Strategies for real-time video games

Tom Hastjarjanto

March 26, 2013

Abstract

Game developers spend a large portion of their time developing and tweaking the artificial intelligence in video games. Problems related to productivity in the development of AI have been solved using various modeling techniques in the field of AI, language design and easier to use editors. Using a domain specific language to assist in describing AI can increase productivity in this area. In addition to this, game developers can be relieved from irrelevant tasks such as worrying about performance, correctness of the implementation, memory management and optimization data structures and focus on the high level description of the game play. In this thesis, we focus on real-time video games and we investigate the development of a domain-specific language containing the necessary elements to describe and execute strategies to achieve goals in a real-time video game. We develop a domain-specific language to express strategies for computer controlled actors using techniques commonly found in embedded domain-specific languages, and in particular embedded domain-specific languages in Haskell. To demonstrate this language we have developed a prototype of a real-time strategy game that uses strategies implemented using the domain-specific language developed in this thesis.

Contents

1 Introduction

Recently, academia have started looking into ways to formalize various aspects of computer games to have a better understanding of the characteristics of modern computer games. Formalizing games was mostly done by trying to determine elements, patterns and theories commonly found in computer games [2]. Parts of the description of a game can be found in game design documents or in the actual implementation of the game. Currently, if there is an update in the game design document, we need to update the implementation as well. Many AAA games [1] are implemented in a scripting language on top of a game engine. Part of the implementation is the behavior of computer controlled actors. These actors behave in an intelligent way and interact with the game and player. This thesis discusses how we can conveniently describe the behavior of computer controlled actors in games. We can identify several issues related to the development of computer games which relate to implementation ofthe behavior of computer controlled actors.

- Productivity - Players expect better and better games over time. These expectations are sometimes hard to meet due to rising costs and limited resources. Productivity gains in speed of implementation and ease of modification and flexibility are therefore preferred over performance [5] [20] [16].
- High level descriptions - Because game play specific code should tailored to the task of creating fun game play, game developers should be relieved from irrelevant tasks such as worrying about performance, memory management and optimization data structures and focus on the high level description of the game play. Limiting the scope of the language to these tasks also reduces the number of bugs that a game designer can create [20] [16].
- Declarative languages - Current scripting languages or DSL's are often based on imperative programming languages. These languages rely on specific development patterns and clever use of control structures to create performing code. Declarative languages make it easier to perform analyses and parallelize parts of the code [20].

In our thesis we will first discuss the advantages, approach and related work in the field on DSLs and AI, then we will discuss our problem domain and current solutions. The main part of this thesis develops a DSL to express the strategies for computer controlled actors using techniques commonly found in embedded DSLs, and in particular embedded DSLs in Haskell.

[1] The game industry speaks of an AAA game when the production costs of a game are among the highest in the industry. Such games are typically produced by large game companies. The term originated in the movie industry

2 Problem description

2.1 Research question

Problems related to productivity in the development of games have been solved using various techniques in the field of AI, language design and easier to use editors. Game developers spend a large portion of their time developing and tweaking the AI. Using a DSL to assist in this task can increase productivity in this area. Describing a domain in a domains-specific language offers several benefits. Benefits of using a DSL are verification of the proper implementation of the strategies through type checking in compilers, automatic generation of source code and documentation and alternative ways to program interactive actors that interact with the game, control non-playing characters or emulate human players in multiplayer games. We will define a DSL to model game strategies to achieve objectives in a game. Using strategic programming[12] we can implement the AI of computer controlled entities. To investigate a DSL in video games we look at the following research question:

Is a DSL an effective way to describe and execute strategies to achieve goals in a real-time video game?

2.2 Research method

We will investigate if we can develop a DSL and interpreter for this DSL to describe strategies and execute the result of each step in a strategy in a game to progress towards completing the specific goal for which the strategy is written. We will do this by developing a framework inspired by the work of Heeren, et al. [11] and apply it to a prototype of a real-time video game written in Haskell.

First we will look into the concept of DSLs and summarize the benefits and the approach to developing a suitable DSL. Then we will look at what we should take into account to describe a game. Once we have a description of the game and its entities we can formulate strategies using these elements to play the game. To formulate these strategies we will first look at traditional approaches in AI, then look at the concept of strategic programming and other approaches to creating combinator based DSL's and finally design and implement a DSL for games, which can describe the game and implement playing strategies.

3 Domain-specific languages

We look at several approaches to modeling decision making in AI. In the next section we will explore a couple of concepts from functional programming which can be used to define a new approach to modeling decision making AI. Using combinators we can formulate a strategy involving a sequence of behaviors or actions to be executed based on the observations of the current state.

3.1 General Domain-Specific Languages

DSLs are programming languages which are designed to solve problems in a specific domain. DSLs are often more expressive and easy to use but less general than general-purpose languages. There are several benefits to designing a DSL for a specific problem domain rather than using a general-purpose language to write software for the domain. Mernik, et al [15] highlight the following advantages of a DSL.

- **Domain-specific notations or operators** – Some domains have specialized notations for rules or data which should be processed by programs written in the DSL. A DSL can encode these constructs to provide a more natural way to encode the specific domain in a program.
- **Domain-specific constructs and abstractions** – A DSL can abstract from low-level details to give a high-level description of the program.
- **Analysis, verification, optimization, parallelization, and transformation of the source code** – Programs written using the constructs and functions of a DSL automatically benefit from the limited scope of a DSL which makes it easier to analyze and optimize.
- **Reuse** – Constructs from a DSL can be reused by different applications within the problem domain which offers a productivity increase for programmers working with the language.

Embedded DSLs are DSLs defined within a general-purpose language. Hudak [18] outlines some benefits of using Haskell for EDSLs. Often it is too time consuming and difficult to implement a programming language from scratch. We can benefit from the work done in a general-purpose programming language by using its features to define a language inside the programming-language. This way the focus is the interpretation of the semantics of the embedded language rather than the implementation details of a language in general.

3.2 Domain-specific languages in games

Many games are already partly written in a DSL. Many game engines contain a scripting component tailored to producing game play specific code. Often these languages are embedded general purpose languages which interact with the low level game engine. Gavin [16] from Naughty Dog specifically wrote a DSL called "Game Oriented Object Lisp" to increase productivity in writing code for object control, in particular the flow of control and state. The motivation

was that game play specific code needs to be rewritten many times during development to achieve appealing game play for the player and mainstream languages are not particularly suited for that. Game Oriented Object Lisp tried to address this issue by making it easier to write code that still is performant but is easier to use for the level abstraction of game play code. Using this DSL, the authors benefit from a way to quickly prototype new code while still keeping maintainable source code. Another motivation to build a DSL for games is that they can relieve game designers from concern for unnecessary focus points such as performance. A DSL should ideally relieve game designers from any other detail than creating fun game play.

A general approach to developing a DSL for video games has been described by Furtado, et al. [7]. First a clear vision is needed about what particular subset of games or part of games should be part of the DSL has to be established. After defining the scope of the DSL, several games are analyzed to identify the features of the particular subset for which the DSL should be written. Based on the features of the analyzed games, the common features can be extracted and encapsulated in a DSL.

4 Video game development

4.1 Implementation of real-time video games

To implement and integrate game strategies in video games we will first summarize how video games are implemented. Since the implementation of a game depends on its type, we limit our scope to real-time video games. These games run with continuous updates, unlike turn-based games, such as chess, which may pause between moves.

In this section we describe the concurrent nature of real-time video games as well as the game play code and the game engine. We will also describe the most important components of the game engine such as graphics, physics, user input. Other components which are part of computer games are resource loading, audio and networking, but they are less relevant for our work and will not be discussed.

4.1.1 Concurrent nature of real-time video games

Many aspects of a game behave *concurrently*. A single state is used to keep track of the game. In real life, multiple participants and objects of a game behave in parallel; they can make decisions and execute them in parallel. Historically, video games have been running sequentially on single core machines. Because CPUs can only perform a finite amount of calculations per second, real-time video games are limited in how much they can simulate. To achieve a perception of a fluently running real-time game, games update the state of the game multiple times per second although the visual rendering of the state is what makes the game appear smooth and fluent. These updates to the state of the game are done in a game loop, which interacts with all the components. The game loop is a function that processes the updates for the state in a loop or interval. Each time these updates are processed we talk about an iteration.

4.1.2 Game engines and video games

Video games are time consuming and costly to develop, especially for game development for games with a large asset- and code base. To minimize the time to develop a video game, a collection of reusable components is developed or used. This collection is often called a *game engine*. Components which are part of the game engine, are the rendering engine, physics engine, networking library, threading library, memory management, input handling, model loading and scripting environment. When developers implement a video game, they will write code that interacts with this game engine using the API provided by the engine.

Creating the core infrastructure of the game as part of an engine is work that can evolve over the years. The limitation of this, is that it sometimes is difficult to integrate with other components written in other programming languages or paradigms such as functional programming. These components are often written in a low level language such as C, or C++ or assembly and the interface to these components are of an imperative nature.

4.1.3 Game play code

As every video game is different, there is a lot of code that is not reusable. The code that is specific for a game is often written on top of a game engine. The *game play code* describes the properties and boundaries of the game. For example, the velocity of a bullet, the rules of a game of football and the interaction based on the input of a player. Unlike the development life cycle of a game engine, the game specific code is more often developed from scratch in a limited time frame. It is essential that productivity in creating game specific code is optimized by the availability of tools and features from the game engine.

