DATABASE DRIVEN MULTI-AGENT BEHAVIOUR MODULE

RADOS JOVANOVIC

A THESIS SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF

MASTER OF SCIENCE

GRADUATE PROGRAM IN COMPUTER SCIENCE
YORK UNIVERSITY
TORONTO, ONTARIO

APRIL 2013

ABSTRACT

Due to performance restrictions, video games have to make a tradeoff between having few independent and intelligent computer controlled agents, or many but simpler ones. Allowing games to have thousands of clever, independent agents would allow for new gameplay opportunities.

In this thesis, I develop a model for an agent behaviour software module, which is capable of processing decisions and their consequences for thousands of agents per second, and of modifying the game state accordingly without artificially limiting the expressive power of behaviour scripts. The module is self-contained, client independent, and has a minimal interface for communicating with the game engine. The model is designed to utilize a database system at its core, taking advantage of its features and optimizations. Additionally, I introduce a set of parameters that aim to quantize the cost of a behaviour, and to analyze the performance of the module. These parameters are considered when designing behaviour scripts and when building the world. Conducted tests show that the parameters provide a good estimate of the cost of a behaviour script. Despite of the created module not being implemented most efficiently, results show better performance compared to current state-of-the-art solutions.

ACKNOWLEDGEMENTS

I would like to thank my supervisor, Parke, for his guidance made this possible.

I would like to thank my family for their support, patience, and love.

I would like to thank all great minds, past and present, whose invaluable knowledge continue to shape me throughout life.

TABLE OF CONTENTS

ABSTRACT ii

ACKNOWLEDGEMENTS iii

TABLE OF CONTENTS iv

LIST OF FIGURES vii

1 Introduction 1

1.1 Motivation 1

1.2 Glossary 2

1.3 Methodology 3

1.3.1 Database Agent Module 4

1.3.2 DAM Flow 7

1.4 Contributions 8

1.5 Outline 9

2 Related Work 10

2.1 SGL and Cassanova 10

2.2 Handling Moving Objects 12

2.3 Game Development Platforms 13

3 Methodology 15

3.1 Game Space Model 15

3.1.1 Agent Computational Cost Per Second 15

3.1.2 Number of Agents 17

3.1.3 Agent Density 17

3.1.4 Putting it all together 18

3.2 The Conceptual Model....18
3.2.1 Databases and Games....20
3.2.2 Queries as Actions....20
3.2.3 Objects and Databases....21
3.2.4 Triggers as Events....21
3.2.5 Communication with the Engine....22
3.2.6 Tick....24
4 DAM Prototype Design and Implementation....25
4.1 Prototype Requirements....25
4.2 Architecture....26
4.3 Agents....28
4.3.1 Updating Agent's State....30
4.3.2 Agent's Cost Metric....31
4.3.3 Fighter Agent....31
4.3.4 Droid Agent....35
4.3.5 Controller Agent....36
4.4 Sensors....38
4.5 Effects....39
4.5.1 Change Team Effect....40
4.5.2 Immunity to effects....40
4.6 Obstacles....41
4.7 Messaging....41
4.8 The Client....42
4.9 Strengths and Weaknesses....43
5 Experiments....45

5.1 Experimental Setup....45
5.1.1 Testing Hardware....45
5.1.2 Server Tuning....46
5.2 Tests....46
5.2.1 Number of Agents....47
5.2.2 Cost Per Second....48
5.2.3 Density....49
5.2.4 A Battle....50
5.3 Summary....52
6 Conclusions and Future Work....53
6.1 Conclusions....53
6.2 Future Work....54
6.3 Selling a Database Solution....55
Bibliography....57
Appendices....60
Appendix A: The DAM SQL scripts....60
Appendix B: The DAM C source code....71

LIST OF FIGURES

Figure 3.1: Two types of cps functions....17
Figure 4.1: An example of creating SQL triggers....27
Figure 4.2: Examples of binding C functions....27
Figure 4.3: Agent table schema....29
Figure 4.4: State trigger definition....29
Figure 4.5: Agent storage structure hierarchy....31
Figure 4.6: Sending a new projectile message....32
Figure 4.7: Intercepting agent's health loss....33
Figure 4.8: Per Tick Update....33
Figure 4.9: Change team effect creation....36
Figure 4.10: Egocentric field steering sensor input and output....38
Figure 4.11: Effect table schema....39
Figure 4.12: Change team effect's constructor parameters....40
Figure 4.13: Static obstacle schema....41
Figure 4.14: Message tables schema....42
Figure 5.1: Test action query....47
Figure 5.2: Change of the number of agents....47
Figure 5.3: Change of cost per second....49
Figure 5.4: Change of the density....49
Figure 5.5: A battle scenario....51
Figure 5.6: A screenshot of a battle....51

Intelligence is the art of good guesswork.

-- H.B. Barlow

1 Introduction

1.1 Motivation

Because artificial intelligence for games can be computationally expensive, game developers need to make a tradeoff between having complex behaviours for a handful of agents, or support larger numbers of them with simpler behaviours. (An agent is an autonomous entity which observes an environment and acts upon it.) On one end of this spectrum, there are games like Neverwinter Nights which support elaborate behaviours, but only a few agents can be active at a time. Warcraft III supports few hundred agents, but each of them can only act in a simple manner. On the other end of this spectrum are games like Rome: Total War, where thousands of units are on the battlefield, but these units are grouped together in a few large battle groups which have identical behaviours for every unit belonging to a same battle group [20]. If games like Rome:Total War could have units with individual and complex behaviours like agents in Neverwinter Nights, it would provide new opportunities for gameplay which are not currently possible.

Database systems can quickly run computations (queries) on large sets of data “in parallel”. Traditionally, queries were crafted for intelligent data retrieval, but recently databases are being used to solve other problems as well, for example, in bioinformatics [11], data mining [13], and business intelligence [16]. In my research, I apply database

systems to compute behaviours of agents in video games.

A standardized and widely adopted language to write queries in is the SQL programming language [23], which is supported by most major database systems. The SQL programming language is expressive enough to model video game character behaviours as well [20]. My goal is to design and create an AI module for agents which is capable of being easily integrated in current video game engine architectures, powerful enough to model typical video game agent behaviours, and scalable so that it can support thousands of such agents making decisions and interacting with their environment at the same time. For this thesis work, I narrowed the focus to talk only about real-time strategy games, although the module could also be used for other game types as well.

In this work, I define a game space model with different game parameters which can be used to describe and tune the technical parameters for game agents. When designing an AI module for games, we must consider how many autonomous agents are in the game at a time and doing work, how complicated the AI behaviours are, and what the spatial distribution of agents in an area is (which dictates how many agents can potentially influence one another). In a game, units generally do not need to have extremely complicated AI scripts, but the scripts should be expressive enough to create interesting behaviours.

1.2 Glossary

Listed are a few key terms used commonly by video game developers, and used in this thesis.

- game engine: a system designed for the creation and development of video games consisting of many different components working together.
- agent: an autonomous entity which observes an environment and acts upon it.
- action: an act which inspects or changes an environment.
- behaviour: doing actions according to some predefined rules.
- firing an action: executing a single action.

- cooldown: a time which needs to elapse before a recently fired action can be fired again.
- tick: the smallest unit of time recognized by a system.
- interval between two ticks/duration of a tick: interchangeable; it is a time interval which needs to elapse for one tick to finish and before the next one can start.
- effect: it is often a temporary game object which does an arbitrary computation which, in turn, could change agents' states.

1.3 Methodology

Central to this thesis project is a system for processing agent behaviour using a database system, which I call the database agent module (DAM). The prototype is coded part in SQL and part in C, and uses the PostgreSQL database system for computing agent behaviour. PostgreSQL is a proven database system which is free and one of the more powerful and comprehensive systems. Due to it being open source, it has been extensively used in academic research. DAM is designed to be a separate, stand alone module from the rest of the game engine so that it can be integrated into other engines with a minimal restructuring of the receiving code. This is not too complicated to achieve since DAM is based on the already external SQL programming language to define its behaviour and a separately running database server process resolving the SQL scripts and accompanying C functions.

The DAM processes each agent independently of one another. Each decision is made separately, and the DAM does not group behaviours to simplify the computation. Every 500 milliseconds, the DAM can process decisions for close to 4000 agents which are aware of the of the environment around them, and make decisions how to react to the environment and change it. They avoid obstacles in their way and move in formation with fellow agents. The most limiting factor when computing behaviour is scanning the area around an agent. How that information is used is controlled by algorithms written in C.

The DAM is evaluated on an average gaming machine (Intel Core i7, four 1.73GHz cores, 4GB of RAM), and does not require any external servers to run efficiently. The DAM does not make use of a graphic processing unit for its computation, so a GPU can be ignored in the machine specification.

1.3.1 Database Agent Module

There are a few limitations of database systems which we need to keep in mind when designing a system such as the DAM. Since agents interact with one another and the player, and these interactions occur only if the agents are within a certain Euclidean distance from each other, we have to work with multidimensional coordinate data. Databases use indexes to retrieve quickly the correct data objects from the database, but they are natively optimized to do computation over one dimensional indexes. Thankfully, there are extensions for indexing multidimensional data, which is crucial for distance queries in a 2D or 3D game space. PostgreSQL provides support for GiST (generalized search tree) indexes which under the hood use R-Trees to store spatial information. This makes distance queries fast to execute. Another limitation is that frequent calls to the SQL's user defined functions can result in poor performance. In DAM, if a custom function is identified as a bottleneck, it can be re-written as a C function and called from the SQL query when needed. Likewise, an SQL query can also be invoked from inside of a C function. PostgreSQL natively supports such cross-language referencing.

The data is stored inside of a database as rows of values, which are residing inside of a table. Complete data which is used to describe a same object can be distributed among several tables. For the purpose of this discussion, I will refer to this data describing the same object but stored in multiple tables as a data object, and so a database is a set of such data objects, or a data set. Because of indexing, intelligent query plan construction, already implemented concurrency control inside of the database system, and various internal optimizations, database systems can efficiently work with huge data sets. In the case of games, if we represent an agent as one data object in the data set, and the decisions agents make as queries, we will be able to process decisions for a large number of agents by taking advantage of all optimizations in the underlying data-

base system.

The architecture for the database agent module is comparable to the one described in [20]. The DAM determines the behaviour of agents each AI-simulation tick. For real-time games, the AI system is notified of a tick by the main game engine at discrete time intervals. Upon receiving the tick message, DAM proceeds to process actions of one or more agents. Each tick, a single agent can perform an arbitrary amount of actions, and many agents can act during the same tick. If an agent's action spans for more than one tick, or an arbitrary amount of time, it will be assigned a cooldown time before the agent can act again so that the taken action has time to complete. Therefore, if an agent has an action in progress when behaviours for a new tick are being computed, this agent will not do any computation that tick cycle. The method of not processing certain agents during a clock tick also allows the developer to control the number of agents which are allowed to take actions (do computation) per tick. In scenarios with a world consisting of many agents but which do not all have to take actions every tick, this feature could yield a large performance benefit. For real-time games, implementing a cooldown is a necessity. The applications of this feature are explained below. Each agent's action can influence the game world in any number of ways, which practically is only limited by the CPU speed. A traditional practice in game design is that when multiple agents act during the same tick, they act simultaneously, and the effects of those actions are resolved simultaneously.

