Creation of a video game and learning of an intelligent agent

Auteur : Lorent, Nicolas
Promoteur(s) : Ernst, Damien
Faculté : Faculté des Sciences appliquées
Diplôme : Master en sciences informatiques, à finalité approfondie
Année académique : 2015-2016
URI/URL : http://hdl.handle.net/2268.2/1665

University of Liège

Faculty of Applied Sciences
Montefiore Institute

Development of a Video Game and Creation of an Artificial Player

Master Thesis Submitted for the Degree of
MSc in Computer Science

Author:
Nicolas LORENT

Supervisor:
Prof. Damien ERNST

Academic year 2015 - 2016

Acknowledgements

First, I am grateful to Prof. Damien Ernst for approving the subject of my thesis, and allowing me to combine my passion with my studies.

I also wish to express my sincere thanks to David Taralla and Aaron Qiu, for their dedication and invaluable advices. Their help and support considerably improved my work, and I am thankful for the time that they dedicated to reviewing this thesis. I was really pleased to meet these friendly fellows who share the same passion for card games, and I honestly could not wished for better co-supervisors. You guys rock!

Purpose

The purpose of this work is twofold.

In a first part, we develop a real-time strategic card game from scratch. It includes the design of game rules, a user interface, cards, decks, and the implementation of the game in java with neither a game engine nor an external library. The game is playable online or offline, against other human players or artificial players, and with pre-built or custom decks.

In a second part, we consider the problem of creating an artificial intelligence to play the card game. Two artificial players are proposed. Firstly, a heuristic player that evaluates moves by quantifying the amount of resources it gains (or loses) versus the amount of resources its opponent gains (or loses). Secondly, a more advanced player that simulates the long-term consequences of a move until the end of the game to predict its outcome. Lastly, the performances of both artificial players are assessed by comparing their win rates.

Table of Contents

Part 1
Development of a Video Game

The developed game is a digital collectible card game for personal computers.

In a collectible card game (**CCG**, also referred to as **trading** card game, or **TCG**), two players compete against each other with customizable decks. A custom deck is built by selecting a limited number of cards from a large pool. During a match, players make use of cards from their respective decks with the objective to reduce the opponent's health points to zero.

Collectible card games first emerged in the 90s as physical games. However, digital versions have been developed later. Most of digital card games originate from an existing physical card game, but some of them are solely digital.

This part describes the creation of such a digital game from scratch. The game is neither an adaptation of a physical game, nor a copy of an existing digital game. The pursued objective was to create a brand new game, which involves the elaboration of rules and the creation of dedicated content.

It consists in the following chapters. Chapter 1 analyses the major existing card games, by comparing their forces and weaknesses, and identifies the specificities of the digital space. Chapter 2 describes the process behind the creation of the game rules. Lastly, Chapter 3 details the implementation of the game.

Chapter 1
Small Tour in the CCG World

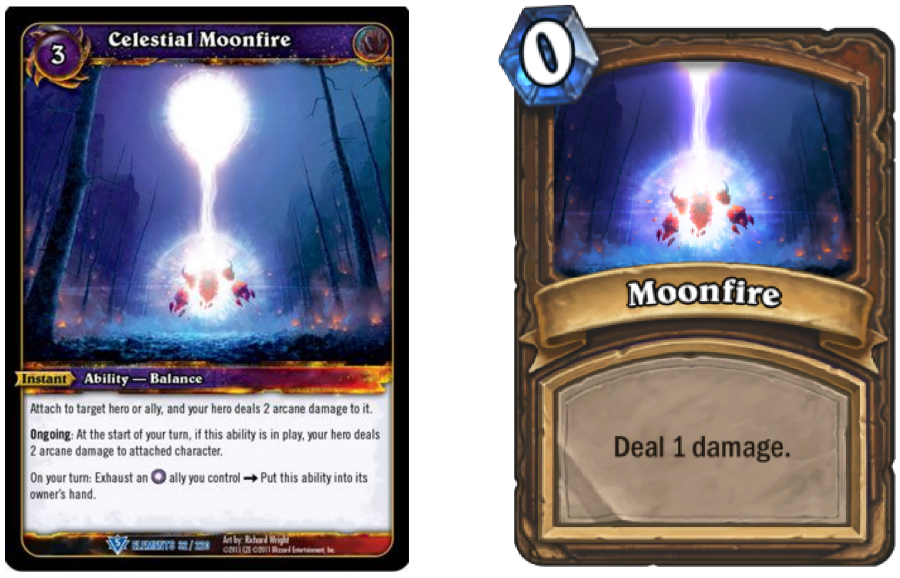
“ A journey is a succession of choices. The first choice is to start the journey. ” - Unknown author

1.1 Hearthstone

On March 2014, Blizzard Entertainment released *Hearthstone: Heroes of Warcraft*, an **online** collectible card game based on the *Warcraft* universe. Hearthstone was not the first CCG based on the franchise, as it also featured the **physical** *World of Warcraft Trading Card Game* (WoW TCG). But that physical game didn’t pan out, and was discontinued on August 2013. By contrast, Hearthstone met an incredible success compared to its physical predecessor with more than 50 million registered accounts as of April 2016 [1], [2].

Although both games are based on the same lore and share many characters, abilities, and artworks, they are highly different games. Beside the clear advantage of the digital format (allowing the game to be free-to-play), Hearthstone’s accessibility was enhanced by a simplification of the physical game rules [3]. Indeed, Hearthstone is significantly easier to play than WoW TCG (Figure 1).

During Hearthstone’s development, some features (such as the “Combat tricks”) were intentionally removed from the game in order to improve its speed. It was found that a faster gameplay made the game more fun, even at the cost of reducing strategic possibilities [4]. The resulting game is thus minimalist, with accessibility being the main reason of its ultimate success.



Attach to target hero or ally, and your hero deals 2 arcane damage to it.

Ongoing: At the start of your turn, if this ability is in play, your hero deals 2 arcane damage to attached character.

On your turn: Exhaust an ○ ally you control → Put this ability into its owner's hand.

Deal 1 damage.

(a) WoW TCG (b) Hearthstone

Figure 1: Comparison of a spell card present in both games

In Hearthstone, a deck is composed of 30 cards divided among minions, spells, and weapons.

- **Minions** are played on the board (Figure 2). They own an attack value and a health value. Once per turn, each minion can attack an opposing minion or directly the enemy hero. When a minion attacks another, they both lose health points equal to the other's attack value. After the attack, any minion with a remaining health less or equal to zero is removed from the board. If a minion attacks a hero, only that hero loses health points because heroes have no attack value. The game ends when a hero's health is reduced to zero.

- **Spells** trigger one time effects to alter the game state. There is a large variety of spell effects, from dealing damage to drawing cards. A spell card is removed from the game immediately after its effect has been resolved.

- **Weapons** give an attack value to heroes, allowing them to attack in the same way minions do. However, a hero with a weapon only deals damage when he initiates the attack. Minions can still attack an equipped hero without taking damage. In practice, weapons are less used than minions and spells.

Cards require the player to spend a magical resource called "mana". The amount of mana accessible to players increases each turn, opening the game up to increasingly expensive cards. The order in which cards are played in a single turn has crucial importance, because it can lead to different scenarios.

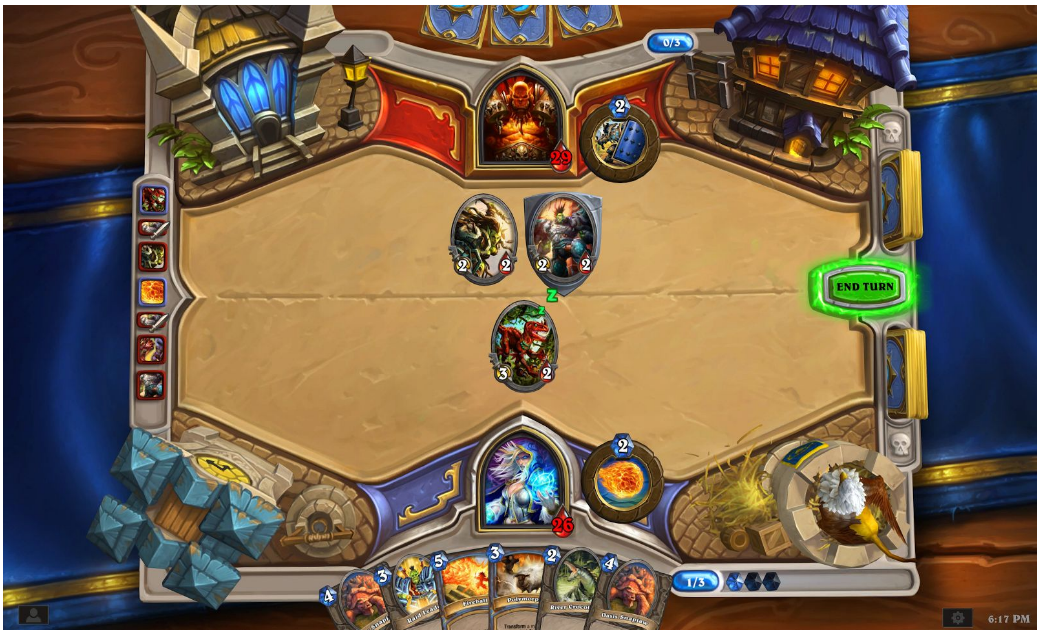


Figure 2: Hearthstone's board

The unique specificity of Hearthstone, beside the cards proper to the game, is the hero power. Before a match, players must choose a hero among 9 classes, each one giving access to a different hero power. Once per turn, a player may spend 2 mana to trigger his hero power. Hero power effects are similar to small spells: they can deal damage, restore health points, draw a card, etc. The advantage compared to a spell is that it doesn't consume a card and can thus be reused multiple times. On the other side, hero powers are much more expensive than spells with comparable effects, and only have a limited impact on the game. Hero powers were designed to spend the leftover mana and are worse than any 2-mana card [5].

Along with accessibility, Hearthstone exploits innovative game mechanics specifically designed for the digital format. Such mechanics could not fit into a physical game, because they require a third party to be resolved. In a digital game, that third party is incarnate by the game itself, allowing the cards to do pretty much anything. Examples of such mechanics include:

- **Randomness:** In physical games, random effects are highly limited because they must be easily resolvable with a coin or a dice. But the digital format doesn't suffer from this limitation since the randomness is carried out by the game. This allowed the emergence of cards like *Unstable Portal* (Figure 3a) requiring a uniform drawing out of 495 possible minions [6].