Another very important part of game specific code is the AI. Most video games contain some form of AI. For example, the players of a football team are controlled by the game. Enemies a player might face in a *first person shooter* act and behave on their own. Many modern computer games contain computer controlled actors which may perform the same actions, and often want to achieve the same goal, as the player. Such actors have access to the same actions and information as human players, but their strategy is implemented by means of a computer program, usually called the AI. The implementation of AI can be seen as a description of how a game can be played by a specific character. Section 4.3 discusses the AI in more detail.

4.1.4 Graphics

To display the current state on the screen, games implement the rendering component on top of a graphics library such as OpenGL or DirectX. Each frame, the rendering component looks at the game state and renders the visible components on screen. OpenGL and DirectX are libraries to communicate with a range of GPU drivers. The API provided acts as an interface to a state machine, which we use to send commands to the GPU drivers. In early OpenGL versions the developer either sends graphic primitives, such as triangles or polygons, or he sends commands to the pipeline which modify the primitives in a certain way. Current versions of OpenGL and DirectX rely on the concept of *shaders* to manipulate the graphics pipeline.

4.1.5 Physics

Many video games simulate a physical world, often simply called *physics*. Common aspects of a physics component are the handling of gravity, velocity and collision to simulate a real-life environment. The component of a game that takes care of physics takes the current state and applies the modifications to all the objects in the game world. For example, the physics component modifies the position of a ball based on the velocity and time difference since the last simulation. Because the physics calculations are based on the time differences between simulations, the accuracy depends on the size of this time difference. A common bug that occurred in old games was a situation where objects traveling with a very high velocity missed a simulation round where they would collide with other objects. This issue is referred to as tunneling. In order to prevent it, it is necessary to complete enough simulation rounds per second such that the objects collisions are captured by the physics component. Because of tun-

neling, we cannot instantaneously apply big state changes ourselves without missing collisions. The physics component directly influences the game state.

4.1.6 Input

Every game relies on input from the user through an input device. The input from a user is mapped to actions which influence the game state. Often they control a character in the game, such as directing movements, velocity and other actions. Input handling is part of the game play code to apply the actions mapped to the user actions to the state.

4.2 Description of the game strategies

A game strategy defines what steps under different circumstances should be taken to complete a particular goal in a game. Traditionally, classical board and other non-video games are described by means of the instructions that are allowed to reach a particular goal. Following these instructions in such a way to win the game is a strategy to play the game successfully. Using natural language for describing game strategy is undesirable, because it is hard to describe a game strategy completely, consistently, and unambiguously in natural language.

Game strategies are described in general purpose programming languages to perform decision making for actors and to respond appropriately to user input and in-game events. Although a game strategy written in a general purpose programming language is a very precise and accurate description, it consists of many implementation details that are irrelevant for a high-level description.

4.3 Artificial intelligence

There are several approaches to implementing the AI of games with computer controlled actors. These depend on the nature of the games as there is no general strategy that would work efficiently for all types of games. For example, the amount of possible states in the game of tic-tac-toe is much smaller than the amount of states in Pacman. For games with limited state it is possible to explore a set of future states using the minimax algorithm[21] which tries to determine the next best step to achieve the highest advantage over the opponent. The alpha-beta pruning algorithm[21] can be used to reduce the amount of states explored. However, this algorithm does not scale up to games which change state multiple times per second as the tree of possible states to explore simply becomes too large or the steps explored have minimal changes over the last state to decide on a meaningful next move.

Most computer games contain a form of AI that responds based on the observations it makes about the current state. In this thesis we are interested in what decisions are taken by the AI rather than how they are performed. A strategy is formed by taking decisions based on the current observations about the state and then considering a couple of possible actions to take. Often these decisions are scripted using simple if-then-else statements, but are modeled using tree or

graph like data structures [14]. We will take a look at some possible approaches to modeling decision making.

4.3.1 Behavior trees

In AI, behaviors are the way an actor interacts with the game. A popular component in video-games is a behavior tree which is used to model decision making to choose a behavior for an agent based on the state of a game. First person shooters such as Halo require actors to work with similar procedures, resources and rules as a player. The AI designers of Halo [2] argue that actors which respond to situations in a variety of ways are seen as intelligent, in which more ways are seen as better and more realistic. In essence the design of the Halo AI had two goals:

- Implement a variety of behaviors, different procedures an actor can take to respond certain situations.
- The way actors choose their behavior based on the analyses of the current situation is what is called behavior integrity. A player perceives it as 'good' when an actor reacts to a situation in the same way as the player would.

To achieve these goals a number of challenges had to be tackled [2].

- **Coherence** - Can the behavior of the actor be adjusted smoothly during state transitions?
- **Transparency** - Can the player make reasonable assumptions about what the actor might do?
- **Mental Bandwidth** - Is the behavior and implementation of the actor comprehensible by a designer?
- **Usability** - How easy is it to customize the behavior of an actor in different scenario's?
- **Variety** - Are the actors able to respond in a variety of ways depending on the instance
- **Variability** - Are the actors able to pick among a selection of applicable responses given a certain situation?

To implement the AI of actors, the AI designers of Halo [2] chose to use a behavior tree or hierarchical finite state machine, more specifically a behavior DAG (directed acyclic graph). In this model, the nodes contain the behaviors and are indexed by state. The nodes can possibly transition to multiple other states. To determine which state of the children of the current state has preference for the transition, each node has a certain priority. Higher priority children are chosen in case there is a transition. In certain situations fixed priority is undesirable. For these situations there are triggers which provide an alternative state to transition to instead of the nodes with fixed priorities.

4.3.2 Goal oriented action planning

Goal oriented action planning (GOAP) is a decision-making architecture developed for the game No One Lives Forever 2 [17]. GOAP is not only responsible for what decision is taken, but also how a decision is taken. GOAP does not replace finite state machines but simplifies producing one. State transitions will not be hard coded but are made as part of a plan. There are three key concepts in GOAP:

- **Actions** - Actions are the steps performed by the actor that makes it do something. Actions know which preconditions have to be met to be able to run, they also know the effect they will have on the state.
- **Plans** - Plans are a sequence of actions to satisfy a goal determined by the actor.
- **Goals** - Goals are conditions that should be satisfied by an actor. In GOAP the goal contains a set of preconditions that should be met before the goal is satisfied. Based on these preconditions an actor can produce a plan of actions that results in satisfying the goal.
- **Plan Formulation** - The planning component will take the most relevant goal determined by the actor and search a sequence of actions based on their preconditions to go from the current state to a state satisfying the condition of the goal. This process is similar to path finding, an implementation can be based on the A* algorithm.

Advantages of this architecture are possible solutions to problems the programmers have not thought off when designing the AI. Sequences of actions to satisfy a certain goal are generated after a runtime search based on the current states and preconditions of available actions instead of hard coded by the programmer. Another benefit of this behavior is that actions can be coded in a smaller scope and are bound together by the planner, which bundles a sequence of actions to satisfy preconditions. Hard coded plans are harder to maintain than an automated machine checked generated plan by the planning system. GOAP also offers an easy way for varied behavior. A programmer adds multiple actions to achieve the same goal with different preconditions and the planner will select the most suitable one based on the current state.

4.3.3 Hierarchical task networks

Hierarchical task networks are another way to plan and execute strategies [19]. They consist of high-level tasks which each contain a set of subtasks which together form a strategy. The leaves of the hierarchy are called primitive tasks. The hierarchical task network consists of methods which each outlines the current goal, the prerequisites and the subtasks necessary to complete the goal. Hierarchical task networks have both a planning and scripting advantage. The plans can by automatically calculated using a planner component. The planner analyses the prerequisites and selects which subtasks to complete in order to achieve the high-level task. The scripts containing the plans are generated *offline* using the results from the planner. The scripts can be included in the game engine to run at runtime using the current state of the game as input.

5 Strategies

Heeren, et al. [11] describes a framework that recognizes and can be used to construct strategies in the domain of exercises. Students who work on interactive exercises can apply changes to an exercise and the framework recognizes if the change is part of a strategy. The framework provides hints to incrementally reach the end goal based on the current exercise.

In this section we will first discuss the requirements, then we will discuss previous work by Heeren, et al [11] on strategies for the domain of exercises.

5.1 Requirements

Heeren, et al. [9] describes several requirements which should be satisfied by the implementation of the rewrite strategy language for the domain of exercises. The following list represents the requirements for a language to describe strategies for the domain of exercises.

1. The strategy language is generic. The language can be used to implement strategies for various domains in which exercises are solved by applying different rules or actions to an exercise.
2. The strategy framework can be used to automatically calculate feedback based on the current state of the exercise and user actions.
3. The strategies reflect textbook descriptions of procedures for solving exercises.
4. The strategies defined in the language are observable.
5. The strategies are compositional and can be combined to form new strategies or be used as part of other strategies.

Most of the requirements are relevant for a strategy language in another domain, in our case, strategies for real-time video games too.

1. The strategy language is generic. The language can be used to apply actions to any state.
2. The strategy framework is able to calculate the next step based on the state. Requirement 2 from the requirements for a language to describe strategies for the domain of exercises states that the framework should automatically calculate feedback given a strategy and user actions. The feedback is used to perform a next step or give a suggestion for the next step. In our domain, actors will automatically perform this next step.
3. The strategies are observable, the strategy framework for exercise rewriting makes the strategies inspectable, printable and transformable.
4. The strategies are compositional and can be combined to form new strategies or be used as part of other strategies. This requirement is relevant for any domain to which strategies are applied.

The above four requirements form the core requirements for a strategy language for a generic domain, including strategies for real-time video games.

One of the requirements mentioned by Heeren, et al. [9], the cognitive fidelity principle, is only relevant for the domain of exercises. In the domain of video games it is not required that users understand how actors perform a strategy. An unpredictable actor can enhance the fun of a video game.