As mentioned previously, cooldowns control if an agent will do computation during a tick or not. By not having all agents active at all times, we are able to save processing power. In game design, it is common practice to create delays between firing of actions, such as shooting, jumping, or calling for backup. The situations which will put the agent to sleep for a period of time is completely application dependent and will vary from game to game, but any such action will only have to specify the cooldown value.

DAM is designed to be sufficiently modular so that adding new behaviours or unit types takes little to no modification of already written code. To achieve this, the system is structured to take advantage of event-driven programming, using database triggers. Events include a signal that a new tick began, that an agent is created or deleted, or

state changes such that an agent has lost health, among others. Triggers are functions which are fired when a row in the database changes and a user defined condition is satisfied. To add a behaviour for a new unit type to DAM with this mechanism, a user needs to write the behaviour function and a trigger which, once the conditions are satisfied and it fires, will call that function. A simple condition for the trigger activation is that a new tick began, and that the type of the agent for which we want the behaviour function to be called is the newly added unit type. With this mechanism, adding new unit types does not modify the already written code, rather just adds to it. The AI is designed to have similarities to finite state machines, although the design is more liberal. An agent can be at most in one state at any given time and each agent's state is not dependent on, unless specified by design, any other agent's state. If we look at each state belonging to each agent's behaviour as a separate unit type consisting only of that single state, we can write all states as separate functions and create separate triggers for each of those state functions. The conditions for those triggers will include previously defined trigger conditions for the unit type in question with an additional constraint that the agent must also be in a specified state. Using this method, we are able to expand agent behaviour and add new states easily into the module. To further increase the modularity of DAM, a group of stand-alone functions which I call sensors are implemented. Sensor functions will produce some output about the environment based on the input. Each unit type can have an arbitrary number of sensors depending on the unit's need, and the unit is in control when they are fired. Sensors include detecting the closest threat, detecting the most optimal path to take to a given goal, retrieving the health of a specific agent, or retrieving any other information about the current state of the environment. Additional performance affecting parameters which can be tuned using the concept of cooldowns are when and how often an agent activates its sensors. To achieve real-time steering of thousands of agents towards their goals, and to implement efficiently the sensor responsible for it, I use the method described in [9].

Since it is computationally expensive, and not needed in every case, to fire sensors which scan the environment every clock tick for every agent, I implemented within he DAM a storage structure for an agent which stores the agent's state data we would like

to preserve between several clock ticks. This storage is used in combination with cooldowns and is created by the database server inside of its own memory space for fast access when computing queries. The amount of information which can be stored is limited only by the amount of memory the application can use. An example of information which could be stored is an unique enemy id for an enemy agent which came in range of another, and the other agent is stubborn and will not look for other enemies until the present one is killed. This way, the other agent preserves its target information between ticks and does not need to query for enemies again until it has to. Therefore, this agent storage structure can be used to save a result of a query so we do not have to execute the same query every tick, if not necessary. The same storage structure is used to pass data from one state to another. Every different unit type can have its own collection of storage structure data variables.

1.3.2 DAM Flow

The database agent module tick flow is as follows. When the DAM clock tick starts, it does the following steps.

1. It resolves all pending messages from game engine.
2. It then runs the main AI SQL script, which will go through all agents and effects and notify them of a new tick. (As mentioned previously, an agent can opt not to do any computation if it is on a cooldown.)
3. Appropriate triggers fire and call related state functions which will compute changes in the environment.
4. The computed changes in the environment are applied.

After the environment is updated, corresponding objects in the rest of the game engine (e.g., graphics) are as well and any messages passed to the game engine by DAM are resolved. Messages passed to DAM can include a request for creating a new agent or to change the position of an agent's desired location (goal). Messages passed from DAM can include requests for new particle effects to be generated or a projectile to be fired, amongst others. This message passing plays a crucial role in making DAM viable

for games. These messages are stored in special tables inside of the database, which are later queried by the game instance or by DAM. The database agent module runs in a separate thread and notifies the game engine when its tick is done. The game engine then queries DAM's results and updates the world accordingly. This multithreading is possible because the DAM works on a separate data set from the rest of the game engine, and this data set is stored in a database so no thread locks are needed, which usually slow the application down if not carefully designed. During a DAM tick the game engine can send messages to it which will be queued and processed for the next tick.

1.4 Contributions

With this research, I have made the following contributions.

1. Game space model. I define a useful way to talk about agent AI in games, their game parameters, which help with the design and analysis of agents. (Section 3.1.)
2. AI/database synergy. I apply the advantages of database systems to improve video game AI systems.
3. A framework for the database agent engine. I design an architecture which allows an AI module to communicate with the rest of the game engine, and vice versa, and also is expressive enough to allow tuning for specific purposes and workloads (i.e., fewer but more intelligent agents or many but simpler ones). (Section 3.2.)
4. Efficient agent module. I investigated ways to handle agent behaviours efficiently and better than the current state-of-the-art. I created a module for processing agent AI scripts which is able to do computation and make decisions for thousands of individual agents per second, and is modular and usable by a modern game engine for strategy and simulation games. This module does not require any external servers to compute efficiently the results of the AI scripts for this scale. (Chapter 4.)
5. Experimental evaluation of proof-of-concept. I demonstrate that the proposed

architecture improves over state-of-the-art for scalability and performance. (Chapter 5.)

1.5 Outline

This thesis is structured as follows. In Chapter 2 I describe the most relevant work, their advantages and their shortcomings. This sets the stage for what problems need to be solved. Chapter 3 is divided into two sections. First, I describe a game space model. This introduces concepts which are needed when talking about AI in games, designing them, evaluating their computational cost, and setting goals for an AI agent module. Next, I introduce the conceptual model for the module. This model describes how we can apply the database technology to help game AI and connect it with the rest of the game engine. In Chapter 4, I use the introduced models to create a prototype. The prototype is a fully functional artificial intelligence module for video games. It communicates with a game engine created for this thesis to create a real-time game prototype. In Chapter 5, I report the results of tests used to evaluate the module. In Chapter 6, I conclude this research and summarize what I have discussed in this thesis, and outline areas for future improvement and future work.

2 Related Work

There has not been much research related to combining database technology and artificial intelligence for autonomous agents in computer games and simulations. My thesis topic is fairly new in what it is covering, and is experimental in nature. There is, however, research being done on different isolated components which are important for creation of such synergy.

2.1 SGL and Cassanova

The closest research to my proposed topic, and the motivation for me pursuing this research direction further, is a paper entitled Declarative Processing for Computer Games [20]. The mentioned paper described a framework which uses database technology to simulate behaviours of thousands of agents. The paper introduced new and interesting concepts related to AI and databases, provided a scripting language to aid with the development of AI scripts, SGL, but it did not describe or discuss critical components which would be needed to make their system usable in development of a complete game. I will briefly elaborate on these shortcomings, and where further development is required.

SGL uses a state-effect pattern. In that pattern, the authors separate the game object attributes into two disjoint sets: states and effects. States are columns in the game object table (for consistency reasons, I will refer to this table as an agent table) which hold

all information describing an object; effects are columns which hold all information about how an object's states should be changed (e.g., a state is health, and an effect is damage taken). In their implementation of this pattern, they add more columns to the agent table for each new effect type, so when a new effect type is introduced to the game, the database schema needs to be modified to accommodate the new effect. Another problem is that if we would want to create a long lasting effect (the effect which spans for more than one clock tick, like being slowed down for a certain amount of time) we would have to add at least two new columns into our database schema, besides the ones needed to describe the effect itself: one column for the remained duration of the effect, and one column which will hold the value the affected column will take after the effect expires (when a slow effect expires, an agent returns to its normal speed).

The SGL framework is built around one global update query. All effects and game rules and AI scripts for different unit types will have to be hard-coded into that one query. The SGL scripting language provides some assistance when creating that big global update query by dividing it into many smaller sub-queries which are connected together via the UNION operator after they are complied. But the end result is the same: a big, cumbersome, and not easily readable query. (Likewise, this cumbersome SGL query likely will not be optimized well and executed efficiently by the database system.)

The authors of [20] pointed out that one of the challenges with their implementation of the state-effect pattern is integrating it with other parts of the game engine. In other words, not every game component can be represented with this pattern so that the SGL framework can make use of it. For example, it cannot support graph traversal algorithms such as A*. Another limitation with their system is that the update query must not contain any form of iterative loop other than a JOIN.

Their system is focused in solving one problem, while leaving many crucial and important problems to be solved later. The authors stated they had problems with simulating different effects and the interplay between them. As well, they did not describe the communication between the game engine and the AI module. In [20], the updating the database after the effects have been computed is not considered, and their benchmarks do not include time spent doing so. The time it takes to update the database with new

values is not negligible, and must be considered when designing a system for the purpose of video games.

Cassanova [4] is similar to [20] in its intended goals, to simulate behaviours of thousands of agents. Cassanova is sufficiently fast to achieve its goal, but in order to do so, certain limitations on the types of allowed behaviours are imposed. The architecture lacks support for arbitrary algorithms, such as A*, script rules (predefined functions which govern how columns are updated) cannot execute imperative code (code which modifies the current state), at most one rule may be associated with any given column of the game state, all rules are always applied at every tick of the simulation, and communication between it and the the game engine is not considered. Due to these restrictions, the Cassanova framework cannot be integrated into game engines as-is.

Cassanova and SGL also introduce their own scripting languages. While they simplify the use of those complex systems, introducing a new scripting language has its drawbacks. In order to make a fully functional scripting language which would be useful in big projects, a proper debugger and editor must also be created, and the language implemented and supported. With a standard language such as SQL, all of this is already in place.

Neither the SGL nor Cassanova systems are designed with modularity and extensibility in mind. It is necessary to tackle and solve the issues mentioned above to make an usable AI engine for current generation of video games. This motivates my work and DAM.

2.2 Handling Moving Objects

There are also components which are not necessarily related to artificial intelligence and video games, but are necessary when constructing a world filled with agents. The paper Indexing Moving Objects using Short-Lived Throwaway Indexes [6] looks at the problem of efficiently handling a database filled with moving objects. Their solution provides a way for indexing, updating, and querying huge data sets. It can update millions of objects and query for positions of thousands of them per second. But in order for their sys-

tem to show such results, they ran it on a collection of server machines, each with two separate processors containing two cores, connected by a hi-speed network. There is plenty of other research done on the topic of efficiently working with moving objects since it has applications in military [1] and air and ground traffic control [6]. However, all proposed solutions include a network of multiple server machines doing the needed computation. This is unacceptable for video games ,since most of them run on a single machine.

Another topic related to my thesis are steering algorithms. Egocentric Affordance Fields [9] and ClearPath [15] are two algorithms which provide efficient solutions to how to handle steering, local path-planning, and obstacle avoidance in a world with thousands of moving agents. These algorithms are not related to databases, but necessary when you are constructing a world filled with agents.

