areas, special process called portalisation is done. In this process, areas are bound by faces, which are polygons either representing solid walls or leading to other areas. Then the reachability can be established much easier. The AAS has several types of reachability, similar to types of edges in navigation graph. Special multi-level algorithm with caching is used for path planning purpose. Its cache stores per goal area, the travel times of areas toward this goal and the first reachability to use from these areas toward this goal. It calculates cache at two levels, both for areas in a cluster and for cluster portals. The AAS also holds information about entities in it, similar to types of navigation points in navigation graph.

The conclusion highlights the capability of create the AAS programmatically, without the need for human intervention that is required for navigation graphs. It also addresses the problem of rounding errors while creating the BSP tree for big maps, which should be solved by computing individual BSP trees for chunks of a map and then combining those smaller trees into one final structure.

3.2 Hierarchical A* algorithm

The Hierarchical A* (HA*) (12) algorithm uses homomorphism abstractions of the search space to create heuristics to improve the performance of the A* search algorithm. Valtorta's barrier is defined as a number of nodes expanded when blindly searching the state space. Homomorphism abstractions are abstractions which group several states of the original state space into a single state in the abstract space. The abstraction technique used is the max-degree STAR, which takes a state with the greatest degree and groups it together with its neighbors within certain distance, defined as an abstraction radius, into single abstract state. This process is repeated recursively until a space with only one state is created.

[2] „Binary space partitioning (BSP) is a method for recursively subdividing a space into convex sets by hyperplanes. This subdivision gives rise to a representation of objects within the space by means of a tree data structure known as a BSP tree." from Wikipedia. (16)

Naive HA* algorithm uses this abstraction to compute heuristics for the search in the base state space. As all of the searches in the abstract space has same goal during one search in the base space, their results are cached until the goal changes. Result of the naive HA* algorithm were very bad and thus several optimizations were presented. First optimization is called h*-caching and it caches the exact distance from all nodes of the resulting path to the current goal as it is known at the end of the search. Second optimization caches the whole path from each state with known h*, as the shortest path from such state is already known, and is called optimal-path caching. P-g caching saves heuristic for each state opened during the search and its value is P-g(S), where P is the length of the resulting path and g(S) is the distance from the start state to the given state. This technique absorbs the h*-caching as P-g(S) = h*(S) where S is on the solution path.

The resulting algorithm combines optimal-path and P-g(S) caching and breaks the Valtorta's barrier in 5 of 8 of the tested search spaces. This algorithm is combined with varying of the granularity of the abstraction, which experimentally determines the ideal abstraction radius for given search space, to finally break the Valtorta's barrier on all of the tested spaces. The future work should lead to the development of a better abstraction technique.

3.3 Shortest Paths with Arbitrary Clearance from Navigation Meshes

Author of this paper presents a Local Clearance Triangulation (LCT) (13), which is a new type of navigation mesh allowing efficient computation of optimal paths for arbitrary clearance. The LCT is composed of triangles, which represent not the walkable areas as in conventional navigation meshes, but they represent the obstacles in the environment.

The creation process of the LCT starts with a triangulation of input segments representing obstacles. A planar space is then triangulated and obstacles are reflected into the result as constrained edges of the triangulation. A local clearance property is added, which guarantees that only local clearance test will be required in the search algorithm, and using this property, the triangulation is refined into the LCT. A free path is such path, which does not traverse any constrained edge of LCT and a channel is a union of triangles a free path traverses through. A locally optimal path for the given clearance *r* is the shortest path on the given channel which remains in the distance *r* from all constrained edges in the LCT. A globally optimal path is the shortest path from all possible channels connecting the start and the end of the path.

Three main algorithms are presented, the first computing a channel of given clearance between the two points. The second is the extended funnel algorithm, creating the shortest path of given clearance in the previously found channel. The

third one is an algorithm for finding globally optimal path by investigating alternate channels until the globally shortest path is found.

Results show that the search algorithm for locally optimal paths is highly efficient and the search for global optimum is marked as much slower and presenting only small reduction in path length. It is concluded that locally optimal paths are perfectly suitable for navigation and can be computed very fast using the LCT.

4. Analysis

There were different sources of possible improvements to the navigation system in PogamutUT2004. Some issues were known and were reported directly by the PogamutUT2004 developers and users. Other issues were related to the proposed system changes and to the introduction of the navigation mesh as a new default abstraction of environment and a path planner. To gather all of the issues and information about them, the thorough analysis had to be done.

We will now describe the reduction of paths we studied, the clustering of maps to work with in both analysis and evaluation, the process of analysis of the navigation system, its state, limitations and proposed changes to it. First step in our analysis was identification of paths that the current navigation had problems following. We label them as navigation cases. We take the navigation case as the process of navigating from one place on the map to another.

4.1 Relevant paths

There are maps of very different sizes in UT2004, where even smaller maps have thousands of navigation cases and the largest CTF-Face3 map has over a million of them. To analyze and evaluate all navigation cases would take very long time. We could simply limit the number of processed navigation cases on each map (and we do so for some large maps eventually), but we wanted to find out if some of the navigation cases are more important than others. We did so by studying typical bot behavior in the game.

Basic filling of the game is the combat with bot's enemies. In DM mode, it is the only goal. There are different goals in other modes, but for fulfilling them, the bot always needs to best its enemies. Starting properties and equipment of all bots are the same and we assume that the combat skills are also the same. By combat skills we mean the parameters of the bot that can be set in the game, like accuracy of shooting, not the quality of the AI controlling the bot. The AI can make the bot dodge incoming projectiles or choose proper weapon for current combat conditions (available weapons and ammunition, distance of the enemy). With these same starting conditions and without the focus on the combat AI, we want to gain an edge for our bot and the way to do so is by picking up items. There are many types of items that can help the bot in combat: better weapons and enough ammunition for them, an armor to increase durability, health packs to replenish lost health (or to gain more than the base health with the super health pack), an adrenaline to be able to start special combos and a double damage bonus. Picking up items can be the difference between losing and winning.

Gathering of items is very important after spawning of the bot, when it has only basic equipment, but also after a fight when the bot can be exhausted by it and needs to replenish health or supplies. From this observation we determined the

more important navigation cases as the ones leading between items or player starts (places where the bot is spawned, so it has to travel from such places) and we mark them as relevant paths. We define a relevant point as a navigation point which is of the type of InventorySpot or PlayerStart and a relevant path as a navigation case between two relevant points.

We ran the initial evaluation and the basic evaluation only on relevant paths. All paths were evaluated in the thorough evaluation on limited number of maps only as is described in the following section.

4.2 Map clustering

There are over 100 maps in UT2004. Each map is designed for one or more of the game modes and PogamutUT2004 does not support all of them, so we limited ourselves only on maps for the supported modes. There are 91 of such maps. Working with all of these maps, studying and evaluating navigation system on all of them would be very time consuming, so we divided the maps into 4 groups.

We prioritized maps for DM and CTF modes, as those are the most popular ones. We also ran a basic analysis on each map, to determine its size and relative navigation difficulty. We counted the total number of paths and a number of paths, which can be built by FWMap path planner without using jump edges, lifts or both. The percentage of such paths can be seen in Figure 4.1 for several maps. On the most of the maps we also ran the full evaluation of the existing navigation system, at least on limited number of paths. This allowed us to divide the maps into 4 groups: Milestone, Reference, Excluded and Standard.

Milestone maps are such maps, where the original navigation system behaved rather poorly and thus there was the biggest room for improvement. There are also maps, which are often used in Pogamut lectures, tournaments or by the team of Pogamut researchers. There are 10 maps in this group. This group will serve as a main source for navigation issues and navigation on its maps will be studied in detail.

Reference maps are the opposite of Milestone maps. Navigation was pretty smooth on them even with the original navigation system. They were picked to control, that by improving navigation on Milestone maps, we do not worsen it on other maps. This group contains 8 maps.

Excluded group consists of maps, which were not suitable for our process. They were either too big (6 maps), so the evaluation would take too much time, there were different gravity settings (7 maps), which the navigation system is not designed to handle for, or we were not able to extract their level geometry to construct the navigation mesh at all (2 maps). Since these conditions cover only a few maps and we had many other maps to evaluate navigation on, we excluded

them. In total, we excluded 15 maps from the evaluation.

Rest of the maps is in the Standard group. These maps were evaluated only on relevant paths and were used mainly for gathering of results and their concrete navigation problems were not studied. There are 58 maps in this group.

Full evaluation was run on all of the maps from Milestone and Reference group for both relevant paths only and for all paths. Standard group was evaluated on the relevant paths only and in some cases, number of evaluated paths was limited due to the size of the map. Excluded group was kept out of the evaluation process completely.

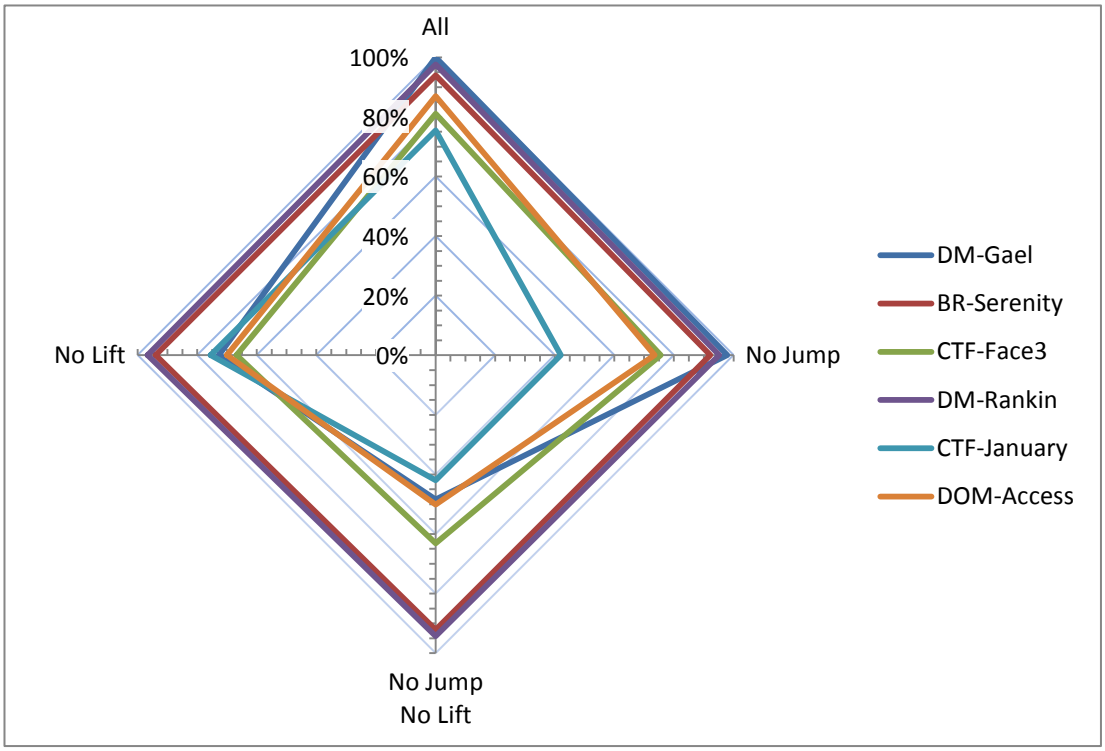


Figure 4.1: Percentage of built paths. All - No restriction on paths, **No Lift** - Paths built without using lifts, **No Jump** - Paths built without using jump edges, **No Jump No Lift** - Paths built without using both lifts and jump edges. The smaller the percentage is for a given map, the more intricate navigation is.

4.3 Work process

We investigated the source code of the navigation system and we also watched the navigation system in action. To watch the navigation system in action and detect possible problems, we proposed solution, which would provide us with the necessary data about the navigation system and ultimately be used for evaluation

and comparison of navigation systems. This solution is described in detail in 7.3. Replays of navigation cases were the most important output of this step of work.

Replays in UT2004 are captured by the game server and are played directly in the engine of the game. They are not videos, but only logs of bot actions in the game, so they have particularly small size (About 100 KB for 10s of the replay). This allowed us to capture and store them in great count. To sort these replays, additional information about navigation cases were captured. We saved the start and the end point of each case, providing us with enough data to reproduce it later while debugging the PogamutUT2004, in order to gain live insight into the interior of the navigation system. We also kept information about the exact location of and the nearest NavPoint to where the navigation failed for each of the failed cased. We grouped them by such NavPoints to assess the impact of issues reported by the failed cases. Some NavPoints are placed poorly thus causing navigation errors and this will also help us find such NavPoints.

Before we started improving the navigation system, we ran the evaluation on the maps marked as important by the Pogamut developers and on most of the smaller maps. We formed the Milestone and Reference groups based on the results of this evaluation. We studied the issues mostly on the Milestone maps. Results of this initial evaluation can be seen in Figure 4.2 for Milestone maps.