- **Uncheckable conditions:** Some cards, such as *Reno Jackson* (Figure 3b), rely on secret information. In a physical version, the card would require the player to reveal his entire deck, which violate the game's rules.

- **Physical limitations:** Some effects duplicate cards or create new cards on the fly. Cards like *Chromaggus* (Figure 3c) could not fit into a physical game for practical reasons.

Another example of digital feature is the golden cards, which are illustrated by an animation instead of a static picture. The aforementioned mechanics, among others, are the reasons Blizzard has not released a physical version of Hearthstone [7].

2
Unstable Portal
Add a random minion to your hand. It costs (3) less.
(a) Unstable Portal

6
Reno Jackson
Battlecry: If your deck contains no more than 1 of any card, fully heal your hero.
4
6
(b) Reno Jackson

8
Chromaggus
Whenever you draw a card, put another copy into your hand.
6
Dragon
8
(c) Chromaggus

Figure 3: Hearthstone’s mechanics examples

Hearthstone: Heroes of Warcraft is a perfect example of successful digital card game. By focusing on the ergonomy and taking advantage of the digital format, Blizzard developers found the perfect balance between accessibility and strategic possibilities. Hearthstone is currently ranked as one of the best digital card game [8] [9], and is rightfully described by its authors as “deceptively simple and insanely fun” [10].

1.2 Magic: The Gathering

An analysis of most influential card games can definitely not omit the very first trading card game created [11], *Magic: The Gathering*, invented by Richard Garfield and patented by *Wizards of the Coast* in 1993. Magic was the first game to combine collectible cards with dynamic rules inspired by role playing games.

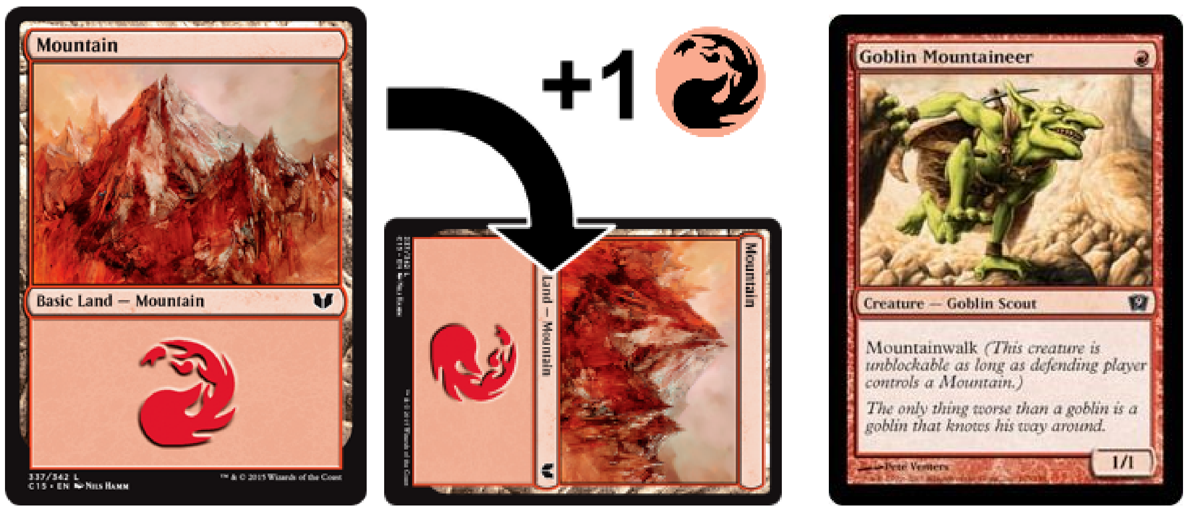
This new gaming genre was an immediate success for Wizards of the Coast. Numerous revisions followed the base game and new expansions are still released on a regular basis nowadays. The game won various awards since its creation, and was so successful that many other companies have tried to emulate it. Twenty-two years after its publication, the Magic community continues to grow with approximately twenty million players as of 2015 [12] [13].

In addition to being the first TCG, Magic also offers an unequaled diversity of gameplay by having substantially more cards and more complex rules than many other card games. With a 36 pages long "basic rulebook" [14] and over 13,651 different cards available as of January 2015 [15] [16], it is easy to get lost in the depths of its strategic possibilities. For the sake of simplicity, this review only gives an overview of Magic's rules. The reader can refer to [17] for a complete insight into the rules.

The core mechanic of Magic is referred to as "**tapping**", and involves changing orientation of a card in play to indicate use. Some actions require the player to tap a card as part of their cost. Tapping limits the number of actions that can be undertaken, because an already tapped card can not be tapped again. Since cards untap at the beginning of their controller's turn, each card can be tapped only once per turn.

Mana in Magic is produced by land cards. Lands cost no mana to play, but no more than one land per turn may be played by each player. When a land is in play, it taps to produce mana. There are five colors of mana (white, blue, black, red, and green), and one type of land for each of them. The amount of mana that must be spent to play a card, indicated in the upper right corner, is a combination of colored and colorless mana, where colorless mana can be paid with any type of land.

For example, a player plays a *Mountain* (Figure 4a), then taps that mountain to gain 1 red mana (Figure 4b), and uses that mana to summon a *Goblin Mountaineer* (Figure 4c). At the beginning of the next turn, the mountain will untap and be available for tapping again. The player will also have the possibility to play a second land, that would give him an additional mana to spend.


Mountain
Basic Land — Mountain
Land — Mountain
Mountain
+1
Goblin Mountaineer
Creature — Goblin Scout
Mountainwalk (This creature is unblockable as long as defending player controls a Mountain.)
The only thing worse than a goblin is a goblin that knows his way around.
1/1

(a) Mountain (a red land)

(b) The mountain taps to produce 1 red mana

(c) *Goblin Mountaineer* (a creature that costs 1 red mana)

Figure 4: The tapping mechanic

Creatures are characterized by two values in the lower right corner: power and toughness. Power is the amount of damage the creature deals in combat, while toughness is the amount of damage that must be dealt to the creature in a single turn to destroy it. A *Goblin Mountaineer* has 1 power and 1 toughness.

During his turn, the player has the ability to attack with the creatures he controls. The particularity of Magic is that the attacking player only declares which creatures attack. The defending player freely chooses which creature(s) he blocks with, if any. Attacking creatures that aren't blocked deal damage equal to their power to the opposing player. If an attacking creature is blocked by a defending one, they both deal damage equal to their power to each other. Any creature that is dealt damage equal to or greater than its toughness is destroyed.

That small control accorded to the defender makes all the difference, because he can either (a) block in such a way the combat results in a beneficial outcome for him (i.e. attacking creatures die and defending one survive), or (b) accept the damage and keep his creatures alive. Unlike many other card games, the best move for the attacking player may be not to attack at all.

Since attacking and defending both cause the creature to tap, a creature is not allowed to perform both actions in a single turn. As a consequence, each creature may be used either to attack during its controller's turn, or to defend during its opponent's turn. In addition, creatures can also own abilities that often require tapping. In that case, a creature would neither attack nor defend in order to use its ability during that turn.

In addition to **lands** and **creatures**, card types also include:

- **Sorceries** and **instants**: they represent one-shot spells that take effect and immediately leave the battlefield;
- **Enchantments** and **artifacts**: they represent persistent magical effects that alter some aspect of the game as long as the card remain on the battlefield. Some enchantments and artifacts are attached to other cards in play;
- **Planeswalkers**: they were introduced later into the game, and push the complexity even further by providing sets of abilities that can be activated by paying a special resource called “loyalty”.

An amazing aspect of Magic is the possibility to play cards during the opponent’s turn. When a player uses a card or an ability, it does not immediately resolve but instead is placed on a stack. The other player gets a chance to respond by playing another card on top of it that will resolve first, often countering the previous card. All players can play as many cards as they want in this way. When no one wants to play anymore, cards in the stack resolve in a *LIFO* order (Last In, First Out). Abilities may also be triggered during the opponent’s turn and added to the stack.

Magic can be played in a variety of formats, in one-versus-one or with more than two players (teams or free-for-all).

The Digital Magic

The *Magic: The Gathering* franchise gave birth to a variety of video games (see [18]). The latest title is named *Magic Duels* (Figure 5) and was released on July 29, 2015. Unlike its predecessors of the *Duels of the Planeswalkers* sequel, that were released on an annual basis, the game is free-to-play and is updated as new physical card sets are released.

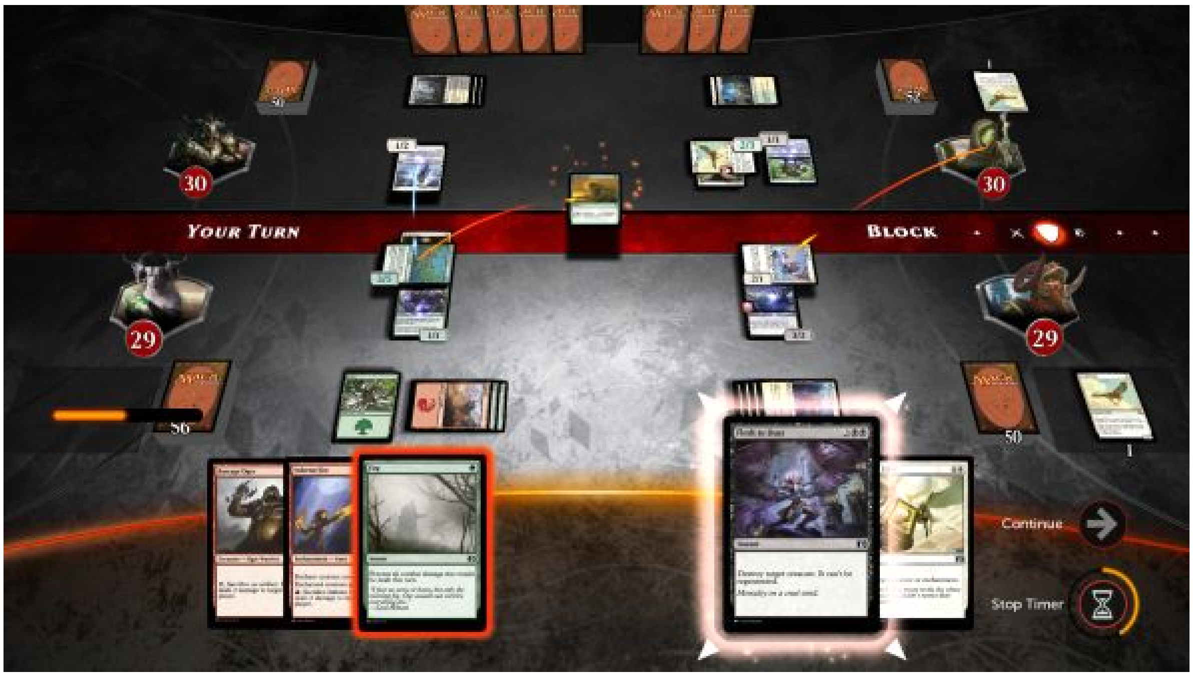


Figure 5: Magic duels

Despite the dominance of Magic in the physical CCG market, the digital versions have met mitigated success with *only* 1 million players of *Magic Duels* as of february 2016 [19] (fifty times less than *Hearthstone*). Unlike Hearthstone, Magic has not been adapted to the digital format in order to preserve the coherence between the coexisting physical and digital versions.