2.3 Game Development Platforms

Since this thesis is being researched and developed for the purpose of it being used in video games, I will list a few video game development platforms (or simply, game engines) available and what is known about them. Video game companies keep their source code well guarded and generally do not publicly publish their algorithms. So I can only look at their official documentation, study games which use a particular video game engine, and create test cases using those engines to gauge the scope of their capabilities.

Duplicating the test cases across those engines and the module prototype DAM created for this thesis would be well beyond the scope of this work, to first get to know the engines and then properly replicate the tests across all of them. So I leave the comparisons and evaluations of the commercial engines for future work. Instead, I use the official documentation and case studies of video games to compare my thesis to those engines.

One of the more popular game engines, Unreal 3, is used by many games today [22] to great success. It has a reputation of being one of the more capable and professional engines available. Unreal engine claims to support “hundreds of characters” powered by

their crowd system [3], but under a few assumptions: those agents will not collide with each other and the world, their AI is disabled (they will not engage in complex interactions with their environment), and will simply move as a flock between predefined (manually placed or automatically generated) navigation-nodes scattered around a level. Other powerful and widely used engines, such as Source Engine [19] and Unity [17], do not give any hints at how many autonomous agents they can handle. However, games made with those professional game platforms do not generally have more than 20-30 agents acting at the same time in the same area.

On the other hand, there are strategy games, such as Warcraft, Starcraft, Total War, Command & Conquer which can have hundreds of units per side. But those units do not have complex individual behaviour scripts; rather, they are controlled by a centralized global AI entity which usually gives the same commands to groups of units. And games like Total War have thousands of units, but units are grouped together which act as one, creating 10-20 different groups. So an individual agent is a collection of equal thinking and acting units only told apart by the rendering engine.

Video games have gone a long way in enabling the display of thousands of units on a screen at one time. We just need to make them all act as individuals.

3 Methodology

3.1 Game Space Model

When choosing between different AI components to use for a video game, it would be beneficial to have a way to compare raw performances of said components in similar situations. Performance of an artificial intelligence component depends on several factors which influence its overall running time. Encapsulating and quantifying these factors, or parameters of the AI component, provide a way to talk about the features of an artificial intelligence component in a more measurable fashion. Parameters which influence the performance of an AI component are collectively titled as the Game Space Model. The model assumes that agents have uniformly distributed action activation times across their respective cooldown intervals, and that the distribution of agents in an area is uniform.

3.1.1 Agent Computational Cost Per Second

Agent's cost per second (cps) is a metric used for quantifying how computationally expensive an agent is. Any action an agent does potentially requires a non-negligible amount of processing power to be resolved. One such action is a request for a path to a goal to be found. Since path finding is a computationally nontrivial action, it will have a

cost > 0 associated with it. An agent's cost is a sum of the costs of all actions the agent can fire.

To evaluate the total cost of an agent in turn-based games, where actions fire once a new game turn begins, we will add the costs of all actions an agent can execute per AI tick. In real-time games a tick can have an arbitrary firing interval which will not depend on user input (pressing the next turn button). Instead, the firing interval is controlled automatically by the game engine. In those cases, the cost of an action which fires every tick will depend on the time between two ticks; the more frequent the ticks are, the more processing power an agent consumes per second for that action. Therefore, for real-time games it is more informative to represent the cost of an action which fires every tick as its cost per second. The cost per second of an action which fires every tick is:

$$\mathrm{cps}(\mathrm{action}_{\mathrm{tick}}) = 1 / \mathrm{tick}_{\mathrm{interval}}$$

Behaviour of agents does not necessarily need to depend on the tick duration, instead an action can have a cooldown time before it can be fired again. In that case the cost per second of an action which fires once per time interval is:

$$\mathrm{cps}(\mathrm{action}_{\mathrm{cooldown}}) = 1 / \mathrm{action}_{\mathrm{cooldownInterval}}$$

Figure 3.1 shows the cost of an action which fires once per tick, and an action which fires once per second. In practice, some actions are more expensive than others. An action of firing a weapon is not as computationally as demanding as an action to compute the shortest path to a goal. To reflect this we add a multiplier to actions cps values. The multiplier can simply be 1 for trivial actions, and 2 for non trivial.

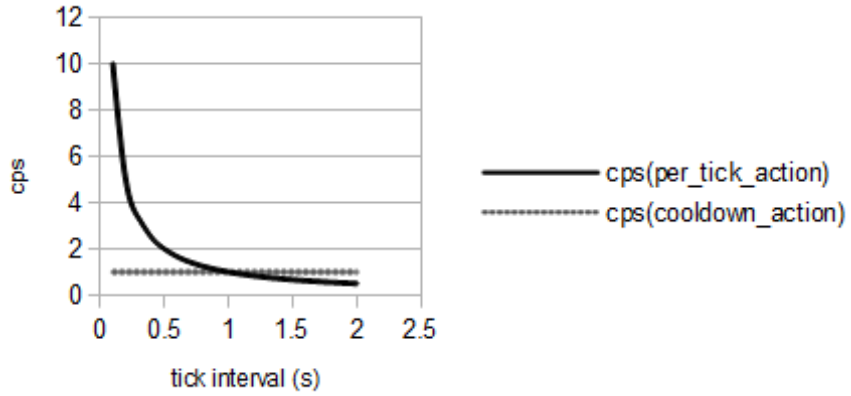

Figure 3.1: Two types of cps functions.

3.1.2 Number of Agents

Number of Agents is a metric about how many agents must be processed every tick. It takes the total amount of agents in the world not considering if they are on cooldown or not. The effect of cooldowns is reflected in the cps parameter.

3.1.3 Agent Density

Agent density is a metric used for determining the percentage of agents which are inside of another agent's area of influence.

$$\text{density} = \text{num}_{\text{agentsInAreaOfInfluence}} / \text{num}_{\text{agents}}$$

Given a density we can then find the size of the world which contains a uniform distribution of agents necessary to to achieve the given density.

$$\text{size}_{\text{world}} = \text{radius}_{\text{influence}} / \text{density}$$

A density value of 1.0 indicates that all agents in the world can influence each other, where a density value of 0.5 indicates that approximately half of the agents in the world are visible to any one agent.

3.1.4 Putting it all together

The Game Space Model is used to quantify a cost of an agent so we can analyze and compare agents in terms of their approximated computational complexities to identify areas for optimization. It defines two tightly linked parameters, the number and density of agents, which determine the needed size of the world to support those agents. After adapting this model to a specific implementation of an AI component it can be used to set requirements when designing agent behaviours and game worlds. But before requirements are set, the AI component is tested against the parameters of the Game Space Model. This gives a better idea of what the component is capable of. Requirements can be set afterwards. For example, an agent's state must have a cost per second value of less than 5, that the system must be able to resolve actions of 4000 agents in under 200 milliseconds, in a world with density of 0.1. The model is used to test the limits of an AI component implementation and to compare it with other, differently implemented, AI components.

The model uses approximated values and abstract parameters since real-world performance of an AI component depends on how well other areas of a game engine which use the AI component are implemented and run. That is also a disadvantage of this model: it is using approximate values of agents real costs, and its accuracy depends on how well the cost values are defined. But once a component has been implemented, modifying these parameters and measuring how the system behaves can be used to test the accuracy of the created model for a particular game and if any adjustments need to be done to it before it is used in level or behaviour design.

Although informative, this model is a forgiving approximation of the actual performance of an artificial intelligence system and should be used as such.

3.2 The Conceptual Model

A decade ago, a video game developer had a choice to purchase an available video game engine or, if available engines did not support the developer's vision, to create most of the code himself in order to create a game. A game engine is a big software sys-

tem which usually is required to be used as a whole; modifying it to perform in a way it was not designed for is not cost effective. Because of that all-or-nothing approach, recently there has been an emergence of modules, smaller systems designed to solve specific categories of problems. Today, different modules exist which are designed to compute rigid body dynamics [10] or just to render trees [5]. The trend in the video game industry is leaning towards integrating modular and well interfaced solutions to problems rather than creating new solutions. It saves money to do so.

The following sections describe my model for an AI module which is built on top of a database server. An AI module specializes in resolving behaviours of video game autonomous agents. The model tries to address various aspects of creating an independent, self-contained module which can be integrated with a game engine.

When designing the model, two important things need to be kept in mind. Firstly, the model has to allow for easy extensibility. The way objects are implemented inside of an AI module must be independent of each other, and adding or removing objects should not require any major restructuring of the code. Game objects are coded as isolated components of the AI module, and communicate through a common interface. Secondly, the module should not limit artificially the expressive power of behaviours. Imposing limitations can help with optimizing a module, but it reduces its set of possible applications. For that reason, the model will assume actions to be a mix of SQL queries and C functions, where appropriate. Both C and SQL are widely used languages and have their own debugging tools. There is ample literature, examples, and support on how to best use them. A goal of this model is to take advantages of database systems, not try to translate every sub-system into a database query.

I refer to the introduced database AI model as the EDA model (elementary database AI model). An implementation of a database AI model is referred to as a database AI module, or simply an AI module. (The database part is implied for the remainder of this writing.) Chapter 4 describes an implementation the EDA model.

3.2.1 Databases and Games

All information which is to be used by a database server must reside inside of it in the form of a table. (Or in a form of a memory block inside of the database process heap space accessed via custom made functions.) For that reason, a database must hold all game state variables relevant to the game AI computations. We will need to store certain copies of data both in a database server and in some other area of a game engine. Both the database AI module and the UI module might need to know the health of an agent, for decision making and rendering, respectively. There is a set of duplicate information between modules, but also different modules store their own data necessary only for their own execution. Such duplication of data is unavoidable because an external database server is running in a separate process with its own memory space so a game engine cannot get a direct pointer to objects stored inside of the database. Even if a database server and the game engine shared the same memory space, a pointer to shared data may be difficult since the database server has its own preferred way how to encode and store data. The memory footprint is not the main issue with duplicated data since if we have four thousand agents which share 80 bytes of data (20 floating-point variables) each we will only have approximately 0.3 megabytes duplicated. A bigger problem is copying this object data between the two processes which may take up noticeable simulation time if implemented improperly.

The need for duplication of certain data is one of few limitations this model imposes. But that has its advantages. If we store a game state in a database table we can make use of many features a database system is providing, such as indexing, query optimization, concurrency control, parallelization, and the SQL programming language.

3.2.2 Queries as Actions

There is a similarity between the concept of actions (from an AI perspective) and database queries. A database query is a program which can retrieve or change rows in a database table; an action is a computation which inspects or changes game state variables. Due to the expressive SQL language [20], paired up with a C interface with the

database server, we can look at query answering as an arbitrary computation, and an action an agent can do as a query.

3.2.3 Objects and Databases

A game object is any entity in a video-game which can, directly or indirectly, influence the outcome of the video-game simulation. An example of a game object is an agent or an effect attached to one. A game object is stored in a database as a row, or rows, of columns belonging to one or more tables. A column can hold information about the object's current position, speed, health, or any other variable an AI system needs to keep track of about a specific game object.