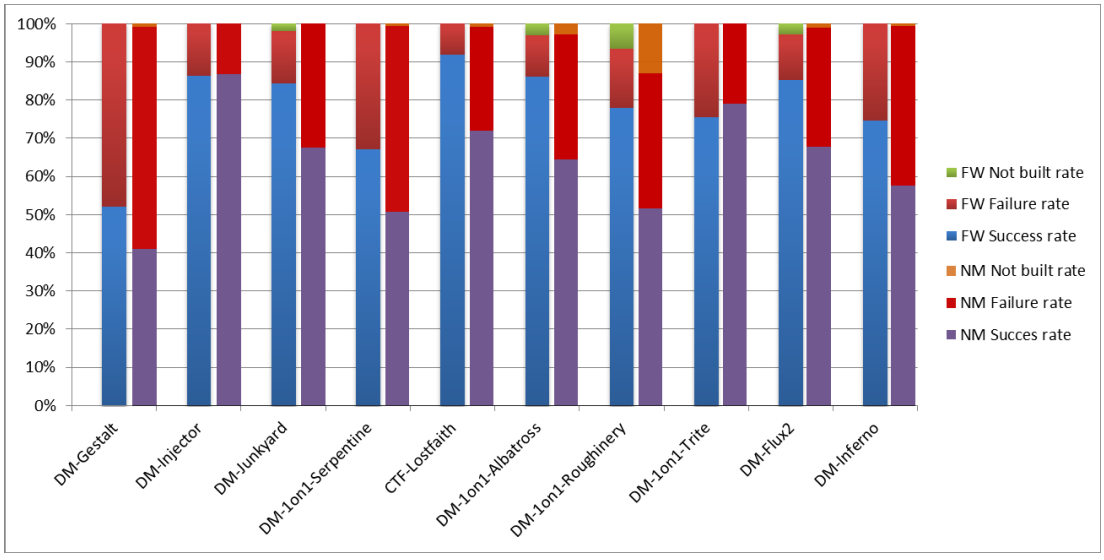


Figure 4.2: Initial evaluation results of Milestone maps. FW – original navigation system with FWMap path planner, **NM** – original navigation system with NavMesh path planner

To allow us to study navigation cases more closely and directly, we created the `NavigationTestBot`. It gets one navigation case and tries to follow it. After

reaching the target or after the specified time limit, it is respawned[3] to the start point of the case and it follows the navigation case repeatedly.

We set up the following process for studying issues of the navigation system on a single map:

- We grouped the failed results by the NavPoint where they failed.
- We ordered the grouped results by the count of failed cases for each NavPoint descending.
- We took several of the results from the top and for each:
    - We selected a few replays and played them.
    - We looked if the individual failures have the same cause and if the cause is something we could repair or prevent.
    - If we were not sure or wanted to study the case more closely, we used the `NavigationTestBot`, let it run the case repeatedly in debugging mode and either studied the computations directly in code or investigated the detailed log produced by its run.

Then we looked for similar issues across the studied maps and assessed if we would try to repair them. We were not able to repair some issues, as they were caused by poorly placed NavPoints, which would have to be solved by moving each such NavPoint manually, or by some other problem described in the limitations section following this one.

We repeated this process as a part of the development after every bigger change in the navigation system. We tested the impact of repairs or new features on the Milestone maps. We often ran the evaluation also on the Reference maps to see if we did not broke some other aspect of navigation.

4.4 Limitations of navigation on the navigation graph

We present limitations of navigation system working with a navigation graph as an abstraction of the environment we discovered during the process of analysis.

Lack of information about the environment. The navigation graph provides little information about the actual environment, as it consists only of immaterial points and links between them. This problem manifests itself mainly when there is the need to jump to successfully follow an edge. The agent has no information about the ideal take off point or about the ideal landing point and no sure way to determine those points, as it has no data about possible obstacles in the way of the

[3] Respawning is an action of reviving of a player after his death in the game to some PlayerStart NavPoint. It can be initiated from PogamutUT2004 to any location by a special command, which kills the bot and then spawns it at the proposed location.

jump. This leads to jumps with wrong take off point or power and they result in the failure of the path following.

Poorly placed NavPoints. Some navigation points are placed poorly and therefore navigation through them fails. It is the result of manual placement of the navigation points by game designers. A typical case is a navigation point placed too near to an edge, so when the agent switches to the link leading from such navigation point it has not enough time to perform the needed jump, to rotate itself to the desired direction or to gain enough speed for the jump. This limitation also often applies to the original navigation system with the navigation mesh, because the mesh contains off-mesh points generated from navigation points of the navigation graph, and these poorly placed NavPoints often lies outside of the mesh.

Low frequency of navigation logic. The frequency of 4 Hz is enough for general agent's logic cycle, but it is insufficient for precise navigation. The agent has to switch nodes early to avoid unnecessary slowing and stops during path following, which can result in crashing into some obstacle which would be avoided if the agent would run straight through the navigation point. The agent moves with maximal speed of 440 UT units per second while on the ground, so it moves quite a bit between logic cycles and it makes precise jumping impossible in some cases. This limitation also applies to the original navigation with the navigation mesh, as it uses the same frequency of navigation logic cycle.

4.5 Limitations of original navigation mesh

Aside from limitations mentioned above and also relevant for navigation system with the navigation mesh, there are some specific problems discovered in the original navigation meshes. The original navigation meshes are available as Attachment 6.

Bad dimensions of the mesh. Edges of the mesh are sometimes too close to walls or other obstacles, in some cases even lead right through them. There are also unnecessary holes in the navigation mesh and for some areas of maps the mesh is missing completely.

Planned path is outside of the mesh. There are cases, when the path created by the NavMesh path planner leads outside of the mesh, which almost always causes failure of the path following.

Unnecessarily longer paths. In maps with teleports, we observed NavMesh path planner creating paths without teleports, which could be much shorter if they would use the teleport.

Strange lift boarding behavior. The agent sometimes waits for the lift to return to the original position very far from the lift.

Jump pad ignorance. The agent traverses through jump pads when following path which does not anticipate the usage of such jump pad. It often results in jump pad deflecting the agent from its path and causing a failure of the path following.

Low quality meshes on maps with a terrain. The level geometry of some maps contains special terrain textures and such geometry typically produces meshes of lower quality. The terrain object covers whole surface of the map and it is replaced by standard static meshes on some places only. There is a problem with the export or the identification of such places and the navigation mesh is created from the terrain object even over those places.

5. Navigation mesh refurbishing

When we ran the initial evaluation of the existing navigation system, it showed us poor results of the navigation system which used NavMesh path planner. Some of the problems were accounted to the navigation system being built with the navigation graph as an environment abstraction in mind and therefore not so great at following the paths produced by NavMesh. However, closer examination showed that some problems emerged from the low quality of the meshes. We found several different issues regarding the meshes and then decided that we need to create better meshes for the mesh based navigation to be competitive. To create better meshes, we studied the navigation mesh creation process closely, to find bugs and possible improvements in it.

Better meshes would have impact on both path planning and path following. Meshes with higher quality represent the map geometry more precisely and therefore path planner is able to construct paths, which are possible to follow in the game. The path following part of the navigation system would also profit from better meshes, firstly by getting better paths to follow and secondly by an access to more reliable information about the environment, which could be used for the improvement of the path following.

5.1 Navigation mesh creation process

Original navigation mesh creation process consists of following steps:

1. UShock (14) map geometry extraction.
    - Input: map data of the UT2004 (*.ut2 files, textures and static meshes)
    - Output: XML file containing extracted data (static meshes, terrains, BSP tree)
2. Map geometry data conversion.
    - Input: output of step 1
    - Output: OBJ file (coordinates between -100 and 100), scale file (scaling coefficient), center file (original center point)
3. Generation of the navigation mesh by Recast (9).
    - Input: output of step 2, parameters specifying generation process and bot's attributes
    - Output: navmesh file (list of vertices and polygons)
4. Processing of the mesh in PogamutUT2004.
    - Input: output of step 3, navigation graph of current map
    - Output: processed file (serialized object representing processed mesh)

The UShock map geometry extraction is done by the altered version of the UShock program. The original UShock program serves for viewing maps of UT2004 and other related games and was created by (14). Map geometry data are extracted from the internal game data and saved in XML format. The resulting file contains definitions of static meshes, terrains and a BSP tree.

Map geometry data conversion is needed because the Recast program has strict requirements for input data. This conversion is performed by UT2004LevelGeom utility and it converts the data from XML format to OBJ format. OBJ files consist of the definition of vertices and triangles between them. Recast limits the coordinates of the input OBJ file to span between -100 and +100 and the transformation to this span is a part of the conversion. Information about the transformation, which consists of the original center of the map and the used scale factor, is saved for later retransformation of the resulting mesh.

The generation of navigation mesh is done by the altered version of the Recast program. Recast takes several parameters specifying the properties of generation, such as dimensions of tiles, maximal edge length, maximal edge error and maximal number of vertices per polygon. Other parameters specify attributes of the agent, its height, radius, maximal climb and maximal slope. Resulting mesh is then transformed to map's original coordinates.

The processing of the mesh in PogamutUT2004 happens the first time the mesh is loaded in PogamutUT2004, when only pure version of the mesh is available and its further processing is needed. Pure version is simply the output generated by Recast program. Other input is the navigation graph of the map, which is used for detection of reachable polygons and for adding off-mesh points and off-mesh connections. First step in the processing is the creation of BSP tree for the mesh's polygons. Unreachable polygons are removed in the next step and BSP tree is rebuilt without them. A polygon is reachable if some NavPoint lies in it, or if it shares an edge with a reachable polygon. Last step is the creation of the off-mesh points and off-mesh connections. A NavPoint is added to the mesh as an off-mesh point, if it lies outside of the mesh or if some off-mesh edge leads from or to it. An off-mesh edge is each edge, which does not lead completely on the mesh or it is a special type of edge (teleport, lift). The processed mesh is saved as a serialized object which represents it.

5.2 Mesh issues

We will now present the found issues related to meshes along with our solution of them.

5.2.1. Double radius computing

When planning a path on the navigation mesh, one must take into account agent's radius. This can be solved either by moving the points of the path away from the edge, if they are too close to it, or by creating the mesh with the agent's radius explicitly. It is especially convenient, if there are only a few different radii of the agents, which is our case, as all agents has the same radius. The original solution combined these two approaches, as part of the radius was reflected to the mesh generation and the rest was used to modify paths in path planning.

There was no advantage in handling the radius of the agent in two different places and the path modifying algorithm was buggy, so the agent often got stuck when moving on the edge of the mesh. We decided to transfer the responsibility of handling agent's radius solely on the mesh generation done by the Recast program. We increased the value of the parameter specifying the agent's radius and recreated the meshes. There was some improvement of the agent's movement along the borders of the mesh, but sometimes it still got stuck.

5.2.2. Borders of the mesh

To found out why the bot still got stuck, we observed the mesh directly in the game by using the draw feature, which draws the whole mesh of the current map along with off-mesh connections directly to the map. We found out that the borders of the mesh are sometimes not far enough from walls. The path was planned correctly on the mesh, but the mesh itself was the problem. There are several parameters in Recast, some specify agent's attributes and other specify the generation process, both having influence on the quality of the resulting mesh. We observed that parameters used in the mesh generation for UT2004 had same values as the default settings when opening the original Recast program. We tried to randomly change the values of the parameters to confirm that a better mesh can be created by their modification and we succeeded. After that, we tested many configurations of the parameters systematically to find the one most suitable for UT2004's maps.

5.2.3. Max edge error

While testing different configurations of the parameters for Recast, we found a bug in it. The max edge error parameter serves for setting of a maximal edge difference from ideal mesh for the smoothing process. This process results in creation of smaller and smoother meshes with lowered accuracy of them, as Figure 5.1 and Figure 5.2 illustrate. It takes the fragmented border of the edge and replaces it with a single straight edge, if error of such edge is lower than the max error parameter. The new edge lies whole in the original mesh, so the mesh is reduced, but the edge fulfills the requirements on the mesh. The problem was in this process, as the smoothing process produced edges which did not lie in the original mesh, therefore creating meshes which did not fulfill the limitations forced by the parameters.

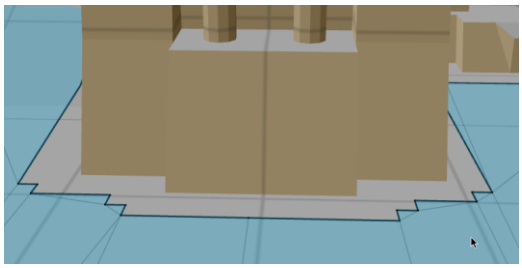


Figure 5.1: Navigation mesh with minimal edge error.

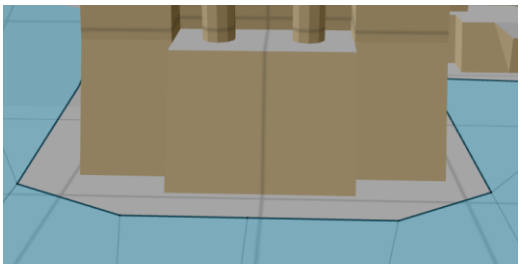


Figure 5.2: Navigation mesh with higher edge error.

We looked for help to the original developer of the Recast program and found out that newer version of the program was released and the bug was fixed in it. We took the latest version of Recast and recreated the alterations done to the older version for our needs. Smoothed edges of the mesh were no longer created outside of the original mesh and did not lower its quality.

5.2.4. Missing, extra, malformed and misplaced objects

When we observed meshes drawn in the map directly in the game, we took notice of strange holes in the mesh and that sometimes the mesh lead right underneath some object. Figure 5.3 and Figure 5.4 contain an example of such errors. We studied the creation of meshes with such issues in the original Recast program, which visualizes the map geometry and the generated mesh. There were suddenly objects in the map geometry in places of the holes in the mesh, and same objects were missing in places where the mesh led through them in the visualization in the game. Other objects looked deformed in the Recast program.