5.2 Strategy combinators

The framework for writing rewrite strategies described by Gerdes, et al. [10] is built on top of a core set of strategy combinators defined in a strategy grammar written in Haskell.

```
data Strategy a
  = Fail
  | Succeed
  | Rule (a → a)
  | Strategy a :∗: Strategy a
  | Strategy a :|: Strategy a
  | Fix (Strategy a → Strategy a)
```

The strategy is polymorphic in its parameter a which refers to the type of the domain it is applied to. The basic elements of a strategy are the *Fail*, *Succeed* and a *Rule* $(a \rightarrow a)$. A rewrite rule gets a the state of type a and returns a new state of type a. :∗: represent the sequence of two strategies, *:|:* represents the choice of either one of the strategies and *Fix* represents the fixed-point combinator for strategies.

In addition to these core language constructs the framework has support for *interleaving* combinators [9]. When students solve a particular exercise, the order of rules applied to an exercise can be non-deterministic. This means the steps can be in any order as long as they are all applied. Interleaving combinators offer the ability to recognize non-deterministic strategies.

```
data Strategy a
  = ...
  | Atomic (Strategy a)
  | Strategy a :%: Strategy a
  | Strategy a :!%: Strategy a
```

Atomic prevents the strategy from being executed interleaved. :%: interleaves the right- and left hand strategies and :!%: interleaves the left- and right hand strategies with a preference for the first strategy. To retrieve the first rule of a strategy the framework defines a function *firsts*. To define *firsts* we need to define the function *empty*, which checks if a given strategy returns an empty result and *split*, which splits the interleaving strategies and combines the possible rules in a list. The function *empty* is defined as follows:

```
empty :: Strategy a → Bool
empty Fail        = False
empty Succeed     = True
empty (Rule _)    = False
empty (Fix f)     = empty (f Fail)
```

$$
\begin{aligned}
&\mathit{empty}\ (a :|: b) &&= \mathit{empty}\ a \wedge \mathit{empty}\ b \\
&\mathit{empty}\ (a :*: b) &&= \mathit{empty}\ a \vee \mathit{empty}\ b \\
&\mathit{empty}\ (\mathit{Atomic}\ s) &&= \mathit{empty}\ s \\
&\mathit{empty}\ (a :\%: b) &&= \mathit{empty}\ a \wedge \mathit{empty}\ b \\
&\mathit{empty}\ (a :!\%: b) &&= \mathit{False}
\end{aligned}
$$

The function *empty* is used in *split* to check whether we need to proceed in splitting the right strategy of the choice combinator.

split is used to calculate a list of possible steps from a strategy. To implement *split*, we make use of properties that hold for the interleaving and left-interleaving combinators as described by Heeren. To define *split* for $a :\%: b$ we make use of the following fact:

$$a :\%: b = a :!\%: b \bigcup b :!\%: a$$

Interleaving a and b is the same as the result of first interleaving a and then b or first interleaving b and then a. Describing the split for $a :!\%: b$ is more involved. We first split a in an atomic part followed by remainder in the form of $\mathit{Atomic}\ a_1 :*: a_2$. We can write the atomic $\mathit{Atomic}\ a_1$ part as $\mathit{Atomic\ (Rule}\ r :*: a_1\mathit{)}$ to illustrate that we return a rule followed by a succeed. Now that we have defined the atomic part, we can interleave the rest of the strategy which leads us to the alternative form for a in $a :!\%: b$:

$$\mathit{(Atomic\ (Rule}\ r :*: a_1\mathit{)} :*: a_2\mathit{)} :!\%: b$$

Now that we have defined the left-hand side of :!%: as $\mathit{(Atomic\ (Rule}\ r :*: a_1\mathit{)}$ $:*: a_2\mathit{)}$, we can transform other strategies to this form and write a recursive split function that returns a list of strategies. The details of the transformations of other elements of the grammar are described in the original research [9].

$$
\begin{aligned}
&\mathit{split} :: \mathit{Strategy}\ a \to [\mathit{Strategy}\ a] \\
&\mathit{split}\ \mathit{Fail} &&= [\,] \\
&\mathit{split}\ \mathit{Succeed} &&= [\,] \\
&\mathit{split}\ (\mathit{Rule}\ r) &&= [\mathit{Atomic}\ (\mathit{Rule}\ r :*: \mathit{Succeed}) :*: \mathit{Succeed}] \\
&\mathit{split}\ (a :*: b) &&= \mathit{split}\ a +\!\!+ \mathit{split}\ b \\
&\mathit{split}\ (a :|: b) &&= [\mathit{Atomic}\ (r :*: x) :*: (y :*: b) \\
&&&\quad \mid (\mathit{Atomic}\ r :*: x) :*: y \leftarrow \mathit{split}\ a] +\!\!+ \\
&&&\quad \textbf{if}\ \mathit{empty}\ a\ \textbf{then}\ \mathit{split}\ b\ \textbf{else}\ [\,] \\
&\mathit{split}\ (\mathit{Fix}\ f) &&= \mathit{split}\ (f\ (\mathit{Fix}\ f)) \\
&\mathit{split}\ (\mathit{Atomic}\ s) &&= [\mathit{Atomic}\ (r :*: (x :*: y)) :*: \mathit{Succeed} \\
&&&\quad \mid (\mathit{Atomic}\ r :*: x) :*: y \leftarrow \mathit{split}\ s] \\
&\mathit{split}\ (a :\%: b) &&= \mathit{split}\ (a :!\%: b) +\!\!+ \mathit{split}\ (b :!\%: a) \\
&\mathit{split}\ (a :!\%: b) &&= [\mathit{Atomic}\ (r :*: x) :*: (y :\%: b) \\
&&&\quad \mid (\mathit{Atomic}\ r :*: x) :*: y \leftarrow \mathit{split}\ a]
\end{aligned}
$$

Given the split function we now define the *firsts* function which takes a strategy and returns a list of pairs containing the next rule and the remaining strategy.

$$
\begin{aligned}
&\mathit{firsts} :: \mathit{Strategy}\ a \to [(\mathit{Strategy}\ a, \mathit{Strategy}\ a)] \\
&\mathit{firsts}\ s = [(r, x :*: y) \mid \mathit{Atomic}\ (r :*: x) :*: y \leftarrow \mathit{split}\ s]
\end{aligned}
$$

The strategy language in the framework has been used to both parse rules of a strategy and to produce them. Our thesis will focus on the production of rules in a strategy. The above definitions of *firsts*, *empty* and *split* as described by Heeren, et al. [9] do not reflect the entire capability of the actual framework. The complete framework contains a definition of the language which accepts a state to which the strategy is applied. The result of the actual version of *firsts* contains the step, the resulting state and the remainder of the strategy. We will go into detail in this implementation in a Section 7.2.

5.3 Example strategies

Using the language of Section 5.2 we can provide a set of example strategies that demonstrate how we can construct strategies using the combinators of the language.

The basic element of a strategy is a *Rule*. In our examples we will use our domain of computer games rather than the domain of exercises. The example shows a strategy for a game state which returns a rule to shoot a gun. The implementation of the rules is not relevant at this point, but it modifies the state in such way that a bullet is shot.

$$\textbf{data}\ \mathit{GameState} = ...$$

$$\mathit{shoot} :: \mathit{Strategy}\ \mathit{GameState}$$
$$\mathit{shoot} = \mathit{Rule}\ (...)$$

$$\mathit{pointAtClosestEnemy} :: \mathit{Strategy}\ \mathit{GameState}$$
$$\mathit{pointAtClosestEnemy} = \mathit{Rule}\ (...)$$

Using the basic *Rule* element of our language we combine multiple *Rules* to form a higher level strategy. We can use the *sequence* and *choice* operator to sequence or choose between multiple *Rules*. The following code shows how to implement a higher level strategy to move to a flag using the atomic move rules.

$$\mathit{move} :: (\mathit{Float}, \mathit{Float}) \rightarrow \mathit{Strategy}\ \mathit{GameState}$$
$$\mathit{move}\ (x, y) = \mathit{Rule}\ (...)$$

$$\mathit{moveToFlag} :: \mathit{Strategy}\ \mathit{GameState}$$
$$\mathit{moveToFlag} = \mathit{move}\ (20, 30) :*: \mathit{move}\ (50, 60) :*: \mathit{move}\ (70, 80)$$
$$\qquad :|: \mathit{move}\ (20, 60) :*: \mathit{move}\ (70, 80)$$

To produce a strategy in which the order of the executed rules is irrelevant we can use the interleaving combinator to produce a list of possible first steps by means of the *firsts* function.

$$\mathit{captureFlag} :: \mathit{String} \rightarrow \mathit{Strategy}\ \mathit{GameState}$$
$$\mathit{captureFlag}\ \mathit{flagId} = ...$$
$$\mathit{captureFlags} = \mathit{captureFlag}\ \texttt{"a"} :\%: \mathit{captureFlag}\ \texttt{"b"} :\%: \mathit{captureFlag}\ \texttt{"c"}$$

Using the fix combinator we can create new combinators such as *repeat*, *try* and *option*

$$\mathit{many} :: \mathit{Strategy}\ \mathit{GameState} \rightarrow \mathit{StrategyGameState}$$
$$\mathit{many}\ s = \mathit{Fix}\ (\lambda x \rightarrow \mathit{Succeed} :|: s :*: x)$$
$$\mathit{shootEnemy} = \mathit{many}\ (\mathit{pointAtClosestEnemy} :*: \mathit{shoot})$$

6 Challenges in implementing strategies in real-time video games

In this section we introduce the main challenges with the implementation of a language to describe strategies in real-time video games. The previous section describes an approach to constructing strategies for the domain of exercises using a small core of strategy combinators. In Section 5.3 we have shown how to write simple example strategies for real-time video games using these combinators. However, using these combinators and functions in real-time video games provides us with a number of challenges due to the concurrent nature of real-time games. First we will discuss the problems related to the concurrent nature and then we will discuss how we adapt our implementation of the strategy library to handle these characteristics.