Depending how table columns are queried, they can be indexed for faster access. Building an index for a column adds an additional memory footprint, but it is not significant, and can be ignored when designing a system such as this. For video-games where objects are distributed across a game world, it is important to create a multidimensional index on the position column to speed up spatial queries. Other candidates for indexing are the columns holding an agent's current state, its team, and a unique id for easier referencing.

3.2.4 Triggers as Events

Modularity and extensibility are important when crafting a module to be used in a wide variety of different projects. Changing a well made modular software is less likely to cause major problems, and it can be reused thereby reducing project development time and cost. The event-driven programming is a powerful paradigm which provides a clean and easy way to abstract software into individual components which will do computation when a specified criteria has been satisfied. Video game engines make elaborate use of event-driven programming.

Databases support events in form of triggers. A trigger is a procedural code which is automatically executed as a response to a change in a database table. When a trigger is created a condition is specified which must be satisfied for a specified function to be ex-

ecuted. One such condition is that object's health column has changed, and that the new value is below zero. A trigger can be defined to execute a function for each row in a table which satisfies an event condition. Other examples of events are firing a notification a new agent has been added to the world, an agent has lost health, a car ran out of fuel, or an agent has moved, amongst others. Any such even can be represented as a trigger.

Triggers can fire before or after a change in a table has occurred adding to their expressive power. A trigger which fires before a table row has been changed can control how and if columns will change their values. For example, if an agent would take damage, a decrease to its health column value, any trigger set to fire before a change to the health column value occurs can intercept and modify the amount of damage the agent took. If that particular agent is immune to damage, the trigger function would simply state that the new column value is the same as the old one.

3.2.5 Communication with the Engine

Any external module which works alongside a game engine needs to communicate with it and exchange information. That is achieved through a common interface bridging the gap between two different systems. A well made module has a minimal interface (API) needed for communication, and does not require any drastic changes to the client code using the module. The primary interface between a database system and its client are tables and queries.

In game engines, references to objects are typically stored as handles. A handle is an integer or a string index which is used to access the object instead of a direct pointer. Handles are (preferably) stored in data structures which allow for fast insert/delete/find operations such as a sorted array or a sorted binary tree, depending on the underlying structure. Working with handles is necessarily slower than with direct pointers, but not by much. (We can achieve a O(1) object access times if we store handles in an array and the handle itself is an offset from the start of the array.) With handles we are protecting contents of objects and hiding their memory locations, which allows us to change memory addresses of objects and preserve the values of their handles. The latter is useful in memory read/write intensive applications with a longer lifespan. In such applica-

tions memory fragmentation is inevitable; memory has to be defragmented, changing physical memory addresses of objects. Handles provide a meaningful way for modules/processes/libraries to communicate between each other. Columns in a database can be handles to the actual data they are pointing to. Databases support fast access handles via primary keys. (Primary keys are responsible for many other things as well.) Handles to objects will be the primary way to work with objects which were created inside of an AI module.

An AI module is responsible for resolving agent behaviours and changing the game state. The changed game state is stored in a database table, which is then queried by a game engine. The game engine proceeds to update the rest of game objects with the newly read data. This method for retrieving the game state data can be used to read any expected and constant game state information (e.g., agent positions) but the method is not well suited for event-based message passing (an agent fired a shot).

To avoid extending the game state tables to accommodate for every message type a AI system can pass, two separate tables which will hold messages are created: one to hold messages sent by an AI module, and one table holding messages intended for an AI module. When an AI module sends a message to the game engine, it inserts a new row in the outgoing message table, and populates that row with the message information. The game engine will then query the outgoing AI module message table and resolve all pending messages.

Messages can be different with respect of the information they carry. They can have any number of variables connected with them. These variables will not be individually queried and are intended only for transmission of data. For that reason the message tables need only to have only one column whose type is a variable length array of bytes. How messages are encoded in byte arrays depends only on the design of the interface and has no restrictions.

Using these two methods in combination, querying for a new state and processing messages, allows modules to exchange any data between each other using a simple interface.

3.2.6 Tick

A game engine is responsible for notifying the AI module that a new tick has begun. At the start of a new AI tick, the AI module resolves all pending messages from the game engine. After messages have been processed, both message tables are cleared. The AI module then executes its global per tick script which informs all game objects of a new tick.

Game objects that have triggers set up to listen for a new tick will inspect if other conditions have been satisfied, and if so, fire appropriate functions. Functions which modify the game state can do so, either right away or save the world updates to be applied in a batch at the end of a tick. Either method will produce the same result since a database server will not modify any data during a same transaction anyway (a transaction being an execution of a SQL query, a global tick query in our case). All changes to table values will be applied at the end of a transaction, even if they have been modified in the middle of it.

Even though actual changes in table column values occur at the end of a transaction (that is, the changes are not seen outside of the scope of the query or queries within the transaction until the transaction has committed), triggers which depend on a change of a column value will fire as soon as a query which does so is parsed by the database server (which happens right after a query has been sent to a database server). This allows us to create events which fire before a change has occurred and override it with another value. This is useful for objects which are immune to specific events, such as a loss of health notification, and undo the effects of the events.

4 DAM Prototype Design and Implementation

The database agent module (DAM) is an implementation of the EDA model described in Section 3.2. The model prototype is built on top of the PostgreSQL database server and interfaced with a game engine coded in C++. The DAM is intended as a proof of concept and to test the validity and expressiveness of the model. It is a prototype and is not necessarily the most optimal way to implement said model.

This chapter is organized as follows. In Section 4.1, I outline the requirements for the DAM prototype. In Section 4.2, I present the overall prototype architecture and its implementation. In Section 4.3, I focus on how agents and their state machines are implemented. In addition to agents, three other game objects are supported by the DAM. These are sensors (Section 4.4), effects (Section 4.5), and obstacles (Section 4.6). In Section 4.7, I describe the implementation of the messaging system. In Section 4.8, I outline the demands and requirements of the client. In Section 4.9, I investigate the strengths and weaknesses of the current implementation of the DAM.

4.1 Prototype Requirements

The DAM is designed to be a self contained, engine-independent module, following the trend of the industry for modularization. The prototype has as few constraints as possible, it supports behaviour scripts of an arbitrary complexity, and it is flexible enough so

that adding new behaviours and objects is streamlined and does not need restructuring of existing code. It is an external module which communicates with the game engine through a minimal interface.

4.2 Architecture

The DAM is coded in SQL and C and packaged as a PostgreSQL DLL. When an application starts, game functions and triggers from said DLL are loaded into the database server. The entire collection of DAM game logic and behaviours are stored in, and processed by, the database server.

The DAM and the game engine run in their own individual threads. Unfortunately, at the time of writing, PostgreSQL does not support parallel computation of single queries. Even though it supports multiple queries being computed in parallel, the primary goal of the DAM is not so much to achieve the best possible performance as it is to provide a test bed for the EDA model. Thus, the PostgreSQL feature for running multiple queries in parallel is not exploited. The DAM interfaces with the database server through a single connection which can execute a query at a time. These queries can have nested subqueries, but they will all be considered to be a part of the same transaction and resolved in a sequential fashion.

The module makes use of storage structures for storing additional game object data. Every game object type has a trigger which fires when a new row in the object table has been added. The trigger calls the objects constructor function which allocates space in the server memory to hold a user defined C structure, the object data. Similarly, when a row has been deleted from the object table, a destructor function will be called to free the allocated memory space.

Similar to C #define directives, SQL programs for the use by the DAM can contain string constants in them, allowing for easier readability of those programs. Using string constants instead of numbers decreases the chance of introducing an error which will need to be identified and resolved. The DAM replaces those strings with predefined numeric values before the SQL program is sent to the database server for execution. Fig-

ure 4.1 shows the use of a FIGHTER_AGENT_TYPE string constant. The same string constants are used inside of the C program in the form of #define directives. Ideally in a future version of DAM, one file of constants would be used for both.

```
CREATE TRIGGER onAgentFighterInsert
AFTER INSERT ON agent FOR EACH ROW
WHEN (NEW.type=FIGHTER_AGENT_TYPE)
EXECUTE PROCEDURE agentFighterInit();

CREATE TRIGGER onAgentFighterDelete
BEFORE DELETE ON agent FOR EACH ROW
WHEN (OLD.type=FIGHTER_AGENT_TYPE)
EXECUTE PROCEDURE agentFighterDelete();
```

Figure 4.1: An example of creating SQL triggers.

In the previous example, the triggers defined in SQL are calling two functions defined in C: agentFighterDelete and agentFighterInit, respectively, which are loaded from the DLL, and connected to the PostgreSQL database server using the SQL code in Figure 4.2.

```
CREATE OR REPLACE FUNCTION agentFighterInit()
RETURNS TRIGGER
AS 'game functions.dll', 'agentFighterInit'
LANGUAGE C;

CREATE OR REPLACE FUNCTION agentFighterDelete()
RETURNS TRIGGER
AS 'game functions.dll', 'agentFighterDelete'
LANGUAGE C;
```

Figure 4.2: Examples of binding C functions.

The two C functions create an agent's storage structure using malloc and delete using free. Pointers to the created structures are saved in the object table as an integer column.

A side effect of the DAM architecture design, and how it is integrated with the game engine, is that the database agent module has a one clock tick latency. The DAM immediately starts computing the new game state when a new clock tick starts, but a certain amount of time needs to elapse before those computations are resolved and the results presented to the game engine. The game state which is available at the end of a tick (after completing the needed computation) is the state of the world at the time when the tick began, since that is when the database system read the game state and started to do computation. It is not surprising that computations take time and results are not immediately available, but it is important to accommodate this phenomenon when designing the prototype.

4.3 Agents

Agents are game objects which execute actions to change the game state, and which make decisions based on the game state. Agents influence one another, and query for other agents' states. Their behaviour is loosely based on finite state machines. An agent must be in one state at a time, but the DAM implementation does not pose any rules on how states transition. State transitions are controlled by the design of state functions. An agent's current state is primarily stored in a table with the schema presented in Figure 4.3.

The agent table consists of state variables which the game engine, or an action query, will use. However, an agent can have additional variables describing various aspects of its state saved in its storage structure. State functions are coded in C, and invoked with triggers.

In order for a trigger function to be called, a change in a table column needs to occur and a specified condition to be satisfied. The agent table, and other tables which require per-tick functions to be called, has a column named activate. The activate column is used to signal an agent that a new tick has begun. Every tick the DAM loops across all agents and changes the activate column value. The actual value is not important, just that an update query has been invoked on the column. Figure 4.4 shows a per-tick trigger which calls an idle state function if the agent is currently in the idle state.

```
CREATE TABLE agent(
        id INTEGER PRIMARY KEY,
        position POINT,
        speed REAL,
        radius REAL,
        health REAL,
        type INTEGER,
        team INTEGER,

        state INTEGER DEFAULT 1,
        currentspeed REAL DEFAULT 0,
        direction POINT DEFAULT POINT(0,0),
        storagedata INTEGER DEFAULT 0,

        activate BOOLEAN
);
CREATE INDEX agent_position_index ON agent USING spgist(position);
CREATE INDEX agent_team_index ON agent USING btree (team);
CREATE INDEX agent_id_team_index ON agent USING btree (id,team);
CREATE INDEX agent_state_index ON agent USING btree (state);
CREATE INDEX agent_state_type_index ON agent USING btree (state,type);
```

Figure 4.3: Agent table schema.

```
CREATE TRIGGER onAgentFighterStateIdle
AFTER UPDATE OF activate ON agent FOR EACH ROW
WHEN (NEW.type=FIGHTER_AGENT_TYPE AND
NEW.state=FIGHTER_AGENT_STATE_IDLE)
EXECUTE PROCEDURE agentFighterStateIdle();
```

Figure 4.4: State trigger definition.

