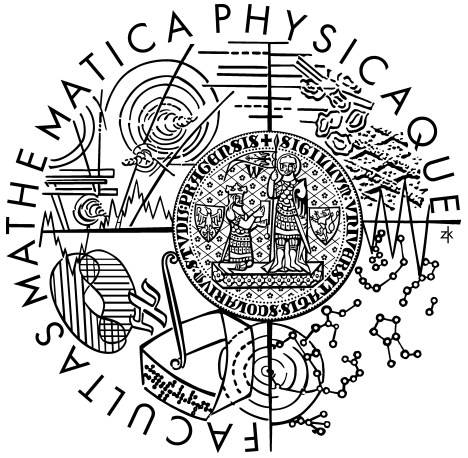
Charles University in Prague
Faculty of Mathematics and Physics

BACHELOR THESIS



Tibor Baláž

Generic realtime strategies

Department of Theoretical Computer Science and Mathematical Logic

Supervisor of the bachelor thesis: RNDr. Filip Dvořák, Ph.D.

Study programme: Computer Science

Specialization: Programming

Prague 2015

I would like to thank my advisor Filip Dvořák for ideas and comments on both my thesis and the project development. His guidance on this project was priceless. Many thanks belong to my friends and family that pushed me further.

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

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

In ..... date ..... signature of the author

Název práce: Generické realtime strategie

Autor: Tibor Baláž

Katedra: Katedra teoretické informatiky a matematické logiky

Vedoucí bakalářské práce: RNDr. Filip Dvořák, Ph.D., Katedra teoretické informatiky a matematické logiky

Abstrakt: Strategické hry jsou oblastí digitální zábavy, která vždy lákala velké publikum jak hráčů, tak výzkumníků, zda-li to už bylo kvůli modelování vojenských strategií, ekonomických principů nebo automatizovaného rozhodování a umělé inteligence.

Bakalářská práce se zaměřuje na vytvoření prostředí, které nabízý možnost efektivního prototypování strategických her s vysokým stupněm abstrakce. Teoretická část práce definuje co je strategická hra, podává úvod do herních enginů a popisuje nový jazyk pro definování strategických her. Praktická část popisuje jak je jazyk interpretovaný do herního enginu a jak pod jeho vlivem probíhá hra.

Klíčová slova: Unity 3D, herní engine, strategické hry

Title: Generic realtime strategies

Author: Tibor Baláž

Department: Department of Theoretical Computer Science and Mathematical Logic

Supervisor: RNDr. Filip Dvořák, Ph.D., Department of Theoretical Computer Science and Mathematical Logic

Abstract: Strategy games are a field of digital entertainment that has always attracted large audiences of players and researchers - be it for modeling military strategy, economical principles or automated decision making and artificial intelligence.

The thesis is focused on developing an environment that allows to efficiently prototype strategic games with a high level of abstraction. The theoretical part of the thesis defines what is a strategy game, gives introduction into the game engines and describes a new language used for defining the games. The practical part describes how is the language interpreted into the game engine and how the execution of the game proceeds.

Keywords: Unity 3D, game engine, strategy games

Contents

Introduction

Popularity of game development has increasing trend in recent years. With the use of new technologies, companies and developers are creating new and more usable game engines, that attract the masses. With these engines, it is much easier to develop your own games. Developer's only worry is about his own imagination and maybe some engine tweaks to perform by his ideas. At the same time great focus is directed towards artificial intelligence that compose behavior of the game world around the player. There are tournaments[2] where game agents, with their own intelligence, compete among themselves. Games have become places of research.

We continue in this trend and we try to push boundaries even further. We try to make developer's work easier. The project allows even beginner to create strategy games extremely fast. But this is not the end. The goal of the project is targeted to researchers. They might create many games in a relatively short time and test their research on a variety of strategy games with many different prospects that they implement. The most outstanding idea is the future interconnection between the project and planner which might handle artificial intelligence in a game.

As we mentioned, we focus our development to strategy games. We present a formal definition for them. And with formal descriptions of strategy game we choose a game engine which presents our graphical output. But before that we introduce the language and language processing software that accepts files written in our new strategy game definition language. Next part of the thesis covers interconnection between theory and components of the actual game. The game processing software. It continually communicates with the game engine and creates reflection of game world shown by the game engine. Then it process actions defined by world definition language and posts changes made by them into the game engine to show them to an user. This is also the part where we introduce basic understanding for usage of game engine with the project. Penultimate chapter covers the details of implementation of initialization and process of the project. This part elucidates project's inner working, at the start of the game as well as during its process, to the reader. In the last chapter we focus on future improvements and possible development of the project.

The contribution of the project is the possibility of creation of a strategy game to a user without necessary knowledge of any programming language. We achieve it with the creation of new language, which is simpler and offers build in support for strategy games. At the same time, language construction is an attempt to prepare a connection between the planner and the game. With the project, generating strategy games would be more automatic and artificial intelligence driven by planner would present usable scientific background for future artificial intelligence research with good graphical output. One more contribution of the project is game creation without unit behavior implementation in the game engine.

Result of author's work is new language for strategy game definition with its own interpreter implemented in C# working with Unity 3D game engine. Interconnection between Unity and interpreter is demonstrated by example of a strategy game.

1. Strategic games

Preview

The chapter present introduction to the game theory used through the project. First part develops a theoretical description of any strategy game world. In the second part we target our focus to basic game definitions.

1.1 Introduction to strategic games

Throughout this work we will use and reference the term strategy video game. In short, usually also referred as the strategy game. For gaming community it is common term, but exact definition is frequently not precise or does not exist. In this article, we are using the term strategy game to describe set derived from game theory and in this context, we use it as a basis for our work. In the case of game theory, we have in mind latter:

Game theory is science discipline, within its scope are defined variety of concepts. Among others there are different kinds of games. For example games of perfect information[10]. Roger B. Myerson states in his book[1] the following.

> Game theory can be defined as the study of mathematical models of conflict and cooperation between intelligent rational decision-makers. Game theory provides general mathematical techniques for analyzing situations in which two or more individuals make decisions that will influence one another's welfare. As such, game theory offers insight of fundamental importance for scholars in all branches of the social sciences, as well as for practical decision-makers.

Strategy video game is a subset of a set defined by game theory which focus is targeted to long deliberate decisions. Individuals think many moves ahead and try to take into account as many environment variables as possible to positively influence their chances to achieve some of winning goal. Usual signs of strategy games is the control over a greater amount of units. As is the game focus redirected to making more sophisticated and time consuming decisions on a global scale, actions of the units are simplified and more automated for the sake of maintaining reasonable game control. Also, as the term video suggests, video game restricts this subset of games played on computers, tablets, consoles and other digital devices that allows graphical interface.

1.2 Common features of strategy games

Following on, this section is devoted to general or common features of strategy games. This part is crucial for dynamic strategy game creation program, because we target these similarities to ease their usage in developing any strategy game. Only as a reminder, we are creating definitions of RTS, converting them into a

grammar that allows description of game world and its behavior. Then creating interpreter for the grammar's language and run simulation with the interpreter. In other words, we play a game.

While we are defining similarities we do so with a task to create the right level of abstraction. It means, we want to describe the game world and events taking place there, but we do not want to make too much detailed description. It would negatively affect language, based on the definitions, created later on. We have in mind this for instance. It is possible to take a set of magical units and tell that their attacks will be placed to spell set. And then take into account other units, this time from the army-based units, and tell, that their attacks belong to physical attack set. If we look at those two sets, then it will be apparent that those sets can be united into one group and it would be set of attacks. As our target is to describe as many attacks as possible, then ideal approach is to define the world of strategy game in maximally generalized concept. With this in mind, we can say that even attack is quite specific in cases of strategy games. For example, action covers, except of attack, also movement and building possibilities. Of course the more we generalize the game description, the more is user pressed to specify a process of the action. In case of the attack there was only necessity to describe damage detail, which will be inflicted upon units, maybe possibility to miss the target of attack. When a concept is shifted to the action, the user has to say that action will target someone and express other assumptions and consequences linked to executing the action of the attack. Though it is a small price compared to the case when we did not think of any kind of the attack and user has to make it by himself.

Therefore, when we think about description of game world we can think of two basic things seen there. Those two are object that exists in the game world and actions that objects perform. Of course, reader can argue that there are actions as *snowing*, that are performed by no object, but on some level of abstraction you can imagine, that *weather* is an object and it performs any weather associated action. Another argument might be that object *cloud* performs action *snowing*, but that is usually not happening, but it is possible.

When we have this basic description of the game, we can look into objects a bit deeper. When we do, we came up with the idea, that object are just instances of some types. Those three, actions, types and instances of types, are enough to completely describe the whole game world and interactions in it.

What we do not describe and what we do not need are definitions to cover game space. Border of the game map for example, can be established by an action. It would be as a first to be evaluated move action that check position and moves away from borders. Other approach might be manipulation directly with game engine and setting boundaries there.

We also avoid adding those environment descriptions to the definitions of strategy games due to creating 3D world that can be difficult to describe. For example, defining all positions of objects on game plate with 3D matrix can be costly as for length of description so in structural representation in a program interpreting it. And there is no need for it when game engine has its own position system.