But some rules inherited from the physical game does not translate well for the digital format. For instance, creatures do not keep their damage from a turn to the next, because it is practically difficult to keep track of damages in the physical version. This rule is no longer justified in the digital game. Another example is the tapping mechanic, that provides a satisfying game play experience in the physical game, but loses much of its interest when the computer does it for you.

The digital game also suffers from the ability to respond to any of the opponent’s actions. Each time a player uses a card, he must wait for a timer to expire in order to give a chance to the opponent to respond. Furthermore, each turn is divided into phases and subphases, slowing and complicating the gameplay even more. That complexity and slowness can discourage newcomers and are a real barrier to the game’s accessibility.

Nevertheless, the digital game is not meant to be a standalone product but rather is a tool to drain players to the physical version. Around 25% of players who play the physical card game have come to it through one of the digital titles [20]. Although it has some serious competitors in both digital and physical markets, Magic is still the most played physical collectible card game in 2015 and remain the biggest success story of the CCG world [21].

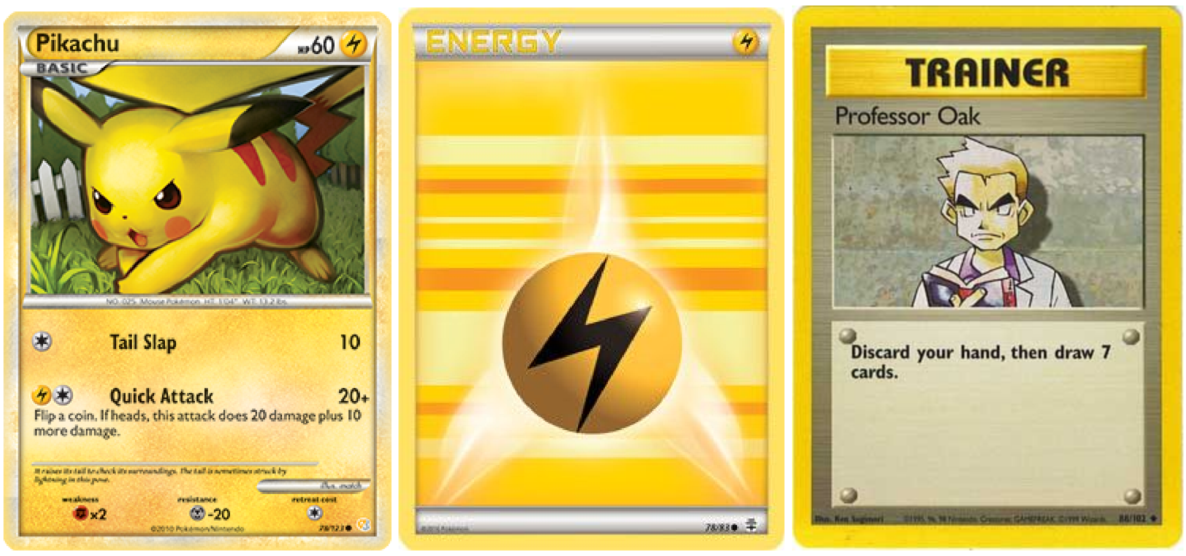
1.3 Pokémon

The *Pokémon Trading Card Game* is another physical game that have met with success. It was initially published by *Wizards of the Coast* (the same studio behind *Magic: The Gathering*) in 1996, but is controlled by Nintendo since 2003 along with the video game series. The game's popularity, enhanced by the Pokémon animated series, was so huge that Wizards sold ten times more cards than expected in the first six weeks [22].

Pokémon TCG took some inspirations from *Magic*, but is oriented toward a younger audience and uses simpler rules to make the game accessible to kids. Furthermore, the game play is not the primary concern of kids, who are mostly interested in collecting the cards rather than playing the game. "It is estimated that only about a third of people who collect the cards actually play by the official rules" [23].

The game is built around three categories of cards including Pokémon, energies, and trainers. **Pokémon** cards (Figure 6a) are the main type of cards and represent creatures. They cost nothing to play and are placed on the board. To mimic the video games, only one chosen Pokémon of each player is active at a time. Others are not part of the battle and are placed face down in a zone called the "bench".

Pokémons have a health value but no attack value. Instead, they own a few skills and attack by using one of them. In order to use a skill, a Pokémon requires to be attached to enough **Energy** cards (Figure 6b). Energy cards are similar to land cards in *Magic*, with the differences that they must be attached to a specific Pokémon and do not need to be tapped.


Pikachu
HP60
BASIC
Tail Slap 10
Quick Attack 20+
Flip a coin. If heads, this attack does 20 damage plus 10 more damage.
weakness ×2
resistance -20
retreat cost
ENERGY
TRAINER
Professor Oak
Discard your hand, then draw 7 cards.

(a) Pokémon (Pikachu) (b) Energy (Lightning) (c) Trainer (Professor Oak)

Figure 6: Pokémon TCG card types

There are 9 types of energies and each skill requires a set amount of colored or colorless (✱) energy. Like in *Magic*, colorless energies can be paid with any kind of energy.

Skills typically deal damage to the opponent's active Pokémon. When the accumulated damage equals or exceeds that Pokémon's health points, it is knocked out and replaced by another Pokémon from its owner's bench. The active Pokémon can also be intentionally replaced by paying a retreat cost. Players win after defeating six of their opponent's Pokémons, or if their opponent runs out of Pokémon.

Trainer cards (Figure 6c) are comparable to spells of other card games. They alter the game by either modifying a rule or applying an instant effect to the game state. Some trainers may be attached to a particular Pokémon, whereas others have one time effects and immediately leaves the battlefield.

In February 2015, the free-to-play *Pokémon Trading Card Game Online* (Figure 7) was released. The digital game follows the same rules than the physical one. However, the game suffers from some technical issues and is criticised for the poor quality of its graphical interface [24] [25]. Despite the initial fan base provided by the franchise, *Pokémon TCG Online* failed to conquer the digital market. "It's not as strategic or complex as *Magic: The Gathering*, nor as accessible and aesthetically pleasing as *Hearthstone*" [26].

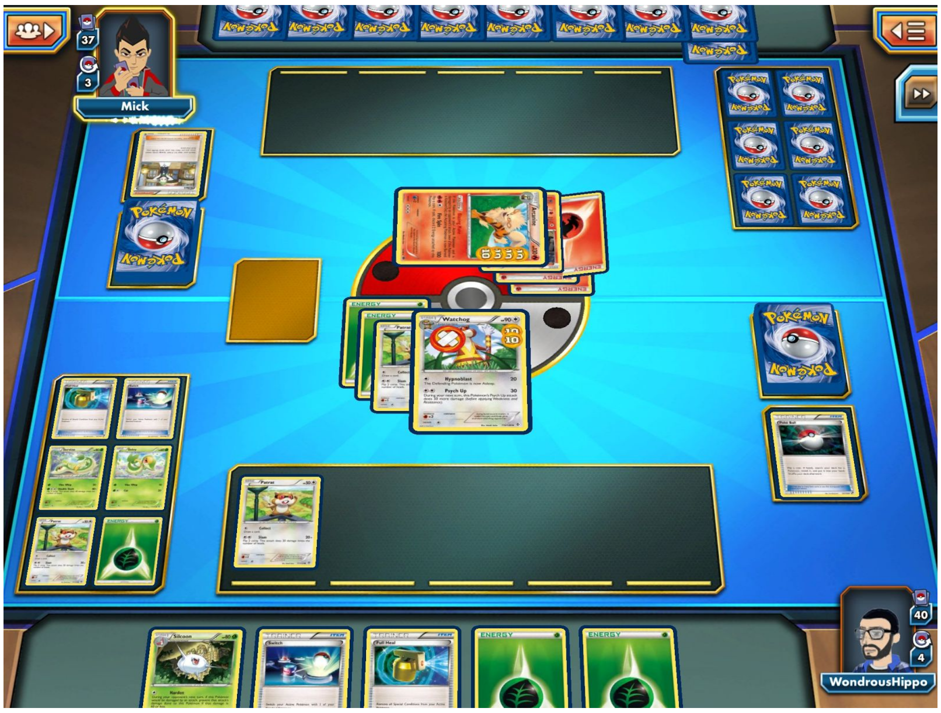
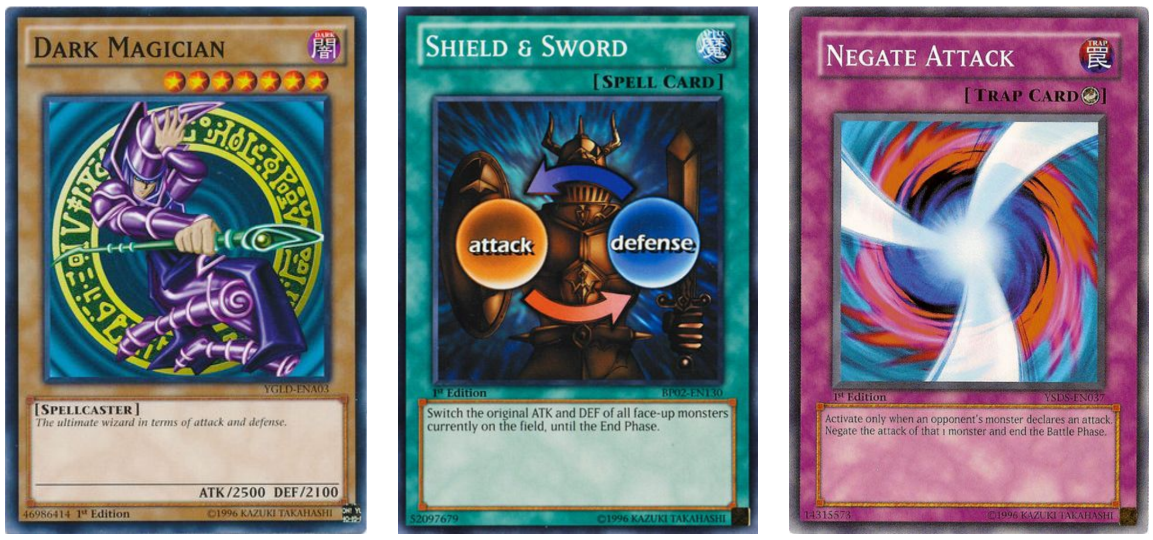


Figure 7: Pokémon TCG Online

1.4 Yu-Gi-Oh!

The *Yu-Gi-Oh! Trading Card Game* is another major actor of the physical CCG market. It was published by *Konami Digital Entertainment* in 1999 and became the best-selling collectible card game in the world in 2009 [27] [28], even surpassing *Magic: The Gathering*. Like *Pokémon*, its success was greatly enhanced by an animated series based on the same franchise.