All agents of all types are added to the same agent table. Different types of agents share the same interface, but the state functions and their implementation may differ. Adding a new agent type involves defining its behaviour function, and registering the corresponding state-function trigger, for that new type. The same steps are taken when

adding additional states to already existing agents.

In Section 4.3.1, I consider specifics of updating the agent's state. In Section 4.3.2, I present a cost metric for gauging the cost of state updates. Next, as a proof of concept, I implement three different types of agents in the DAM, each designed to test and challenge different aspects of the module. These are the fighter agent (Section 4.3.3), the droid agent (Section 4.3.4), and the controller agent (Section 4.3.5).

4.3.1 Updating Agent's State

Within the same transaction, changing the agent table using queries will produce equivalent results regardless of the order in which queries are being resolved. The database system's concurrency control guarantees that even if table values change in a middle of a transaction, the current transaction will not see the changed values and will continue its execution as if no change has occurred.

Database systems are optimized to process queries in batch. They utilize in-memory buffer pools which hold copies of data read from the datafiles that reside on the hard drive. Before processing a query the database system copies the needed data into its buffer pool. If subsequent queries work on the same data set, as they usually do in batch processing, the data in the in-memory buffer pool is likely to be used eliminating the need for costly hard drive operations. Frequently alternating between calling different query types (INSERT, UPDATE, SELECT, DELETE) produces longer per-query execution times. (I suspect this occurs due to having to flush the contents of the buffer pool and read new data with each different query type.) The SGL [20] made use of batch processing optimization by storing aggregate values of how columns should be changed as effect columns belonging to the agent table, and then those changes were applied as a batch after the tick has been resolved. The DAM works in a similar way, but the values with how columns should be changed are not stored as database columns. Rather, they are saved in variables inside of the agent's storage structure written in C. An agent holds a pointer to its storage structure and so reading from and writing to it is nearly instantaneous.

Different agent types share the same state schema, such as their health or position, and their storage structures, by design, are derived from a common agent storage structure parent. The common agent storage structure contains all variables which dictate how an agent's state should be changed. Since C does not support object hierarchy, this relationship is created by placing the “parent” agent's storage structure along the first bytes of “child” agent's storage structure (Figure 4.5). That way we can extract the common agent data without knowing the child agent's storage structure's actual type.

```
struct ChildAgentData
{
        struct ParentAgentData* parent;
        int data;
};
```

Figure 4.5: Agent storage structure hierarchy.

4.3.2 Agent's Cost Metric

The Game Space Model described in Chapter 3.1 defined the computational cost of an agent's state as a sum of costs of all actions an agent can execute in the given state. In the DAM, an action is a query. This query's cost is dominated by the CPU cost and the disk I/O cost. Because mixing INSERT queries with more common SELECT queries slow down the computation of behaviours considerably, I have labelled every SELECT query to have a cost of 1, and every other query to have a cost of 2. Using this cost metric is to discourage mixing different types of queries when designing agent behaviours. An agent's total cost is the cost of the most expensive state the agent can possibly be in.

4.3.3 Fighter Agent

The primary unit type in the prototype is the fighter unit. The unit serves as a test bed for the module and is the most optimized out of the three implemented agent types. It is the only unit used in a battle scenario described in Section 5.2.4. They are the grunts of this prototype. They have good defensive abilities, a fast firing weapon, are fast them-

selves, and can efficiently maneuver through an obstacle field. The fighter agent can be in three states: idle, move, and shoot.

In the idle state is the initial state in which an agent will scan for enemies or if its target position (goal) changed. If an agent's goal has changed, the agent will transition to the move state, or, if an enemy agent comes in range, it will change to the shoot state. When an agent is in its move state, it fires its egocentric movement sensor which returns the most optimal direction and speed the agent can take to reach its goal. (Sensors are described in Section 4.4.) When in the move state, it can switch to its idle or shoot state, depending on if the agent reached its destination or an enemy comes in range, respectively. The requirement for an agent to switch to the shoot state is that an enemy agent came in range. When this occurs, the agent which detected the enemy saves the enemy's unique id to its storage structure, so that when it enters the shoot state, it does not need to query again its surroundings to see which enemy is in range. When transitioned to the shoot state, an agent will stop any movement and shoot at its target.

When an agent decides to shoot, two things will happen: The shooting agent gets a pointer to the target agent's storage structure and increments its damageTaken variable for the current tick. This will modify the game state. The shooting agent will then send a message to the game engine (see Figure 4.6), by inserting into the message table that a shot has been fired. When the game engine receives this message, it will create appropriate particle effects.

```
struct AIMessageCreateBulletEffect message =
{ AI_MESSAGE_CREATE_BULET_EFFECT };
message.sx = (float)agentPosition->x;
message.sy = (float)agentPosition->y;
message.ex = (float)enemyPosition->x;
message.ey = (float)enemyPosition->y;
sendMessage(&message, sizeof(message));
```

Figure 4.6: Sending a new projectile message.

The sendMessage function inserts a new row into the messagesFromDAM table and copies the sizeof(message) bytes from the &message pointer into the database BYTEA

column. (BYTEA is an array of bytes.) The sendMessage function is used by all DAM objects.

The fighter agent is in possession of a protective shield which negates all damage done to the agent that happened at the same time (in the same tick). For example, if a fighter agent is being hit by bullets from multiple directions at the same time an explosion happens, the fighter agent will not take any damage. Instead, his shields will go down. To implement this behaviour a trigger is defined for the fighter agent which fires before the agent has lost any health (Figure 4.7), and calls a function which checks if its shields are up or not. The invoked function will either not do anything (and the agent will lose health), or it will undo the health loss and remove its shields. If a fighter agent has shields or not is a boolean state variable specific only to that agent type, and is stored in the agent's storage structure.

```
CREATE TRIGGER onAgentFighterBeforeDecreaseHealth
BEFORE UPDATE OF health ON agent FOR EACH ROW
WHEN (OLD.type=FIGHTER_AGENT_TYPE)
EXECUTE PROCEDURE agentFighterTakenDamage();
```

Figure 4.7: Intercepting agent's health loss.

The shields will recharge after some time. The shield recharge mechanism is not dependent on the agent's current state. To model this cleanly, the function which checks if shields should be recharged or not is implemented in a similar way to state functions. Trigger conditions for this function do not have an additional constraint that an agent must be in a specific state (e.g., the idle state). Therefore, the shield recharge function will be called once per tick, regardless of the state an agent is in (Figure 4.8).

```
CREATE TRIGGER onAgentFighterPerTickUpdate
AFTER UPDATE OF activate ON agent FOR EACH ROW
WHEN (NEW.type=FIGHTER_AGENT_TYPE)
EXECUTE PROCEDURE agentFighterPerTickUpdate();
```

Figure 4.8: Per Tick Update.

The DAM is designed to run in real-time and actions which agents fire have cool-downs in seconds associated with them. For the fighter agent, enemy check times have

a cooldown of 2.5 seconds, egocentric steering computation has a 0.3 second cooldown, time between two shots is 1.7 seconds, and the time it takes for the shield to recharge is 2.1 seconds. Some actions are fired only in specific states. The agent's state costs have been computed. In the idle state, an agent only checks for enemies (one select query) so:

$$cps(idle_{fighter}) = 1/2.5 = 0.4$$

In the move state, the agent checks for enemies, as well as fires the egocentric sensor which runs two select queries:

$$cps(move_{fighter}) = 1/2.5 + 2/0.3 = 7$$

Computing the cost of the shoot state is a little different. If an agent is in its shoot state, it is safe to assume that the agent will also be taking damage, which implies that every 2.1 seconds, the time it takes the shields to recharge, the agent will send a message that the shields went down, and a message that the shields went up later on, and once per 2.1 seconds it will also negate taken damage with an update query. In addition to the cost of shields, every 1.7 seconds the agent will retrieve the target agent's location (verifying if the target is still alive), and send a message that a shot has been fired:

$$cps(shoot_{fighter}) = 3/1.7 + 3 * (2/2.1) = 4.62$$

Experimental data supports these approximated state costs. The DAM takes the most time to resolve a tick when fighter agents are in their move state. (Experiments are described in Chapter 5.)

The total cost of the fighter agent in the worst case scenario then is:

$$cps(agent_{fighter}) = 7$$

As we saw so far, following the model described in Section 3.2, we have created a clean way to add features to existing game objects by adding new triggers and defining their functions. The following two unit types were created to test the expressive power of the DAM and are not necessarily coded in the most optimal way.

4.3.4 Droid Agent

A droid is a simpler unit type. It is representing a swift, self steering intelligent bomb. Once an enemy comes in range, a droid will lock on to it, switch on its auto pilot system (thus ignoring any future commands to change its course), enable thrusters to increase dramatically its speed, and launch itself to its target creating a large explosion when it comes within a critical range. Since they do not require anyone to pilot them directly they are small, small enough to assume safely they will never collide with other agents. The functionality not to consider specific types of entities is implemented primarily to further distance the droid agent implementation from other agent types. A droid agent can be in three states: idle, move, and engage.

Its move state is different than the fighter agent's move state in how the egocentric movement sensor has been calibrated. The egocentric movement sensor does not consider other agents when it is computing the steering direction and velocity for the droid agent. Droid agents can only collide with static obstacles, and they try to avoid only those. Another difference is that when a droid agent switches to its engage state, it will stay in that state until it travels the required distance to reach its target. It explodes damaging all agents, friendly or not, caught in the blast radius. When a droid agents explodes, it will query for all agents in range and their storage structure pointers, and increment their damageTaken variable for the current tick. It will also send a message to the game engine to create the visual part of the explosion.

A droid agent's state costs for the idle state is:

$$cps(idle_{droid}) = 1/3$$

The droid agent is a little slower in its response time to an enemy threat, 3 seconds between checks. When not engaging, the droid is also slower than the fighter agent to avoid obstacles, and checks for them every 0.35 seconds.

$$cps(move_{droid}) = 1/3 + 1/0.35 = 3.2$$

When engaging, the droid agent makes a few adjustments to itself. Besides ignoring further commands and increasing its speed, it also becomes more agile, checking for obstacles as soon as it can (every tick). The cost of the engage state is then:

$$cps(engage_{doid}) = 1/tick_{interval}$$

It is not unreasonable to limit the DAM tick intervals to be once every 0.3 seconds. Mentioned agents do not have actions being fired with a higher frequency. In that case the cost 1/0.3 is the cost of the droid agent:

$$cps(agent_{droid}) = 3.33$$

4.3.5 Controller Agent

The controller is a specialized unit type. It is slow, vulnerable, and has no direct way to attack or defend itself. Its only weapon takes a long time to power up, 3.2 seconds, during which time the controller agent cannot move. When the agent finally manages to power up its weapon and shoots, the weapon creates a blast which temporarily converts all enemy agents in the blast radius to fight for the controller's team. The effect which the controller creates lasts 2.7 seconds after which the affected agent will switch back to its original team. When the controller's weapon is fired, it queries for all agents in an area, and creates an effect to change an agents team for each of the affected agents.