Definition 1.1. *Finite set of variables (properties)* $O = \{o_1, \ldots, o_n\}$*, where the variables can be of type integer, real, boolean, etc.*

When a reader looks at the definition, he will immediately realize, that this is a common definition of the variable with the domain. But for convenience, we mention it as specific definition. More important is an interpretation of the definition. Imagine that you have just two tanks in a game. What would set O look like? It would consist of all variables of both tanks. For example, there would be variable health two times. Once it would be linked with first tank and second health occurrence would be linked to the second tank.

Following on the first definition, we define game world or more precisely game state.

Definition 1.2. *State of the world $s = \{v_1, \ldots, v_n\}$ is an assignment of values to the variables in O. S is the set of all possible states of the world.*

If we interpret a group of variables as an object, then state of the world is state of objects. By defining state of the world, there arise need to shift world from one state to another. This possibility is given by the action.

Definition 1.3. *Action is a transition function $S \rightarrow S$ (in state space it is directed edge, which can be used to move from one state to another - the edge exist only when begin state meet all conditions of transition function).*

In other words, action changes states of objects if they meet the conditions for action execution. For example conditions for an action shoot can be that the shooter and the target are relatively close. What is left is the beginning and the end of the game.

Definition 1.4. *Game start I is some state of the world. Game end G is set of some states of world.*

Usually, game start will be just a few objects, for example main building, builders and some resources. While game end can be achieved by destroying all enemy units or completing a goal. Therefore game start is just one state, but game end has multiple states.

We present a few additional definitions. They are used throughout the thesis and their purpose is transparency improvement and more intuitive game presentation for the user.

Definition 1.5. *Game object or just object is a set of variables such that no two objects share a variable and the union of all objects equals O.*

The definition makes it clear, that game world is set of objects which in reality are variables describing them.

Definition 1.6. *Object type is a classification of object, where each object has exactly one type and if the two objects share a type then they share all variables types (reals, booleans, etc.).*

Definition of object type might not be very intuitive, but the reader should remember, what the definition 1.1 states. With this in mind it can be easily seen that, both objects(tanks) are of the same type consisting of health. Beware, if one tank would consist of more variables, then they would not share the same type. This is easily imagined as a class and its instances from OOP.

2. Language

Preview

In this chapter, we will take a quick look at definitions from Section 1.2 and introduce a grammar, that prescribe language which will be used to describe game objects and actions. This chapter will also include information about grammar processing software and motivation which lead to choosing it.

2.1 Language and processing software

Right now, we have a formal description of general features, which we want to use for creation of any strategy games. Grammar, we create, based on definitions from the first chapter is not the only one grammar available for the game definition. There is also *VGDL* with its own implementation [5]. But we did not used it and created our own grammar. This is mainly due to possibility of planner usage which requires slightly different grammar construct. We would like game strategy definitions and their grammar construct correspond to the ANML [6] like grammar. Also, we do not use *ANML* grammar because we want game engine related tasks incorporated into it. First step toward the *ANML* like grammar was grammar 2 created by RNDr. Filip DvoĽ^{TM}Ă˙k, Ph.D., my supervisor. Though, it still needed game engine related changes. And so this led to creation of new lightweight language that incorporates knowledge from both grammars, adds game engine related tasks and still offers enough expression power to describe any 3D RTS game.

With the decision of new grammar comes new language and need to process it. The processing part is done by parsing tool that has to satisfy two conditions. Firstly, parsing tool has to support language which is also scripting language of a game engine chosen for the project. Secondly, parsing tool has to be able to generate a parser that can build and walk parse tree, allowing user interaction during the walk thru the tree.

Game engine, Unity, supports three languages. They are JavaScript, Boo and C#, while Unity 5 will drop Boo. Therefore, we have chosen C# from the rest of the languages. It is due to previous experience in development and better developer's environment which support also developing projects from Unity and the last but not least, wider variety of parser generators.

In this regard, we have chosen ANTLR as parser generator. It is freeware with long development history and good community base which help during the first steps. And as a first time user, we really appreciated this fact.

2.2 Grammar construct

We presented mathematical definitions in the Chapter 1. And we are using language to present a method of defining those definitions to the users. But it is not the sole purpose of language. Language form a bridge between definitions and

game engine. It covers common features of strategy games and basic graphical output of those definitions.

Language has a syntax that is defined by the grammar. During creation of the grammar, we have based it on the definitions from the Chapter 1. In a game environment, there occur units which, between definitions, correspond to the game objects 1.5. While creating those objects, it is necessary to know their properties. Therefore, there arises a need for types from definitions 1.6. The type is mainly important as a template while creating new units. Lastly, we need something that will express changing of game world. It is represented as action 1.3, including preconditions which represent conditions of action.

As of now, we have types, instances of types (corresponding to the game objects) and actions. But the grammar of the language offers a bit more. Types are sets of variables with the domain, but at the same time they carry information about prefab [7] associated with it. It is property of the game engine, that every type has its graphical representation. It is better to say, that unit has a graphical representation, but all units of the same type have the same graphical representation, therefore this property is associated with types and not with units.

Something similar can be seen in grammar structure of instances. Every instance has its position in the game. We incorporate this information. Thru it every instance sets its position in the game thru the language. There is a difference between instance and game object. Game object has all his properties assigned, but instance variables are left uninitialized. Their assigning is shifted to the action in the grammar structure. This is motivated by the idea ,that many units of the same type will have the same settings of many variables so we set them by an action instead of writing initialization for all of the variables.

Another important addition that is not incorporated neither in *ANML* nor in *VGDL* is action evaluation priority. The need for this addition is due to possibility of multiple action execution. For example, we have two move actions. First one moves object away from right border second actions moves object to the right and we are at the edge of right border. Of course object may move only once per game engine evaluation. Therefore we have perform just one action. For the game interpreter to execute the correct action we have to set it higher priority. Otherwise interpreter might take the wrong one and object would cross the game border which is unacceptable.

Back to game definition and they respective mirrors in the grammar. Action corresponds to the definition 1.3 in its whole range. Action changes state of instances and therefore performs change of game world. Only difference between implementation and definition is that we implement many actions instead of one. Action in grammar has also addition of two sets of effects which are performed upon instances. The first set of effects is immediately executed and the second set is performed after time duration. Action has implemented support of user input in the game engine. It is support for set of click events related to the strategy games.

As of this moment, we know the content of grammar, but we are lacking some structure. We were inspired by *ANML* planning language when we developed language therefore structure is really close to it. The motivation is, of course already mentioned, possibility of interconnecting the language with a planner, where the world description would represent the domain definition of the planner's

language.

The last idea was not tested because at the end, we were just inspired by the planning language *ANML* and we did not adopt the language grammar fully. Therefore our grammar might need some changes to be able to be used. However, changes should not be severe. For example, the structure should preserve only with some delimiter changes.

2.3 Grammar rules

This section is dedicated to introduction of grammar to the reader. We present rules and their production rules with a brief description of their meaning. Grammar rules are written as regular expressions where quoted text is actual text that has to be written and unquoted text represent grammar rules. We picked most important rules and whole grammar can be found at the end as Attachment 1.

- **Rule** type:
  Production rule 'type' NAME 'prefab' NAME ('extends' NAME)? '{' variable*'}'
  Description:
  The rule determines definition of a game object's type. During using of this construct, user will fill name of type, prefab and if it is filled then name of a type it extends of, instead of occurrences of *NAME* in respective order. At the end user will fill all of *variable*s that will occur.

- **Rule variable:**
  Production rule opt=('boolean'|'number') NAME ';'
  Description:
  The rule is definition for variable of boolean or number domain.

- **Rule** instance:
  Production rule 'instance' NAME NAME '(' INT ',' INT ',' INT ')'
  (',' NAME '(' INT ',' INT ',' INT ')')* ';'
  Description:
  The rule defines instance of object. Variable *NAME* represents, in respective order, name of already defined type, name of instance. Last variable *NAME* is reserved for another names of instances of the type. Three *INT* variables carry information about x,y,z axis of instance in the game engine respectively.

- **Rule** action:
  Production rule'action' NAME '(' (NAME NAME (',' NAME NAME)*)?
  ')' '{'
  'duration' '=' expression ';'
  ('priority' '=' (INT'.')*INT ';')?
  'pre' '{' (precondition (precondition)*)? '}'
  'effs' '{' (effect (effect)*)? '}'
  'effe' '{' (effecte (effecte)*)? '}'
  '}'
  Description:

The rule defines action between game objects. Variables *NAME* in the first row stand for name of action, name of parameter's type and name of parameter respectively. Last two occurrences of variable name are reserved for additional action's name of parameter's type and name of parameter. Second line represents duration between start and end effects of action. Third line is definition priority of the action. Priority is expected to be sequence of zero to n even numbers finished by odd number. Dot is used as number delimiter. Moving on, if we want to execute the action, we have to meet its conditions. These conditions are represented on the forth line of action's rule. Sixth and seventh line define effects of the action. The difference between *effs* and *effe* is in evaluation. *Effs* are evaluated as soon as action meets all of its conditions, meanwhile *effe* are evaluated after the meeting all of conditions and elapse of *duration* period.

- **Rule** precondition:
  Production rule id op=('=='|'!='|'<='|'>='|'<'|'>') expression ';'
  Description:
  This rule states that precondition is transformed into boolean equation where on the left side is identifier and on the right side is expression.
  Production rule functionCall
  Description:
  This rule states that precondition can be also function call. Expected return value is boolean.

- **Rule** functionCall:
  Production rule NAME '(' (NAME)? (',' NAME)* ')' ';'
  Description:
  The rule describes function with its name and its parameter's names. Type is not necessary because it is predefined.