The particularity of *Yu-Gi-Oh!* is the absence of any kind of resources. **Monster** cards (Figure 8a) possess a level indicated by the number of stars under their name. A level 4 or lower monster can be summoned to the battlefield freely once per turn. Summoning a higher level monster requires to sacrifice weaker monsters that are already on the field. Level 5 and 6 monsters need one sacrifice, while level 7 and higher need two sacrifices.


Dark Magician
[Spellcaster]
The ultimate wizard in terms of attack and defense.
ATK/2500 DEF/2100
Shield & Sword
[Spell Card]
attack
defense
Switch the original ATK and DEF of all face-up monsters currently on the field, until the End Phase.
Negate Attack
[Trap Card]
Activate only when an opponent's monster declares an attack. Negate the attack of that 1 monster and end the Battle Phase.

(a) Monster (Dark Magician) (b) Spell (Shield & Sword) (c) Trap (Negate Attack)

Figure 8: Yu-Gi-Oh! card types

Monsters can be played either in attack or in defense position, and may change their battle position each turn. The power of a monster is defined by two values, **ATK** and **DEF**, which represent its strength in attack position and defense position, respectively. Monsters in attack position are oriented vertically, whereas defending ones are turned sideway and placed face down such that their power is unknown from the opponent.

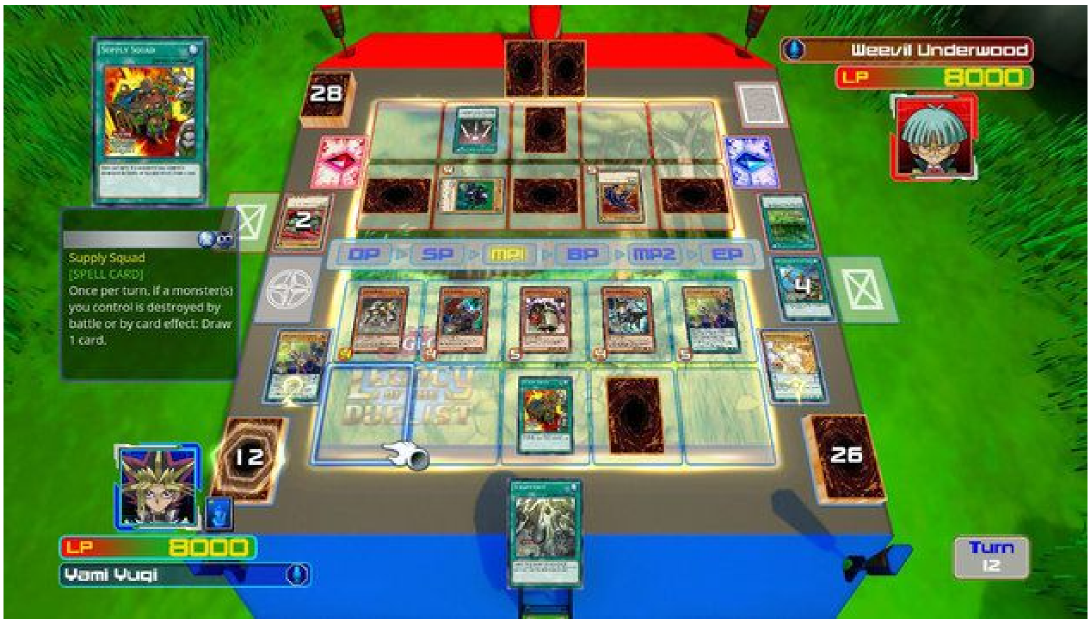
Each monster in attack position may attack once per turn by targeting one of the opponent's monsters, or directly the opponent if he has no monster. When an opposing monster in attack position is targeted, the ATK points of the attacking and the attacked monsters are compared. The monster with the lowest points is destroyed, and its owner takes damage equal to the difference between the two monsters' ATK.

When the targeted monster is in defense position, that monster is revealed and its DEF points are compared to the attacking monster's ATK. If the defending monster loses, it is destroyed but doesn't deal any damage to its owner. On the contrary, if the attacking monster loses, its owner takes damage equal to the difference of points, but the monster stays alive. Each player begins with 8.000 life points and wins by reducing the opponent's life to zero.

Yu-gi-oh! also makes use of the traditional **spell** cards (Figure 8b). These cards are played from the hand, and are divided in a few categories: *normal* spells must be played during their owner's turn outside of combat and immediately leave the battlefield after their activation, *quick play* spells can be used even during combat, *continuous* spells remain on the board until they are destroyed, *equip* spells only affect a specific monster they are attached to, and *field* spells affect both players.

The third type of cards, **trap** cards (Figure 8c), is played face down on the board and trigger in response to a specific opponent's action. Like in *Magic: The Gathering*, multiple traps and spells may be activated at the same moment by both players, and resolve in reverse order. Unlike *Magic*, traps need to be played in advance, reducing the potential interactivity.

The Yu-gi-oh! card game was emulated by a myriad of digital incarnations (see [29]). The latest title is *Legacy Of The Duelist* (Figure 9) and was released in July 30, 2015. Since most of Yu-gi-oh! video games are exclusively available on consoles, they are not directly competing with *Hearthstone* and *Magic Duels*. However, *Konami* announced the release of an online game based on the *Yu-gi-oh!* franchise for both PC and consoles scheduled in winter 2016 [30]. That future title may be a serious competitor for already established digital card games.



Supply Squad
[SPELL CARD]
Once per turn, if a monster(s) you control is destroyed by battle or by card effect: Draw 1 card.

Figure 9: Yu-Gi-Oh! Legacy Of The Duelist

1.5 Discussion

Our analysis of the CCG market revealed a large variety of game play. While most card games share similar components, each of them brings small changes into the genre that create a game experience different from the competitors.

Some successful digital card games emerged from an existing physical game. But the coexistence of both physical and digital versions is rather a constraint than an advantage. Although the digital game can benefit from the fan base already established by the physical one, they require to keep the same rules for the sake of coherence. That's an issue because mechanics designed for a physical game aren't necessarily suitable for a digital game.

In particular, digital card games suffer from complexity and meet greater success when the game play is simpler and faster. Reducing the complexity of the game rules is an effective way to enhance a game's accessibility for newcomers. However, the appropriate trade off between accessibility and strategic possibilities remain difficult to identify, as an oversimplified game may lose much of its interest in the long term. For that reason, digital card games often make use of a learning curve, in which players are gradually exposed to increasingly complex aspects of the game play.

Another specificity of digital card games is their ability to exploit mechanics that are not possible in physical games, or would require a judge. Online games can take advantage of such mechanics to provide a gameplay experience that rises above games based on physical rules. Indeed, some adaptations of physical games even introduced a few digital-only cards specifically designed for their digital version.

Lastly, ergonomy and polishing are crucial to ensure a game's ultimate success. These qualities can be the main reasons of success, as in *Hearthstone*, and even predominate over the gameplay itself. Conversely, a neglected graphical interface may ruin the potential of a successful franchise like *Pokémon*.

	Hearthstone	**Magic: The Gathering**	**Pokémon TCG**	**Yu-Gi-Oh!**
Format	Digital only	Physical & digital	Physical & digital	Physical & digital
Access (digital version)	Free-to-Play	Free-to-Play	Free-to-Play	Pay-to-Play
Players (both formats)	50 Millions [1] [2] (April 2016)	20 Millions [12] (October 2015)	Unknown	Unknown
Complexity	Low	High	Low	Medium

Table 1: Card games comparaison

Chapter 2
Game Design

“ If you wish to make apple pie from scratch, you must first create the universe. ” - Carl Sagan

This chapter describes the process behind the creation of the game. The pursued objective was to create a card game that is at the same time innovative and fun to play. To achieve that objective, the various topics encompassing the essence of card games are reviewed. For each of them, the common weaknesses of current card games are identified, and a solution is proposed and justified.

2.1 Questioning the Basics

The card games reviewed in Chapter 1 share a common point: they are turn-based. This analysis starts by investigating the relevance of turn-based systems.

Turn-based Advantages

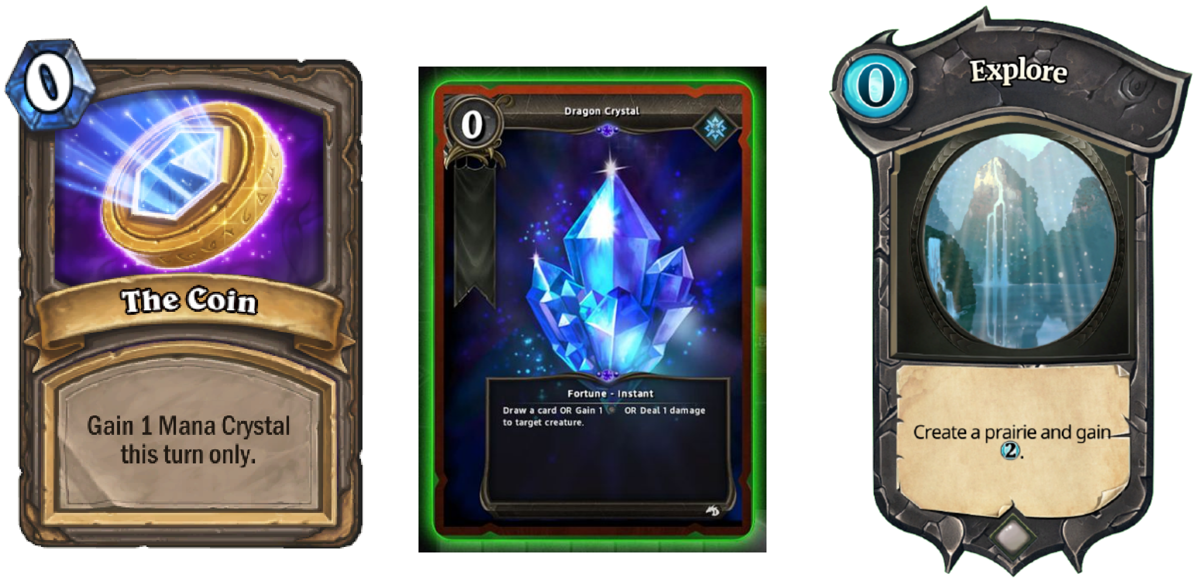
Turn-based games have some advantages. Firstly, they give time to players to think and make appropriate decisions. Turn-based games can be seen as a succession of puzzles that require to be solved. That problem solving is one of the main components that make card games fun.