PosgreSQL allows for parameters, identified by the dollar sign and a number, in SQL queries whose values are set before the database server executes a query. In Figure 4.9 the parameters, which are set by calling agents, are the agent's team, the effect centre position, and the effect blast radius. The fourth parameter is a byte array which holds any constructor parameters the effect requires to be passed to it. Effects are described in more detail in Section 4.5.

```
WITH agentsInRange AS (
SELECT id, team
FROM agent target
WHERE target.position <@ circle ($2, $3) AND
target.team != $1 )
INSERT INTO effect(type, tagentid, constructorparams)
SELECT CHANGE_TEAM_EFFECT_TYPE, id, $4 FROM agentsInRange;
```

Figure 4.9: Change team effect creation.

The controller agent is not optimized and is introduced to the DAM to test its expressive power, mainly if it is capable to create cleanly arbitrary effects, such as the effect to temporarily change an agent's team. The controller agent can be in these four states: idle, move, charge up, and fire. It can only reach its fire state from the charge up state, and the charge up state can only transition into the fire state.

A controller agent checks for enemies every 3 seconds, and every 0.5 seconds it checks for obstacles while moving. So the controller agent's idle and move states have the cost:

$$cps(idle_{controller}) = 1/3$$

$$cps(move_{controller}) = 1/3 + 2/0.5 = 4.33$$

During its charge-up state, the droid does no computation. It waits for a certain time, 3.2 seconds, to elapse. After its weapon charges up, the agent transitions to its fire state, creating effects for all enemy agents in the blast area. The fire state must wait for the charge up state to finish, so the cooldown for the controller fire action is the weapon's charge-up time. In the fire state, a controller agent checks where the enemy moved, then it creates effects for every enemy agent in range. When an effect is created, the effect reads the affected agent's current team (that value is stored and later used to return an agent's team to how it was), changes it to the new team, and also initializes and saves a pointer to the effect's storage structure (Section 4.5.1). Therefore, the controllers agent's fire cost depends on the amount of affected agents, which depends on the game design (eg., blast radius and agent density). The controller agent's cost for being in a fire state is:

$$cps(fire_{controller}) = 1/3.2 + 2/3.2 + m*(1/3.2 + 2*2/3.2) = 0.94 + m*1.56$$

where m is the number of affected agents. The controller's cost per second is:

$$cps(agent_{controller}) = max(cps(move_{controller}), cps(fire_{controller}))$$

4.4 Sensors

Sensors are encapsulations of different SELECT queries to separate frequently used components and increase the modularity and readability of the code. Although they can do an arbitrary computation, they are designed so that they do not modify the game state. These helper functions are internal to the DAM and accept sensor defined C structures as inputs and outputs.

```
struct FieldSteeringOutput
{
        float speed;
        Point direction;
};
struct FieldSteeringInput
{
        // per agent constant
        int id;
        float radius;
        bool staticCheck;
        bool dynamicCheck;

        // per tick variable
        Point* position;
        Point* direction;
        float speed;
        Point* goal;
};
```

Figure 4.10: Egocentric field steering sensor input and output.

An example of sensor inputs and outputs is given in Figure 4.10. The sensor input and output structures are defined once, and stored in agent's storage structures. Storing the sensor saves memory allocation and deallocation time, and also variables which are agent specific, and do not change every tick (such as an agent's id or its radius), can be assigned once and reused later.

4.5 Effects

Effects are game objects stored in the effects table. Each effect is bound to a single agent, changing its state, either permanently or temporarily. A new effect is created by inserting a new row into the effects table, invoking appropriate effect constructor functions. (The schema for the effect table is presented in Figure 4.11.) The DAM notifies effects of a new tick by updating the effect activate column, similar to how agents are notified. An effect which lasts permanently does not need to create a trigger that listens for a new tick. Temporary effects require to create such a trigger. A temporary effect per-tick function keeps track of the effect's lifetime and removes itself from the table when it expires.

Effects are not required to change the game state. An effect which represents a cloud of smoke following an agent will, when constructed, send the game engine a message to create a smoke particle system on the agent. When this effect expires, a message to remove the particle system is sent.

```
CREATE TABLE effect(
        id SERIAL,

        tagentid INTEGER,
        type INTEGER,
        constructorparams BYTEA,

        target agent DEFAULT NULL,
        activate BOOLEAN DEFAULT TRUE,
        privatedata INTEGER DEFAULT 0
);
```

Figure 4.11: Effect table schema.

An effect may require certain parameters to be passed to it during its creation which are necessary for proper initialization of the effect. Constructor parameters are passed when a new row is inserted into the effect table, through the constructorparams column. The constructorparams' type is an array of bytes and it holds custom data.

4.5.1 Change Team Effect

The controller agent creates a temporary effect which makes enemy agents change their teams, and become the same team as the controller agent. After a short period of time, the effect will expire and the affected agent will switch back to its original team. (In the DAM, the change team effect expires after 2.7 seconds.) The effect's storage structure keeps track of the remaining time. The effect's storage structure is also used to save the affected agent's original team. The agents desired new team is passed to the effect as its constructor (Figure 4.12).

```
struct ChangeTeamEffectParameters
{
        int agentID;
        int newTeam;
};
```

Figure 4.12: Change team effect's constructor parameters.

For this effect, a constructor and destructor trigger functions are set up, which will change the agent's team to the new value, or revert it to its original value, respectively.

4.5.2 Immunity to effects

It is not uncommon for game agents to get a temporary, or permanent, immunity to certain effects. This is not implemented in the DAM prototype, but it can be accomplished using similar methods used to construct the game logic so far.

To stop an effect being applied to an agent, we tighten the condition for the trigger which fires on an insert into the effect table. In its WHEN clause we would add

```
AND NOT isAgentImmune(NEW.targetid, NEW.type)
```

The isAgentImmune is a function returning a boolean value, which searches through all applied effects on the agent (a SELECT query on the effect table, searching using the targetid index). For each found effect, the function will check a simple map to see if it is in conflict with the effect we are trying to add.

4.6 Obstacles

Obstacles are non moving game objects which interfere with agent movement. Obstacles are defined in a separate table. They are considered by the egocentric steering algorithm, and since agents cannot pass through them, obstacles directly influence the outcome of the steering algorithm. The obstacle table schema is given in Figure 4.13.

```
CREATE TABLE staticobstacle(
        id SERIAL,

        position POINT NOT NULL,
        radius REAL NOT NULL
);

CREATE INDEX staticobstacle_position_index
        ON staticobstacle USING gist(position);
```

Figure 4.13: Static obstacle schema.

The DAM only supports radial obstacles, although all mechanisms are in place to create polygonal obstacles if desired. (PostgreSQL supports spatial queries on objects defined as arbitrary polygons.)

4.7 Messaging

Two tables are constructed to hold messages, one holding messages passed to the DAM, and one holding messages passed by the DAM. They each consist of one column, a variable sized byte array (see Figure 4.14). These tables do not require their columns to be indexed (making INSERTs and DELETEs quicker) since for the current implementation of the DAM, searching over messages is not used. All messages are resolved in between ticks, and removed after.

When the DAM sends a message to the game engine, it inserts a new row into the messageFromDAM table and copies the binary message data into the data column. The

binary format is shared between the game engine and the DAM, so that when the game engine receives the message, it only needs to cast it to a known structure type. A game engine cannot liberally insert messages intended for the DAM. As mentioned previously, current implementation of the DAM limits the system to the execution of one query at a time. Messages intended for the DAM need to be queued, and only when the database finishes its tick they can be inserted into the messageToDAM table. This is a limitation of the prototype that could easily be removed by having two simultaneous connections to the database server, one for the DAM, and one for the game engine. The game engine could then freely execute queries while another query is being processed. The database server will process the queries in parallel while taking care to access the table data in a transactionally correct way.

```
CREATE TABLE messageFromDAM(
        data BYTEA
);

CREATE TABLE messageToDAM(
        data BYTEA
);
```

Figure 4.14: Message tables schema.

4.8 The Client

The client is a software package which is using the DAM module, a game engine in this case. The game engine built for this prototype is coded in C++, but since the interface between the game engine and the DAM are SQL queries, the client can be in any other programming language which is capable of communicating with the PostgreSQL server.

The client is running the DAM in a separate thread, and keeps track of when a DAM tick has completed to process messages sent by it. When the DAM signals that a tick has completed, the client sends all queued messages to the DAM. In this prototype, the client loads, and executes, SQL script files which initialize the game object tables, bind C functions, and create triggers. Additional responsibilities which currently lie in the client,

but which could and should be taken care of by the DAM ultimately, is table maintenance, and applying changes to the game state after they have been computed by the DAM.

Since the position column in the agent table is changing every tick, the agent table must undergo regular maintenance (using the VACUUM ANALYZE operation) at the end of each tick. Without this maintenance, spatial query answering becomes increasingly slower as the spatial data in the agent table becomes more unorganized.

The client is also cleaning up the effects bound to dead agents. All this functionality should be taken care of by the DAM and not burden the game engine.

4.9 Strengths and Weaknesses

Even though my DAM prototype is not the most efficient implementation of the EDA model (described in Section 3.2) that would be possible, it demonstrates the powerful features which the model affords.

- The DAM runs fully on the database server and is interfaced with the game engine through a minimal, well defined interface.
- It does not require the game engine to be structured in any particular way.
- The game engine can be created in any programming language which can communicate with the PostgreSQL database.
- The game logic being processed by the database server is largely coded using the C programming language. The expressibility of what types of behaviours can be implemented is limited only by C.
- Using triggers and sensors, the DAM is created as a component-based module that consists of specific, and reusable, components used by different game objects.

However, there are several disadvantages to using my DAM prototype.

- The main issue with the DAM is that it is built on top of the PostgreSQL database server. Prior to using the DAM, the PostgreSQL server must be installed on the

client's machine.

- The DAM slows down considerably with careless use of database queries, primarily alternating between using different query types (SELECT, INSERT, etc.).
- Lastly, certain parts of the DAM need to be re-designed. The implementation of the messaging system is not optimal (it needlessly duplicates data before it stores it into a table, and waits for the DAM tick to finish before processing messages).

5 Experiments

The database agent module (from Chapter 4) is tested against the parameters of the Game Space Model (from Section 3.1). Each of the tests modify the value of one parameter. As the value changes, the time the DAM took to finish its computation is measured.

5.1 Experimental Setup

5.1.1 Testing Hardware

Tests are conducted on an ASUS G73-JH laptop. The laptop has an Intel I7 CPU with four cores, each running at 1.7GHz, 8 gigabytes of RAM, and the Windows Windows 7 operating system. Both the prototype and the PostgreSQL database server (version 9.2) are 32bit applications.