Figure 5.3: Example of holes in the navigation mesh. Holes are the red marked polygons. (They are not convex, so they cannot be parts of the mesh.)

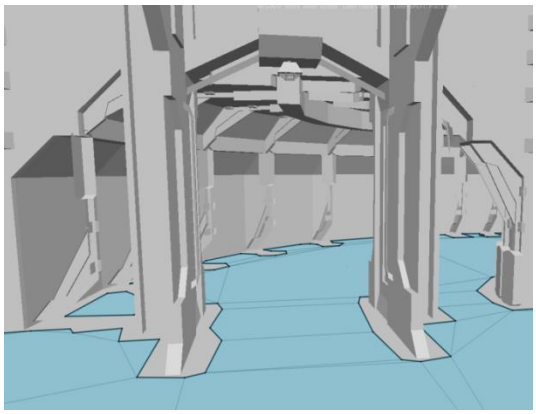


Figure 5.4: Example of holes in navigation mesh. Level geometry of the same place visualized in Recast program.

We investigated this by studying the same maps in UnrealEd (It is an editor of maps for several Unreal games, including UT2004.) and by studying the outputs of both the modified UShock program and of the UTLevelGeom utility. We identified the misbehaving objects and figured out that when they are transformed in the UTLevelGeom utility, the transformations are made in different order than in the UShock program (and therefore also in the game). We fixed the order of the transformations and the objects appeared in the right places in the correct shapes. Both original and modified versions of the UTLevelGeom utility are available as Attachment 4.

5.2.5. Semitransparent floor

Some floors in UT2004 are made of netting and so the player can partially see through them and can sometimes even shoot through them. Such floors are not exported by the modified UShock program and therefore meshes do not cover areas with such floors. Examples of missing floors in the exported level geometry are shown in Figure 5.5 and Figure 5.6. We did not find any way to detect such floors in the UShock program. This issue remains unsolved and is mentioned in the discussion chapter as a possible future work.

Figure 5.5: Semitransparent floor. Screenshot of semitransparent floor from DM-Asbestos map.

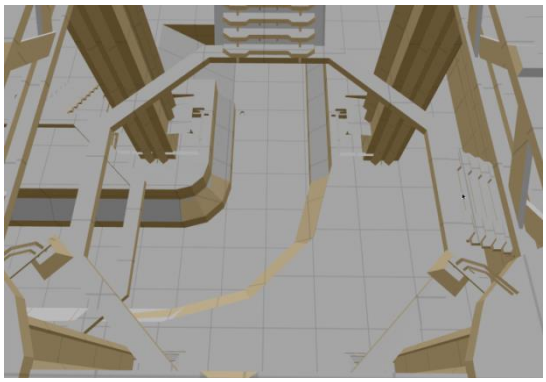


Figure 5.6: Semitransparent floor. Visualization of exported level geometry with missing semitransparent floor.

5.2.6. Map envelope

Map geometry of UT2004's maps often contains objects not related to navigation. Most of them are textures displaying sky or objects which purpose we do not know. They are often placed far from the existing geometry of an actual level and therefore greatly increase the span of the coordinates of the map. As the coordinates must be translated to the range of only [-100,100] for the mesh generation, the greater coordinate span reduces both the precision and the speed of this process.

We decided to lower the coordinate span by removing objects unrelated to navigation from the map geometry. We created an envelope block, constructed by resolving minimal and maximal coordinates of NavPoints in the map and by adding a margin to them. We did this by creating new task and bot for our evaluation framework which is described in 7.3. We used the envelope block in the UTLevelGeom utility in transformation of the map geometry to the Recast's input. We completely removed all triangles which had all of their vertices outside of the envelope block. For triangles which had only some vertices outside of the envelope block, we moved such vertices to the border of the block.

5.2.7. Big difference in quality of meshes

We studied meshes of several different maps and found out that the quality of the meshes differs greatly. We suspected that the cause was different quality of the extracted geometry of the maps, as we knew that there were sometimes problems with extracting map geometry for maps with terrains, but eventually we found the main reason. The parameters for the mesh generation process were not scaled along with the map geometry and therefore the process gave different results for differently scaled maps. Many of the maps had similar scale, as the map's space was often much greater than the map geometry needed, but even then, the scale coefficients differed from 20 to 200. This issue was deepened by the enveloping of the maps, which reduced their space to the actual size of the map. Some of the maps had almost nonexistent mesh, because the unscaled radius of the bot was greater than the width of its corridors where in other maps the unscaled radius resulted in too small distance of the mesh to the walls and the bot got stuck often.

We scaled the parameters along with the map geometry and regenerated the meshes. Improvement was significant.

5.2.8. Jump pads in navigation mesh

We watched the replays of failed navigation cases from the navigation system using the navigation mesh and discovered that the agent often had problem with jump pads. A jump pad gives the agent significant boost in the up direction, but it also leads the bot to specified place, so in fact it performs the whole jump instead of the agent. This problem did not occur in the case of navigation using navigation graph. In it, there were only one edge leading from the jump pad and the bot stepped on the jump pad only when it wished to use such edge. In contrast, the jump pads were covered by the navigation mesh and the agent crossed them unknowingly while pursuing path which did not use that jump pad's edge. We needed to prevent that.

We needed to exclude the area of the jump pad from the navigation mesh and only allow access to it through off-mesh connections. We could modify the processed mesh directly in the code of the new navigation system, but we decided that the most natural way would be to prevent the generation process from placing

the mesh over the jump pad in the first place. We created a new bot and a task in our evaluation framework, which saved all locations of jump pads for each map. In the map geometry transformation process, we added a pyramid object to the map geometry, preventing the creation of the mesh over each of these locations. The agent now steps on a jump pad only if it wants to explicitly use it. Example of the map geometry with the pyramid object over the jump pad is in the Figure 5.7.

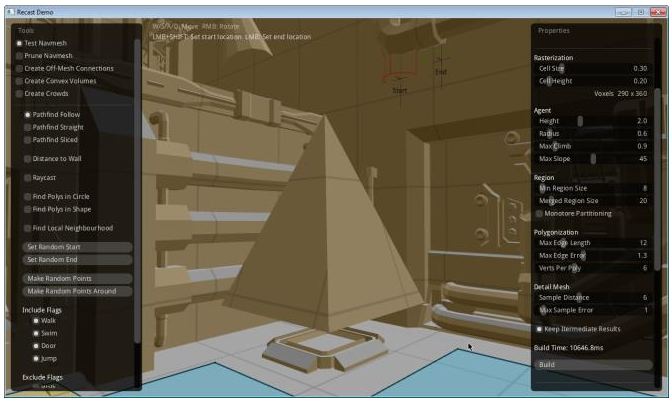


Figure 5.7: Hidden jump pad area. The area of the jump pad is made unavailable for the mesh generation by adding of an extra pyramid object over the jump pad.

5.2.9. Lift boarding

Path following through lift had certain rules in the old navigation system which used FWMap as a path planner. Agent went to the LiftExit NavPoint, where it checked if the lift was in favorable position for boarding and if it was not, it waited for it to descend to such position. After the lift was in the correct position, agent boarded it by going to the LiftCenter NavPoint. Getting-off the lift was done through navigating to another LiftExit NavPoint after reaching necessary height.

The problem with lift boarding with usage of navigation mesh originated from the fact that LiftCenter NavPoints were added as off-mesh points to the mesh and could be navigated to directly, without using a LiftExit NavPoint first. The problem manifested itself, when the agent had to wait for the lift to descend. The agent often waited too far from the lift when the link leading to the LiftCenter NavPoint was long and as the navigation system waits for the favorable lift position at the last node before the LiftCenter NavPoint. It looked strange and was impractical if someone was to compete with the agent for the use of the lift.

We tried to solve this issue by adding an extra node to the path, which would divide the last link to the lift. It would be close enough to the lift to look natural, but not too near to be under the lift, where the waiting agent would block the lift in its descending. Lifts have different sizes and only reliable information about lifts we get from the game are the updated positions of LiftCenter NavPoints, which move with the lift, so we did not have any way to determine the correct location to place the proposed node in. We wanted to overcome this deficiency of

information by simply using the length of the link between the LiftExit and the LiftCenter NavPoint for given lift, but lifts are sometimes rectangular and it would not have to work in such cases.

In the end, we removed LiftCenter NavPoints from the list of possible off-mesh points, therefore such NavPoints were not placed on the mesh, forcing the usage of an off-mesh connection from LiftExit NavPoint to be used when boarding a lift.

5.2.10. Bugs

We identified and fixed several bugs in the code of the classes handling and using the navigation mesh. Polygons of the navigation mesh are indexed from 0, but in some methods, the index 0 was considered a failure value. Neighboring polygons are often looked up while constructing paths. The neighboring polygon was identified by the shared edge, but when calling the method recursively, instead of passing the already found polygon, only the location predicted to lie in it was passed and it was looked up again. When off-mesh points were detected, they were added to the nearest polygon, but there was a bug in the computation of the distance of the polygon to the off-mesh point and it sometimes resulted in placing the off-mesh point to a bad polygon. The result of this were off-mesh connections linking different floors, as some off-mesh point was placed to a polygon on a different floor than it in fact lied on. There was also a bug in the funnel algorithm, where it did not shift the polygon in which it looked for the next node of the path, resulting in invalid paths leading outside of the mesh. All of these found bugs were fixed and that improved the quality of the path planning.

5.3 Refined navigation mesh creation process

In process of fixing the issues related to meshes, we significantly modified the navigation mesh creation process. Here is the structure of the refined version of the navigation mesh creation process:

1. UShock map geometry extraction.
    - Input: map data of the UT2004 (*.ut2 files, textures and static meshes)
    - Output: XML file containing extracted data (static meshes, terrains, BSP tree)
2. Extraction of additional map information from UT2004.
    - Extraction of jump pads for blocking them on the navigation mesh.
        i. Input: navigation graph of current map
        ii. Output: list of jump pad's locations on current map
    - Construction of the envelope block for current map.
        i. Input: navigation graph of current map
        ii. Output: coordinates of the envelope block without margin

3. Map geometry data conversion.
    - Adding of jump pad blocking pyramids to the map geometry, cropping the map geometry by the envelope block and transformation of the map geometry to OBJ format with limited coordinates.
    - Input: output of step 1, outputs of step 2
    - Output: OBJ file (coordinates between -100 and 100), scale file (scaling coefficient), center file (original center point)
4. Generation of navigation mesh by Recast (Attachment 5).
    - Parameter's values fitted to UT2004 and scaled to current map.
    - Input: output of step 2, fitted and scaled parameters specifying generation process and bot's attributes
    - Output: navmesh file (list of vertices and polygons)
5. Processing of the mesh in PogamutUT2004.
    - Off-mesh points assigned to correct polygons, lift centers accessible only by off-mesh connections, polygon with index 0 is valid.
    - Input: output of step 3, navigation graph of current map
    - Output: processed file (serialized object representing processed mesh)

Meshes generated by this refined process are available as Attachment 7.

6. Navigation system upgrades

We present and explain proposed system changes, which emerged from the analysis, their solution and their impact on the quality of the navigation system.

6.1 Faster ticking of navigation logic

An agent's logic runs in the interval of 250 milliseconds and is invoked by receiving of the synchronous batch message from GB2004, which contains actual information about both the agent and the environment. This is closely described in the Background chapter (Ch. 2). The navigation logic runs in the same interval after the agent's logic is finished, to reflect the latest commands from the agent.

This interval was determined by the performance of the UT2004 server, GB2004 module and an agent's Java code and is sufficient for most of the tasks of the agent. However, it is not sufficient for navigation purposes, as path following requires some tightly timed tasks:

- Switching to the next node – If the agent reaches the current destination before it is switched to the next node, it simply stops there and waits for a new move command. This is partially solved by the possibility to specify the next consecutive target in the move command and by switching to the next node early. NavPoints are usually far enough for this to work in the navigation system with the FWMap path planner, but when using the NavMesh path planner, nodes of the path are often very close to each other, especially when avoiding some obstacle with a complex shape, and it does not work. Switching to the next node early can result in crashing to some obstacle or a wall.
- Jumping – Performing a jump is a difficult task with such long interval. It can be too early to jump in one tick of the logic and too late in the other, especially when the node is placed on the edge, which JumpSpot NavPoints often are, or when performing a jump directly after the switching to next node, which often results in change of the direction and decrease of the speed, so we do not have ideal jumping conditions.

Problems with these path following tasks led to the decision, that the navigation logic will be evaluated more often. It will be invoked by a new type of the GB2004 message, `LocationUpdate`. We will pass some information about the agent inside the message, but we must limit the amount of the information to be able to send it often enough. The `LocationUpdate` message contains information about agent necessary for path following:

- location –3d coordinates of agent's current location (3 decimals),
- velocity–agent's current velocity in each direction (3 decimals),

- rotation – agent's current rotation in each direction (3 decimals).

We bound the navigation logic evaluation cycle to the reception of the `LocationUpdate` message and tested the performance of the related systems. We changed the interval for sending the `LocationUpdate` message and tried to run increasing number of agents on several maps. We came to the conclusion that the navigation cycle can run 5 times more often than the agent's logic cycle. At this rate we were able to run 12 agents on the CTF-LostFaith map without any delay in message receiving and processing.