Secondly, turns allow the introduction of advanced game mechanics. For example, Magic’s tapping mechanic permits each minion to either attack during its controller’s turn or to defend during its opponent’s turn. Turns also constitute an intuitive temporal unit for recurrent actions, generally happening at the start of each turn, such as drawing a card or gaining resources.

Thirdly, since players are acting one after the other, the game’s leader usually alternate each turn, producing satisfying little victories for both players. Eric dodds, game director on *Hearthstone*, argues that “little victories are in a lot of ways more important for the player than the final outcome of a game” [31]. Indeed, the player feels much better after a game punctuated by a few comebacks even if it ended up on a loss.

Turn-based Drawbacks

On the other side, turn-based card games also suffer from some issues. The first issue is the fact they confer a slight advantage to the first player to go, because that player has repeatedly access to a given amount of resources before the other player can answer with the same resources. For instance in *Hearthstone*, the starting player was 20% more likely to win during alpha despite an extra card given to the second player [32] [33]. As a consequence, the second player usually starts with a unique card specifically designed to counter-balance that disadvantage (Figure 11).


0
The Coin
Gain 1 Mana Crystal this turn only.
0
Dragon Crystal
Fortune - Instant
Draw a card OR Gain 1 OR Deal 1 damage to target creature.
0
Explore
Create a prairie and gain 2.

(a) Hearthstone (b) Duel of Champions (c) Faeria

Figure 11: Balance cards given to the second player in various games

A second issue is the lack of interactivity between players due to the inability to act during the opponent's turn. A few physical games, such as *Magic: The Gathering*, ingeniously allow the player to instantly react to his opponent under certain circumstances. However, such a mechanic considerably slows down the game, and has been shown to be unsuitable for a digital game. The digital *Hearthstone* originally allowed to play a card on the opponent's turn, but that feature was canceled and replaced by "secrets", a weaker form of the same mechanic [34].

A third issue is the loss of unused resources between turns. Since they do not accumulate, players are encouraged to spend all the resources available on each turn in order to minimize the amount lost. As a consequence, players have limited control on the moment they play a card, because consuming all the resources may be more rewarding than playing the cards at the appropriate moments. Furthermore, this rule increases the influence of luck on the game, as the player's ability to spend the exact amount of resources available is restricted by the cards in his hand. Recently, we have seen the development of a new card game, namely *Faeria,* that accumulate resources between turns.

Lastly, a fourth frequent complaint about card games is their overall slowness. Turns are typically divided into phases, or even subphases, that horribly slow down the gameplay. For that reason, card games tend to be disregarded by gamers in favor of faster games.

Proposed Solution

The highlighted weaknesses, which are inherent to turn-based games, led us to question the presence of turns in a video game. Originally, card games were designed around turns because they were meant to be played with physical cards. However, this concept is no longer required in a digital game, as the flow of the game can be controlled by the computer.

In this work, we explored the idea of creating a real-time card game. That is, as opposed to turn-based games, players would be able to act simultaneously. This small - but ambitious - change causes huge repercussions on the game rules. Hence, the remainder of this chapter explains how the fundamental card game rules were adapted to the real-time space.

2.2 Resources

Card games are generally based around a resource system controlling the flow of the game. This system balances the cards strength by adapting the amount of resources needed to play the cards.

One may ask if resources are essential to card games. A few games (e.g. *Yu-gi-oh!*) use no resource at all, by simply allowing the utilisation of a single card per turn. However, the absence of resources has two main disadvantages. Firstly, it is not possible to combine two or more cards simultaneously when players are limited to one card at a time and, secondly, the individual strength of every card must be almost equivalent as differences of power can't be balanced by resources. For these reasons, we included resources in the created game.

Resources Production

Resources can take various forms, including cards (e.g. *Magic: The Gathering, Pokémon TCG*). In that case, they are part of the player's deck and drawn in the same way creatures and spells are. Although players have total control over the proportion of resources constituting their deck, a well-known optimized proportion is usually used in practice (which is around 40% of the total deck size). The main issue with this type of resources is the randomness in the number of resources drawn. In particular, an opening hand with too few or too many resources may induce a disadvantage for the player concerned.

In other card games, usually digital ones (e.g. *Hearthstone*), resources are gained automatically. Generally, the amount of resources available increases each turn, opening the game up to increasingly powerful cards. We adopted such an automated resource production, because it ensures that both players get the exact same amount of resources. Since there are no turns anymore in a real-time game, the resources are produced continuously, similarly to a real-time strategy game. Players can use a card as soon as they own the required resources.

Resources Types

Whereas simplest card games are based on a single resource type, others, such as *Magic: The Gathering*, are built around several colors of resources. In the latter case, each card in the available pool cost a combination of colored resources. In this work, we used 6 resource types symbolizing schools of magic: *Fire*, *Frost*, *Nature*, *Arcane*, *Light*, and *Shadow*. The types of resources generated automatically depend on the hero incarnated by the player (see section 2.4).

Often, the number of cards drawn from the deck is highly limited and, therefore, also considered a resource. In traditional turn-based card games, the player draws a card at the beginning of each turn. By contrast, in a real-time game, there are no turns anymore.

Instead, drawing cards cost a special resource, called “**devotion**”, that is produced continually in addition to card resources. Every time a player gains a *devotion*, he can draw a card.

2.3 Deck Building

Decks used by players to compete against each other are fully customized by their respective owner. Players build their deck by selecting a limited number of cards from a large pool.

Deck Segmentation

The card pool is habitually segmented into distinct categories, sometimes referred to as *classes, factions*, or *colors*, that are each specialised into a particular archetype of decks. This segmentation is crucial to ensure the coexistence of a wide diversity of decks.

In most basic games, the cards selection is restricted to a single chosen class (or faction), as seen in *Hearthstone* or *Duel of Champions*. However, since the tactical options available inside a single class are often too limited, each class is generally dominated by a few competitive decks used by everyone. As a consequence, what is supposed to be customization is actually rather deck *selection* than deck *creation*. In addition, as an unwanted side effect, players may not be able to incorporate a newly acquired card into their deck, because it may belong to a different class.

By contrast, we do not restrict the cards selection to a single class, but rather allow the player to combine any number of colors in his deck (as seen in *Magic: The Gathering*). Whereas increasing the number of colors in a deck unlocks more strategic possibilities, on the other side, it also reduces the consistency of play (because it reduces the probability of drawing cards of the resource type(s) currently available). In practice, a viable deck does not include more than 2 colors. With this system, any card can be combined with any other, while still preserving the diversity of decks.

Card games often feature neutral cards, that are generic colorless cards whose cost can be paid with any type(s) of resource. When playing such a card, games with different resource colors thus ask the player to select the combination of resources to use. The advantage of neutral cards is the fact they can be included in any deck regardless of its color(s). However, a real-time game may not leave the time to choose among the many potential ways to pay a cost. Instead, in the game we created, neutral cards cost *devotion* (the resource used to draw cards) as a kind of colorless resource. Some neutral cards may also cost no resource at all.

Deck Size

Another question is the number of cards allowed in a deck. In existing card games, deck size varies between thirty (*Hearthstone*) and sixty (*Magic: The Gathering, Pokémon TCG*). The number of copies of the same card that can be included in a deck is also limited for reasons of diversity. The limit ranges from 2 (*Hearthstone*), sometimes 3 (*Yu-gi-oh!*), up to 4 copies

(*Magic: The Gathering*). Unusual cards may be labeled as “Unique” or “Legendary” to indicate only one copy of it is allowed in a deck.

The game created in this work currently imposes no limits on the deck size nor the number of copies of each cards in a deck. This allows the user to add newly acquired cards without removing any other, as well as testing cards interactions easily (by bringing a small deck). As examples, three pre-constructed decks containing around thirty cards are given.

2.4 Hero

Players are traditionally represented by an entity with health points, referred to as a *Hero* or *Champion*. The game ends when a player’s health is reduced under zero. Whereas most card games consider the hero as being just a punching bag, some allow him to interact in various ways with the other components of the game.

In *Hearthstone*, the choice of a hero gives access to hero-specific cards and determines a unique hero power available to the player. Once per turn, a player may spend 2 mana to trigger that hero power, whose effect is similar to a small spell. In addition, the hero is also able to perform attacks, in the same way minions do, by equipping a weapon that gives him attack points.

Following a radically different system, *Magic: The Gathering* introduced Planeswalkers, a card type subject to particular rules that represent an allied hero. Planeswalkers are not able to attack directly, but instead own a few skills, and may use one of them per turn. However, they remain distinct from the player itself by owning a separated health value, and can be defeated without its owner losing the game. Unlike *Hearthstone*, Planeswalkers are part of the deck and can thus be customized.

In the game we created, heroes also have a central role that directly affects the gameplay. A hero is represented by a card, and a valid deck must contain exactly one chosen hero card. Just like other cards, heroes are collectible, and may modify a particular game’s rule. For instance, a hero may reveal the top card of its owner’s deck, or slowly regenerate health with time. Unlike other cards, heroes are not shuffled in the deck but rather start the game immediately on the board. The game ends when one of the two heroes has been defeated.

Although a hero initially starts with 0 attack points, he may be targeted as any creature by spells and effects increasing his attack value, providing him the ability to attack. Some heroes also own a special power that may be triggered by paying a *devotion* cost.

The choice of a hero determines which resources are produced automatically. A hero may produce one or several resource types, and the production speed of each type may differ. Furthermore, each resource color is not restricted to a single hero, but rather offers several potential heroes with different strategies.

2.5 Protecting & Positioning

Among the various card games available, a large variety of systems defining how creatures attack and defend themselves co-exist. There are, in fact, almost as many ways to defend creatures as there are card games.

In *Magic: The Gathering*, the attacker declares which creature attacks, while the defender freely chooses, on the attacker's turn, which one of his own creatures will bloc. Despite the fairness of the system, it is incompatible with a real-time game as the continuous actions would not leave time to the defender to perform his choice.