According to a hardware survey [18] conducted by Valve Corporation [21], at the time of writing this thesis, an average gaming computer is a machine with a 2.3GHz CPU with four cores, 8 gigabytes of ram, and a Microsoft Windows 7 operating system. The testing hardware is similar in power to the reported average gaming computer. The video card is not considered because the DAM runs entirely on the CPU.

5.1.2 Server Tuning

I adjusted the following PostgreSQL server parameters to increase its performance, and removed some features not required by a video game.

- Maintenance work memory is set to 16MB.
- Stack depth is set to 3MB.
- Temporary and shared buffers are set to 32MB.
- Work memory is set to 64MB.
- Forced synchronization is turned off.
- Synchronized commits is turned off.

Database synchronization mechanisms help with protecting the data in an event of a server crash. If forced synchronization is enabled, the database server writes any database changes onto the hard drive as soon as possible. If synchronized commits are enabled, the database server waits on a query to finish, and its changes fully written to the disk, before it reports the result. Typically, video games do not have systems implemented to preserve data in events of catastrophic failure, so these synchronization mechanisms are disabled.

5.2 Tests

The following tests evaluate the performance of the DAM. For the tests, the three parameters are as follows.

1. 2000 agents in the world.
2. Agents fire actions with a cost of 1 cps.
3. Random distribution with density of 0.2. (A single agent, on average, considers 20% of the total agents, 400 in this case.)

The action is a query which scans an agent's surroundings and records all agents in range (Figure 5.1). The action is fired once per second. For each test, one parameter is varied, the other two parameters are held constant.

```
SELECT target.id, target.position
FROM agent target
WHERE target.position <@ circle ($1, $2);
```

Figure 5.1: Test action query.

Tests measure the time, in milliseconds, the DAM takes to complete a tick. The measured time includes tick computation (behaviours and effects), updates to the game state, and per-tick database maintenance. For each test, the time it takes for the tick to complete is calculated. The test time is the average time it takes the tick to complete spanning over ten seconds of computation. Each test is repeated nine times and their average time to complete a tick is reported.

5.2.1 Number of Agents

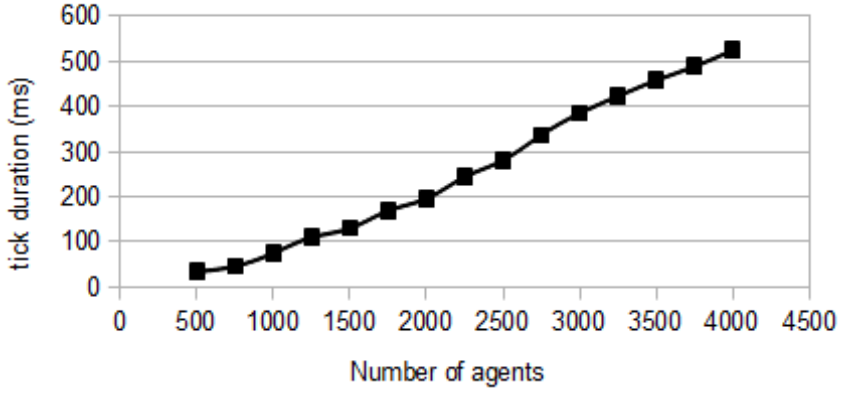

Figure 5.2: Change of the number of agents.

This test is designed to measure how the DAM behaves as more agents are added to the world. In order to keep the average number of agents inside of another agent's area of influence at 400, the size of the world is scaled as more agents are added to the world (Section 3.1.3). Figure 5.2 shows the results of the test. Data is taken at intervals of 500

agents, in a range of [500, 4000].

PostgreSQL uses R-trees for indexing spatial data. An R-tree is a balanced search tree. In the worst-case, when all nodes overlap, a search query on an R-tree has $O(n)$ time complexity. Generally, R-trees perform much better on real-world data [14]. A search query on an optimally constructed R-tree with no overlapping nodes has $O(\log_m n)$ time complexity [12], where n is the number of nodes and m is the maximum number of children a node has (typically 4 for a 2d space). An algorithm to construct an R-tree from a set of points has $O(n \log_m n)$ time complexity [7]. (We bulk-load our data into an R-tree since we destroy and reconstruct it every frame.) A bulk loaded R-tree is worst-case optimal for certain variations of R-trees [8].

During a tick, each agent queries an R-tree, and at the end of the tick the R-tree is reconstructed (with updated values, if any). Therefore, the total time complexity of the tick function is $n O(\log_m n) + O(n \log_m n) + C$. For our purposes (where n is of a reasonable size), $\log_m n$ is fairly small, causing it to add a negligible effect, making the entire graph give a linear appearance.

5.2.2 Cost Per Second

This test is designed to measure how the DAM behaves as agents change the frequencies of firing their actions, modifying an agent's cps parameter (Section 3.1.1). Result of the test is shown in Figure 5.3. The figure demonstrates that if the frequency of firing an action is equal or greater than the frequency of ticks, an action fires every tick, and cool-downs are ineffective in distributing the workload across several ticks. Increasing the action firing frequency further does not slow down the system more. Data is taken at intervals of 0.2 cps, in a range of [0.4, 3].

The plateau that appears on the graph depends on the computer hardware that is running the simulation. On a more powerful machine, the plateau ceiling will be lower, and begin manifesting itself at higher values of cps.

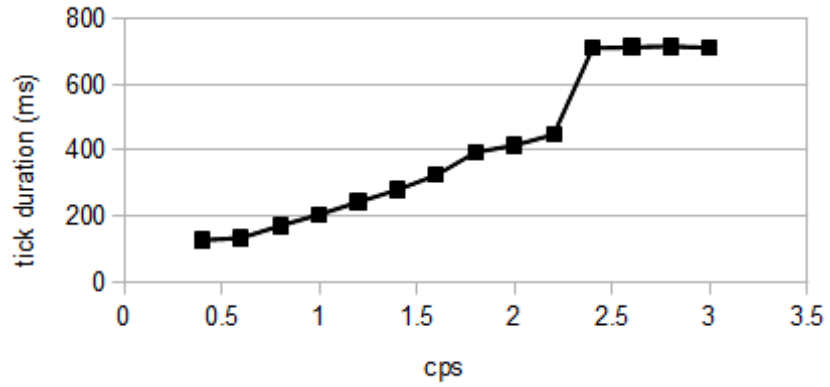

Figure 5.3: Change of cost per second.

5.2.3 Density

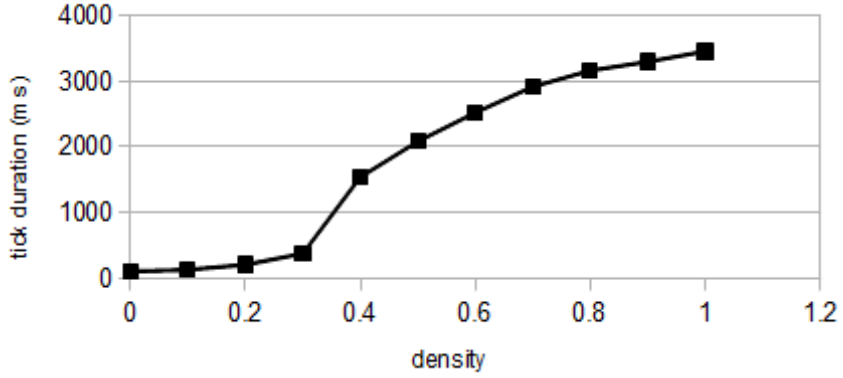

Figure 5.4: Change of the density.

This test is designed to measure how the DAM behaves as the density of the world changes. The density value is initially set to zero (no agent is inside of another agent's area of influence), and is incrementally increased until it reaches the value of 1 (all agents are in each others area of influence). The density value is increased by decreasing the area in which agents spawn (Section 3.1.3). Results of the test are shown in Figure 5.4. Data is taken at intervals of 0.1, in a range of [0, 1].

Initially, the time it takes a tick to complete is increasing steadily. At a certain critical point (approximately the point when 700 agents on average are visible to any other) a

big drop in performance occurs. At that critical point approximately 1,400,000 query results are generated per tick (2000 agents, each considering 700 agents in their vicinity). A similar test which generates 1,400,000 query results, but where density is kept constant at 0.2, continues to scale somewhat linearly with respect to the number of queries the module generates (Figure 5.2, point with 3500 agents). I suspect this jump in the tick time occurs when the database server can no longer work with a single query mainly in-memory and must use the hard drive to cache parts of it. The number of results returned by a single query influences the performance of the database more significantly than the number of queries invoked.

5.2.4 A Battle

The previous tests are testing the raw power, and the limits, of the DAM system, using the parameters of the Game Space Model. For the next test, I simulate a battle scenario consisting of fighter units (described in Section 4.3.3) in an environment populated with obstacles. I exclude other agent types due to their unoptimized state, and as such, it would not be a good idea to put them in a game. For this simulation, fighter agents have all of their abilities, but the agents are tweaked to have a very large health pool. Because of their nearly infinite health, the number of agents is kept constant throughout the course of the battle. Half of the agents are engaged in combat, and half are navigating through an obstacle field. No agents are kept idle. The agents are initially placed into the world so that the average number of agents which are inside of another agent's influence area is 500. But as the battle unfolds and agents shift around, the fighting agents form clusters, increasing the density of agents in those clusters. I waited for all fighting agents to finish getting in place before I started to collect the results.

The results are shown in Figure 5.5. Like the density graph, the battle scenario graph has a critical point where the performance starts to decrease dramatically. This is due to the increase in the number of agents which are clustered together and engaged in combat. Before this critical point the performance of the DAM scaled linearly with respect to the number of agents in the world. Data range is [1000, 500] with intervals of 500.

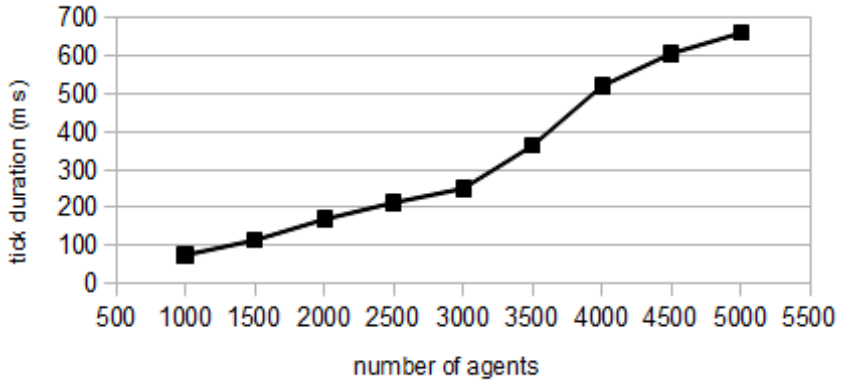

Figure 5.5: A battle scenario.