The new interval of 50 milliseconds for navigation cycle logic provides us with ability of more precise path following, but we have to adjust the whole navigation system to this new interval.

6.2 Link filtering

UT2004 contains many tricks how the player can get to the otherwise inaccessible place or how to get to some place by shorter way than by a standard movement. Some opportunities for such tricks are marked in the navigation graph and native bots from UT2004 can perform them there too. We limit ourselves to description of the native bot's tricks marked in the navigation graph, as we have no other way to find places appropriate for these tricks.

We identified these types of tricks:

- **Wall jump**– Some maps contain special places, where the wall is not vertical, but only very steep, or there is some object leaning against the wall, forming a steep surface. Such places can be used for the wall jump trick. It consists of performing the dodge move against the steep surface, which does not result in the movement away from it as normally, instead it moves the bot up. At the highest point, an air jump can be performed, together with the boost from the dodge, allowing to reach otherwise unreachable heights.
- **Lift jump**–Some lifts provide the possibility to reach certain places by using a lift jump. These places are often natural 3rd floors of such lifts, but where the lift does not ride. A lift jump consists of performing double jump while riding a lift. The success of such jump depends on the bot's speed and direction of its movement as well as timing of the jump. Some of these lift jumps are pretty easy, even average player can perform them successfully almost always, but other require precise movement and great timing.
- **Weapon jump**– Projectiles of some weapons in UT2004 affect the vicinity of its impact point (e.g. rockets). This can be used to gain extra boost for jumping. Along with the boost from such shot, the bot also suffers damage from it. The rocket launcher and the loaded Shield gun can be both effectively used for weapon jumping.

- **Translocator** – In some maps, the bot is equipped with a special device called Translocator. It gives him the capability to shoot a beacon to some place and later use the secondary mode of the Translocator to teleport itself to the place where the beacon lies. With good aim, the beacon can be shot to otherwise unreachable places and then the bot can teleport itself there.

We discussed possible disclosure of some tricks to the navigation system. Wall jumps could be implemented, but there is a problem with how successful would we be in the precise timing needed for the jump after the dodge. Also there is not any known way to identify the wall jump specifically from the data provided by the GB2004 module. Some lift jumps are quite easy from the player's point of view, but others are very hard and we have no clear method to differentiate between them. They are also very sensitive to timing, and even our new faster navigation logic cycle would not have to be fast enough. With weapon jump and Translocator, we would need to use resources not otherwise available to the navigation system, because we would have to temporary take over control of the weapon selection as well as shooting from weapons. In addition, weapon jump would hurt the agent and could even kill him instead of providing the desired jump. This goes well beyond the competency of the navigation system and it could not be part of the internal logic of the system. At most, it could be provided as a special feature, available outside of the navigation system on an explicit request from the agent, but it would not be very useful.

We rejected the implementation attempt of weapon jump and Translocator trick and after facing the difficulty with identification of suitable dodge and lift jumps, we rejected them as well, but we leave them open for possible reconsideration in future work.

Special tricks are marked by special flags on navigation links of the navigation graph. These flags were evaluated and selected links were filtered out of the links available for computation of Floyd-Warshall matrix in FWMap path planner. Original implementation of the NavMesh path planner also removed some link from consideration as off-mesh connections, but used slightly different evaluation process. We unified this process for both path planners, we fixed a bug related to the identification of lift jumps and we moved the process to the new `UT2004EdgeChecker` class, where it is available as a static method.

6.3 Jump module

When we studied the replays of failed navigation cases, we noticed, that apart from failures caused by a bad path provided by the path planner, most of the failures were caused by problems with jumping. This is to be expected, as jumping is hardest of the path following tasks. We used the advantage of more information about the environment, given to us by the navigation mesh, to properly plan and

execute the jump sequence. To do this, we created a new Jump module, which takes responsibility for all computations related to jumping.

Jumps are the responsibility of the path runner. In each navigation logic cycle, it analyzes the current navigation link to decide, if a jump is necessary. It decides using information about the link from the game engine. If it chooses, that there will be a jump, it tries to determine, whether it is already the right time to start it or if it should wait and reevaluate it in the next cycle. The whole jump processing sequence looks like this:

1. Determine if the jump is needed on the current link – Each of these is a cause for jumping:
    a. signal from path navigator,
    b. jump flag on the link,
    c. force double jump flag on the link,
    d. needed jump data present on the link.
2. Resolve the jump – Check whether to force the start of the jump, based on the distance to target, or if the jump is needed at all.
3. Prepare the jump –If the jump is not forced from the previous step, we check whether it is a right time to initiate a jump sequence. To continue with the jump computation, following conditions must be fulfilled:
    a. Agent has a sufficient velocity. (>= 200 UT units/s)
    b. Agent is not too far from the target of the jump. This is decided by computing maximal distance the agent is capable of jumping based on its current velocity and by comparing it to the agent's distance from the target.
    c. Deviation of the agent's movement vector from the direction to the target is small enough. (< 20 degrees)
4. Based on the difference in z coordinates of the agent and the target, either jump or fall sequence is initialized:
    a. In the jump initialization, type of the jump and its power is computed, and if the jump is not forced, it can be delayed if the computation is not successful. Reasons for failed power computation are:
        i. Difference in z coordinates of the agent and the target is too great. (> 130 UT units)
        ii. The target is too far from the agent.
    b. In the fall initialization, the distance of the fall itself is computed and jump is used to overcome the rest of the distance to the target.
5. If the jump was not delayed, it is performed by sending the jump command with information about the type of the jump, single or double, the possible delay for double jump and the power of the jump to GB2004.

The jump processing sequence uses a lot of heuristics and observed constants. We improved this process by using the advantages of faster navigation logic cycle and more information about the environment from the navigation mesh.

6.3.1. Jump curve analysis and interpolation

The current path runner computes the needed power of the jump by using multiple methods with equations, created by watching the agent jump in the game and by analyzing of the agent's properties during the jump. We want to improve the accuracy of the computation of the power for the jump.

We could try to interpret data about the jump provided by the game engine. Each jump is represented by the vector with three values, one of them supposedly the proposed power of the jump. Interpretation of this data was already tried before by Pogamut developers and concluded as ineffective, so we went different way. We used the faster navigation logic cycle to gather more precise data about jumping and created new jump estimations from them.

We created a bot which logged all information from the `LocationUpdate` message (location, velocity, rotation) after receiving it. We let this bot run on the flat surface and periodically repeat jumping. We changed the type of the jump and its power to capture its relation to the jump's height and distance. Eventually we let bot perform a fall down jump, to gather more data about falling down.

We took the log and extracted data describing one jump. We got the parameters of the jump command and list with entries for each tick of the navigation logic cycle, containing a timestamp, a location and a velocity of the bot. We extracted several jumps with different command parameters and interpolated each with quadratic curve representing relation between time and a difference in z coordinate for given jump. We found the linear relation between the power of the jump and its curve. We extracted equations describing these relations which take time and desired z coordinate difference and output necessary power of the jump to reach the desired height in given time.

The equation for a single jump is:

$$power = \frac{(targetZ\ +\ 475\ *\ time\text{^}2)}{time}$$

The equation for a double jump has an extra parameter called delay, which represents the delay between the first and the second jump, and the equation looks like this:

$$power\ = \frac{(targetZ\ +\ 475\ *\ time\text{^}2\ +\ 20\ *\ time\ -\ 140)}{\big(1.066\ *\ (time\ -\ delay)\big)}$$

It is also limited to time values greater than the delay value.

We get the *targetZ* as a difference between z coordinates of the target and the agent's current position. The time is simply computed from the 2d distance (ignoring z coordinate) between the agent and the target and from the agent's current velocity. We determined that there is a speed boost invoked by jumping, so the final equation for computing time parameter is:

$$time = \boldsymbol{d} + \frac{(distance2d - velocity * \boldsymbol{d})}{(velocity * \boldsymbol{boost})}$$

, where ***d*** = 0.1 and ***boost*** = 1.08959

The constant ***d*** represents the average delay between sending of the jump command and the speed boost exhibiting itself and the ***boost*** constant holds the rate of the boost. For very short distances this equation is supplied by simple division of the *distance2d* by the *velocity*.

In UT2004, if a bot collides with the environment during jump, the maximal height it reaches during the jump is not affected, so we expressed the relation only between a power of the jump and its maximal height to be used in cases where we predict a collision. This equation has the following form for a single jump:

$$power = 3.87 * height + 111$$

It is valid only if the height on the input is lesser than 60 UT units. For heights between 60 and 130 UT units, where a double jump is needed, the following equation applies:

$$power = 3.136 * height + 287$$

All of these equations are part of the Jump module as functions and together with heuristics for different nonstandard situations, which are described later in this chapter, form the core functionality for computing the required jump power.

6.3.2. Jump boundaries

When the path runner processes a jump sequence, it has just a little information about the environment. It does not know where the jump could possibly start at the latest, only the start point of the jump link, or where the agent could land the earliest, only the end point of the jump link. In UT2004, there are often long links, much longer than the jump which needs to be performed to follow such link successfully. Even when the agent's distance would allow him to land directly on the end point of the link, it can be impossible to do so, due to some obstacle in the air,

like low ceiling or some pipe, and the agent must in fact land much earlier to avoid such obstacle and to succeed in the following of the link.

We used the information provided to us by the navigation mesh to find the points at the edge of the mesh, therefore giving us the latest take off point and earliest landing point and use them in the path following to avoid problems described above. We created the `JumpBoundaries` class, which carries useful information about the jump for the given link.

`JumpBoundaries` contain following information:

- Jumpable – flag signaling whether the jump on the current link is doable,
- Navigation target – an end point of the current link,
- Landing target – a point on the edge of the mesh from the direction of navigation target or the navigation target itself if it lies outside of the mesh,
- Take-off min – a point between the start point of the current link and an edge of the mesh from the direction of start point, closest to the start point, from which some jump would reach the landing target,
- Take-off max –a point in the same range, closest to the edge of the mesh, from which some jump would reach the landing target,
- Take-off edge direction – a direction of the edge of the mesh through which the link crosses from the start point direction (null if the start point lies outside of the mesh),
- Target edge direction – same as before but for the direction from the end point (null if the end point lies outside of the mesh).

This data are used for determining of the right moment to start the jump and to compute the correct power for it. Mesh edge's directions are used for heuristics for the falling down and for the collision prediction.

6.3.3. State prediction

When a jump command is sent to GB2004, it takes a moment, before the jump begins. In this moment, the agent's attributes can change significantly, influencing the jump, which was already computed and started. We cannot eliminate this delay, but we measured it and present predictions of the most important changes affecting the jump.

An average delay between sending of the command and a start of the jump is about 50 milliseconds, which corresponds to one navigation logic cycle and we have no tools to measure it more precisely, so we work with this delay.

There are two main attributes affecting a jump, which can change before it starts: the agent's velocity and the angle deviation from the desired direction. We

analyzed the collected data from various logs of bots following paths and determined the following equation for the agent's velocity prediction:

$$correctedVelocity = 0.9788 * velocity + 111$$

The corrected velocity is the predicted velocity at the time of the take-off and is limited by maximal velocity.

For the angle deviation, we determined that it is simply halved during the delay.

We use these corrected values in the jump computation to achieve more precise jumping. In this case, predictions allow us to reduce the risk of overpowering the jump and agent landing farther then wanted.

6.3.4. Jump collision heuristic

Sometimes, the bot collides with a wall during a jump. Such collision affects its speed only in the horizontal direction, the vertical speed stays the same and the bot reaches the same height as without the collision. If the jump has a sufficient power to reach the target height, the bot will still reach it after the collision, but it will probably land sooner than computed, so it would have to finish the path to the original target by running there. The only possible complication is when the target platform is sloped and the bot approaches it from such direction, which is not perpendicular to it and it leads up to the higher part of the platform. In such case, the bot is moved along the colliding wall to the higher part of it and therefore needs stronger jump to reach its top.

We use the information about the target's edge of the mesh direction, extracted when computing `JumpBoundaries`, along with the approaching direction of the bot, to detect such case. We compute the actual point of the collision, taking into account the radius of the bot, which acts as a bumper from the side of the bot and also as a reserve between the actual edge of the wall and the mesh. After we know the point of the collision, we use it to determine the time spent in the collision, which moves the bot to the higher part of the wall and create a boost coefficient for the original jump power, to successfully reach the top of the wall in the higher place.

We tested this heuristic in action, and it showed promising results, but sometimes the collision did not occur and the overpowered jump took the bot far beyond its original target. Video of the overpowered jump is part of the Attachment 8. But the positive influence beats these rare failures, so we embedded this heuristic to the Jump module.

6.4 NavMeshNavigator and Runner

A navigator and a runner are the two main classes in the PogamutUT2004 navigation system responsible for path following. The original `LoqueNavigator` and `KefikRunner` were both fitted to work with paths built on the navigation graph. When the navigation mesh was first introduced to the PogamutUT2004, this was one of the reasons why the NavMesh path planner performed worse than the FWMap path planner. We created `NavMeshNavigator` and `NavMeshRunner`, fitted to work with paths built by the NavMesh path planner. They are also aware of the faster navigation logic cycle and of the new Jump module, which `NavMeshRunner` uses for all jump related computations and decisions. The structure of the new navigation system is shown in Figure 6.1.