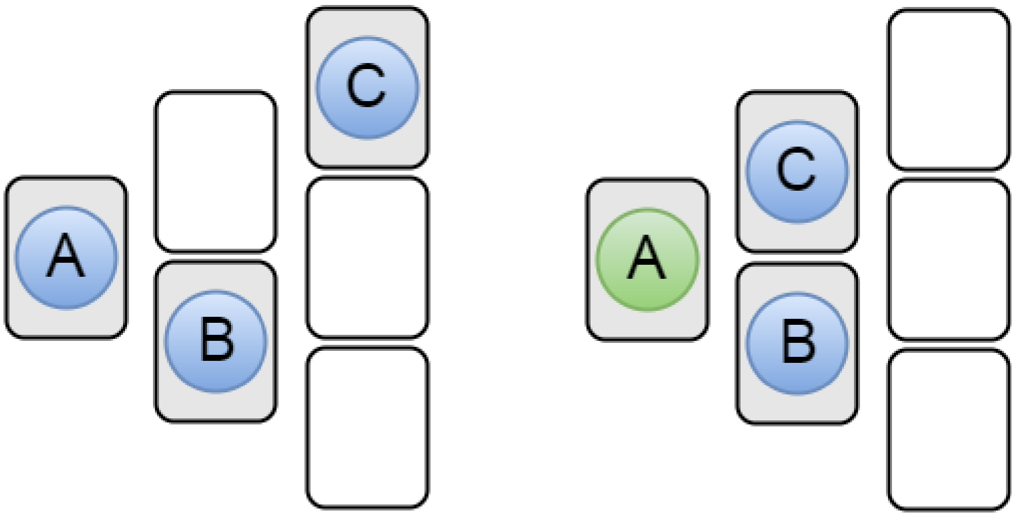
A far distinct approach was adopted by *Hearthstone*. In this game, a minion may own the "Taunt" keyword, an ability that forces enemies to eliminate creatures with *Taunt* before attacking the other creatures or the hero. The use of a keyword successfully removes any action required by the defender, but loses at the same time the opportunity to choose the blocker. As a consequence, a creature with *Taunt* can't be protected and, symmetrically, creatures without *Taunt* have no way to be used as protectors.

In *Yu-gi-oh!*, a player can't be targeted as long as he owns at least one monster. In a sense, the game behaves as if every monsters had the *Taunt* keyword. But that only protects the player, while usually we would like to defend particular minions that are more vulnerable than others.

This is partially achieved by *Pokémon TCG*. In this game, only one pokémon is active at a time, and can be retreated to be replaced by another. Still, this is unsatisfactory, because creatures are not able to interact with each other when there is only one of them active at a time.

Some other card games, such as *Duel of champions*, take into account the cards position on the game board. The battleground is divided into a few rows, where minions are placed in, and a minion can only attack the front opponent in the same row. Players can thus efficiently protect any arbitrary creature by placing another in front of it. Positioning also has the advantage of enabling cards with particular areas of effect (e.g. spells targeting a specific row, minions boosting adjacent allies).

Coupled with a real-time dimension, positioning adds even more tactical depth, by allowing players to move their creatures at any moment. However, the grid configuration is already largely exploited by many other games (e.g. *SolForge*, *EarthCore*). Instead, we adopted a triangular formation (Figure 12), a slight variation where two creatures are needed in front of a third to protect it. Notice in Figure 12b that creatures B and C can themselves be protected by placing in the same way two creatures in front of them. The position of a creature doesn't affect its capacity to attack the opponents.



(a) None of the creatures are protected. (b) Creature A is protected by B and C.

Figure 12: Triangular formation

We already mentioned that the hero is himself a card comparable to minions. As a consequence, he occupies one of the triangle's slot and can move like any other minion does. This also gives him the ability to protect another minion.

2.6 Simplicity

Card games have the rightful reputation of being complicated and intimidating for newcomers. Hence, a major challenge when designing a card game is to make it accessible to as many people as possible. Unnecessary complexity should be avoided because a person who doesn't understand the game is not able to play.

A typical source of complexity in card games lies in the division of turns into phases, each one restricting the kind of actions that can be done. Thanks to the real-time space, such complexity was avoided by allowing players to perform any action in any order.

Complexity can also appear in cards description. Ideally, players should be able to understand instantly a card's effect by reading it once. This is particularly needed in a real-time game, where the time constraint forces players to read cards quickly. For that reason, we pushed the complexity to the interactions between cards, rather than the cards themselves, by limiting cards to one or two sentences. Frequent mechanics were also replaced by keywords to further reduce descriptions length.

2.7 Randomness

Another recurrent complaint about card games involves the presence of too much randomness. Some games directly include luck into the cards themselves, like *Pokémon TCG*, using mechanics such as flipping a coin or rolling a die. By contrast, other games are based on highly complex but deterministic rules such as *Magic: The Gathering*. In both cases, a random factor inherent to card games is the order in which the cards are drawn.

It is easy to fall into the misconception that randomness is opposed to skill. According to Ben Brode, lead game designer on Hearthstone, “Randomness can actually increase the amount of skill required to play the game in some scenarios” [35], [36]. Indeed, randomness challenges the problem solving capabilities of players by bringing a lot of stochasticity into a game. The same arguments were advanced by Magic’s creator Richard Garfield [37]. Both game designers came to the conclusion that, despite luck’s influence on the game’s outcome, better players still win more on average.

Randomness also opens the door to mind-blowing scenarios. *Hearthstone* is a good example of card game in which incredibly fun stories can happen due to random effects. Since randomness ends up being challenging and fun, the game we created **does** include some random effects.

2.8 Game Art

As in every video game, the user interface must be consistent and attractive to ensure the success of the product. This subsection exhibits the images created for the game, and justifies how the information is displayed on screen. A card game requires the design of two main components: the cards and the game board.

Cards

A unique template was created for each card’s color (Figure 13) to easily identify the type of resources required by a card. As a potential future improvement, slightly different templates could also be used to differentiate minions from spells.

(a) Fire (b) Frost (c) Nature (d) Light

Figure 13: 4 card templates (out of 6)

Cards were designed by taking into consideration that they are not totally visible when held in hand (Figure 14). In practice, players tend to identify cards by their illustration rather than their actual name. Hence, the card's name is written in the middle (instead of the top) in order to increase the portion of the illustration that can be seen from a card in hand. The cost of the card, which is a crucial information, is located in the top-left corner, the only one visible when the card is in hand. In game, the totality of a card can be displayed by placing the mouse over that card.

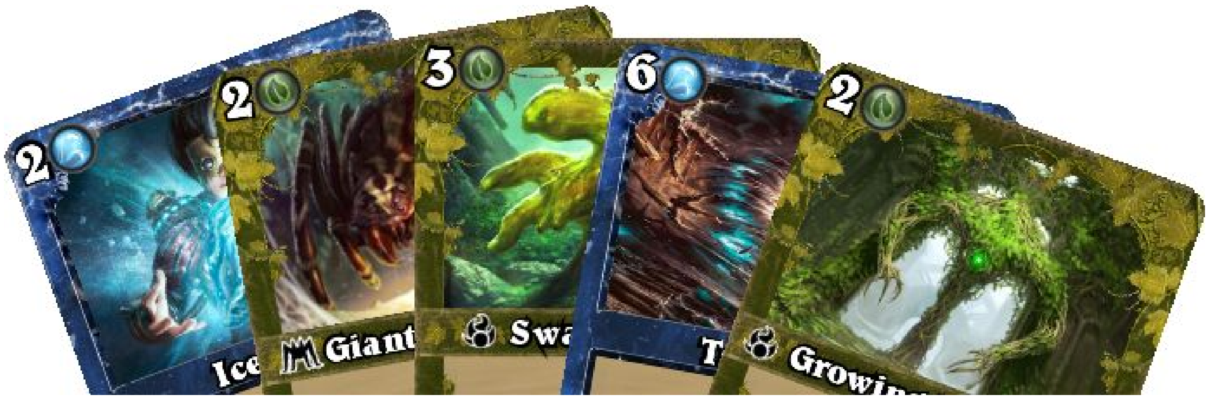


Figure 14: A hand with 5 cards as seen in-game.

Board

Most digital card games are oriented vertically, with the local player's cards displayed at the bottom of the screen and the opponent's ones at the top. This is possible because the usual absence of positioning allows them to align each player's creatures on a single line. By contrast, the triangular positioning we adopted requires creature cards to be displayed on distinct lanes. We thus opted for a large horizontal board (Figure 15), in which players are positioned on the left and right of the screen.

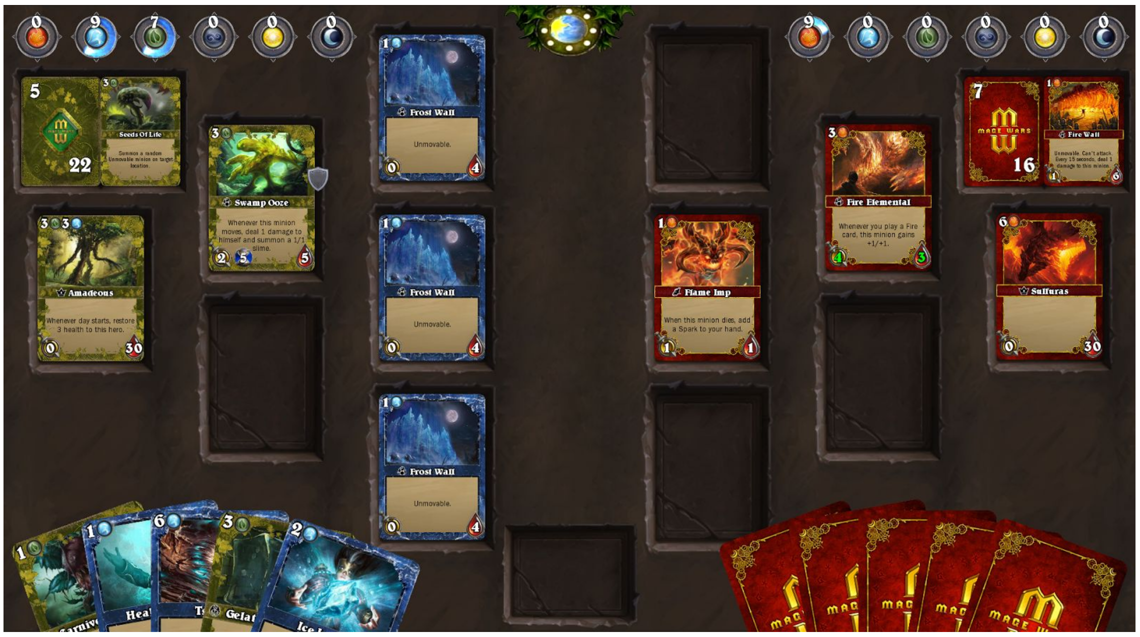


Figure 15: Game board

Chapter 3
Implementation

The game is implemented in the java language. The source code is composed of 254 files and is available on the Matheo platform (Master Thesis Online) as an annex to this paper. Neither a game engine nor an external library was used.