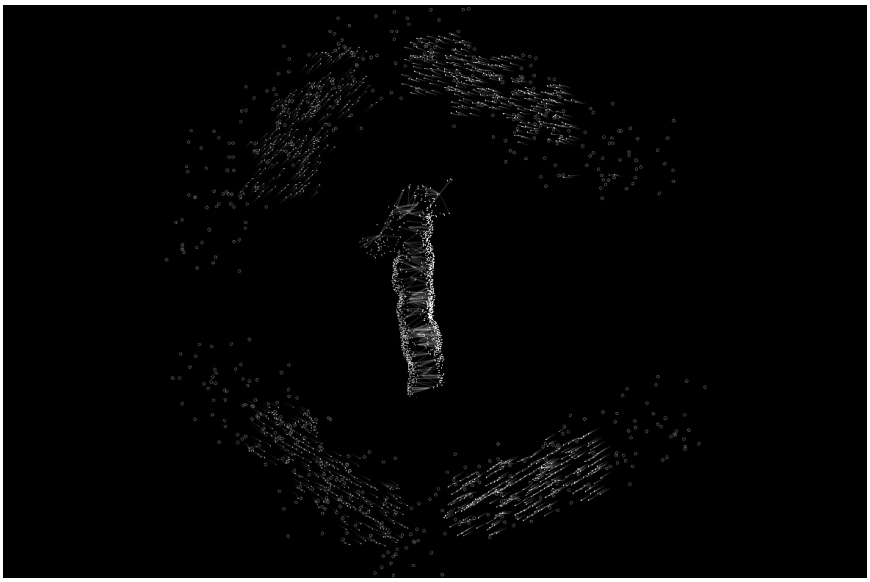

Figure 5.6: A screenshot of a battle.

Figure 5.6 shows an example of a battle between two armies. Units in the middle are forming lines and firing lasers at their enemies (they are trying to push trough). On the outside are scattered obstacles (little bigger, hollow circles) and units navigating through them, avoiding collision. Three thousand units are portrayed in the figure.

5.3 Summary

Data which a database system uses usually resides on the hard-drive, and fetching data from and writing to the hard-drive is costly. For that reason, database systems make heavy use of caching. Query planners are complex systems which try to ascertain the most efficient way to execute queries, which among other things includes minimizing disk I/O by making the most use of cached data.

According to the results of the experiments, the DAM scales fairly linearly with respect to the number of agents in the world, but as long as the number of agents in another agent's influence area is kept constant. Data caching is most likely responsible for the linear increase in time. Increasing the density had a critical point after which the performance of the module decreased dramatically. I suspect that this is so because the database server needs to use the hard drive more often in those cases, slowing the simulation down. An agents cost per second scaled linearly until the frequency of actions surpassed the frequency of the tick.

These experiments were designed to test the raw capabilities of the module. In practice, the movement action has a smaller density value, and the shoot action has a longer cooldown period, resulting greater performance of the system, as demonstrated by the battle scenario.

6 Conclusions and Future Work

6.1 Conclusions

In this thesis, I defined the EDA model (Section 3.2). The EDA model uses database technology to implement efficiently behaviours for thousands of agents, without limiting the expressive power of their artificial intelligence scripts.

An implementation of the model (Chapter 4), titled the DAM, is created as a proof of concept, and to serve as a test bed for new features. Even though the DAM is not taking the advantages of all features the database server provides, the server and its optimizations enable the DAM to process behaviours for thousands of agents each second. The performance of the DAM can be improved further by using other database server features as well, such as parallel query processing. In addition to the increase in performance, taking advantage of the database system features (rather than implementing, and testing, similar functionality from scratch) saves development time and resources, which can be directed elsewhere.

In addition to the EDA model, I introduced the Game Space Model (Section 3.1), a set of game parameters relevant to the game AI. The Game Space Model is used for evaluating the performance of an AI behaviour module, identifying areas for optimization, and quantifying the computational costs for different behavioural scripts. It is used when

designing a game, and when comparing the raw performances of different AI modules. For each of the Game Space Model parameters, a test is designed and administered to evaluate the performance of the DAM. Experiments demonstrate that the model has an improved performance over comparable state-of-the-art solutions (described in Section 2.3), while not imposing artificial limitations to the expressive power.

6.2 Future Work

The work presented in this thesis is a novel approach to AI design and implementation using database technology for game engines, and even though the created module has a satisfactory performance, there is room for improvement of both the model and the module.

The database AI model can be redesigned to run on an external server machine, or on a cluster of server machines, to act as a game server for multi-user games. PostgreSQL can natively distribute server computation across multiple cores on a single server machine, and across several machines, to support many concurrent users.

To increase the performance of the module in situations where both the client and the server are on one machine (single player games), the underlying database system can be changed to a in-memory database system such as SQLite [2]. SQLite is a mature database system which supports all features needed to implement the model. (It supports spatial indexing, triggers, C functions.) Using an in-memory database will eliminate the slow disk I/O operations.

In the current implementation of the EDA, the game engine and the artificial intelligence module communicate with the database sequentially, through a single user connection. Allowing parallel access to the database would improve the messaging system. If the game client has the freedom to access to database at any time, message queuing (and sending the queued messages in bulk later) would not be required. Due to the concurrency feature of a database server, no logical errors will arise from sending messages while a tick is being computed. Another improvement of allowing parallel database access, computations for a new tick could be started immediately after the previous one

ends and the game client starts to read the most recent game state (not waiting for the read to complete).

To compare effectively the performance of different artificial intelligence modules, the Game Space Model needs to be extended. More standardized tests need to be developed for proper comparison of different AI modules. As mentioned previously, the practical performance of an AI module depends on factors other than the module itself, it matters how it is integrated with the game engine and how they communicate. Creating comparable tests for different game engine structures and game types will not be an easy task, as there are many variables influencing the outcome.

Traditional database systems are designed to contain persistent data mostly unaffected by time. As a result, rapidly and constantly updating table rows, and re-indexing them afterwards, is a costly operation, taking up roughly half the reported simulation time. In contrast to traditional database systems, real-time databases are designed to handle workloads whose state is constantly changing [REFERENCE]. More research needs to be done do see to what extent real-time databases would be suitable for this purpose.

6.3 Selling a Database Solution

Databases contain a set of powerful features which can be exploited for doing optimized computation on sets, which are ideal for processing behaviours for multi-agent games. Representing the game state as a database table enables use of built-in single or multi-dimensional indices, speeding up the execution of behaviour scripts.

Databases support writing arbitrary algorithms in C, and seamlessly intertwine C and SQL. (SQL is used when working with the game state, and C is used for performance intensive and complex algorithms.) The game engine can interface with the database and read its game state via SQL queries, allowing the engine to be created in any language that supports working with databases, while the behaviour is still written in C and SQL.

Database systems scale efficiently with an increase in object stored. They automatically offload parts of work onto the hard drive if it cannot all fit in memory.

Database systems are tried, tested, and heavily optimized systems, and their features can be used directly for game development. It would take many man-hours and resources to build a reliable system such as PostgreSQL from scratch. PostgreSQL is a free, fast, open-source database management system that supports all features needed for implementing game logic.

Bibliography

[1] C. M. Procopiuc, P. K. Agarwal, S. Har-Peled. Star-tree: an efficient self-adjusting index for moving points. In *Proceedings of the Fourth Workshop on Algorithm Engineering and Experiments*, 2002

[2] D. R. Hipp. Retreived October, 2012, from SQLite: *http://www.sqlite.org/*

[3] Epic Games. Retreived October, 2012, from Unreal AI: *http://www.unrealengine.com/en/features/artificial_intelligence/*

[4] G. Maggiore, A. Spanò, R. Orsini, G. Costantini, M. Bugliesi, M. Abbadi. Designing Casanova: a language for games. In *Proceedings of the 13th conference on Advances in Computer Games*, 2011

[5] Interactive Data Visualization, Inc. Retreived October, 2012, from SpeedTree toolkit: *http://www.speedtree.com/*

[6] J. Dittrich , L. Blunschi , M. A. Vaz Salles. Indexing Moving Objects using Short-Lived Throwaway Indexes. In *Proceedings of the 11th International Symposium on Advances in Spatial and Temporal Databases*, 2007

[7] L. Arge, K. H. Hinrichs, J. Vahrenhold, J. S. Vitter. Ecient bulk operations on dynamicR-trees. In *Proceedings of the 1st Workshop on Algorithm Engineering and Experimentation*, 1999

[8] L. Arge, M. de Berg, H. J. Haverkort, K. Yi. The Priority R-tree: a practically efficient and worst-case optimal R-tree. In *Proceedings of SIGMOD*, 2004

[9] M. Kapadia , S. Singh , W. Hewlett , P. Faloutsos. Egocentric Affordance Fields in Pedestrian Steering. In *Proceedings of the symposium on Interactive 3D graphics and games*, 2009

[10] Nvidia Corporation. Retreived October, 2012, from PhysX physics engine middleware: *http://www.geforce.com/hardware/technology/physx*

[11] P. A. Fujita, B. Rhead, A. S. Zweig, A. S. Hinrichs, D. Karolchik, M. S. Cline, M. Goldman, G. P. Barber, H. Clawson, A. Coelho, et al.. The UCSC Genome Browser database. In *Nucleic Acids Res. 39*, 2011

[12] R. Göbel. Proceedings of Towards Logarithmic Search Time Complexity for R-Trees. In *Innovations and Advanced Techniques in Computer and Information Sciences and Engineering*, 2007

[13] R. J. Bayardo. Efficiently Mining Long Patterns from Databases. In *Proceedings of the 1998 ACM SIGMOD international conference on Management of data*, p. 85-93, 1998

[14] S. Hwang, K. Kwon, S. K. Cha, B. S. Lee. Performance Evaluation of Main-Memory R-tree Variants. In *Proceedings of International Symposium on Spatial and Temporal Databases*, 2003

[15] S. J. Guy, J. Chhugani, C. Kim, N. Satish, M. Lin, D. Manocha, P. Dubey. ClearPath: Highly Parallel Collision Avoidance for Multi-Agent Simulation. In *Proceedings of the 2009 ACM SIGGRAPH/Eurographics Symposium on Computer Animation*, 2009

[16] U. Dayal , M. Castellanos , A. Simitsis , K. Wilkinson. Data Integration Flows for

Business Intelligence. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009

[17] Unity Technologies. Retreived October, 2012, from Unity Game Engine: *http://unity3d.com/*

[18] Valve Corporation. Retreived December, 2012, from Steam Hardware Survey: *http://store.steampowered.com/hwsurvey*

[19] Valve Corporation. Retreived October, 2012, from Source Engine: *http://source.valvesoftware.com/*

[20] W. White, B. Sowell, J. Gehrke, and A. Demers. Declarative Processing for Computer Games. In *Proceedings of SIGGRAPH SandboxSymposium*, 2008

[21] Wikipedia. Retreived December, 2012, from Valve Corporation: *http://en.wikipedia.org/wiki/Valve_Corporation*

[22] Wikipedia. Retreived October, 2012, from List of Unreal 3 Games: *http://en.wikipedia.org/wiki/List_of_Unreal_Engine_games#Unreal_Engine_3*

[23] Wikipedia. Retreived November, 2012, from SQL: *en.wikipedia.org/wiki/SQL*