6.4.1. Differences in paths produced by FWMap and NavMesh path planners

In a path from the FWMap path planner, built on the navigation graph, all nodes are also NavPoints, containing additional information connected to the other nodes by links of the navigation graph, also carrying more information about themselves. The existing path navigator and runner sometimes rely on such information, but in a path from the NavMesh path planner, nodes are often common points of the navigation mesh, short of the needed information. The same applies to links connecting nodes of the path. We removed the dependency on the additional information from both classes, but sometimes the NavMesh path contains NavPoints (off-mesh points) and navigation graph links (off-mesh connections) and in such cases, we take advantage of the additional information they carry, we just do not depend on it.

6.4.2. Faster navigation logic cycle

A navigator and a runner make decisions, which are time related. The navigator determines if it is already the right moment to switch to the next node, the runner decides if the time is right for beginning of the jump. Both of these decisions depend on the current situation of the bot, but also on how long it will be until next such check. And this is what has changed by making the navigation logic cycle faster, as it is now only 50 milliseconds instead of 250 milliseconds between such checks. This has great influence on conditions used for determining results of those decisions and we had to modify both the navigator and the runner significantly.

6.4.3. Jump module integration

We integrated the Jump module into the new `NavMeshRunner` class. This gives us the ability to decide about jumps and compute them more precisely. Thanks to `JumpBoundaries`, we now often know how far we are from the latest take-off point and if we have the sufficient reserve before reaching it, we tighten the rules for starting the jump and possibly wait for better conditions before jumping. It

allows us to wait for the situation, where the agent has a stable speed and moves in the correct angle to the current target, minimalizing the impact of the inaccurate predictions of the acceleration and the angle correction. In `NavMeshNavigator`, we can switch to the next node later, minimalizing the risk of changing the direction too early and colliding with some obstacle as a result.

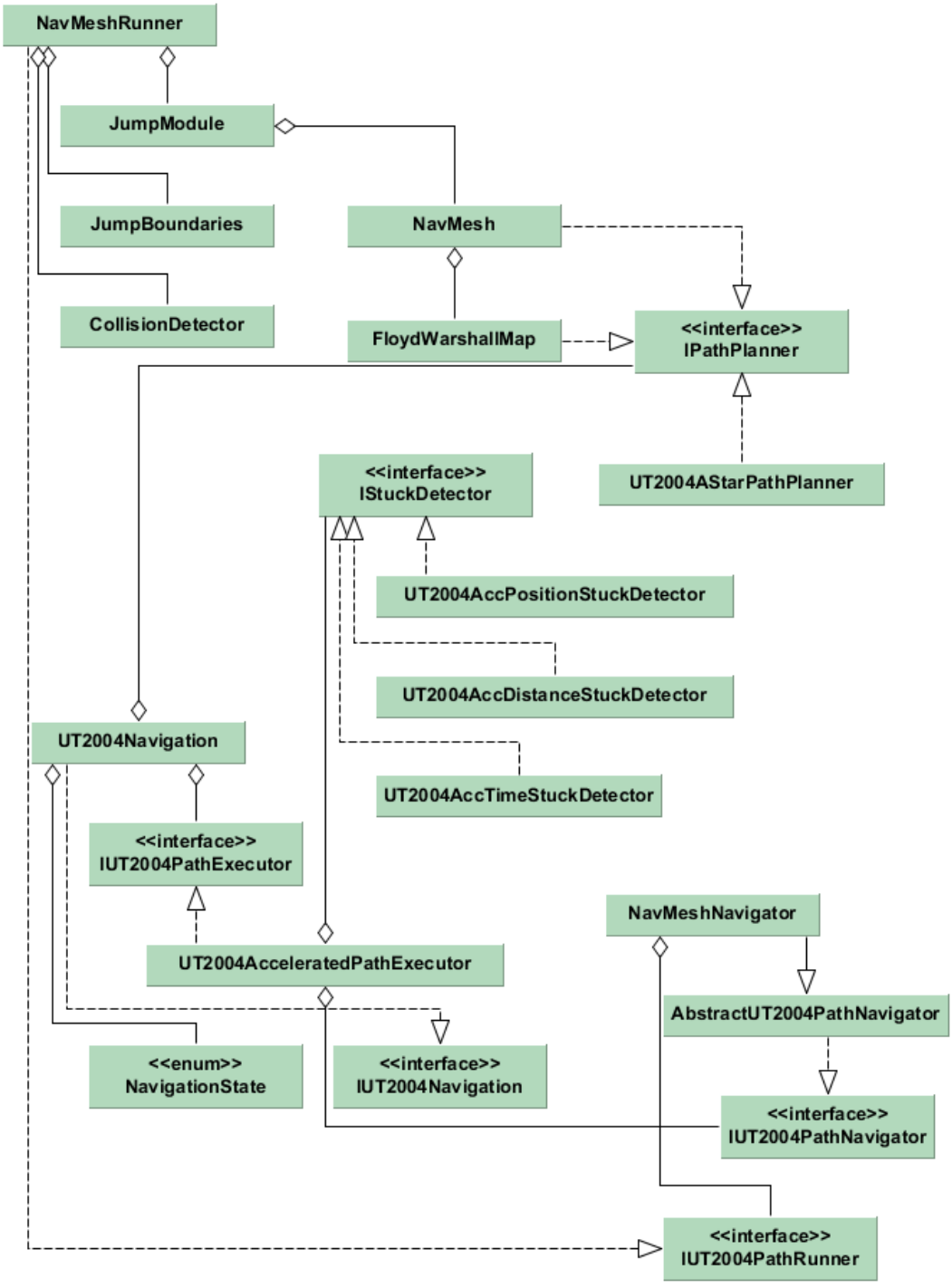


Figure 6.1: Structure of the new navigation system.

6.4.4. Fall down heuristics

When the agent reaches the edge of the floor, it should simply fall down and we count on it when computing the fall down jump. But in UT2004, there is a mechanism, preventing the agent to simply fall down, if it approaches the edge under small angle, resulting in sliding along the edge in the direction of the approach. In such cases, we have to have the agent jump a little, overcoming the edge and correctly falling down. To detect the case when we are approaching the edge under the angle resulting in the sliding behavior, we use the `JumpBoundaries`. It contains information about the point on the brink of the mesh and about the direction of the mesh's edge on which such point lies on. We compare the direction of the edge with the approaching vector of the agent and if it is needed, we add the little jump instead of letting the agent simply fall down. This prevents the agent from sliding along the edge away from the desired path.

6.4.5. Collision detection

Even after the refurbishing of the navigation mesh creation process, navigation meshes are not perfect which can lead to the agent colliding with the environment even when following a correct path from the view of the mesh. Other situations for such collisions rise after jumping, if the jump is underpowered of overpowered, it can lead the agent away from the original path and again, result in a collision. The most common objects causing collisions are railings, blocking the movement of the agent, while staying invisible to us on the navigation mesh.

There is a built in system, where we receive a `WallCollision` message from GB2004 and we can react to it in navigation logic. There are some situations, which we take as a collision, but we do not receive the `WallCollision` message and to detect them, we created `JumpCollisionDetector`, which takes current information about the agent's movement, evaluates it against the previous data and determines, if the agent is colliding. If the agent is colliding, we try to release it by performing the full double jump, possibly overcoming some, for us, invisible obstacles.

The combination of listening for the `WallCollision` message and evaluation by `JumpCollisionDetector`, we have a fair chance of detecting and handling a collision, before stuck detectors signal that the agent is stuck, causing failure of the path following process.

6.5 Teleports in navigation mesh

Many maps in UT2004 contain teleports, which can significantly shorten the distance of a path between two places, as they transport the agent immediately from one end of the teleport to the other. The FWMap path planner is aware of teleports and uses them in the path planning process to plan the shortest possible path. As the Floyd-Warshall algorithm builds the matrix with shortest paths from

each NavPoint to every other NavPoint, it begins with paths between directly connected nodes and in cycle it checks for shorter paths one node longer. This process assures that the information about the teleport spreads itself and in the end, the teleport is considered for every path.

On the other hand, the NavMesh path planner is not aware of teleports and uses them only, if it encounters them during the path planning. NavMesh uses the A* algorithm for the path planning and it works with the heuristic metric to estimate the remaining distance to the end. The current A* algorithm uses the Euclidian distance as such heuristics. It also has a depth first approach and only returns to alternate paths when the shortest estimated one does not exist. The combination of this approach and the Euclidian distance metric result in the state, when only way to use the teleport is if it lies on the shortest estimated and existing path without the use of the teleport link. We changed this situation and use teleports every time they lead to the shorter resulting path.

We could solve this by using a different metric for estimates in the A* algorithm, one which would take teleports into account. We considered this approach, but in the end, we decided on a different solution.

We already had a tool with the sufficient information for effective use of teleports, FWMap. In most cases, the NavMesh path planner builds similar paths as the FWMap path planner as paths of the NavMesh path planner are navigation mesh aware optimizations of the FWMap's paths. Where the FWMap path leads between NavPoints, the NavMesh path is not limited to NavPoints and can plan custom lines on the navigation mesh, but if we acknowledge that the navigation graph is dense enough to contain links in every reasonable direction, the resulting paths of both these path planners will lead through the same areas. We decided that this is the case for most of paths and we took advantage of it in our approach to teleports.

To plan a path with the NavMesh path planner which will use the advantage of teleports, we proposed following process:

1. Check if the current map contains teleports. If not, build the path directly with the NavMesh path planner. This is done only once and stored in the NavMesh class.
2. Build a path using the FWMap path planner and check if it contains teleports. If not, plan the path with the NavMesh path planner.
3. For each teleport in the path:
    a. Split the path at the teleport.
    b. Plan the path from start to the teleport with the NavMesh path planner.

c. Repeat for the rest of the path. (From the teleport exit to the end.)

4. Join sub paths into the resulting path.

This process is possible thanks to the speed with which the path is planned in the FWMap path planner (Paths are pre computed and only looked up in the matrix.), so it does not delay our path planning process much.

Our solution fulfills our requirements, it allows the NavMesh path planner to fully use the potential of teleports and its computation is fast enough. Attachment 8 contains videos with agents navigating from and to the same location, one is from the original navigation system without the use of teleports and the other one uses our solution for using teleports.

7. Evaluation

We needed to evaluate our new navigation system and compare it with the existing ones. Every change we made and any improvement we proposed and implemented should enhance the capabilities of the navigation system. But we need to confirm that this will still be true, when they are all combined together. We also need to compare our navigation system against its predecessors, to certify the fulfillment of our goals.

We used the evaluation from the beginning of our work on the new navigation system, first to gather the information about original navigation systems and their deficiencies and problems, then to gather same information about our new navigation system when testing it after implementing every new feature.

7.1 Compared navigation systems

During our evaluation, we compared 3 navigation system configurations:

1. The original navigation system with the FWMap path planner – This was the default navigation system configuration in PogamutUT2004.
2. The original navigation system with the NavMesh path planner – The NavMesh path planner brought the navigation mesh to PogamutUT2004 and was supposed to replace FWMap as a default path planner. But the imperfection of meshes and incompatibility with the original navigation system resulted in lower success rate than the first navigation system and prevented the replacement from happening.
3. Our new navigation system with the improved NavMesh path planner – Result of our work, with a goal to overcome and replace the original navigation system, while bringing the advantages of navigation mesh to PogamutUT2004.

Results of these configurations served for their comparison, as well for identifying of different kind of problems. First configuration's result showed the imperfections of the original navigation system as a whole. Results of second configuration pointed at possible issues when building a navigation system with the usage of the navigation mesh. And the continuous results of our navigation system showed its bugs and problems during development stage.

7.2 Evaluation process

A single task for evaluation is to assess data about performance of a given navigation system on a given map. The only method for gathering data is to actually have the bot run in the environment and to have it use the given navigation system as it is impossible to simulate navigation outside UT2004.

A base unit of an evaluation is a path. Each path consists of a start point and an end point. It is a task of the path planner to prepare the actual path between those two points. Then the bot tries to follow this path and collects data about navigation. It collects data about both the path planning and the path following. Data used for the path planning evaluation are the result of the path planning process (whether the path was built successfully), the total length of the path and number of jumps and lifts on it. For the evaluation of the path following process the result of the path following, the total time it took and possible data about the location of the failure are collected. In order to produce comparable data between old and new navigation systems, we limit the selection of start and end points to navigation points of the original navigation graph. Then we create paths by joining each point to all of the other points. There are more than 100 000 paths for some maps, which is too much for manageable evaluation. So we reduced this number without lowering the relevance of the evaluation by using the concept of relevant points and relevant paths, presented in the Analysis chapter. (Ch. 4)

The depth of the evaluation process differs for different groups of maps, which we created earlier in the map clustering process (4.2). The scope of the evaluation on individual maps of these groups looked like this:

- Milestone group– Evaluated both relevant paths only and all paths. For relevant paths only, the evaluation was run repeatedly to assess data about the safety of paths. The safety is the attribute expressed in percentage, which represents the success rate of the path following for given path. Evaluation of the all paths and repeated evaluation of the relevant path was done only for navigation configurations 1 and 3, as they are the main competing configurations.
- Reference group – An evaluation of the same extent as for Milestone group.
- Standard group – An evaluation of the relevant paths only for all configurations. The evaluation on the big maps was further limited to the maximum of 8000 paths.
- Excluded group – Maps from this group were not evaluated at all.