6.1 Concurrent nature of games

One of the key differences between the application of strategies in the domain of exercises and video games is the concurrent nature of video games as discussed in Section 5. The best way to complete a video game depends on the actions of other players and the in-game events. We will discuss the influence of these factors on the implementation and execution of strategies in real-time video games in the following section.

In the previous section, the basic component of a strategy is a *Rule*. This *Rule* is defined with the purpose to rewrite the current state of the exercise. In the domain of real-time video games we use the type action as the basic element of our strategies. We will discuss the differences in interpretation and semantics between rules and actions in the next section, but we will use the action to refer to the result of an application of a strategy in this section.

6.1.1 Actors

In real-time video games, a number of human- and computer controlled characters simultaneously take part in the same game. These can be characters with varying abilities and goals. We will refer to computer controlled characters as actors. Strategies are associated with each actor. Actions which the actors perform are applied to the current state which results in a new state in which the actor is a step closer to achieving his goal according to the strategy. However, this progress can be interrupted or intervened. In most games, there are multiple actors competing for the same goals or having conflicting goals which only one of the parties can achieve. Each of these actors has its own strategy and its own goal. Some actors might have a shared goal, but each actor will need to derive its own best next action based on a shared game state. If we have n actors who are part of a game, we have to apply n *actions* to a shared state simultaneously.

To conclude, a real-time video game has multiple actors that influence the same game state.

6.1.2 Time

Another aspect of real-time video games is that they often rely on a state that changes over time and changes result in different states depending on the time difference between the previous and current state. For example, suppose we have a game with two actors, actor a and actor b. Actor a wants to capture the flag, and actor b wants to defend the flag. If actor a determines its first step of the strategy, it will move towards the location of the flag. If actor b determines its first step, he will shoot a bullet at actor a. Both steps will be derived based on the same state. If we simply execute the actions one before the other, actor a will be at the flag, and the bullet fired by actor b missed its target. However, if we interleave the execution of the actions while taking duration into account, we see that the bullet hit actor a while he was traveling to the flag. The simulation of the duration of actions is essential to mimic reality.

Considering the above example, we identify two problems. The first problem is that shooting a bullet costs far less time than moving from one location to another. We have to adjust our strategy language to deal with multiple strategies returning several steps at different points of time. An actor might perform n *actions*, while another has just completed one action due to the duration of each action.

The second problem in the example is that if the bullet hits the actor who was moving to a target location, the strategy might change because the state has changed. The actor might have died, thus making his active strategy and action irrelevant, or he might have been hit, after which he wants to shoot back at the shooting actor as part of a better strategy in that situation. We have to make our system capable of interrupting *actions* in progress to produce a new action if the situation requires to do so.

Thus, dealing with time introduces two problems namely

- *Actions* take time to complete, multiple *actions* from different actors do not necessarily take the same amount of time to complete
- Due to the different durations of *actions*, the state for which an action has been selected can be obsolete, which requires a new action to be chosen.

6.1.3 Interleaving concurrent actions

With multiple actors working on the same state and actions that take time to complete we have a concurrency aspect when combining these two aspects. In the strategy framework for exercises as discussed in Section 5.2, all *actions* or *rewrite rules* directly rewrite the state. This means that subsequent *rewrite rules* work on a different state than the previous *rewrite rule*. The order in which these *rewrite rules* are applied matters for the validity of subsequent steps. In the case of real-time video games we have multiple *actions* that are each modifying the state at any point in time. In video games, the end result does not have to be precise in the sense that modifications at millisecond intervals, or the time per frame, are not observable by a player as long as a believable experience is created. To summarize, we have to integrate parallel

execution of actions on the same state to give the player a believable, realistic experience.

6.1.4 Tight coupling

In the strategy framework for exercises as discussed in Section 5.2, applying the *rules* on a state will produce a new state. The problem is that the actions implement how to create a new state based on the state that is passed in as argument. If an actor in a video game has a strategy in which an action results in shooting an enemy, the action itself contains the implementation of how to shoot the enemy. If another actor wants to use this same strategy, but has a different weapon it will be difficult to reuse the action because the implementation of shooting the enemy is coupled with the weapon of the original actor.

6.2 Solutions

The previous section describes some of the aspects that we need to take into account when we want to use a strategy language to implement real-time video games. We will built upon the strategy framework for exercises as discussed in Section 5.2 and adjust or extend it to deal with the specific aspects of real-time video games. In particular we will adjust and extend the system to deal with different actors, the *notion of time*, the *ordering* of executions of actions and the *tight coupling* currently enforced by the feedback system.

6.2.1 Actors

If we have multiple *actions* from multiple actors we might get different end states depending in which order the actions are executed. This means that an action applied to state 0 will result in a state 1, but it might be a different state than expected by the strategy. For example, the strategy returns an action which instructs an actor to move to a target location, but the actor was shot while moving there. The strategy expects a state where the actor is at the target location waiting for the next action of the strategy, but actually, the actor is shot. We can solve this problem by interleaving the execution of multiple actions.

When an actor is shot, he wants to respond to this situation. This problem is also related to the interruption problem where the state changes in such a way that immediate reconsideration of the strategy is required. A solution to this problem is an event based system that fires events when a new action of a strategy is required, such as a goal completion event, enemy spotted event, or character shot event. These events are triggered during the simulation steps.

We also need to take into account that we need to continue the current strategy if the current action is executed successfully. For example, when an actor has reached its target position, we want to execute the next action of its strategy. Continuing or changing the strategy distinguishes two different scenarios for which we need events, namely an event for goal completion and an event to interrupt the current action of an actor. Some events need to pass in certain

information in order to update the strategy context. For example, an action requires an actor to target an enemy. An event containing the selected enemy will allow the next action to make use of that knowledge.

6.2.2 Time

An open problem remains how we deal with actions from multiple actors which take a different amount of time to complete and are executed concurrently. Using the game loop as discussed in Section 4.1.1 we can take the actions and update the state with part of the progression of the execution the actions. By using the iterations of the game loop we can simulate the duration of actions. By using small intervals between iterations of the game loop we can capture events that happen as part of the progression of the execution of an action. These events allow us to interrupt current actions and determine a better step of the strategy to handle the new state.

6.2.3 Tight coupling

In the framework for strategies in the domain of exercises, applying *actions* on a state will result in a new state. If we separate the implementation of the execution of actions and the result of the next step of a strategy, we will have a strategy with a simple action as basic element. The action processor will take these actions and a state and return a new state. It is important to note that the duration of an action involves processing the action multiple times. For example, suppose we set the direction of a character to point at a target location to which it will move and set its velocity. In this case the character starts to move to the target location with a velocity as set by the action, but the state will be changed by the physics engine which gradually changes the actual position based on the velocity and direction of movement. The action data type will act as an intermediate communication language between the strategy and the game engine. Values of the action data type are part of a strategy but it possible to reuse strategies because the execution of each action can be implemented independently from the strategy. The following code shows how we implement the action data type:

```
data Action
   = Move (Int, Int)
   | Shoot (Int, Int)
   | Cover
   | ...
perform :: Action → State → IO ()
perform (Move (x, y)) state = ...
perform (Shoot (x, y)) state = ...
perform Cover         state = ...
```

The simulation performs the calculations necessary to reproduce the state at a given point in time. In real-time video games, these steps run multiple times per second to simulate a fluent animation as described in Section 4.1.1. For example, in these steps the physics engine looks at the velocity and multiplies the velocity with the time difference since the last iteration of the game loop.

After multiple iterations, an actor who moves to a target location reaches its destination. The state of the game, after completion of the step, will be the same as the directly creating a state where the actor is at the target location, given that there are no interruptions or external influences on the state. With this approach we also solve the problem that an action takes a certain duration of time.

7 Contribution

In this section we discuss our research contribution. First we will describe how we implement the grammar for our DSL and how we determine the best actions based on a strategy and a state. Then we discuss how we can integrate the grammar and the functions to determine the best actions in real-time video games.

7.1 Language extensions

Section 6 describes the differences between the domain of exercises and real-time video games with respect to using strategy combinators, and how we handle the unique characteristics in real-time video games. This section describes the implementation of the strategy framework for real-time video games. The core of the framework is the strategy language definition.

```
data Strategy a s
  = Fail
  | Succeed a
  | Context (s → StrategyContext → IO (Strategy a s))
  | Strategy a s :*: Strategy a s
  | Strategy a s :|: Strategy a s
  | Fix (Strategy a s → Strategy a s)
  | (s → StrategyContext → IO Bool) :?: Strategy a s
  | Strategy a s :%: Strategy a s
  | Strategy a s :!%: Strategy a s
  | Atomic (Strategy a s)
 deriving Show
```

We create a strategy language definition similar to the one described in Section 5.2 with an extra parameter a to reflect the usage of an action data type. We use parameter s to specify the game state type with which the strategy works. Moreover, we add extra constructors to the language. Before we introduce the *Context* constructor we will describe the *StrategyContext*. The framework for strategies in the domain of exercises contains a concept to pass around a context with the state of an exercise and additional information about the previous steps. The StrategyContext serves the same purpose.

```
type StrategyContext = M.Map String Dynamic
```

We implement the *StrategyContext* as a *Map* with *Dynamic* data as entries and *String* as key. Because we do not know in advance what the type of the data is we want to store per action we use the *Dynamic* type for the *Map*

entries in *StrategyContext* to provide some flexibility while keeping a common interface to get and set data in the context. The *Context* constructor defines a strategy that uses the state and the strategy context. This strategy is useful when we want to generate a strategy based on information from the state or the previous action. For example, this strategy allows us to determine a target location or to determine the target enemy which was shot by us in the previous action. The result is wrapped in an IO monad, because functions applied on the *StrategyContext* can interact with foreign environments, such as the physics state.