- **Rule** effect:
  Production rule id opt=('='|'-='|'+='|'/='|'*='|'%=') expression ';'
  Description:
  This rule states that effect is transformed into boolean or number equation where on the left side is identifier and on the right side is expression.
  Production rule functionCall
  This rule states that effect can be also function call. There is no expected return value.
  Description:

- **Rule** expression:
  Production rule expression opt='*'|'/'|'%') expression
  Description:
  The rule describes expression as mathematical binary operation. This definition is separated from subtraction and addition because of higher priority.
  Production rule expression opt='+'|'-') expression
  Description:
  The rule describes expression as mathematical binary operation. This definition is separated from multiplication, division and modulation because of

lower priority.
Production rule id
Description:
The rule expression can be transformed to the identifier.
Production rule INT
Description:
The rule expression can be transformed to the non-negative integer.
Production rule '(' expression ')'
Description:
The rule states that we can create priority in expressions by adding brackets.

Throughout the grammar description we frequently used terms *identifier* and *NAME*. Usually, term identifier represents instance's variable written like *instance.variable* or boolean constant *true*. *NAME* represents string without digit at the beginning. Definition of both terms can be found between Attachments 1.

2.4 Not common scripting language

The reader might ask what is the difference between this new language and some scripting language of the game engine? Didn't we create our own scripting language and its interpreter? Where is a benefit?

We can think of the three main differences.

- Firstly, interconnection with planner. It would be quite difficult to connect logic stored in the scripts. But this is not a problem when using language with syntax targeting input syntax of a planner.
- Secondly, the language interpreter has built in support for strategy games. Therefore, user can use simplified methods during handling a game. Allowing him increase his focus elsewhere.
- Thirdly, if the game is published, there is no way to change behavior of objects without the necessity of rebuilding the game. Fortunately, grammar with its own interpreter offers an option to add objects and behavior to those objects. The user is restricted only in the case when he wants to show something that is not shown during game build.

3. Graphical simulation software

3.1 Introduction to game engine

What is a game engine? It is the thing that makes the game go. In this sense it is like a car's engine. The concept of a game engine is quite simple: its purpose is to abstract just the details related to the game development. This includes rendering, physics, and input, so that developers (artists, designers, scripters and other programmers) can focus on the details that make their games one of a kind. Engines offer reusable components that can be used to create games. Loading, displaying, and animating models, collision detection, physics, input, graphical user interfaces, and even portions of a game's artificial intelligence can all be components that make up the engine. In contrast, the content of the game, specific models and textures, the meaning behind object collisions and input, and the way objects interact with the world, are the components that make the actual game.[4]

More formally. A game engine is a system bringing abstraction for the creation of video games. The game engines offer a software tool that developers use to create games. Game engines offer another abstraction between game development and platform dependency. With this abstraction developer can create games without code dependency on any platform. Usually, game engines consist of these tools for game development: rendering engine, a physics engine, sound mixer, scripting possibilities, artificial intelligence, networking, multi-threading and debugging.

3.2 Evolution of game engines

Before the era of game engines, games had to be built from the ground each time. For example, a game for ATARI 2600 console were developed in this manner. Target was the optimal use of display hardware. But it was not the only concern of game engines, memory issues made almost unsatisfiable conditions for data heavy design development that game engines need. Because the newer generation of games would be redesigned to use rapidly advancing hardware complete deletion of code or a little reusable game content would occur frequently.

Some time later in house game engine development became common practice. This engine was prepared to be used with other first or third party software while main technology was kept in house. Companies updated engines as were computers evolving and improvements were needed. LucasARTs's SCUMM is an example of this in house engine and it was core for games by this company around 1980s.

Some engines began as in house engines, but with passing years they also evolved into middleware. This is a case of the Unreal Engine. Later on, because of developing and improving costs of in house engines increased, companies reacted with specialization towards full game engine or middleware for it.

Generation of 3D game engines began with Reality Lab's RenderMorphics, Criterion software's Renderware and Argonaut Software BRender. Microsoft bought Reality Lab and later on, its engine was transformed into Direct 3D.

Game engine as a term began to appear in the middle 1990s. It was mostly linked to the FPS games as Epic Games's Unreal and Id Software's Quake. Both of them build their games with reusable design in mind. It caused team separation to game engine and game content developers. It meant separation of game graphics and logic versus game's physics and platform support.

Separated game engine creation was not only game development advantage. Of course, companies with game engine could develop game sequels faster and much cheaper and that is important in the game industry. But it is not all, another great advantage is a possible revenue from selling a license. This approach is, for example, put to use by Epic and their Unreal engine.

As the game engines evolve, they are more appealing and their usage is becoming wider. Appealing trend is even deepened by implementation higher programming languages as scripting languages. Unity 3D is example of engine with C# as scripting language. The engines became foundation used to develop MMORPGs, strategy games and not just FPS games. But it is not where it ends. War or chemical simulations are other areas where game engines found their usage. Alongside wider areas of usage and new game types, game engines have to cope with increasing range of usable platforms to attract new users or prevent current users from changing the company.

3.3 Game engine abstraction

Game engines can be described as an entity providing visual development tools with a possibility of reusable software component integration. The game engine environment offers elements that allows faster and easier development of games. Those elements are commonly used game properties. For example sound, physics, graphic models and behavior of game objects. All of those components are somehow integrated into the game engine. The development of graphical models can be arranged by graphical editor, behavior of the game objects is usually implemented in the form of scripts attached to game objects.

Even though game engines are middleware software they frequently work with other middleware software. There are many reasons, better performance or engine's complete lack of given functionality. Some engines are built on the idea, that they offer just platform abstraction and span for connection of other game development middleware tools. When we are talking about platform abstraction it is necessary to mention the fact, that games are no longer restricted to computers and consoles, but they are played on phones and in the web browsers. Therefore, game engines have to also support these platforms.

Sometimes, game engines offer only rendering capabilities. Even though they lack common features of game engines, their focus to rendering is significant. They specialize in rendering wide areas or crowded game worlds. In this case, it is up to the user to define all other functionality needed. Those engines are called rendering engines or 3D engines. Genesis 3D is an example of this type of engine.

Next to rendering capability of the game engine, there is another very important function they provide. It is physics simulation. Game engines usually have their own physics engine that runs on another core and calculates physics events in the game. Physics engines offer calculating of collision detection. Collision detection plays main role in game's physics appearance and during the years there

are multiple methods used to solving them. The difference is caused by different representation of the game object's body.

Usually, game objects are seen in the game engine as two separate shapes. One of these shapes is detailed and visible to the player while second much simpler is invisible. For example fence with many holes is visible shape and block is its simplified version. The purpose of this approach is performance. While the player can see the first one, physics engine sees only the second. In this manner physics engine calculates collision between simplified shapes. Therefore, it makes it impossible to interact with holes in the fence, because they do not exist for the physics engine. Even if some hole would be big enough for your character to go thru, physics engine would not allow you to do so because for him it is block without holes.

3.4 Choice of engine

It is well known, that if we want our project to be useable and have meaning it has to attract users. It might be done by some added value or its simplicity of usage. With these ideas in mind, we direct our choice of the game engine on the next points.

- **Large community** Usually, when we start with new technologies is very hard. Being able to use knowledge of others is a great help. Large communities of developers develop and post tutorials or other educational related materials. Most importantly, if we are stuck with some problem, there is high probability that someone already has encountered the problem. Therefore, we are able to find solution quite quickly.

- **Easy access** In other words, cheap. We were looking for game engine, that is affordable by anyone or nearly anyone. The project is not useable by itself, you need an engine to run it. And purpose of the project is more for an academic circles, beginners or game enthusiasts. All these groups would prefer lower costs in exchange of loosing some features.

- **Application possibilities** Another appealing factor is useability. If we support rich variety of platforms, we have greater chance to put our project into application.

- **Intuitive usage** User can begin with no knowledge of game development and he is still able to produce some useable output. This can be mostly covered by GUI of game engine.

Complying with these requirements, we have chosen two, most suitable, engines. Now, we present their comparison and features that led to the final choice.

- **Unity 3D**

  - **Advantages**
    Free and pro version can be used depending on demands.
    Easy integration with 3D editor like Maya or 3ds Max.
    Large asset store offers paid and free assets and game starters.

Active community which can help when you are in peril.
Unity uses Mono as script host. There arise possibility of usage C# for assembly.
Unity support reflection and it allow changes while game is running.
Garbage collector frees user from managing memory issues.
Easy to begin with. Most work is drag and drop.

- **Disadvantages**
  You can silently run out of memory on a 32bit editor.
  Unity focus is not on raw performance but on a new features for pro version.
  Garbage collector has trouble when working on a bigger project.
  Outdated Mono.
  Unity API methods are not thread safe.
  Source code is inaccessible and therefore performance problems are hard to locate.

- **Unreal engine**

  - **Advantages**
    Monthly fee is 19$.
    Purchase of license makes available source code. Even when your license expire you can still keep source codes. You will only lose access to updates.
    As of Unreal 4, Epic let go of UnrealScript and engine brings C++ as scripting language possibility.
    Unreal 4 replaced Kismet by more intuitive Blueprint.
    Easier detection of sources consumption because of access to source code.

  - **Disadvantages**
    The engine is harder to understand that to the Unity.
    Unreal 4 cannot be used to create games for previous generation of consoles.
    Older versions of Unreal used UnrealScript, what creates user to learn new language.
    Memory management is necessary.
    there is not run time separation between editor and code that you are writing. Therefore, code changes requires closing editor module, rebuild and relaunch.