Data gathered during the evaluation process were used in various metrics to evaluate and compare tested navigation system configurations.

7.3 Evaluation framework

To perform the evaluation, we had to create some system capable of running the agent on the map and gather data about its navigation. We decided to develop a more general system, which would cover the needed functionality but would be capable of more. We created the evaluation framework (Attachment 2), which is capable of performing general evaluation tasks on the selected map. The

documentation of the evaluation framework along with an example usage is available as Attachment 3.

Our evaluation framework is multi-platform, multi-task framework, which runs each task in a separated JVM to minimalize the impact of single task's failure or a problem with a UT2004 server. The number of maximal parallel runs is given by the number of processor and by a multiplier parameter from the framework's configuration, specifying how many tasks can run on a single processor.

From our experience, it is possible to run multiple tasks on a single processor. The actual number of tasks that can be run simultaneously depends on the specification of the computer (processor speed, memory size) as well on maps the tasks run on. The Java process running our navigation evaluation on a small map usually takes about 50-100 MB of memory. On big maps it is often over 500 MB per process. Requirements on the processor are not very high, except from cases when the pure navigation mesh is processed for the first time or when the processed navigation mesh is loaded from the file at the start of the bot.

7.3.1. Task, Bot and Parameters

To be able to run some evaluation in our framework, you must prepare 2-3 specific classes.

Task. A task class contains information needed for starting the evaluation task by the framework and must implement `IEvaluationTask` interface. It must hold a name of the map on which the task should be run, a class of the bot used for the task, a class of the bot parameters the bot uses, an instance of this parameter class, a path where the log should be stored, a required level of logging, a name of the file representing the task (tasks are stored as XML files) and a path to the directory where the possible outputs of the evaluation should be stored.

Tasks are stored as XML files, serialized and de-serialized by the XStream library. (15)

Bot. A bot class represents the agent performing the evaluation of a given task and must extend the `EvaluationBot` class. The `EvaluationBot` class is simply an extension of `UT2004BotModuleController`, which is the base class for all agents in PogamutUT2004. The only extra information it provides is the flag signaling that the agent completed its work.

Parameters. Each bot can be started with parameters, which are contained in the `UT2004BotParameters` class or its extensions. This class is only voluntary to implement, as the original `UT2004BotParameters` class can be used instead.

Optionally there is one other specific class holding the immediate results of the evaluation and capable of exporting them at the end of the evaluation.

Each evaluation task is defined by the triple of task, bot and parameters classes. This triple is then supplied to the framework and it runs it. The task instance is passed to the framework as the XML file which is then de-serialized by the framework.

7.3.2. Structure of the framework

The structure of our evaluation framework can be seen on a diagram in Figure 7.1.

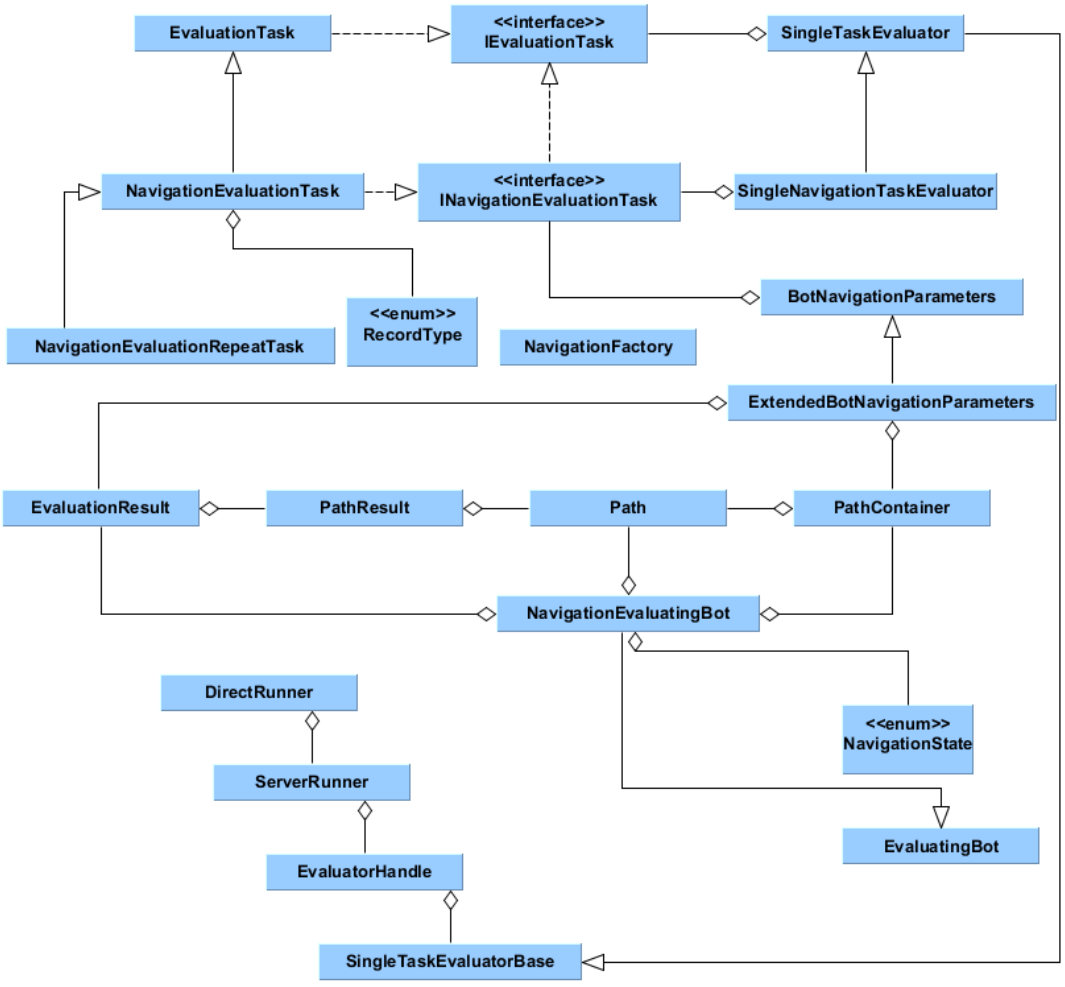


Figure 7.1: Structure of the evaluation framework.

The main class of the framework is the `ServerRunner`, responsible for loading tasks from files and creating individual task evaluators, which run a single task. It works in 5 second cycles, each cycle refreshing status of the currently running tasks, checking if all the tasks are already completed, checking for capacity to start another task and, if the capacity is available, starting the task by creating

`EvaluatorHandle` and letting it create the instance of `SingleTaskEvaluator`.

`EvaluatorHandle` holds the connection between `ServerRunner` and `SingleTaskEvaluator`. It starts `SingleTaskEvaluator` as a separated process with an own JVM and lets `ServerRunner` check the state of this process.

`SingleTaskEvaluator` is responsible for running a single evaluation, represented by a task file. It starts an instance of UT2004 server, using the `UCCWrapper` class, and then runs the required bot with given parameters on that server.

`DirectRunner` is a special runner used when there is not enough capacity or tasks to run multiple tasks in parallel. It does not start separate process, but it starts the evaluation directly in the current environment instead.

7.3.3. Batch creation of the task instances

We often want to create tasks with almost the same configuration, but for many maps. To define each task instance in code and then serialize it or to create it directly in the XML format would be time consuming in both cases. To fasten the creation of such tasks, we created `TaskFileGenerator` and batch creating methods for each task type. Batch methods accept the desired configuration for given task, but for some parameters (most often maps) it takes list of values instead of a single value. It then creates an instance of the task for each combination of parameters and returns them to `TaskFileGenerator`, which serializes all of the created task instances to given directory.

7.3.4. Multiple uses of the evaluation framework

During our work, we found several use cases, where the evaluation framework came in handy. We created new types of tasks and by ran them by the framework.

We used the evaluation framework for following tasks:

- Navigation evaluation – This is the main use case, we describe it in detail in the next section.
- Map paths statistics – We gathered data about paths for all maps to be able to cluster them and to determine their path following difficulty. We used the FWMap path planner for this task. For each map we extracted a total number of all paths and relevant paths and how many of all and relevant paths can be built:
    - without any limitations,
    - after removing lift edges from the navigation graph,
    - after removing jump edges from the navigation graph,
    - after removing both lift and jump edges from the navigation graph.

- Envelope computation – We computed the envelope box of all NavPoints on the map. We used the envelope box for cropping of the map geometry before generating the navigation mesh, as it is described in 5.2.6.
- Jump pad detection – We gathered information about locations of all jump pads on the map. We used this in creating pyramid blocks in the map geometry to solve issue with the jump pads on navigation mesh, described in 5.2.8.
- Jump inspection – We created a task to compile data for the jump analysis and the interpolation described in 6.3.1.

7.4 Navigation specific extension of the evaluation framework

For our main use case of the evaluation framework, we created the extension of the evaluation framework called the navigation evaluator. It contains the implementation of several task triples we used for evaluation of the navigation systems as well for other tasks related to our work. It also contains an extension of the `SingleTaskEvaluator` called `SingleNavigationTaskEvaluator`, which allows restarting and resuming of the evaluation tasks. Restart of the evaluation is needed when we want to store replays of the evaluation, as UT2004 server crashes after making 300+ replays. To prevent that, we added parameter to the framework's configuration, specifying after how many paths the bot should be restarted. The resume is beneficial for long running tasks, which could fail or be interrupted, which can now be simply resumed from latest saved state.

New tasks using the extra features of the navigation evaluator can be created similarly to the process described in the previous section. The only difference is that the task class must extend the `NavigationEvaluationTask` class, a bot must be an extension of `NavigationEvaluatingBot` and the parameters class must extend the `BotNavigationParameters` class. These classes are the triple we used for our main evaluation, but since the logic method of the bot can be overridden, they can still be extended to create a task with a different purpose.

7.4.1. NavigationEvaluatingBot

`NavigationEvaluatingBot` is the bot we used for the main part of the evaluation. Its parameters, `BotNavigationParameters`, contain specification of the navigation system as well as specification of the paths on which it should be run.

`NavigationFactory` provides the bot with the desired navigation system by constructing it according to parameters specifying whether to use the original or the new navigation system and the NavMesh or the FWMap path planner.

`PathContainer` is responsible for working with paths. A list of paths is constructed from parameters specifying number of paths to test and whether to test all paths or only the relevant ones. If the number of paths is limited, paths are selected randomly and only the desired quantity of them is kept.

The bot switches between following states:

1. NotMoving – state at the start of the bot and in between navigating paths, signals that the bot is ready for next path,
2. OnWayToStart – the bot has a path assigned, but has not reached its start yet,
3. Navigating – the bot is in the middle of path following process,
4. AtTheTarget – the bot has reached the target of the path and is waiting for the result to be processed,
5. Failed – the bot has failed during the path following and is waiting for the failure to be processed,
6. FailedOnWayToStart – the bot has failed to get to the start of the path and is waiting on the processing of its state.

Each logic cycle, the bot executes some action corresponding to its state and possibly switches to other state.

The bot collect following data about the navigation process:

- Aggregate results – consists of numbers of paths for each of these flags:
  - Processed,
  - Completed,
  - Failed,
  - Not built.
- Result for each path:
  - result of path following,
  - length of the path,
  - time the path following took,
  - number of lifts,
  - number of jumps,
  - for failed paths extra: exact location of failure, NavPoint nearest to the location of the failure.

Data of the evaluation are held by the instance of the `EvaluationResult` class, and are saved periodically to the disk before each restart.

7.4.2. Results storing

Results of evaluations are stored in the directory structure. The top level contains folders with names of navigation system configurations. In them, results from the same map are stored in the same folder with the same name as the map. There are individual folders with evaluations. Name of each folder is created from the date and time of the start of the evaluation.

Each evaluation contains XML file with the definition of the task instance, data about following of the paths in a file *data.csv*, aggregated results in a file *data.aggregate.csv*, files with logs and files with replays. We created utility which organizes these files in following steps:

- Move log files to the *logs* folder.
- Group records by the NavPoint where they failed and move them to folder structure in the folder *records* (There is a folder for each NavPoint with at least one failed record.).
- Create a *data_failed.aggregate.csv* file with information about the number of failed paths for each NavPoint.

This organization was particularly useful when investigating problems in navigation, when we easily found NavPoints with most failed paths and corresponding records of such failures.

After gathering results from many evaluations, we discovered disadvantages of this structure and difficulty to compare the gathered data directly. We created other utility, which transfers data into a SQL database, where they can be organized and queried more effectively.

7.5 Metrics

To be able to verify fulfillment of our main goal, to create a competitive navigation system overcoming the original navigation system with both path planners (Goal 1, in 1.1), we had to establish a set of metrics to use for comparing of the navigation systems. We will now introduce them, explain their meaning and data used to compute them.

7.5.1. Success rate

This is one of the most important metrics for comparison of the navigation system configurations. It reports about quality of the path following, but path planning also influences it. If the path planner builds a bad path it is hard to follow it successfully. This metric expresses the rate of the paths navigated successfully for given map. It can be simply described as:

$$successRate = \frac{number\ of\ successfully\ completed\ paths}{number\ of\ processed\ paths}$$

It was used both for results with all paths and relevant paths only. Paths which were not successfully navigated either failed during path following or were not built at all, which express the following metrics.