The second new constructor, :?:, uses the *StrategyContext* and the game state to check if a condition holds. Based on this check, the right hand side of the combinator is returned. This is useful in case we want to use specific strategies based on a condition.

The following example from our example game described in Section 8 shows how we can use the :?: combinator and what a condition check can look like. The implementation of *attackSequence* is not relevant for this example, but it is the resulting strategy when the the *enemiesInTheSameRoom* condition is satisfied. The definition of the state in this example can be found in Section 8.2. *enemiesInTheSameRoom* is a function that first retrieves the position of the actor that is associated to the strategy. Then the area in which the actor is will be used to perform a search for enemies which are also located inside the same area.

```
killEnemies = enemiesInTheSameRoom :?: attackSequence

enemiesInTheSameRoom :: State → StrategyContext → IO Bool
enemiesInTheSameRoom = λstate context →
  do let gameObjectId = fromJust $ getGameObjectId context
     let gameObject = fromJust $ M.lookup gameObjectId (objects state)
     pos ← getPosition gameObject
     let area = getRoomByPos pos (areas $ level state)
     let filter = (keyFilter gameObjectId)
     let otherObjects = M.filterWithKey filter (objects state)
     M.fold (λobj bool →
       do b ← bool
          p ← getPosition obj
          return $ b ∨ (insideRoom p area)) (return False) otherObjects
where getGameObjectId context =
        do gameObjectIdDyn ← M.lookup "currentGameObject" context
           (fromDynamic gameObjectIdDyn) :: Maybe String
      keyFilter gameObjectId k a = k ≢ gameObjectId ∧
                                   case unitComponent a of
                                     Just _ → True
                                     _ → False
```

Although the implementation of the of the *enemiesInTheSameRoom* function looks quite verbose, it is a realistic representation of what kind of code is typically written when using the *StrategyContext*.

To conclude, the strategy language takes an extra type argument to reflect the return type of a strategy, and has two additional constructors that return a strategy based on information in the state or the context of a strategy.

7.2 Determining the next action of a strategy

The strategy framework is responsible for the construction and the application of strategies. At the core of the strategy system is the strategy data type which contains combinators for describing a game strategy. The previous section outlines the differences between our data type and the data type for a strategy in the framework for strategies in the domain of exercises. To run strategies we use a modified version of the function *firsts* from the framework for strategies in the domain of exercises

The function *firsts* takes a strategy, a state and a strategy context and returns a list of possible actions an actor can take combined with a new strategy environment and the rest of the strategy. The framework for strategies in the domain of exercises uses rewrite rules which take a state and return a new state as described in Section 5.2. The grammar for strategies in real-time video games from Section 7.1 uses actions as basic elements. We do not include the resulting state in the results like in the framework for strategies in the domain of exercises. It is important to note that the result is wrapped in an *IO* monad because the *split* function, described below, applies the context function to the state which also uses *IO*.

```
firsts :: Strategy a s →
  s →
  StrategyContext →
  IO [(a, Strategy a s)]
firsts strategy state context
  = do res ← split strategy state context
       return [(fromStrategy r, x .*. y) | Atomic (r :*: x) :*: y ← res]
```

Firsts is built on top of the *split* function. The split function is similar to the split function described in Section 5.2. *Split* for game strategies works with the game state, the *StrategyContext* and the new *:?:*, *Context* and *Succeed* constructors from the strategy language.

```
split :: Strategy a s →
  s →
  StrategyContext →
  IO [(Strategy a s)]
split Fail           s context =
  return []
split (cond :?: s1) s context =
  do res ← cond s context
     if res then split s1 s context
     else return []
split (Context r)  s context =
  do res ← (r s context)
     split res s context
split (Succeed r)  s context =
  return [Atomic (Succeed r :*: Fail) :*: Fail]
split (s1 :|: s2)    s context =
  liftA2 (++) (split s1 s context) (split s2 s context)
split (s1 :*: s2)    s context =
  do res ← split s1 s context
```

```
        liftA2 (++) (return [Atomic (r :*: x) :*: (y . * . s2)
                           | Atomic (r :*: x) :*: y <- res])
                    (if empty s1 then split s2 s context else return [])
split (Atomic s1) s context =
  do res <- split s1 s context
     return [Atomic (r :*: (x :*: y)) :*: Fail
       | Atomic (r :*: x) :*: y <- res]
split (Fix f)       s context =
  split (f (Fix f)) s context
split (s1 :%: s2)   s context =
  liftA2 (++) (split (s1 :!%: s2) s context)
              (split (s2 :!%: s1) s context)
split (s1 :!%: s2)  s context =
  do res <- split s1 s context
     return [Atomic (r :*: x) :*: (y .%. s2)
       | Atomic (r :*: x) :*: y <- res]
```

The *Context* constructor contains a function that takes the game state and the *StrategyContext* and returns a new strategy which the split function uses. The *:?:* function returns the right-hand strategy in case the condition in the left-hand side of the combinator holds.

7.3 Integrating strategies in video games

In the previous section we discussed the grammar of the DSL to describe strategies in real-time video games. In this section we will discuss how we integrate our DSL in games. First we describe how we associate strategies with an actor, then we discuss the concept of event handling. Finally we will discuss how we execute the actions which are part of *Succeed* constructor of a strategy.

7.3.1 Strategies per actor

Since each actor has its own strategy, we associate the strategy, the actions and the strategy context per actor. It is possible for an actor to have different goals from others. This is why each actor keeps track of its own strategy. To keep track of certain decisions, some strategies require a state during execution. For example, a strategy to shoot an enemy until the enemy dies needs to keep track which enemy was selected. Selecting the enemy and shooting might be different actions but the next action needs to use information from the previous action. This information is stored in the *StrategyContext*. Since each strategy has its own state, this state is also stored with each actor. The actions from a strategy take time to complete, which is why we need to keep track of the current executing action for the actor. In Section 6.1.2 we described a problem that occurs when an actor executes an action but a state change occurs which requires the actor to run the strategy again to find a possibly better strategy to handle the new situation. If the current action changes to a new action after an interruption, the current action and strategy state change to the new action and the context will be the initial context. We store the strategy, the result and the *StrategyContext* in a strategy component per actor which is defined as follows:

```
data StrategyComponent a s = StrategyComponent {
```

```
  initialStrategy :: Strategy a s,
  strategyResult  :: StrategyResult a s,
  context         :: StrategyContext,
  initalContext   :: StrategyContext
} deriving Typeable
type StrategyResult a s = (a, Strategy a s)
```

The strategy result is a tuple that contains the current action and the next part of the strategy. This result is used to execute the action and to determine the next step of the strategy. The usage of this type is described in the functions discussed below.

7.3.2 Event handling

At any time in the game, events can be fired to notify actors of a change in the state. In case the state changes, the conditions using the :?: combinator might change which will result in a new step of a strategy. Code that executes an action is responsible for firing an event as soon as it completes, we will discuss this in more details in Section 7.3.3. Sometimes state changes are relevant for more than one actor. For example, when an actor captures a flag, the game sends an event to all players to notify them that the flag has been captured. The constructors of the *Event* data type distinguishes between *InterruptEvents* and *CompletedEvents*. Each of these events specify for which actors they are relevant, namely all actors or actors with specific identifiers.

```
data Event = CompletedEvent EventActor
           | InterruptEvent EventActor
         deriving Show
data EventActor = EventActor [String]
                | All
              deriving Show
```

To keep track of events we use an *EventChannel*. Because we do not know in advance to whom we want to send events or what kind of events we can listen to, we use the *EventChannel* as a message bus. The *EventChannel* is implemented using a list to keep track of all events that are fired in the game. To fire events, we use the function *fireEvent*. The *fireEvent* function inserts the event in the channel.

```
type EventChannel = [Event]
fireEvent :: EventChannel → Event → EventChannel
fireEvent eventChannel event = do (event : eventChannel)
```

Because of the nature of the interface to OpenGL, we cannot recursively call the game loop function with an *EventChannel*. This is because the callbacks are triggered by the OpenGL library and return the type *IO ()*. Therefore we need to keep around a reference to an event channel using an *IORef*. We use a modified version of *fireEvent* to use this *EventChannel*.

```
fireEvent' :: IORef EventChannel → Event → IO ()
fireEvent' chan event = do
  channel ← readIORef chan
  writeIORef chan (fireEvent channel event)
```

The function *fireEvent'* is used in our game play code to fire events when actors complete their action or when external events happen that influence the strategies of the actors. Below are two examples how *fireEvent'* is used, the next section shows an example of an action which is processed and uses *fireEvent'* to notify the actor performing the move.

```
fireEvent' channel (CompletedEvent (EventActor [identifier gameObject]))
fireEvent' channel (InterruptEvent All)
```