Finally, we compared these two engines and the final decision is that Unity is the game engine of the project. We based our choice on a variety of usability and the fact, that it supports C# as scripting language. Its memory management is not inconsiderable, but mainly it is easy to learn and to use language. And we also developed our project with the free version to demonstrate lightweight demands of the users.

4. Simulation system

Preview

In this chapter we introduce complex progress of creation and run of user created strategic game to a reader. As first, we describe work of software simulating game. Consequently we will present integration and usage of program for strategic game generation and as the last part of the chapter we will take a closer look at some steps that program process to create strategic game and allow its play.

4.1 Simulation

4.1.1 Simulation prerequisites

This section is dedicated to describe first steps during connecting the project to an environment created in Unity. Before we start mentioning what user has to do. We would like to mention that the project includes basic set of Unity's prefabs that represent structures like buildings, war machines, builders, workers and units. Through the grammar, the project gives access to handling of game keyboard's input for actions of the world defining file. But we presume that in case that it was not sufficient, then user will implement any additional feature that he need e.g. mouse scrolling will distance game's main camera.

Now, what developer has to do. First requirement is syntactically and lexically correct world definition.This definition is an input for the project, which creates simulation behavior based in it. Next, we need something that we use to recognize and process the world definition file. For this purpose is used of ANTLR4 runtime 3.5. The assembly takes care of grammar recognition. Based on its basis is developed *action*, *type* and *instance* creation. Last part is project assembly. It handles parsed input from ANTLR4. It overrides base handling and creates game structure. Last requirement is set of prefabs stored in the directory .\Resources but most suitable storage is in this directory structure .\Assets\Prefabs\Resources. Prefab can be described by this sentence from Unity web page. [7]The prefab acts as a template from which you can create new object instances in the scene.

All necessary items are include on data carrier distributed with the project. Data carrier also contains basic example of implementation of project so in case of any problems feel free to inspire yourselves.

It is author's belief, that resources pool is large enough for any beginner's graphical output. If it is not, then user can change or create new graphical resources for his usage. One more thing he has to keep in mind is carrying newly created instance's parameters into the engine. Every single prefab contains scripts that is filled with variables used in the game e.g. *health*. But any new instance's variable that should be used in the engine should be contained in the script. Also, script must refresh this variable in every update. Instance's variables which are not displayed in the simulation might be in the script but it is not necessary because just interpreter will work with them. We recommend

script implementation and not direct usage of the project's variables because of easier debugging of the simulation and game independence on the the project.

Game start is quite easy. We do not only create interpreter, but whole game project with its scene is already created and distributed with the game interpreter. Therefore, what is left for user is to define objects and action in the grammar and run the game engine scene.

Unfortunately, just most basic properties are implemented in the project. This includes object rotation and movement. Basic HUD for health bar process. Any other, more advanced, features must be implemented by game developer. We hope it is a small inconvenience, because what developer has to change are only Unity's scripts.

Game project was constructed by this tutorial [8]. Game project was constantly changed after tutorial end but similarities like pictures etc, can be easily found.

4.1.2 Simulation start

With new language defined by the grammar that incorporates structure from *ANML*, ideas of game description from first Chapter1.2 and a game engine related actions as units position, object models and action prioritization, our main addition, expect for forming language and its grammar, is the language interpreter.

The language interpreter, as name implies, is program designed, in the first phase, to parse game defined by RTSDL. During parsing, interpreter creates mirror of the game show by the game engine and stores it as can be seen on Fig 4.1 on the left side. This internal structure consist of all object defined by game definition file. This is all the interpreter can see. Any object created in other way, then by this, does not exist for interpreter.

Another interpreter's task is action preparation for following game run. During this preparation is action priority tree build. The tree, after the whole file processing, is decomposed and actions are put into single, by priority, ordered array.

When interpreter finishes these tasks he is prepared to continue as a game evaluation software. At this point changes in the game engine are quite significant. The difference can be seen on Fig 4.2, which represents state before game start and Fig 4.3, which represents game state after parsing game definition file and building game structures.

4.1.3 Simulation progress

Unity 3D is working by updating every object in some time intervals. Project works on the same principle. but during update phase, we call every action and try it on every possible combination of parameters.

This is due to the fact that we do not have actions associated with objects, but objects associated with actions. When you have actions associated with objects, then you check each and every object and evaluate every action on that object. It is common method used in game engines. Where you have object tank and user assign scripts to the tank. The scripts represents actions which tank performs. When you have object associated with action, then you evaluate every action