7.5.2. Failure rate

Another important metric, it captures the rate of failed paths for given map and reports about the quality of the path following. It has the following equation:

$$failureRate = \frac{number\ of\ paths\ which\ failed\ during\ navigation}{number\ of\ processed\ paths}$$

Again, it was computed for results with all paths and with relevant paths only.

7.5.3. Not built rate

This is complementary metric to the previous two. When added together they give result of 1, meaning that all of the paths projected into one of these metrics.

This metric expresses the rate of the paths which were not built by the path planner, so it reports about quality of the path planner and about difficulty of the map. Its equation is:

$$notBuiltRate = \frac{number\ of\ paths\ which\ were\ not\ built}{number\ of\ processed\ paths}$$

This metric together with the *failure rate* gives us important information about the difficulty of the map. Two maps with the same *success rate* can still differ very much in difficulty, because one of the maps can be really hard to navigate, where the other simply contains many places reachable only through paths requiring some of the path following tricks (described in 6.2), which our navigation system is not capable of performing, and the path following on the actually built paths can have very high success rate.

7.5.4. Restricted success rate

To capture the impression an observer of the agent would have from watching the agent move in the environment, we defined a restricted success rate metric. It expresses the success rate, but only on the successfully built paths.

Navigation on the map with bad success rate but good restricted success rate would probably look to the observer as a pretty smooth. The observer might notice that the agent does not navigate to some places at all, but since these places are only reachable by special tricks, which can be difficult to perform for live players too, it might not look strange to him.

We define the restricted success rate as:

$$restrictedSuccessRate = \frac{number\ of\ successfully\ completed\ paths}{number\ of\ processed\ paths - \ number\ of\ paths\ which\ were\ not\ built}$$

The number of processed paths in all previous metrics could be replaced with the total number of paths of given type with the exception of evaluation on the largest of the evaluated maps, where the number of processed paths was restricted (to a maximum of 8000 paths).

7.5.5. Average path length

To compare the quality of path planners and of paths they produce, we introduced the average path length metric. Shorter paths are preferable for navigation and should look more natural.

To improve the relevance of this metric, we limited the set of navigation cases which paths we will measure. A navigation case will be in the measured set only, if all of the evaluated navigation system configurations constructed a path for it and if this path was subsequently successfully followed. This prevents distortion of the results of navigation systems, which path planner builds the path, but its path follower is not able to follow such path.

An average path length for given map is defined as:

$$\text{averagePathLength} = \frac{\sum_{p \in P}^{P} \text{length of p}}{|\text{P}|}$$

P is a set of paths for selected navigation cases.

7.5.6. Average navigation time

An average navigation time is the second metric created to compare the quality of paths produced by the path planner, but this metrics also compares the capabilities of path following to navigate paths quickly.

Again, we use only the navigation cases from the same restricted set as in the previous metric. We define an average navigation time as:

$$\text{averageNavigationTime} = \frac{\sum_{p \in P}^{P} \text{navigation time of p}}{|\text{P}|}$$

P is a set of paths for selected navigation cases.

7.5.7. Failures overlap rate

To compute the similarity of problems in two navigation system configurations, we define a failures overlap rate.

To determine the failures overlap rate, we specify:

- $failedPaths_{better}$ as a set of failed paths from the more successful configuration
- $failedPaths_{worse}$ as a set of failed paths from the less successful configuration
- metric's equation:

$$failuresOverlapRate = \frac{|failedPaths_{\text{better}} \cap failedPaths_{worse}|}{|failedPaths_{better}|}$$

The higher the rate the more probably compared configurations suffer from the same problems, as they fail on the same paths. We used this metric also for comparison of different evaluations of same configurations.

7.5.8. Failures difference rate

Close to the previous metric is the failures difference rate. It also studies similarity of problems in two navigation system configurations or in two evaluations of the same configuration.

In this metric, we use the information about the NavPoint nearest to the location of the failure on given path. We take a set of such NavPoints from the more successful configuration and remove failures with any NavPoint from this set from the set of failures of the less successful configuration. This can be expressed like this:

- NP_{better} – a set of NavPoints with failed paths in more successful configuration
- F_{worse}– a set of failed paths of less successful configuration
- np(p) – a NavPoint at which path p failed
- equation:

$$failuresDifferenceRate = \frac{|\{p;\ p\ \in F_{worse} \wedge np(p) \notin NP_{better}\}|}{|F_{worse}|}$$

The smaller the failures difference rate is, the more similar the two evaluations are and vice versa. It says how many percent of the worse evaluation failures are unique to this evaluation.

7.5.9. Relevant to All difficulty ratio

We limited our evaluation for most maps to relevant paths. We justified this decision in the 4.1. To compare possible differences created by this limitation, we ran the evaluation on some maps for all paths and we defined the relevant to all difficulty ratio.

For these selected maps we take $successRate_{relevant}$ as a success rate on relevant paths, $successRate_{all}$ as success rate on all paths and define:

$$relevantToAllDifficultyRatio = \frac{successRate_{relevant}}{successRate_{all}}$$

Values of ratio greater than 1 mean, that the navigation of relevant paths is easier than navigation on all paths for given map and vice versa.

7.5.10. Average rates and differences

We ran the evaluation of selected maps multiple times to gather enough data to be able to establish an average success rate, an average failure rate and an average not built rate and differences between individual evaluations. We wanted to study the stability of results gathered from the single evaluation of the map. We expected a small difference in the case of the not built rate, as the process of path planning is deterministic. We presumed that the differences of success rate and failure rate would be rather small (< 1%) due to often repetition of parts of paths, but we needed to certify this assumption.

We ran evaluation 10 times on selected maps and established average rates and both maximal and average differences.

7.6 Results

We defined number of metrics in the previous section and we will present the results measured by them here. We sometimes limited results to some groups of maps or showed only aggregate results and we mention this at corresponding tables or graphs. We often refer to groups of maps as defined in 4.2.

7.6.1. Quality of navigation

The success, the failure and the not built rate are the most important of evaluated metrics, as they provide information about the overall quality of the navigation system. To be able to present accurate results, we ran the evaluation of Milestone and Reference maps 10 times on relevant paths. We ran this evaluation only for the new navigation system (referred to as NEW from now on) and for the old navigation system configuration with the FWMap as a path planner (referred to as FW), as it was superior to the one with the NavMesh path planner in the initial analysis. We computed average accuracy rates on results of this evaluation and found out that our new navigation system is more successful on 16 from 18 maps from evaluated groups. Complete results are presented in Figure 7.2. Average improvement of the success rate was 9.59% for the Milestone group and 2.99% for the Reference group. FW had average success rate of 90% percent on the Reference group and 77% average success rate on the Milestone group, so there was significantly more room for improvement. The largest deterioration was measured on the CTF-Maul map with the value of 3.88%. The largest improvement was made on the DM-1on1-Trite map with the value of 21.86%.

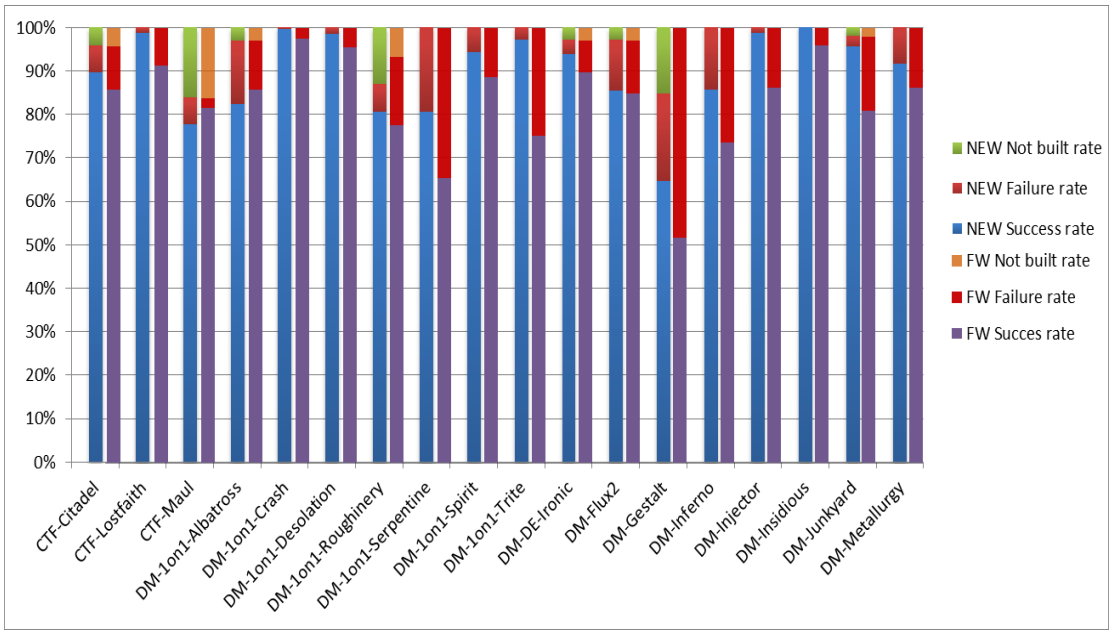


Figure 7.2: Evaluation results of Milestone and Reference groups for NEW and FW systems, showing the average success rate, the average failure rate and the average not built rate.

Comparison of the new and old navigation system with navigation mesh. Our evaluation of the old navigation system with the navigation mesh as a path planner (referred to as NM) was limited to a single run on each map. The more precise data was not that important for us, as the NM system performed significantly worse than the FW system, thus we concentrated on besting the FW system. The NEW system performed better on all 18 maps from the Milestone and the Reference groups, with the average improvement of 23.66%. The largest improvement was measured on the DM-1on1-Desolation map, reaching 46.24%. Complete data are available as Attachment 9.

Quality of navigation on maps from Standard group. We ran the evaluation only one time for Standard maps on relevant paths. We further limited the evaluation to a total of 8000 paths for some of the bigger maps. The NEW system was more successful than FW system on 40 maps and more successful than NM system on all 53 maps the NM system was evaluated on. We were not able to evaluate the NM system on the 5 remaining maps from the Standard group, as the NM system failed in loading or processing of the original navigation mesh. The average improvement was 2.78% against the FW system and 25.86% against the NM system.

We will now concentrate only on the NEW and FW systems. The maximal measured deterioration is 9.61% on the CTF-DE-ElecFields map and the maximal improvement is 15.98% on the DM-DE-Grendelkeep map. The deterioration was larger than 5% on 4 maps where the improvement was greater than 5% on 20 maps from which the improvement exceeded 10% on 8 maps.

Complete data for all evaluated navigation systems for all evaluated maps are available as Attachment 9.

Quality of navigation on all paths. We ran the evaluation also for all paths on selected group of maps again only for NEW and FW systems. We wanted to use Milestone and Reference maps, but 6 maps from them were too large for a full evaluation and we replaced them with 6 different smaller maps for this part of the evaluation. The NEW system was more successful on 17 of 18 maps with average improvement of 4.66%. We used average rates for relevant paths where available (12 maps).

7.6.2. Differences in evaluations

We took results from the repeated evaluations of the Milestone and the Reference groups to compute maximal and average difference in rates between evaluations. This will show us stability of the navigation systems, as well as a quality of data gathered from a single evaluation for the Standard group. We present results in Table 7.1. The stability of both systems is very similar and it does not affect results for most of the evaluations of the Standard group.

	NEW	**FW**
Maximal difference	0.95%	1.51%
Average difference	0.55%	0.73%

Table 7.1: Maximal and average differences between evaluations.

We wanted to study the relative difficulty of relevant paths to all paths through *relevantToAllDifficulty* ratio for maps we ran the full evaluation on. We used average rates for relevant paths where they were available. The success rate of all paths was higher for 11 of 18 maps for the FW system but only for 4 of 18 maps for the NEW system. Results did not show us any pattern related to the difficulty of relevant paths against all paths.

7.6.3. Restricted success rate

Values of restricted success rate mostly copy results of the standard success rate. There are only 2 exceptions in Milestone and Reference maps, both increasing difference in the restricted success rate in the favor of the NEW system. This means that we correctly identified more paths that our system is not capable of following. In the Standard group, the restricted success rate differs noticeably in 2 maps only, again in the favor of the NEW system.

7.6.4. Similarity of issues between navigation systems

We defined two different metrics related to the similarity of issues between navigation systems. A histogram of failures difference rate (defined in 7.5.8) on Figure 7.3 shows that there are many maps, where issues are quite similar, but on some maps the issues are almost distinct for the two navigation systems (NEW and

FW). Results of the failures overlap rate (defined in 7.5.7) are presented in a histogram in Figure 7.4 and lead to similar conclusion as results of the failures difference rate.

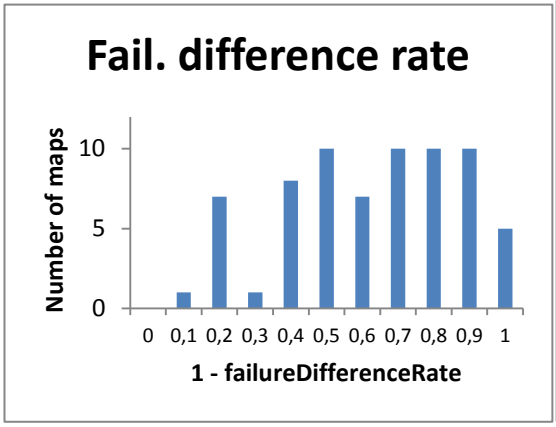


Figure 7.3: Histogram of failures difference rate.