Because all fired events are stored in the event channel, we also need a component that listens to incoming events and reacts on state changes by notifying actors that they need to run the strategy on the state and perform the action from the result. The *processEvents* function is called by the game loop, which runs every iteration of the game loop. The function then calls the appropriate response handler to respond to the changes in the state that influence the steps of the strategies of the actors. The response handler is implemented as the *eventResponse* function which is called with the function *next* or *interrupt* which are described below. The *processEvents* function uses the game state as last parameter and has a different implementation depending on the game. In the rest of this section we will use the following data types for *State* and *Action*.

```
data State = State {
  objects :: M.Map String GameObject
    ...
}
data Action
    =  Move (Float, Float)
    |  GoToRoom String
    |  Idle
    ...
```

We will use the *State* type in the implementation of *processEvents*.

```
processEvents :: EventChannel → State → IO State
processEvents [] state = return state
processEvents (x : xs) state = do
  state' ← processEvent x state
  processEvents xs state'

processEvent :: Event → State → IO State
processEvent (CompletedEvent e) state = eventResponse e next state
processEvent (InterruptEvent e) state = eventResponse e interrupt state
```

The *eventResponse* function takes a function that modifies the *StrategyComponent* based on the state. In the code example above we saw the function *next* and *interrupt*. If the event interrupts the current strategy we run the initial strategy on the state. If the events indicates the completion of a running strategy we can continue the strategy where it left off by computing the next action. Depending on the *EventActor* the *next* or *interrupt* function will adjust the strategy for all actors or a group of actors. The response handling for all actors occurs when an interrupt event is thrown, for example when a flag is captured and all actors should be notified. The details of *next* and *interrupt*

will be explained later.

```
eventResponse :: EventActor
  → (StrategyComponent Action State
    → State
    → IO (StrategyComponent Action State))
  → State
  → IO State
eventResponse All fun state = do
  objects' ← T.sequence $ M.map (liftGameObject fun state) (objects state)
  return state { objects = objects' }
eventResponse (EventActor identifiers) fun state = do
  objects' ← T.sequence $ M.mapWithKey updateActors (objects state)
  return state { objects = objects' }
  where updateActors key = if elem key identifiers then liftGameObject fun state
    else return
```

Because the *processEvent* function is recursive, we return an updated state for the next objects to work with. We update the relevant actors using the *liftGameObject* function which applies the modification function to the *StrategyComponent* of the specific actor. Then we add the updated *StrategyComponent* to the game object using the *addComponent* function.

```
liftGameObject :: (StrategyComponent Action State
  → State
  → IO (StrategyComponent Action State))
  → State
  → GameObject
  → IO GameObject
liftGameObject fun state gameObject = do
  case strategyComponent gameObject of
       Just component → do newComponent ← fun component state
                           return $ newGameObject newComponent gameObject
       _ → return gameObject
  where newGameObject component =
    addComponent "strategyComponent" component
```

The function *next* determines the next action of the current *StrategyComponent*. The *StrategyComponent* is updated with the result of the execution of the action. The function *next* is defined as follows:

```
next :: StrategyComponent a s → s → IO (StrategyComponent a s)
next component state = do
  let (_, strategy) = strategyResult component
  let c = context component
  result ← randomFirst strategy state c
  case result of
    Just s → do return $ component { strategyResult = s }
    _ → return component
```

In addition to determining a next action using the *next* function, we also define a function *interrupt* which takes the initial strategy from the *StrategyComponent* and calculates the best action based on the state. If the current action remains

the best choice based on the conditions that hold for the state, the resulting action would be the same as the current action and thus not modify the current situation.

```
interrupt :: StrategyComponent a s → s → IO (StrategyComponent a s)
interrupt component state = do
  let s = initialStrategy component
  let c = initialContext component
  result ← randomFirst s state c
  case result of
    Just s → do return $ component {strategyResult = s}
    _ → return component
```

7.3.3 Action handling

The last part of the game strategy framework is to take the next action of a strategy and apply the action to the state. In the previous section we describe the application of a strategy to the state and the result of a strategy. The execution of an action is specific to a game, and possibly per actor which uses a certain strategy. For example, we can easily reuse basic behavior of actors in multiple real-time first person shooters, but the actual implementation of the behavior of an actor depends on the properties of the actor in a specific game. Therefore, we only give an example of the implementation of the application of an action to the state.

In our example game, the function *runObjects* traverses through all game objects that contain a *StrategyComponent*. *runObjects* is similar to the function described in Section 6.2.3. These game objects can be considered as actors that need to complete a goal. For each of those game objects, the current action is applied to the state. The action calls the appropriate handler function with the necessary parameters.

```
runObjects :: IORef State
           → IORef EventChannel
           → GameObject
           → IO GameObject
runObjects stateVar channel gameObject = do
  case strategyComponent gameObject of
    Nothing       → return gameObject
    Just strategy → do
      let action = getAction (strategyResult strategy)
      let executeAction = do
        case action of
          Move (x, y)        → executeMove (x, y)
          GoToRoom string → executeGoToRoom string
          Shoot (x, y)       → executeShoot (x, y)
          Idle               → (\_ _ → return gameObject)
      executeAction stateVar channel gameObject
```

The game play code contains code that responds to actions by calculating the next modification of the state for an action. The example function *executeMove* runs every frame as long as the current action is *Move*. This function uses the *Action* data type as described in Section 8.2 from the example game. Each

frame, we adjust the position of the *gameObject* to rotate towards the target location. If it reaches the target location, we fire a *CompletedEvent* and we adjust the velocity to zero. The physics engine takes care of adjusting the position and acceleration based on the current direction of the player. This functionality is not visible in the *executeMove* function but is separated from the game code. The physical properties of the game objects are all updated in the physic calculations which also happen on a per-frame base.

```
executeMove :: (Float, Float)
  → IORef State
  → IORef EventChannel
  → GameObject
  → IO GameObject
executeMove (x, y) stateVar channel gameObject = do
  state ← readIORef stateVar
  case physicsComponent gameObject of
    Just (PhysicsComponent shape _ _ _) → do
      let b = H.body shape
      H.Vector xo yo ← get $ H.position b
      if H.len (H.Vector (xo − x) (yo − y)) > 20 then
        do gameObject′ ← rotateToPoint (H.Vector x y) gameObject
           adjustVelocity gameObject
        else do H.velocity b $ = H.Vector 0 0
           fireEvent′ channel (CompletedEvent
             (EventActor [identifier gameObject]))
           putStrLn "movement done"
           return gameObject
```

7.3.4 Summary

To create a framework to describe strategies for real-time video games, we used existing research in the domain of exercises and added functionality to handle the unique characteristics of real-time video games. To summarize, the following contributions have been made [2]:

- We created a strategy grammar similar to the one described in Section 5.2 and added the :?: and the *Context* data type constructors. The :?: combinator provides functionality to define strategies based on conditions that hold on a state. Combining the :?: combinators with the :|: combinator enables the definition of strategies with priorities. The Context combinator defines strategies that return a strategy based on the state and strategy context.

- When we calculate the next step of an strategy given a state, the result is an action data type. Code that executes actions is implemented separately from the strategies. Because the actions act as intermediate communication language, the strategies are more reusable and are easier to integrate in existing game engines or game libraries.

- The split function uses the interleaving combinators as a way to produce non-deterministic strategies, which uses a similar implementation as the

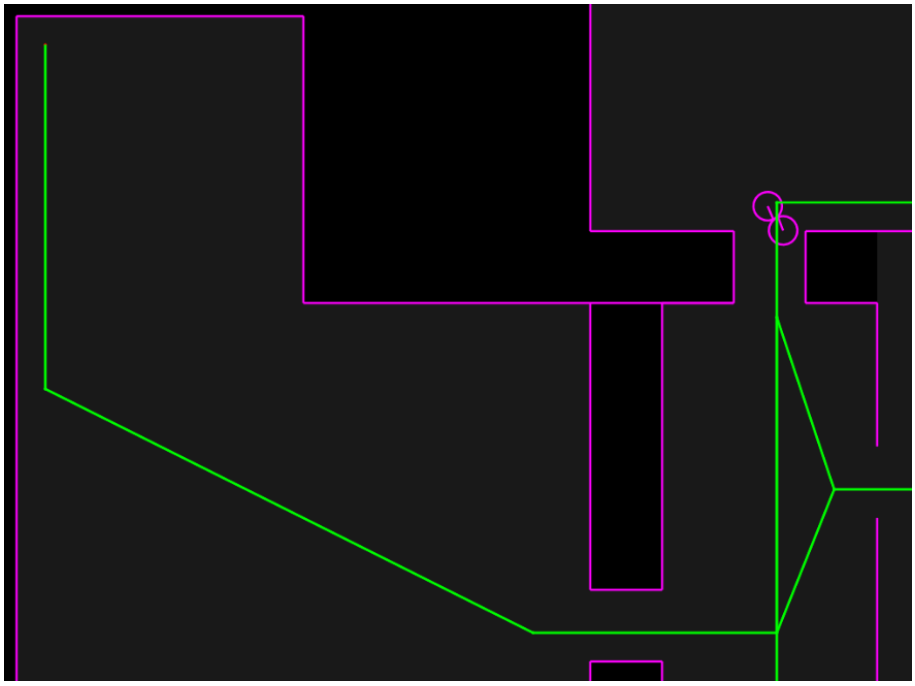
[2]Source code is available at http://intellicode.nl/thesis.html

interleaving combinators described in Section 5.2.

- The game play code should contain code that applies actions to the state. This means that the handling of actions can take advantage of reusable middle-ware components such as physics or animation libraries which are accessible from the gameplay code.
- Upon completion of actions, the game play code should fire events to the event channel to notify the strategy system that it can determine the next step of the strategy.