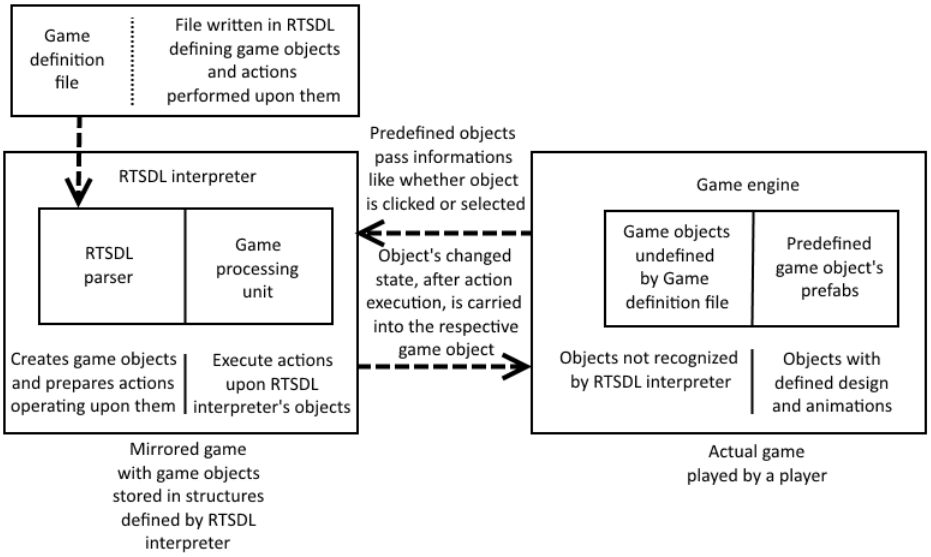


Figure 4.1: Relationship between interpreter and game engine

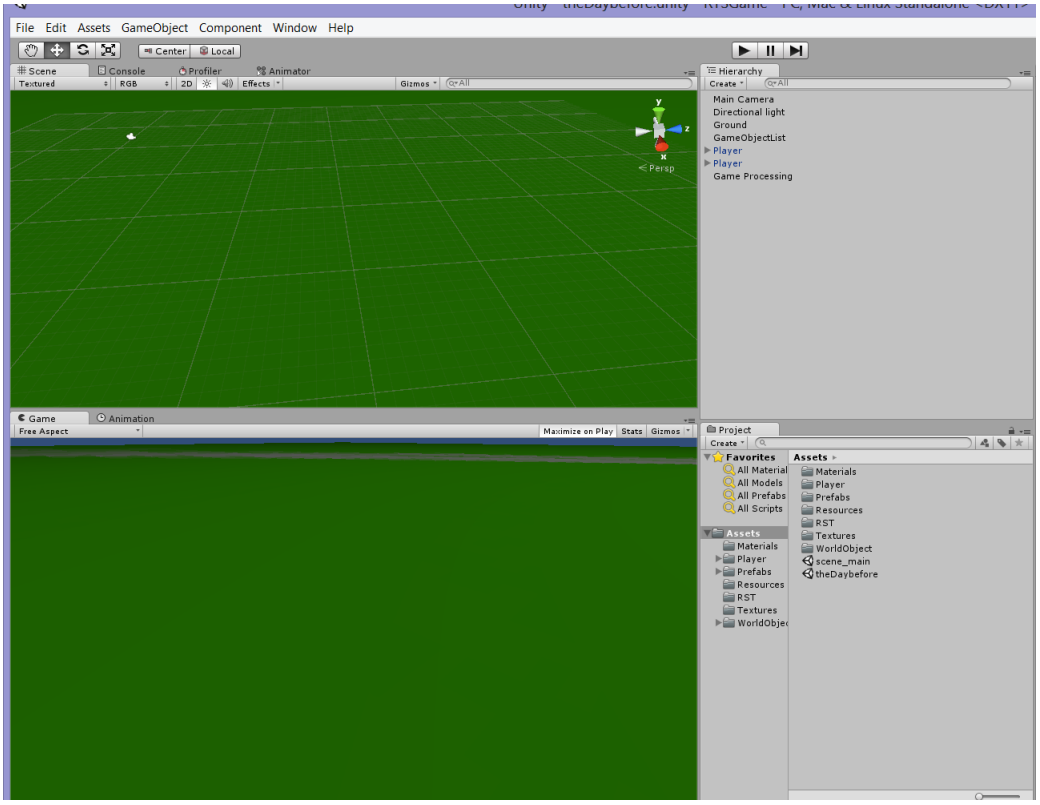


Figure 4.2: Game in egnine befor start

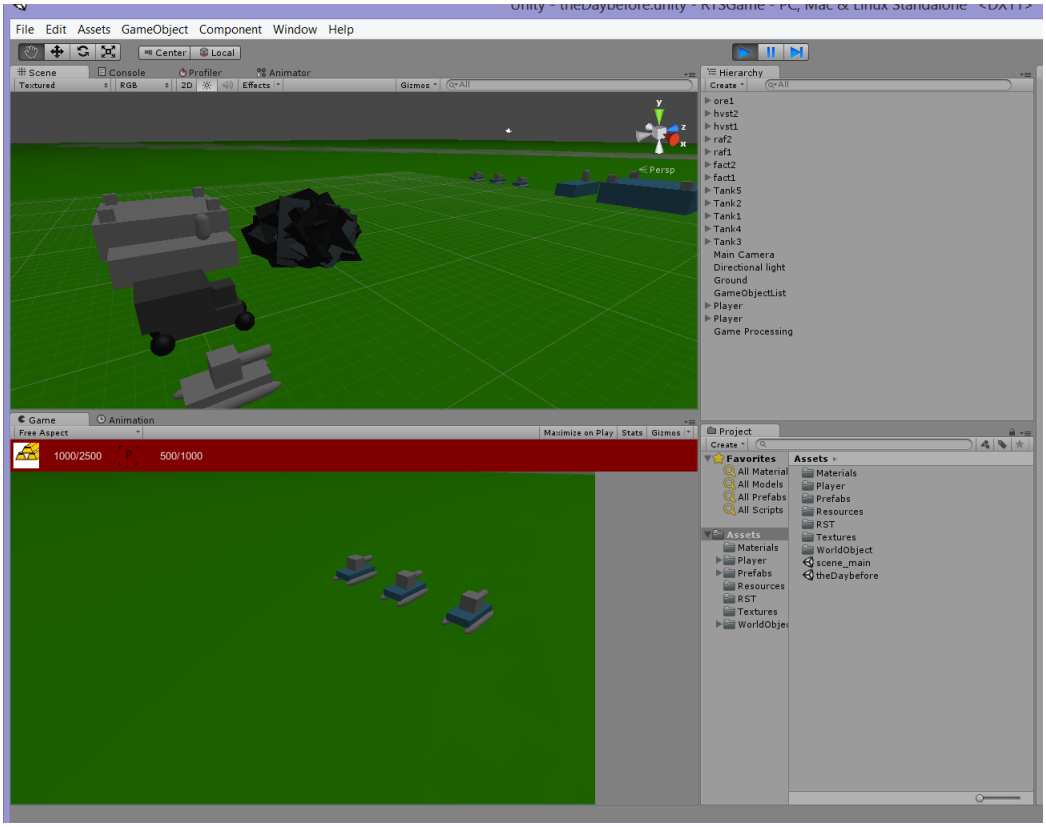


Figure 4.3: Engine evaluation of script and project's action

upon set of parameters assigned to this action. Objects are parameters of actions and evaluation of actions changes states of objects.

This action evaluation happens in the project and as simulation project update it's own representation of game object it is representation defined in the world defining file, user has to read this representation and carry it into his simulation representation.This Fig 4.1 shows communication between game engine and language interpreter. For example, as you can see in the world definition file 3, action *Shoot* works with variable *isShooting* but there is no need to carry this variable into the game engine. In this case, there is only change of variable *health* carried into game engine. It is important to mention, that for standard or basic HUD displays, there is no need for any code to be written, because all of the game prefabs already has their graphical output implemented.

This has it's ups and downs$implementationofadditionalbehaviordisplaying$. Between ups belong possibility to have defined more complex objects in the project and let project actions handle this object. And user will show just a few properties of this object in the simulation. As downs can be counted unclarity. Because we do not know what and when is happening. Another down might be different order of actions calling as an user would like to have. But all this creates acceptable degree of error thanks to short update time of the game engine.

4.1.4 Simulation and project

Whole strategic game behaves just as simulation of *game world*, which we already described in Chapter 1.2. We start in developer defined *game world* W, composed of *objects* $\{O_1, \ldots, O_n\}$. Subsequently simulation proceed step by step. Each step is application of *actions* 1.3 on the game world W. It happens until we reach some of the final states 1.4 of the game. This simulation is performed in game engine, introduced in last Chapter 3.1.

Steps of simulation in the game engine are evaluation of events and physics in time intervals also referred as "ticks" in which *game world* changes from W_i to W_j. In this time, game engine checks every object and call every action on those objects, therefore change *objects* and thru them *game world*. Just one note. This occurs for objects with scripts with redefined one of update methods. We do the same with developer defined *objects*, but engine does not check any of them. Or better said, game engine does not call developer defined *actions*, but calls executive method which is part of the project and this method handles calls of *actions* of those *objects*. It can be seen in Figure 4.4.

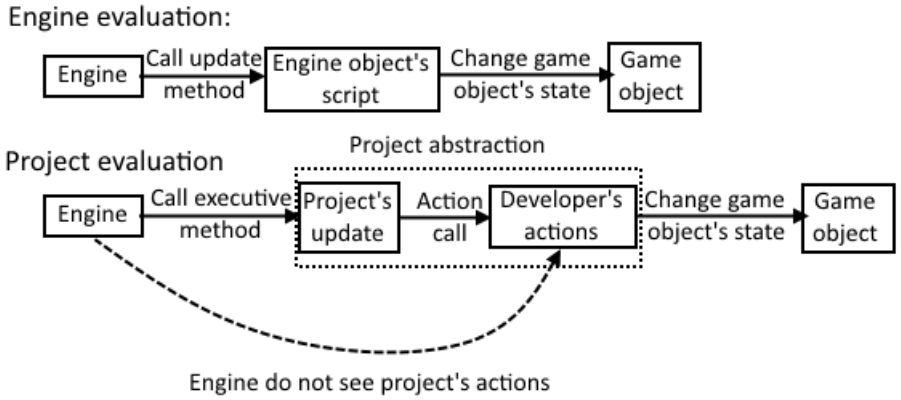


Figure 4.4: Engine evaluation of script and project's action

Also it is good to mention that our *objects* in the game engine are now connected to graphic models used in game by developer. So now not just *object* properties are changed, but also their look in the game, should be changed, depending on change of those properties. For example, when the unit is hit, blood can splash around it.

While we are talking about game engine and simulation of game. We should mention one quite important problem. As we mentioned game engine does not evaluate actions all objects at once, but only one at the time and it goes one by one. We do it in the same way. And there might be stumbling block. We go from one *action* to another and update each *object* that is associated with it. There might happen, that *object* O_1 changed its variables by *action* $\varphi_1(O_1)$, therefore there is O_1' instead of O_1. Later, project will call *action* $\varphi_i(O_1, O_k)$ on some other object O_k, but there is no longer *object* O_1, just *object* O_1'. φ_i will evaluate O_1', if it satisfy conditions. This leads to an error.

Why did it happen? This happened when we allowed user to create multiple action instead of one main action. Unfortunately, with this kind of problem we

have to live on, because creation of just one main function is far too complex for developer. Also code readability would suffer and other flaws might occur.

This problem is called problem of interfering effects of actions and it can be solved

- either by the action evaluation from a snapshot (parallel execution)
- or by defining order of the actions (serial execution)

But even though there are solution, they are too time consuming or excessively developer demanding. Also game engines do not care about this, when they evaluate run of a game. So we will not do it and we will say it is error, but due to small game engine evaluation interval, we presume that objects are just a bit altered and it is acceptable degree of error.

5. Implementation detail

5.1 Game start

Initialization of any game is implemented as processing of the world definition file. Through the definition file we acquire initial state of world with all it's objects and actions that can be done. This leads to repeating the process of reading and parsing world definition file each time we start the game.

Disadvantage of repeated parsing and world recreating is outweighed by easy way to change this world without interference into code or any source structure as long as we do not want to change game's graphical behavior. All user has to change is the world definition file. This also means user can add actions into already functional game without any need for change of script or game's controls. User just has to keep in mind that if he add any new variables or actions into the world definition file, they will not be shown on the graphical output of the game. The project will use it in the way defined, but if it would be variable defining flying, then user has to change graphical behavior of units with the variable in the game engine.

The file reading and processing is done by grammar processing program ANTLR4 introduced in Section 2.1. ANTLR4 reads the world definition file and creates types, instances and actions. Here we present connection between objects of strategic games and those structures.

- **type** - is implementation of object type 1.6 and in game it characterize set of units of the same type. Imagine that you have footman soldiers and they are all different in term that they are on different positions, might have different life, depending upon the fact, that they were in fight and were damaged. But they all have same set of properties defining them. So in term of object type, they all belong to one object type. This is usually used during creation of new units, because in game there is no unit yet, but there exists object type, that predefine object properties and when unit is actually created, then properties are set and real unit appears.

- **instance** - is implementation of game object 1.5 and in game they are actual units or objects like soldiers or buildings. Each soldier is one instance. Anything you create in game is instance. As was, in definition, stated, object is just setup of properties of *object type* and here it is similar in terms that as model for set of properties is used type 5.1.

- **action** - presents implementation of of action 1.3 in game. Action allows developer access to manipulation with objects, which are contained in the game world or eventually their creation. Every movement of unit is action. Shooting between two units is action, buffing, de-buffing from tower or unit are all actions. This is not restricted to unit to unit interaction, but also unit to area interaction, like unit throwing a grenade, is allowed. Of course actions also include creation of units or their destruction.

5.2 Game progress

When the game is started, game world is created. What happens next? World evolution starts and it corresponds to application of action 1.3.