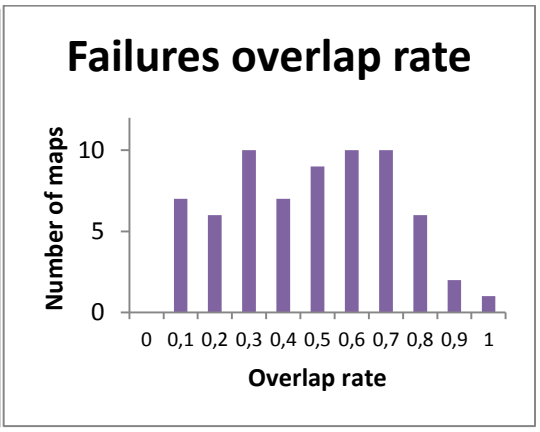


Figure 7.4: Histogram of failures overlap rate.

7.6.5. Stability of paths

The stability of paths was studied on maps from the Milestone and the Reference groups, which were evaluated repeatedly. We present a histogram for each navigation system, showing distribution of paths by their stability, in Figure 7.5 and Figure 7.6. We omitted paths with stability of 0 and 1 as those form most of the paths and the histograms would not show us anything.

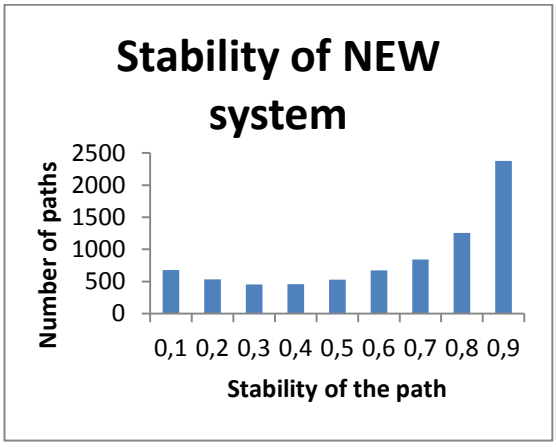


Figure 7.5: Stability of paths in the NEW system.

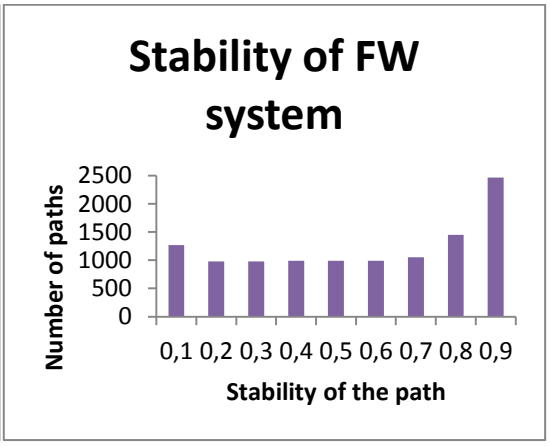


Figure 7.6: Stability of paths in the FW system.

Similarity of issues between evaluation runs. Other way to determine the stability of the system is to measure the failure overlap rate and failure difference rate, but this time for various evaluations of the same map. Figure 7.7 confirms that both systems are quite stable in terms of issues which manifested themselves during an

evaluation. The failure overlap rate is more rigid in qualifying issues as the same, as exactly the same path must fail in both evaluations, so its values are lower. Average value of failure overlap rate is 79.16% for the FW system and 74.98% for the NEW system. Average value of the inverted failure difference rate is 97.52% for the FW system and 96.18% for the NEW system.

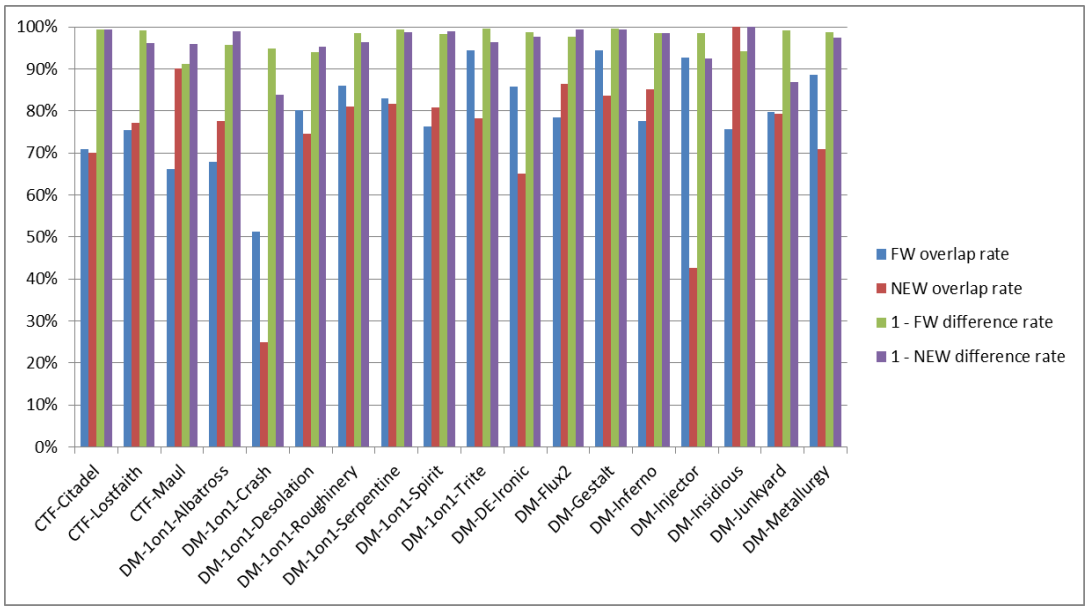


Figure 7.7: Average failure overlap and difference rates for NEW and FW systems

7.6.6. Path statistics

We measured data for individual paths and formed following statistics from it: an average path length, an average time it took to follow the path, a number of jumps on the path and a number of lifts on the path.

	NEW	FW	Difference
Average path length (UT units)	5284	5399	115
Average navigation duration (ms)	11131	11308	177
Average number of jumps (count)	1163	2855	1692
Average number of lifts (count)	407	400	-7

Table 7.2: Path statistics

Table 7.2 shows path statistics, confirming that paths from the NavMesh path planner are shorter and faster than paths from the FWMap path planner. Those paths also use much fewer jumps, as they often find alternate route on the navigation mesh. Slightly higher number of lifts is also the result of alternate routes.

In total, paths were shorter on 58 maps, faster on 54 maps and had lower number of jumps on 73 maps (on 3 maps there were no jumps) all from 76 of evaluated maps. Longer and slower paths on some maps were caused by not

optimal path planning algorithm, which is described in 2.3.3 and its improvement is mentioned as a possible future work.

8. Conclusion

We will now discuss the fulfillment of our goals, show remaining issues and outline possible future work from them.

8.1 Fulfillment of goals

We significantly improved the process for creation of navigation meshes, which resulted in more precise meshes with higher usability and benefits. We improved and extended the creation process, fixed bugs in the NavMesh path planner and generated meshes of higher quality for all maps. We fitted the navigation subsystem to work with the navigation mesh, we refined the computation of jumps and used additional information from the navigation mesh to jump at the right time and to the right place. We increased the frequency of the navigation logic cycle and adjusted the navigation system to this new frequency.

We created a robust evaluation framework allowing us to test navigation systems thoroughly and we performed such evaluation. Results of this evaluation prove the fulfillment of the first goal and the evaluation itself accomplishes the second goal.

The new navigation system is fully integrated into our copy of PogamutUT2004 (Attachment 1) and is prepared to become a part of a public release allowing PogamutUT2004 users to profit from the improved navigation system with the navigation mesh.

8.2 Future work

We created new navigation system with the navigation mesh at the center of attention, it outperforms the old navigation system, but it is not perfect and some problems remain open to future work.

We rely on the original navigation graph with off-mesh connections and the quality of the navigation graph reflects in the navigation mesh. There are some poorly placed navigation points and links between them, preventing the agent from successfully following a path through them. The only solution is the manual adaptation of the navigation graph, which will be very time consuming, but our evaluation framework can help with the identification of these poorly placed navigation points. We experimentally tested this on the DM-Gestalt map, where we moved only 2 navigation points and improved the success rate by 10% as a result. This was possible due to the fact that through these points lead two of only a few paths leading to the upper level of the map.

We analyzed jump tricks but decided not to try to implement them. Some places are reachable only with the usage of those tricks and if we would want to

make a perfect navigation system, we would have to implement at least some of them. Main problem is the need for precise timing, which is hard for us to achieve. We studied some lift jumps which looked doable to perform even with our current system, but there would possibly have to be a specific routine for each lift jump.

The path planning algorithm on the navigation mesh could be improved, as the current algorithm sometimes chooses a path which is not the shortest one possible, as described in 2.3.3. This could be solved by implementing of some extension of algorithms presented in 3.3.

We could further improve the quality of navigation meshes by obtaining better level geometry. The current level geometry extracted through the UShock program has two known problems: missing data about semitransparent floors (5.2.5) and a problem with terrains (4.5).

Bibliography

1. **GEMROT, Jakub, et al.** Pogamut 3 can assist developers in building AI (not only) for their videogame agents. In: Agents for Games and Simulations. Springer Berlin Heidelberg, 2009. p. 1-15.

2. **RUSSELL, Stuart; NORVIG, Peter; INTELLIGENCE, Artificial.** A modern approach. Artificial Intelligence. Prentice-Hall, Egnlewood Cliffs, 1995, 25.

3. *Epic Games, Inc.* [Online] [Cited: 16 7 2014.] http://epicgames.com.

4. **TOMEK, Jakub.** Navigation mesh for the Virtual Environment of Unreal Tournament 2004. Prague, 2013.

5. *Pogamut 3 Lectures.* [Online] [Cited: 16 7 2014.] http://diana.ms.mff.cuni.cz/pogamut-devel/doku.php?id=lectures.

6. *Pogamut Cup.* [Online] [Cited: 16 7 2014.] http://www.pogamutcup.com/.

7. *BotPrize.* [Online] [Cited: 16 7 2014.] http://www.botprize.org/.

8. *Pogamut.* [Online] [Cited: 16 7 2014.] http://diana.ms.mff.cuni.cz/main/tiki-index.php?page=About.

9. **MONONEN, Mikko.** Recast. [Online] [Cited: 28 7 2014.] https://github.com/memononen/recastnavigation.

10. **DEMYEN, Douglas; BURO, Michael.** Efficient triangulation-based pathfinding. In: AAAI. 2006. p. 942-947.

11. **VAN WAVEREN, J. M. P.** The quake III arena bot. University of Technology Delft, 2001.

12. **HOLTE, Robert C., et al.** Hierarchical A*: Searching abstraction hierarchies efficiently. In: AAAI/IAAI, Vol. 1. 1996. p. 530-535.

13. **KALLMANN, Marcelo.** Shortest paths with arbitrary clearance from navigation meshes. In: Proceedings of the 2010 ACM SIGGRAPH/Eurographics Symposium on Computer Animation. Eurographics Association, 2010. p. 159-168.

14. **BERTRAM, Thomass.** UShock. [Online] [Cited: 16 7 2014.] http://sourceforge.net/projects/ushock/.

15. *XStream.* [Online] [Cited: 16 7 2014.] http://xstream.codehaus.org/.

16. *Wikipedia.* [Online] [Cited: 16 7 2014.] http://en.wikipedia.org/wiki/Binary_space_partitioning.

List of Abbreviations

AAS – area awareness system – an abstraction of 3D virtual environment presented in (11)

BR – Bombing Run – a team game mode of UT2004

BSP – binary space partitioning – a way of recursively subdividing a space into convex sets by hyperplanes

CTF – Capture the Flag – the most popular team mode of UT2004

DM – Deathmatch – the most popular single game mode of UT2004

DOM – Double Domination – a team game mode of UT2004

FPS – first person shooter – a type of an action game, where the player sees the environment through the eyes of the controlled avatar

FW – the original navigation system with FWMap path planner – used only in Results section

FWMap – Floyd-Warshall map – a path planner, in PogamutUT2004 used for the navigation graph

GB2004 – Gamebots2004 – a module responsible for communication between UT2004 environment and agent's Java implementation

HA* - hierarchical A* - a hierarchical version of popular A* path finding algorithm, presented in (12)

IVA – an intelligent virtual agent

LCT – local clearance triangulation – a special type of navigation mesh, suitable for agents with different radii

NEW – the new navigation system with the navigation mesh – used only in Results section

NM – the original navigation system with the NavMesh path planner – used only in Results section

UT2004 – Unreal Tournament 2004 – a FPS action game, highly oriented on multi player game

TDM – Team Deathmatch – a team version of the popular game mode of UT2004

Attachments

1. PogamutUT2004 with the new navigation system
2. Evaluation framework
3. Documentation
4. UTLevelGeom – original and modified version
5. Recast – original latest version, modified old version, modified new version
6. Meshes – original
7. Meshes – new
8. Videos
9. Additional results – tables and graphs