8 Example game

To demonstrate how we can implement the strategy framework we implement a small 2D real-time action game that contains actors which use strategies to compute steps to complete the game. The game contains two teams. The goal of the *terrorists* is to capture a flag. The goal of the *counter-terrorists* is to prevent the flags from being captured. The level contains a spawning point for the terrorists and a spawning point for the counter-terrorists. The levels contain two locations where a flag can be captured.



The screen shot above shows part of a level in the example game. Players are rendered by circles, the walls are rendered by purple lines and the navigation structure is rendered by green lines.

8.1 Libraries

The game has been implemented in Haskell using the OpenGL library to render the game on screen and to schedule the game loop. On top of the OpenGL

library we use GLUT which provides commonly used functions to use the OpenGL library such as input handling. To perform the physics calculations we use the Chipmunk physics library using the Haskell bindings via Hipmunk.

8.2 Architecture

The game is implemented using the libraries described in the Section 7. The core components from the game are defined in separate modules:

- Rendering - The rendering in the example game only consists of basic shapes to illustrate how the game behaves. The level is rendered using OpenGL line calls that represent the the level as described by the game objects. To visualize the AI navigational decisions, the way points and possible routes between way points are also rendered on screen. To visualize the players, we use circles and we draw a single point to represent bullets.
- Input - The game is primarily a simulation of AI players. In a game where players are able to play characters the input functionality is more complex. In the game we use the OpenGL functionalities to write callbacks to handle input.
- Level - Levels are described in a JSON based file. A level is parsed and converted into Haskell data types. Based on these data types we create the necessary game objects and add the objects to the physics engine and render them on the screen.
- State - The state of a game is stored in a *State* object as described below. The state is referenced by an *IORef* because the game loop is called using OpenGL callbacks. Since we cannot update the state recursively we have to pass in a reference to the state.

```
data State = State {
  hipmunkSpace :: H.Space,
  objects      :: M.Map String GameObject,
  circleList   :: GL.DisplayList,
  point        :: H.Vector,
  level        :: Level,
  changed      :: Bool
} deriving Typeable

data Action
  = Move (Float, Float)
  | GoToRoom String
  | Idle
  | Action String
  | Shoot (Float, Float)
  | Wait Int
  ...
  deriving (Show, Typeable, Ord, Eq)
```

The most important functions in the *State* data type is the *hipmunkSpace* and the *objects*. The hipmunkSpace returns the state of the physics

engine and is part of the Hipmunk physics engine. The Hipmunk physics engine uses *space* to refer to the state. The *objects* function returns a collection of game objects. The other data is used to load the appropriate level and to cache OpenGL objects. The *Action* data type contains the list of constructors which a strategy can return.

- GameObject - Every object in the game is a game object. The game objects are represented using a component based model[22]. This way we can easily extend the behavior of game objects depending on what components are relevant for the object. For example, a wall has a physics and a render component, while a way point just has a render component.

- Physics - The physics engine is a third party library written in C++. In the game we use the Haskell bindings to communicate with the library. The physics engine contains a state, often referred to as *world* or *space*, which is initialized on the game startup. All objects with a physics component are added to this state together with a description of their shape. An object's physics component contains a *shape* which describes the outline of the shape of the game object. The shapes are used to perform collision checks against other shapes in the physics engine. The shapes created with Hipmunk are pointers to an object in the Hipmunk library and are mutable.

  During each iteration of the game loop we simulate the state of the Hipmunk space by calculating the new positions of objects based on the time delta from the last simulation.

- Strategies - The strategies are implemented using the combinators as described in the Section 7.1. Each game object can have a strategy component which contains the strategy and an initial context. In the game, the actors which play the game have strategies for navigating and shooting at enemies when they are in the same room. The high level structure of the strategies will be discussed in the next section. Each iteration of the game loop, all components with a strategy component will execute their current action based on the state and if applicable fire events when actions have been completed such that new steps can be calculated for the next iteration of the game loop.

- Action execution - Each iteration of the game loop, we perform a function that takes the action and the state and runs the physics engine to calculate a new state. After the action is completed we fire the events as described in 7.3.3.

8.3 Example strategies

In this section we discuss a couple of examples of strategies that are used in our example game. Although we describe strategies, we only describe the high-level structure of our strategies using the strategy framework as described in this thesis.

8.3.1 Moving to the flag

The main goal for one team is to capture the flag, and for the other team, to prevent that from happening. Both teams start in a location from which they need to proceed to the location of the flag. The primary strategy is to move to this location. An actor can move to a location by navigating through several way points in the map to reach the target. We can manually construct a strategy which requires us to create a list of way points to navigate to reach the target destination. Another option is to use a path finding algorithm inside our strategies as is usually done in other video games. We have implemented the A* path finding algorithm to generate a list of way points to navigate to to reach the target destination. This demonstrates that we can easily embed other technologies as part of a strategy.

The *moveToFlag* function takes a position and uses the function *moveToWaypoint* to construct a strategy that is composed of move actions to all way points on the shortest path.

```
moveToFlag :: Pos → Strategy Action State
moveToFlag pos =
  Context (λstate context →
    let s = state
        a = areas $ level s
        w = waypoints $ level s
        target = head $ waypointsInRoom "flag-a" w
    in return $ moveToWaypoint a w target pos)
```

The *moveToWayPoint* function uses the :∗: combinator to construct a strategy to move from the current position to a target way point. The *getShortestPath* function uses the A* algorithm to obtain a list of all way points that are part of the shortest path to the target location, which is the location of the flag for this strategy. When a *Move* action has been completed a *CompletedEvent* will be fired after which the next step from the strategy is chosen.

```
moveToWaypoint ::
  [Area] →
  [Waypoint] →
  Waypoint →
  Pos →
  Strategy Action State
moveToWaypoint areas waypoints (Waypoint _ target _) pos =
  let (Area name _ _ _) = getRoomByPos pos areas
      startWaypoint = getClosestWaypointInRoom name pos waypoints
      path = getShortestPath waypoints (λx → target ≡ x) startWaypoint
      shortestPath = fromMaybe [] path
      d2f = double2float
      move = λ(Pos x y) → (:∗:) (Succeed $ Move (d2f x, d2f y))
  in foldr move Fail shortestPath
```

To summarize, we have shown how we can use the :∗: combinator to build strategies based on an algorithm to calculate the necessary steps to perform using the strategy. We also use the function from the *Context* constructor to query the state for information used in the strategy.

8.3.2 Shoot enemies

In order to capture the flag or prevent the flag from being captured, the actors have to shoot the members of the other team. We have constructed a strategy to implement basic combat behavior.

$$
\begin{aligned}
&\mathit{combat} = \mathit{enemiesInTheSameRoom} \mathbin{:?:} \\
&\quad (\mathit{shootClosestEnemy} \mathbin{:|:} \mathit{throwGrenade} \mathbin{:|:} \mathit{takeCover})
\end{aligned}
$$

The *enemiesInTheSameRoom* function checks in the state if there are any enemies in the same room as the actor. We have seen its implementation in section 8. The strategy demonstrates how we check whether a condition holds in the state. We use the :?: combinator to check the left hand side, and if the result is true, the right hand side of the strategy is used. In this strategy, if there are enemies in the same room, the actor will randomly chose an action to shoot the closest enemy, throw a grenade or take cover. The left hand of the :?: constructor is chosen randomly by using the *:|:* combinator. The *shootClosestEnemy*, *throwGrenade* and *takeCover* functions are omitted, but they will fire a *CompletedEvent* when the action has been completed.

8.3.3 Composing strategies

To combine the combat strategy with the strategy to move to a flag we use the :|: combinator. If there are no enemies in the room, the strategy will fail, and we use the second strategy to move to the location of the flag.

$$
\mathit{gameStrategy}\ \mathit{pos} = \mathit{combat} \mathbin{:|:} \mathit{moveToFlag}\ \mathit{pos}
$$

Now we have a strategy that shoots enemies if they are in the same room or the actor will proceed to the flag. A missing strategy in our game is to return to base when the flag is captured by a member of the terrorist team. Similarly the opposing team needs to retrieve the flag when the flag is captured by a member of the terrorist team or defend the flag. Both teams need to react in case the flag is captured. This will be triggered by an *InterruptEvent* for all actors. Because the strategies are similar for both teams except for the strategy after the flag has been captured, we can take advantage of the combinators and the fact that strategies are first class citizens. We can easily reuse overlapping strategies. The description of each strategy is defined below the example code:

$$
\begin{aligned}
&\mathit{returnToBase} = \mathit{actorHasFlag} \\
&\qquad\qquad\qquad \mathbin{:?:} \mathit{moveTo}\ \mathit{baseWayPoint} \\
&\qquad\qquad\qquad \mathbin{:|:} \mathit{defendTerroristWithFlag} \\
&\mathit{terroristStrategy} = \mathit{gameStrategy}\ \mathit{returnToBase} \\
&\mathit{chaseTerroristWithFlag} = \mathit{flagCaptured} \\
&\quad \mathbin{:?:} ((\mathit{enemyWithFlagInSight} \mathbin{:?:} \mathit{throwGrenade} \mathbin{:*:} \mathit{shoot}) \mathbin{:|:} \mathit{chase}) \\
&\mathit{counterTerroristStrategy} = \mathit{gameStrategy}\ \mathit{chaseTerroristWithFlag} \\
&\mathit{gameStrategy}\ \mathit{teamSpecificStrategy} = \mathit{teamSpecificStrategy} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbin{:|:} \mathit{combat} \\
&\qquad\qquad\qquad\qquad\qquad\qquad \mathbin{:|:} \mathit{moveTo}\ \mathit{flag}
\end{aligned}
$$

Because the function implementations have been omitted we give a short description of what the functions do.