The action is in reality implemented as set of action in game engine. Therefore game world shift from one state to another is evaluation of many partial actions in each tick of the game engine. Actions are called with a parameters. The parameter's n-tuples are generated from set of parameter's types by generator. After generating all of possible n-tuples of parameters of action, they are substituted to the action and it is called.If an action evaluates all of its preconditions to the true subsequently it starts effects evaluation. This is occurrence of the real action from definitions 1.3.

During one tick of the game engine action might be evaluated with many n-tuples of parameters by which every evaluation changes game world bit by bit. Due to this, world state is highly dependent on action evaluation order. Why is that so ? It is because of the way we implemented action.

We can pick from two choices during the action implementation.

- First option is to implement some kind of an world mirror before any action precondition evaluation begin. The mirror would be initial state of the game world at the start of each tick of the game engine and preconditions of actions would be evaluated on this mirror image of the world. Effects of actions would be carried into the actual world.
  - **Advantages**: Initial state of world is the same every time we check preconditions. Thus there is no action that could be done if we would change order of evaluation of actions. This present possibility for easier multithreading in evaluation.
  - **Disadvantages**: Existence of excluding actions. Good example is collision of balls. Excluding action's effects may cause inconsistent state of the game world.
- Second option is evaluation of actions on current state of the game world. Actions are evaluated after effects of action before has changed the game world.
  - **Advantages**: There are no excluding actions. During evaluation of action no other action may interfere. Therefore there is no inconsistent state of the world.
  - **Disadvantages**: Actions that could not be called at the beginning of the engine tick might be called. $\Rightarrow$ Order of action evaluation may produce different game world at the end of engine tick.

We find it sufficient to use second option. Action ordering problem is negligible due to small game engine tick times and it outweigh problem of solving potential world inconsistency.

5.3 Type's internal structure

When do we think about structure of types and instances, what do we need? For sure, we need access to predecessor to be able to acquire his variables. Therefore we have necessity for pointer to parent type. For simplification reasons we create dummy parent type, which is predecessor of all types without explicitly defined predecessor. The dummy predecessor has other advantages we will mention latter. Let's call so far defined structure *type-instance* hierarchy. This structure can be seen on Fig5.1 drawn by black solid arrows.

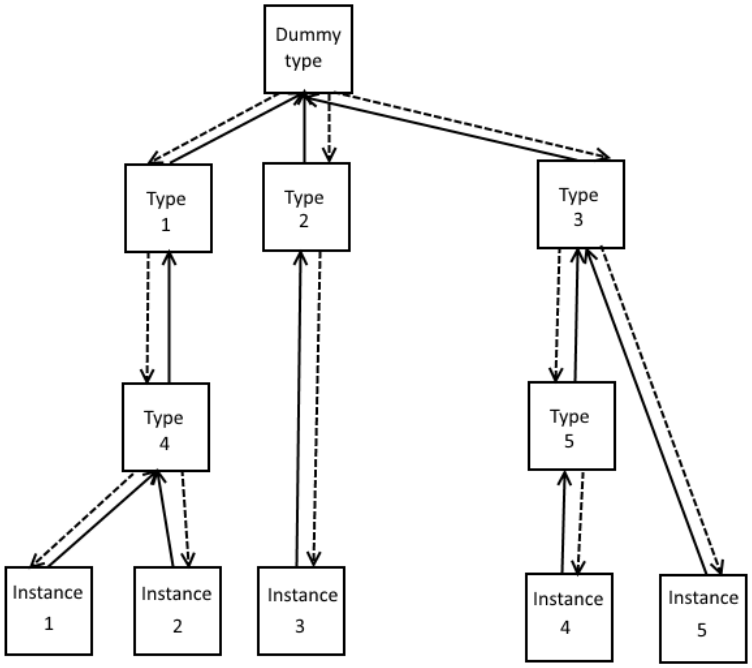


Figure 5.1: Solid line is used for basic inheritance structure. Dotted line display structural addition used for evaluation.

Another addition or change to structure is motivated by need of evaluation of action. Evaluation need to iterate through all of instances of some type. Therefore, we have to look at all types which are successors to this type. Why is that so and also evaluation process is described in more detail in the next section.

With this in mind, we have to add array of pointers to successors. This also implies, that we have to include pointer to instances of types. In action, we are interested in variables of instances, not in types which have variables without defined values. This can be also seen of Fig5.1 drawn by black dashed arrows.

As we speak of actions. They create the need for another structure. Specifically, the idea of direct accessing any type leading to dedicated search structure just for the possibility of direct access of the type. Structure facilitates names of the types and pointers to this specific type, in *type-instance* hierarchy on Fig5.1. We can see this new structure on Fig5.2.

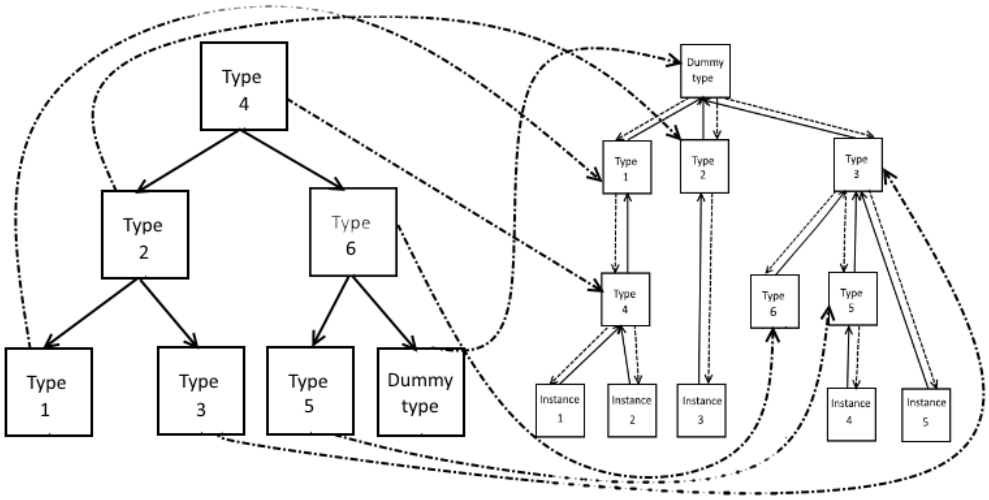


Figure 5.2: Search structure of types with pointer to hierarchy.

As we can see, picture also contains *dummy type*, which allow as to call action which takes any instance as a parameter.

6. Future steps

Preview

This chapter's content covers unimplemented ideas that can be added to the project to expand its evaluation power and speed. There are also parts that make project more user friendly or the project action's evaluation more deterministic.

6.1 Evaluation optimization

It is possible to increase the effectiveness of parameter assigning to the actions. Every action has to try every possible tuple of parameters, but what if, at least one of many preconditions would work only with one parameter from a tuple of parameters? This is chance to filter whole tuple depending just on one parameter.

How can we increase efficiency with this knowledge? In this case, we would take this parameter as first one. We would not care for any other parameter until evaluation of this other parameter is necessary. With this optimization, we have to keep in mind one other detail. That detail is, that it is not enough to find precondition with one parameter. We have to evaluate it as first precondition. So from this point on, it is appropriate to separate preconditions taking just one parameter and group them in the name of it. This can be seen on Table6.1. In the same way as we began, we can continue to create groups for pairs of parameters of preconditions and so on for all other tuples.

Another part of the parameter evaluation, optimization is built on the idea or possibility, that precondition with more usages of the same parameter in one group creates a higher probability of failure. It is because you have to achieve a more complex condition. For this reason, when we encounter groups of preconditions from the first part of optimization. We will sort them by precondition's parameter count in each group. This sort is applied between groups with the same name of parameters. This can be also seen in Table6.1.

6.2 Action precondition formulas

One other upgrade can be extension of action precondition. Right now, an action has to satisfy all of its preconditions that are only connected by a conjunction. In the expressive power of defined language syntax, there is no negation or any other logical operator except conjunction.

So let us say, that we have two preconditions connected with disjunction. Is there no way to express it in our world defining language. Fortunately, there is workaround how users can add disjunction into an action precondition. It can be done be creating the same action as was the action where we want disjunction. Now, we take the first action, the original one, and we put first precondition of disjunction into it. We put second part of the disjunction precondition into a copy of the action.

It is usable solution, but we have to keep in mind a few things. Firstly, you have to be careful if both of preconditions are true, the action is called multiple

WeaponChange(Weapon equip, Weapon invWp)
Preconditions:
equip.owner == invWp.owner;
equip.equiped == true;
equip.shells == 0;
equip.hasShellsToReload == false;
invWp.inInventory == true;
invWp.shells > 0 − invWp.reserveShells − invWp.mag;

Group by parameter

name	**name and occurrence**
parameter: equip	**parameter: invWp**
equip.equiped == true;	**occurrence: 3**
equip.shells == 0;	invWp.shells > 0 −
equip.hasShellsToReload == true;	invWp.reserveShells − invWp.mag;
one parameter: invWp	**parameter: equip**
invWp.inInventory == true;	**occurrence: 1**
invWp.shells > 0 −	equip.equiped == true;
invWp.reserveShells − invWp.mag;	equip.hasShellsToReload == false;
	equip.shells == 0;
parameters: invWp,equip	
equip.owner == invWp.owner;	**parameter: invWp**
	occurrence: 1
	invWp.inInventory == true;
	parameter: invWp,equio
	occurrence: 1,1
	equip.owner == invWp.owner;

Table 6.1: Precondition groups by used parameter

times, e.g. twice in this case. Secondly, if we have more disjunctions in one action, their number will rise linearly or, in the worst possible case, exponentially. It depends on the fact that all the disjunctions are part of just one condition or they are separated in many of them. This corresponds to a conversion of formula to DNF[9] and the creation of the action of each and every clause. See Table 6.2 for example.