The implementation is divided into 5 layers: the network layer, the control layer, the logical layer, the model, and the graphical user interface. In the following explanations, bold words denote class names.

3.1 The Network Layer (src/mana/network)

The network layer ensures the communication between the clients and the server. Its purpose is to allow upper layers to ignore the difficulties of synchronisation and to function as if the game was running as a single application.

Multiplayer Mode

Network communications are built around the ***NetworkListener*** class, which is a *Runnable* class able to send and receive *Serializable* objects through a socket. It is used on both client and server sides, such that any communication is performed by two *NetworkListener* interacting with each other.

The ***Server*** waits for clients to connect over the network, and instantiates a ***Player*** for each client connected. A *Player* is a *NetworkListener* that ensures the communication with a particular client.

On the other side, the ***Client*** connects to the server immediately when it is launched, via an instance of *NetworkListener*. The communication between the two sides is performed by serializing ***Event*** objects. Each event is implemented in a dedicated class that defines its impact on the application (Table 2).

General events		
Name	**Direction**	**Description**
MatchmakerJoined	Client -> Server	Adds the player to the matchmaker.
MatchmakerLeft	Client -> Server	Removes the player from the matchmaker.
GameStart	Server -> Client	Indicates an opponent has been found.
GameLeft	Server -> Client	Indicates the opponent left the game.

Table 2: General events

When connected, a user has the capacity to join the ***Matchmaker***. The *Matchmaker*'s purpose is to store players until an opponent is available to play against. When two players have joined the matchmaker, they are removed from it and linked by a new instance of ***Game***, which represents the association between those players. A single server can handle multiple games at the same time. The creation of a game is summarized by the sequence diagram of Figure 16.

During a game, the user triggers events through the graphical interface. In-game events have the particularity of being bidirectional (called ***PingPongEvent*** in the source code). When such an event is triggered, for example when the user draws a card, the client transmit it to the server but does not change the local game state yet. The server then processes the event (in this example, it determines which card has been drawn) and retransmit the event to all players in the game, including the originator one. The event, now containing the card drawn, is ultimately executed on all clients to modify the game state.

Table 3 contains the possible events occurring during a game. The same event class is transmitted in both directions to avoid the duplication of each event into a request and a response. The handling of in-game events is depicted by the sequence diagram of Figure 17.

In-game events		
Name	**Direction**	**Description**
CardDrawn	Client -> Server Server -> Client	Indicates the user drawn a card from his deck. Adds that card from the deck to the user's hand.
CardPlayed	Client -> Server Server -> Client	Indicates the user played a card from his hand. Removes that card from the user's hand and play it.
CardMoved	Client -> Server Server -> Client	Indicates the user moved a creature on the battlefield. Moves that creature to its new location.
CardDiscarded	Client -> Server Server -> Client	Indicates the user discarded a card from his hand. Puts that card from the user's hand to his graveyard.
AbilityCast	Client -> Server Server -> Client	Indicates the user cast a creature's ability. Applies the effects of that ability.

Table 3: In-game events

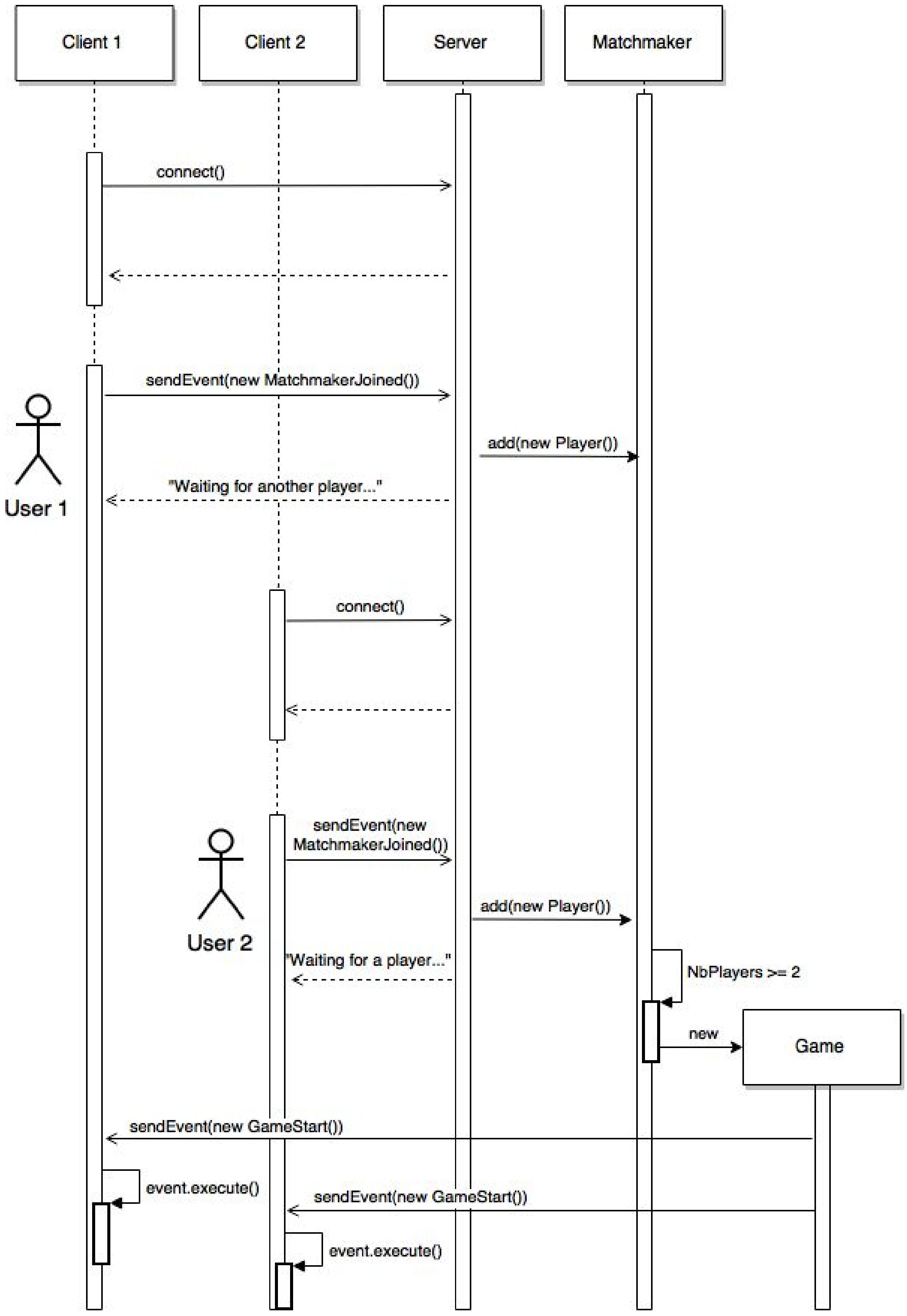

Client 1
Client 2
Server
Matchmaker
connect()
sendEvent(new MatchmakerJoined())
add(new Player())
"Waiting for another player..."
User 1
connect()
sendEvent(new MatchmakerJoined())
User 2
add(new Player())
"Waiting for a player..."
NbPlayers >= 2
new
Game
sendEvent(new GameStart())
event.execute()
sendEvent(new GameStart())
event.execute()

Figure 16: Sequence diagram of a game's creation.

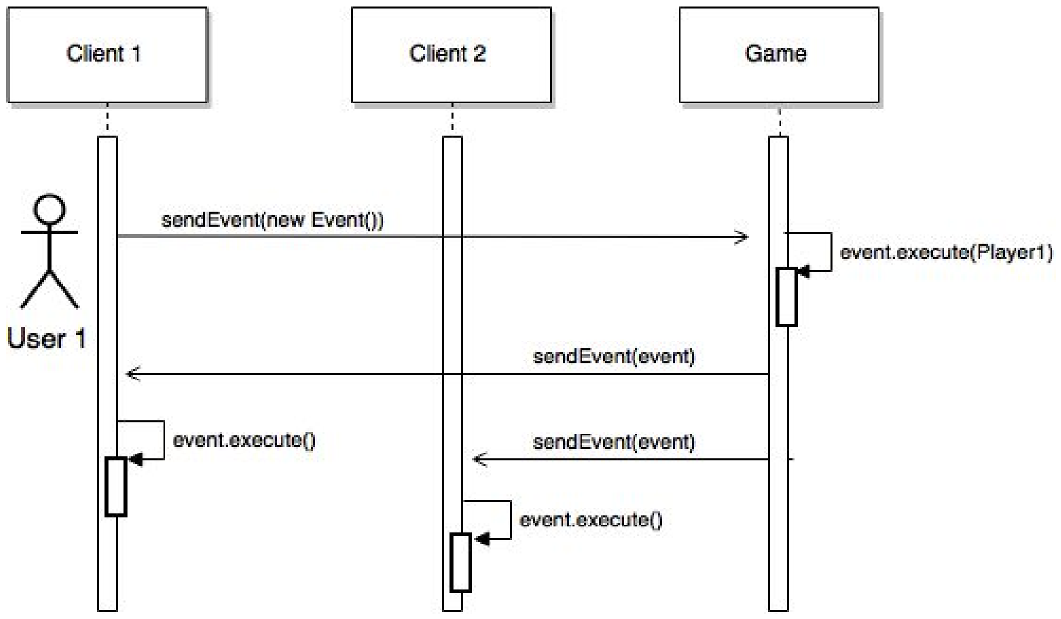


Figure 17: Transmission of an in-game event

Single-player Mode

The single-player mode allows a human player to compete against one of the two artificial players, and is accessible through the main menu via the “Solo” button. This game mode was implemented without modifying the game logic designed for the multiplayer mode. In particular, it still uses a server, but locally, allowing the game to be played without a network connection. From the implementation point of view, a single-player game is nothing but a particular multiplayer game in which the server and both players are on the same device.

The creation of a single-player game is composed of three steps. Firstly, when the user presses the *solo* button, a local server is started in a new thread. That server will ensure the synchronization between the two players and monitor the game updates.