- *actorHasFlag* - An implementation of this strategy will check the state if the actor executing the strategy has the flag.
- *moveTo* - The level file contains several way points, the *moveTo* function will use the A* algorithm to generate a strategy to move to the way point passed as parameter. The implementation is similar to the *moveToFlag* implementation described in section 9.3.1.
- *defendTerroristWithFlag* - An implementation will first check the location of the actor with a *Context* function and store the result in a context. We can combine this *Context* function with another strategy that will move to the location using the :∗: combinator.
- *flagCaptured* - This function will check the state if the flag has been captured
- *enemyWithFlagInSight* - This function will check the state for the current position of the actor executing the strategy. Then the function will iterate through the other actors in the game and if an actor with the captured flag is within a certain radius, the function will return *True*.
- *throwGrenade* and *shoot* - This strategy will return an action to first throw a grenade and then another action to shoot at the actor which captured the flag.
- *chase* - This strategy will check the state to search for the location of the actor that captured the flag and will construct a strategy to move to the location using the A* algorithm.

Some of the strategies that check the state for a certain condition will need to run when the actor has been notified that the state has been changed in such way that we need to apply the strategy again. For example, *enemyWithFlagInSight* requires events to be fired if actors come within a certain radius. If this event is fired, and the actor with the captured flag is in sight, the *enemyWithFlagInSight* will return *True*.

8.3.4 Events

We have described how we describe strategies using the strategy combinators, but for a full implementation we need to implement an event for each *Action* that completes or each event that should notify actors to adjust their strategy. In Section 7.4.3 we gave the following example of a function that executes the *Move* action which is a step from the *moveToFlag* or *moveTo* strategy mentioned in the previous section:

```
executeMove :: (Float, Float)
  → IORef State
  → IORef EventChannel
  → GameObject
  → IO GameObject
executeMove (x, y) stateVar channel gameObject = do
  state ← readIORef stateVar
  case physicsComponent gameObject of
```

```
Just (PhysicsComponent shape _ _ _) → do
  let b = H.body shape
  H.Vector xo yo ← get $ H.position b
  if H.len (H.Vector (xo − x) (yo − y)) > 20 then
    do gameObject′ ← rotateToPoint (H.Vector x y) gameObject
       adjustVelocity gameObject
    else do H.velocity b $ = H.Vector 0 0
       fireEvent′ channel (CompletedEvent
         (EventActor [identifier gameObject]))
       putStrLn "movement done"
       return gameObject
```

Whenever a *Move* is complete, a *CompletedEvent* will be added to the eventChannel. In Section 7.4.2 we described how this event will be used to take the next step in a strategy after an event is received. Without this event, the next step cannot be selected from the strategy, so it is important to add code to fire the events when an action is completed.

The other type of event is the *InterruptEvent* which will be fired whenever there is a change in the state that actors should be notified about. In the example game, when the flag has been captured by a member of the terrorist team we fire the following event:

```
InterruptEvent All
```

This event will trigger the engine to apply the full strategy to the state for each actor and select the best step. This event is also executed when an actor is shot or when there are enemies detected in the same room. In these cases we fire the *InterruptEvent* with specific a specific list of actor identifiers in the *EventActor* data type constructor as in the *executeMove* function.

8.3.5 Summary

We have described how we implement an example game using popular components found in many modern video games with the same constraints as we described in the Section 5. We have also shown how we use the strategy system from Section 6 and Section 7 in the example game to show how we use a modified version of the exercise feedback framework in a real-time video game. Although full implements of actions and functions are outside the scope of this thesis, we have shown how we can describe strategies to achieve goals in a real-time video game.

9 Conclusion

In this section we will evaluate our contribution and solution to our research problem and discuss the opportunities for future work.

9.1 Evaluation

In this thesis we have introduced a DSL to describe strategies for real-time video games. We investigated various existing methods to describe strategies used in existing games to identify the necessary attributes to describe strategies in real-time video games as discussed in Section 4.3. To design our DSL, we used previous research from the domain of exercises as described by Jeuring, et al.[8] and Heeren, et al.[11][9]. The framework as described in these papers is used as source of inspiration. The DSL described in this thesis is built to handle the constraints as outlined in section 6. The following contributions are made in addition to building upon the previous work on a DSL for the domain of exercises:

- Our DSL offers a conditional combinator :?: to model strategies if certain conditions are met and a *Context* combinator to return a strategy based on data from the state or the action specific *ActionContext*.
- The strategies return actions which are are executed asynchronous rather than executed directly. The implementation of the execution of an action is handled by the game play code which allows us to interleave execution and to benefit from the features offered by the game engine.
- To know when to calculate the next step of a strategy we have added an event system to notify actors that actions are completed or interrupted.
- Our DSL is used to determine the next step rather than to parse previous steps of strategies.

To conclude, we have shown that we can develop a DSL containing the necessary elements to describe and execute strategies to achieve goals in a real-time video game. We built upon existing research in the domain of strategies and used components which are typically found in real-time video games. We discussed various ways how strategies are currently described in real-time video games. Existing methods mostly rely on using general purpose control flow elements or use graph- or tree like data structures to model strategies. Our DSL offers a convenient and concise way to describe, include and reuse strategies in real-time video games. Our implementation can be an attractive alternative that has the advantages of a DSL and does not require a graphical user interface to conveniently adjust, prototype and maintain strategies in real-time video games.

9.2 Future work

In this thesis, we focused on describing strategies for real-time video games and implemented an example strategy game. We want to implement other types of games such as first-person shooters, role-playing games and platform games. Another opportunity for future work is to implement related technologies such as behavior trees, GOAP and hierarchical task networks using our DSL to illustrate the power of our language. Using the concepts from this thesis we want to investigate if we can describe other parts of video games such as the flow of a story line, animations and behavior of groups of agents using the DSL

as described in this thesis. Although we can describe strategies that return the best action given a game state, we still depend on the fact that a game needs an implementation of the basic actions. Currently there is no enforcement of the implementation of events that are triggered upon completion of actions. Currently we rely on the fact that a programmer fires an event at the completion of an action. The programmer can use of identifiers in the *StrategyContext* to notify the strategy system that an action of a certain actor is completed. Future work can focus on the relationship between actions and events to see if we can further integrate them and couple the events to actions to remove the need of a way to identify which actor executes the action.

References

[1] D. Leijen *The λ Abroad*, PhD Thesis , 2003.

[2] D. Isla *Handling Complexity in the Halo 2 AI*, Proceedings of the Game Developers Conference , 2005.

[3] E. M. Avedon *The Structural Elements of Games*, The Study of Games, 1973.

[4] T. Fullerton *Game Design Workshop: A Playcentric Approach To Creating Innovative Games*, Morgan Kauffmann, 2008.

[5] T. Sweeney *The Next Mainstream: Programming Language: A Game Developers Perspective* the Symposium on Principles of Programming Languages, 2006

[6] D. Leijen, E. Meijer and J. Hook *Haskell as an Automation Controller*, Lecture Notes in Computer Science, Volume 1608, 1999

[7] A. W. B. Furtado, A. L. M. Santos, G. L. Ramalho *Digital Games Development Automation through Domain- Specific Languages: an End-to-End Approach*, 2008

[8] J. Jeuring, B. Heeren *Recognizing Strategies*, Electronic Notes in Theoretical Computer Science, 2008

[9] B. Heeren, J. Jeuring *Interleaving Strategies*, Conference on Intelligent Computer Mathematics, 2011

[10] A. Gerdes, B. Heeren, J. Jeuring, *Properties of Exercise Strategies*, 1st International Workshop on Strategies in Rewriting, Proving, and Programming , 2010

[11] B. Heeren, J. Jeuring, A. Gerdes *Specifying Rewrite Strategies for Interactive Exercises*, Mathematics in Computer Science - MiCS, vol. 3, 2010

[12] R. Laemmel, E. Visser and J. Visser *The Essence Of Strategic Programming*, draft, 2002

[13] H. Hoang, S. Lee-Urban, H. Muoz-Avila *Hierarchical Plan Representations for Encoding Strategic Game AI*, proceedings of the first artificial intelligence and interactive digital entertainment conference, 2005

[14] A. Nareyek *AI in computer games*, ACM Queue, 2004

[15] M. Mernik, J. Heering, A. M. Sloane *When and How to Develop Domain-Specific Languages*, ACM Computing Surveys, Vol. 37, No. 4, December 2005, pp. 316344, 2005

[16] A. Gavin *Making the Solution Fit the Problem: AI and Character Control in Crash Bandicoot* , Computer Game Developers Conference Proceedings, 1997

[17] J. Orkin *Applying Goal-Oriented Action Planning to Games* Draft AI Wisdom 2, 2003

[18] P. Hudak *Building Domain Specific Embedded Languages* Yale University, 1996

[19] J.P Kelly, A. Botea, S. Koenig *Planning with Hierarchical Task Networks in Video Games* Proceedings of the Artificial Intelligence and Interactive Digital Entertainment Conference, 2008

[20] W. White, C. Koch, J.Gehrke, and A. Demers *Better Scripts, Better Games* Communications of the ACM - Being Human in the Digital Age Volume 52 Issue 3, 2009

[21] S. J. Russell, P. Norvig *Artificial Intelligence: A Modern Approach* Prentice-Hall: Englewood Cliffs, 1995

[22] Eelke Folmer *Component Based Game Development A Solution to Escalating Costs and Expanding Deadlines?* 10th International Symposium, CBSE 2007, Medford, MA, USA, 2007