This multiplication creates necessity for evaluation of the great number of actions, which takes a significant amount of computation time. This unpleasantness can be easily solved if we add other logical operators. Even if we say easily, reader should keep in mind that solving logical expression is NP-complete problem.

6.3 Create and delete

As the project is, actions can influence world in any way possible, except for two basic situations. Lack of those two situations can be omitted by use of build in user defined functions.

Linear increase	**Exponential increase**
$A \wedge (B \vee C \vee D \vee E \vee F)$	$A \wedge (B \vee C) \wedge (D \vee E \vee F)$
result in	**result in**
$A \wedge B$	$A \wedge B \wedge D$
$A \wedge C$	$A \wedge B \wedge E$
$A \wedge D$	$A \wedge B \wedge F$
$A \wedge E$	$A \wedge C \wedge D$
$A \wedge F$	$A \wedge C \wedge E$
	$A \wedge C \wedge F$

Table 6.2: Action precondition increase during implementation disjunction

But with future development it is possible to make two separate actions whose sole purpose would be creating or deleting objects. Action addObject and removeObject. Another solution to this problem is the possibility of adding a new operator, which will handle these situations, but its implementation would be much more complicated than implementation of two specially dedicated actions.

Each solution has its pros and cons. Separate actions are easier to implement because calling for action is already implemented and only different thing will be effect which will destroy or create an instance. But it can not destroy unit at the right time. What do I mean? There is possibility that the check of preconditions of removeObject on an unit was called before an action that lowered health of the unit. Health was lowered below zero, therefore unit should die this round and even if the preconditions of removeObject would be met. It is too late, because the action was already called in this iteration of the game engine. We

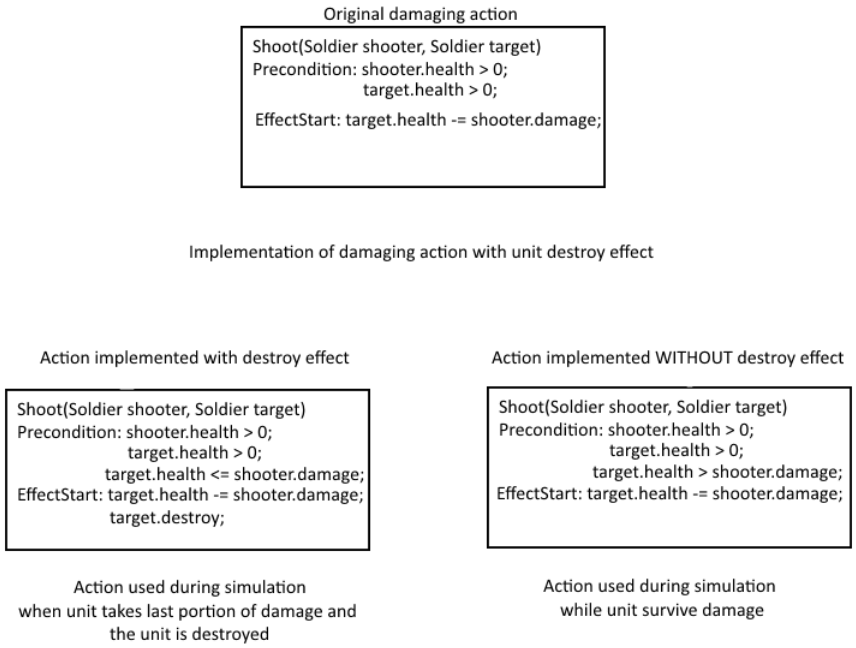


Figure 6.1: Damaging action implementation

can live with this nuisance. The problem arises in the next iteration of the game engine. The unit, which is dead, but removeObject did not destroy it, execute some action. And this is it. This is the problem. There is an action which should not be because the unit is dead or we would like it to be.

The problem can be partially solved if we implement removeObject as the effect of an action and we do not implement it as an action. Then there is the possibility to remove a unit at the right time and it avoids the execution of action after unit's death. The reader might object that this will work only on actions that kill unit. That issue is not solved for damaging actions. It is true, but users may add a precondition to action, which will determine the death of the unit, if this command would be executed. This version of damaging action would be called only if the unit will die after its execution and in any other case, casual damaging action would be called.See Example 6.1.

6.4 Engine callback

There are many things that can be improved during the project or they are drawbacks that can be removed. One of them is delegation of action changes in the game engine.

Right now, the user has to add code to his update method of a game object, that handles the change of the object which has been done by the action. This creates chance for one already mentioned problem. Evaluation in wrong order. Imagine that update method of game object called, but action affecting the game object was not. But it will be called! So object does not see any changes in this iteration of game engine, but there might be after the action is evaluated.

We can prevent it by implementing mechanism of firing events which could be caught by user. The idea is simple. Each and every action that fulfills its preconditions will fire this event. Fired event will communicate with an object and based on this communication the game object will be changed.

We call it communication because it can be implemented in many ways. The first possibility is implementation with direct access to an object and calling of the game object's method dedicated to this event. Another way could be set of all actions executed this engine's iteration and game objects will update themselves after those actions. This requires dedicated storage for actions and their priority execution, but it is a small price to pay, for execution of right order.

One last thing we should remind is that action might have multiple parameters so any implementation of communication between a game engine and project should be able to handle all of the parameters of action.

Conclusion

We created the new language improving development of the strategy games. Its core structure was based on ANML language bringing the possibility of usage with a planner. For its usage, we successfully implemented assembly which represents language interpreter and game progressing software. The assembly target was a game engine that we chose after obtaining knowledge about term game engines and their thorough comparison.

The most challenging task was interconnection between game engine controls associated with strategy game and their project interpretation. At the beginning, we believed that project will handle all click events on any game object, therefore user does not have to implement their events handling in his game. But during implementation of this approach we discovered that cost profit ratio is too high and we dropped the idea. This led to shift the handling of basic controls to the game engine. Trough this approach game engine propagates click events to the project.

But this implementation has benefit. It is in separation of strategy game controls implementation in game engine from its interpretation in the project. In other words we let the user implement game controls as he pleases as long as he propagate game control events into the project. Or, more importantly, game event logic does not have to be implemented by the same person as game units logic is.

In the introduction we mentioned project with its own language replacing knowledge need of any programming language. This target was fulfilled only partially. Due to, already mentioned, project missing click event handling. User still needs language knowledge for implementation of game control logic. To solve this con, we developed environment that handles all of the strategy game basics needed for playing. It also serves as a presentation of the project and as a base foundation of user's future development.

One more target of the project was game units creation and their behavior implementation without usage of game engine. We have completed this task thanks to predefined game environment that we created. User can use language to define units and their behavior and the interpreter will create units at the specific positions in the game and interpreter's action evaluation will handle game units interaction. All of this is done without any interference with the game engine.

Bibliography

[1] Mayerson, Roger B. *Game theory: analysis of conflict*. 1. edition. USA: Harvard University Press, 1997. ISBN 0-674-34116-3.

[2] *Student StarCraft AI Tournament* [online] Available from: `http://www.sscaitournament.com/`

[3] Dvořák, Filip. *Game grammar based on ANML* [online] Available from: see attachment 2

[4] *Ward* Jeff. [online] Available from: `http://www.gamecareerguide.com/features/529/what_is_a_game_.php`

[5] Schaul, Tom. A Video Game Description Language for Model-based or Interactive Learning [online] Available from: `http://people.idsia.ch/~tom/publications/pyvgdl.pdf`

[6] Smith, David E.,William Cushing. The ANML Language [online] Available from: `http://ti.arc.nasa.gov/m/pub-archive/1423h/1423%20%28Smith,%20D%29.pdf`

[7] *Unity Technologies: Unity manual*. [online] Unity Technologies: ©2015 [cit. 29.4.2015]. Available from: `http://docs.unity3d.com/Manual/Prefabs.html`

[8] *Building a real-time strategy game in Unity 4.1 using C# scripting* [online] Available from: `http://stormtek.geek.nz/rts_tutorial/`

[9] Weisstein, Eric W. *Disjunctive Normal Form*. MathWorld A Wolfram Web Resource. `http://mathworld.wolfram.com/DisjunctiveNormalForm.html`

[10] Mycielski, Jan *Handbook of Game Theory*. [online] Volume 1. Elsevier Science Publishers B.V., 1992. Available from: `http://www.math.upenn.edu/~ted/210F10/References/GameTheory/Chapter3copy.pdf`

List of Tables

List of Figures

List of Abbreviations

OOP	Object Oriented Programing
ANTLR	ANother Tool for Language Recognition
ANML	Action Notation Modeling Language
VGDL	Visual Game Description Language
RTS	Real Time Strategy
RTSDL	Real Time Strategy Description Language
DNF	Disjunctive Normal Form
AI	Artificial intelligence
FPS	First Person Shooter
MMORPG	Massively Multiplayer Online Role Playing Game
GUI	Graphical User Interface
NP	Nondeterministic Polynomial time

Attachments

Attachment 1. This file represent grammar rules for world definition file.

```
grammar GStrat;
@parser::header {#pragma warning disable 3021} /* CSLCompliant warning surpas*/
@lexer::header {#pragma warning disable 3021}
/**
 * At the input we have either a type declaration, declaration of an instance of a particular type
 * or a declaration of an action.
 */
root
    : ( type|instance|action )*
      ;

/**
 * We declare the name of the type and the variables (entities) the type consists of.
 * We use single-inheritence of types
 */

type
    : 'type' NAME 'prefab' NAME ('extends' NAME)? '{' variable * '}' # NewType
      ;

/**
 * We may declare a boolean variable or a number variable - integer (TODO: choose what numbers to use across the system,
 * sticking to floats might be better)
 */

variable
    : opt=('boolean'|'number') NAME ';' #NewVariable
    ;

/**
 * First goes the name of type, then any number of comma-separated instances
 */

instance
    : 'instance' NAME NAME '(' INT ',' INT ',' INT ')' (',' NAME '(' INT ',' INT ',' INT ')')* ';' #NewInstances
    ;

/**
```

```
39  * An action consists of set of preconditions (expressions we
        can evaluate) and effects,
40  * we distinguish between effects that occour at the beginning
        and the end of the action.
41  * The action has duration defined by an expression. The
        parameters of the action are the instances
42  * of types.
43  */
44 action
45         :
46                 'action' NAME '(' (NAME NAME (',' NAME NAME)*)
                       ? ')' '{'
47                 'duration' '=' expression ';'
48                 ('priority' '=' (INT'.')*INT ';')?
49                 'pre' '{' (precondition (precondition)*)? '}'
50                 'effs' '{' ( effect ( effect )*)?  '}'
51                 'effe' '{' ( effecte ( effecte )*)? '}'
52                 '}'
53         ;
54 effecte
55         : effect
56         ;
57
58 precondition
59     : id op=('=='|'!='|'<='|'>='|'<'|'>') expression ';' #
           PrecondExpr
60         | functionCall #CallFnPrecond
61     ;
62
63 /**
64  * Calling internal function with some parameters
65  */
66
67 functionCall
68     : NAME '(' (NAME)? (',' NAME)* ')' ';'
69     ;
70
71 effect
72     : id opt=('='|'-='|'+='|'/='|'*='|'%=') expression ';' #
           AssignExpr
73         | functionCall #CallFn
74     ;
75 /**
76 * Calling user defined action with some parameters.
77 * Only actions that operates on types can be called.
78
79 actionCall
80         : 'call_act' NAME'.'NAME '(' ((id|INT) (',' id|INT)*)?
                ')' ';'
81         ;*/
82
```

```
83 expression
84     : expression opt=('*'|'/'|'%') expression #MulDivMod
85         | expression opt=('+'|'-') expression #AddSub
86     | id #ident
87     | INT #int
88     | '(' expression ')' #parenth
89     ;
90 id
91         : NAME('.'NAME)?
92         ;
93 BOOL_ID : 'boolean';
94 NUMBER_ID : 'number';
95 NAME : [_a-zA-Z][_a-zA-Z0-9]*;
96 INT : '-'?[0-9] | '-'?[1-9][0-9]+;
97 MUL : '*';
98 DIV : '/';
99 MOD : '%';
100 ADD : '+';
101 SUB : '-';
102 EQUAL : '==';
103 NOT_EQUAL : '!=';
104 LESS_THEN : '<';
105 LESS__OR_EQ : '<=';
106 MORE_THEN : '>';
107 MORE_OR_EQ : '>=';
108 ASSIGN : '=';
109 ASSIGN_MINUS : '-=';
110 ASSIGN_ADD : '+=';
111 ASSIGN_DIV : '/=';
112 ASSIGN_MUL : '*=';
113 ASSIGN_MOD : '%=';
114 WS  :   ( ' '
115         | '\t'
116         | '\r'
117         | '\n'
118         ) -> channel(HIDDEN)
119     ;
```

Attachment 2. This file represent original grammar rules created by RNDr. Filip Dvořák, Ph.D.

```
1 /*
2  * Author:  Filip Dvořák <filip.dvorak@runbox.com>
3  *
4  * Copyright (c) 2014 Filip Dvořák <filip.dvorak@runbox.com>,
       all rights reserved
5  *
6  * Publishing, providing further or using this program is
       prohibited
7  * without previous written permission of the author.
       Publishing or providing
```

```
8  * further the contents of this file is prohibited without a
       previous written
9  * permission of the author.
10 */
11
12 /**
13 * Examples:
14 *  type Soldier {
15 *    number health;
16 *    number damage;
17 *  }
18 *  type SoldierWithJetPack extends Soldier {
19 *    boolean flying;
20 *  }
21 *
22 *  instance Rambo1, Rambo2;
23 *
24 *  action Init(){
25 *    pre:{}
26 *    effs{
27 *    Rambo1.flying = false;
28 *    Rambo1.health = 100;
29 *    Rambo1.damage = 6;
30 *    Rambo2.flying = false;
31 *    Rambo2.health = 100;
32 *    Rambo2.damage = 8;
33 *    }
34 *    effe{}
35 *  }
36 *
37 *  action Shoot(Soldier shooter, Soldier target){
38 *    duration = 10;
39 *    pre:{ close_range(shooter, target),
40 *          target.flying == false
41 *        }
42 *    effs:{}
43 *    effe:{ target.health -= shooter.damage }
44 *  }
45 *
46 *
47 *
48 *
49 *
50 *
51 *
52 *
53 */
54
55
56
57 grammar GStrat;
```

```
58
59 /**
60  * At the input we have either a type declaration, declaration
        of an instance of a particular type
61  * or a declaration of an action.
62  */
63 root
64     :
65         (type|instance|action)*
66     ;
67
68 /**
69  * We declare the name of the type and the variables (entities
        ) the type consists of.
70  * We use single-inheritence of types
71  */
72
73 type
74     :
75         'type' ID ('extends' ID) '{' variable* '}'
76     ;
77
78 /**
79  * We may declare a boolean variable or a number variable -
        integer (TODO: choose what numbers to use across the
        system,
80  * sticking to floats might be better)
81  */
82
83 variable
84     :
85         ('boolean'|'number') ID ';'
86     ;
87
88 /**
89  * First goes the name of type, then any number of comma-
        separated instances
90  */
91
92 instance
93     :
94         'instance' ID ID (',' ID)+ ';'
95     ;
96
97 /**
98  * An action consists of set of preconditions (expressions we
        can evaluate) and effects,
99  * we distinguish between effects that occour at the beginning
        and the end of the action.
100 * The action has duration defined by an expression. The
        parameters of the action are the instances
```

```
101  * of types.
102  */
103 action
104     :
105         'action' ID '(' (ID ID)? (',' ID ID) ')'
106         'duration' '=' expression
107         'pre' '{' precondition* '}'
108         'effs' '{' effect* '}'
109         'effe' '{' effect* '}'
110     ;
111
112 precondition
113     :
114         ID OPERATOR_COMPARE expression|
115         functionCall
116     ;
117
118 /**
119  * Calling internal function with some parameters
120  */
121
122 functionCall
123     :
124         ID '(' (ID|INT)? (',' ID|INT)* ')'
125     ;
126
127 effect
128     :
129         ID OPERATOR_ASSIGN expression ';'|
130         functionCall
131     ;
132
133 expression
134     :
135         expression ('*'|'/'|'+'|'-') expression|
136         ID|
137         INT|
138         '(' expression ')'
139     ;
140
141
142 OPERATOR_COMPARE
143     :
144         '=='|'>='|'<='|'<'|'>'
145     ;
146
147 OPERATOR_ASSIGN
148     :
149         '='|'-='|'+='|'/='|'*='
150     ;
151
```

```
ID: [a–zA–Z.]+ ;
INT : [0–9]+ ;
```

Attachment 3. World definition file is user defined file which describes game behavior. This behavior includes positioning of game object and their interaction with each other thru the actions.

```
type Unit prefab Unit {
    number health;
    number damage;
    number playerColor;
    boolean inicialized;
    boolean isShooting;
  }
type Tank prefab Tank extends Unit {
    boolean flying;
    boolean resurected;
  }
type Worker prefab Worker{
 number health;
 boolean inicialized;
}
   instance Tank Tank3(10,0,25), Tank4(10,0,20);
   instance Worker workerik(10,0,30);
action Init(Tank soldier){
    duration = 0;
    pre{soldier.inicialized == false; }
    effs{
    Tank3.playerColor = 1;
    Tank4.playerColor = 2;
    soldier.flying = false;
    soldier.health = 100;
    soldier.damage = 3;
    soldier.inicialized = true;
    soldier.isShooting = false;
    }
    effe{}
  }
action Init(Worker soldier){
      duration = 0;
    pre{soldier.inicialized == false; }
    effs{
    soldier.health = 100;
    soldier.inicialized = true;
    }
    effe{}
}

  action Shoot(Tank shooter, Unit target){
    duration = 300;
    pre{ shooter != target;
```

```
45          shooter.health > 0;
46          target.health > 0;
47          close_range(shooter, target);
48          target.flying == false;
49                              shooter.isShooting == false;
50
51          }
52      effs{shooter.isShooting = true;target.health -= shooter.
            damage;}
53      effe{
54                          shooter.isShooting = false; }
55  }
56  action Reincarnate(Tank unit){
57  duration =5000;
58  pre{unit.health <=0;unit.resurected == false;}
59   effs{unit.resurected = true;}
60    effe{  unit.health =100; unit.resurected = false;
61                          }
62  }
```