Secondly, an artificial player is launched in a new process. Artificial players are represented by the abstract ***ArtificialPlayer*** class, that is extended to define concrete implementations of players, such as the ***HeuristicPlayer*** and the ***SimulatingPlayer***. When an AI player is started, it uses its own *Client* to connect to the local server, and emulate a request to join the matchmaker. Since artificial players operate in another process, they own a different process ID than the human player that makes them distinguishable for the server.

Thirdly, the original client used by the human player connects to the local server and joins the matchmaker at its turn. Because two players joined the matchmaker, the server starts a new game. The creation of a single-player game is summarized by the sequence diagram of Figure 18.

During a game, artificial players use controllers from the control layer (see 3.2) to listen to the game and be notified of events. An artificial player’s client can be either shown, to observe the game from its point of view and/or intervene with its actions, or on the contrary hidden, saving a substantial computational effort as most CPU time is spent by the graphical interface. It is also possible to increase or decrease the game speed using the *Time* controller.

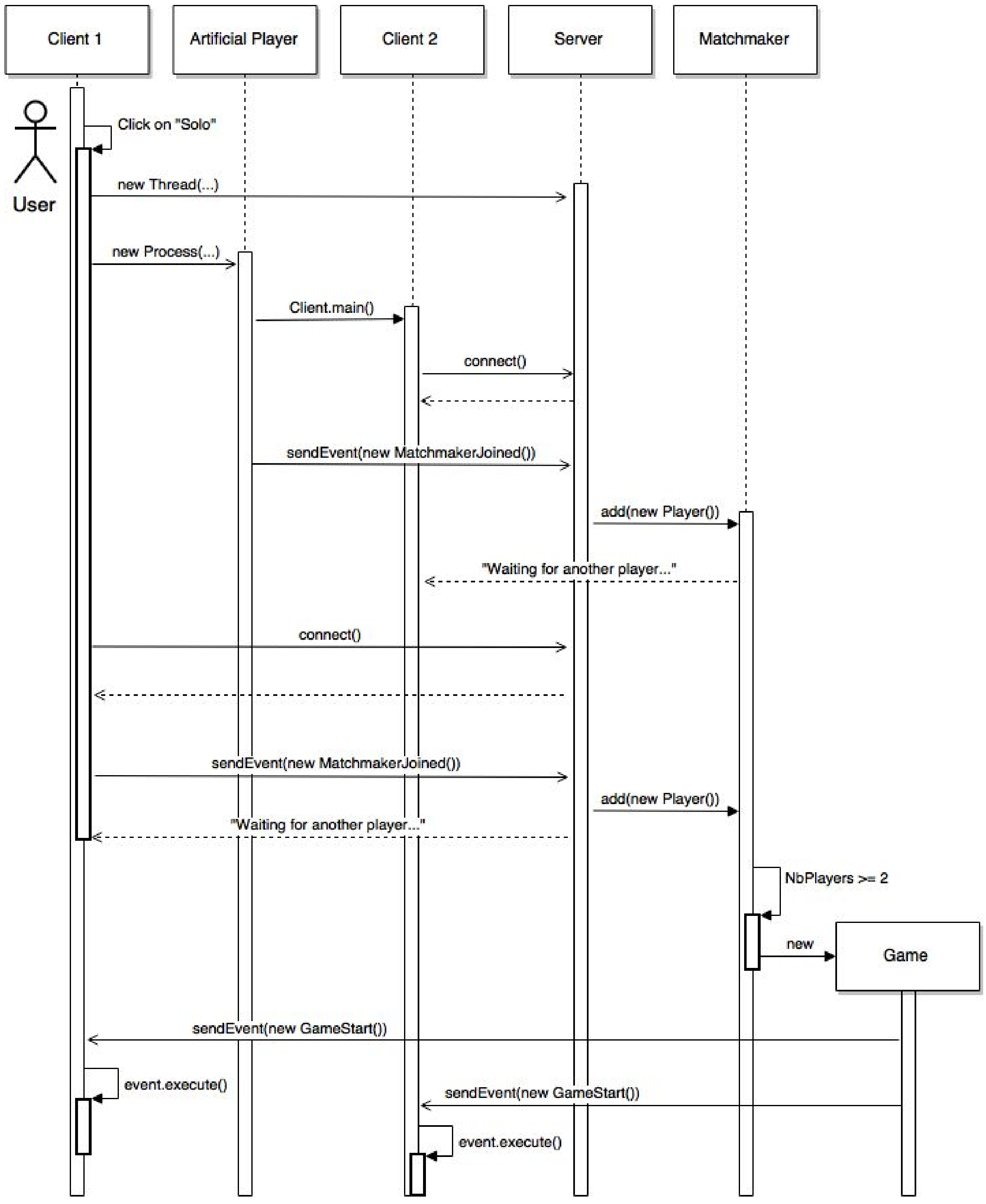


Figure 18: Creation of a single-player game

3.2 The Control Layer

Controllers manage the various events, from user inputs to game events, and dispatch them to any object interested in processing them. An object can register to a controller by implementing the corresponding interface, and calling the controller's specific register method causing the listener to start receiving events. When an event relevant to a controller occurs, all listeners are notified in the order they registered.

Mouse Inputs

Mouse inputs are generated by the ***Mouse*** class and transmitted through a custom ***MouseListener*** interface (not to be confused with *java.awt.event.MouseListener*). Whereas regular java's *MouseListener* requires each listener to examine the given *MouseEvent* to determine which button has been pressed, this custom interface distinguishes itself the left mouse button from the right, and also generates mouse motion events.

Objects interested in mouse events must implement the *MouseListener* interface, and define a *isMouseOver(int x, int y)* method that returns true if the point (x;y) is part of that object. Events occurring are only transmitted to the listeners that are under the current mouse location (i.e. *isMouseOver* returned true), such that listeners do not need to check whether the mouse is over them when they received an event. The implementation of this method is usually straightforward, as most clickable objects are rectangular.

A *MouseListener* object can then register to mouse events using *Mouse.addMouseListener*(*MouseListener* l), causing the *Mouse* to start transmitting events by invoking the relevant methods of the listener (Table 4). The listener can stop receiving events by using *Mouse.removeMouseListener*(*MouseListener* l).

Methods	**Description**
Boolean isMouseOver(int x, int y)	Returns *true* if the point (x;y) is over this listener. Mouse events are only transmitted when this method returns true.
Void mouseMoved(int x, int y)	Invoked when the mouse has been moved to coordinates (x;y).
Void mousePressedLeft(int x, int y)	Invoked when the left mouse button has been pressed at coordinates (x;y).
Void mousePressedRight(int x, int y)	Invoked when the right mouse button has been pressed at coordinates (x;y).
Void mouseReleased(int x, int y)	Invoked when any mouse button has been released at coordinates (x;y).

Table 4: The *MouseListener* interface

Time Steps

The ***Time*** controller, and the corresponding ***TimeListener*** interface (Table 5), control the rate of recurrent actions by sending time steps at specified intervals to listeners.

An object can register to time steps by using *Time.addTimeListener*(*TimeListener* listener, int delay) and specifying the delay in milliseconds between the notifications. Listeners are free to register at any time, and typically do it directly in their constructor, but signals only starts being produced when the game begins. This ensures every game object is started at the same time, regardless of the moment they registered.

Since all game objects are animated by the *Time* controller, the global game speed can be adjusted by a single constant. It is also possible to pause the whole game easily by making the *Time* controller stop sending notifications.

Examples of *TimeListeners* include the mana bar producing resources, the animation of objects moving or rotating fluently, the control of the day-night cycle, as well as the triggering of some card effects.

Method	**Description**
Void actionPerformed()	Invoked every time the delay specified at registration has elapsed.

Table 5: The *TimeListener* interface

Game Events

The ***GameListener*** abstract class controls events relative to the game rules. The class contains a set of empty *Void* methods (Table 6) that are each invoked when a particular game event occurs. An object can listen to the game by extending the *GameListener* class and overriding the methods of interest. Methods not overridden remain empty and thus do not trigger anything.

A *GameListener* can then starts receiving game events by using *GameListener.addGameListener*() and stop receiving them with *GameListener.removeGameListener().* For example, a card typically starts listening the game when it enters play, and stop listening when it leaves the board.

The reason *GameListener* is a class to extend rather than an interface to implement is twofold. Firstly, some objects need to override the *addGameListener()* or *removeGameListener()* methods to add particular operations immediately before or after they start (or stop) listening the game. Secondly, the *GameListener* class include a large number of methods and an interface would require to declare each of them, whereas generally only a few are needed. Using a class allow to set once the default action as doing nothing.

Methods	**Description**
gameStarted()	Invoked when the game starts.
devotionGained(Player, Integer)	Invoked whenever a player gains devotion.
manaGained(Player, School, Integer)	Invoked whenever a player gains mana.
manaSpent(Player, List<School>)	Invoked whenever a player spends mana.
cardDrawn(Card)	Invoked whenever a card is drawn.
cardAddedToHand(Card)	Invoked whenever a card enters a player’s hand (a card may enter a hand without being drawn).
cardPlayed(Card)	Invoked whenever a card is played.
cardDiscarded(Card)	Invoked whenever a card is discarded.
spellCast(Spell)	Invoked whenever a spell is played.
creatureSummoned(Creature)	Invoked whenever a creature is played.
creatureGrabbed(Creature)	Invoked whenever the player grabs a creature.
creatureMoved(Creature, Location)	Invoked whenever a creature moves.
creatureDamaged(Creature, Integer)	Invoked whenever a creature takes damage.
creatureHealed(Creature, Integer)	Invoked whenever a creature is healed.
creatureDead(Creature, Location)	Invoked whenever a creature dies.
abilityReady(Ability)	Invoked whenever an ability is ready.
abilityCast(Ability, Location)	Invoked whenever an ability is used.
eventPlayed(Event)	Invoked whenever an event card is played.
dayStarts	Invoked whenever the day begins.
nightStarts	Invoked whenever the night begins..

Table 6: The *GameListener* class

Audio

The ***AudioPlayer*** controller triggers the lecture of sound effects and musics. It is a *GameListener* that associates an audio file to each game event of Table 6. Another object can also trigger a sound by using *AudioPlayer.play*(String filename). Since every sound is played by the AudioPlayer, the overall audio volume can be adjusted by a single constant.

3.3 The Logical Layer (src/mana/logic)

The logical layer implements the game rules and is responsible for updating the game state according to them.

A naïve way to represent the cards is to implement each of them individually, by coding every time exactly what the card is supposed to do. However, considering a card game usually contains several hundreds of cards, this would lead to redundancy as some effects are reused by multiple cards. For example, when dealing damage, we must check if the target is still alive and remove it from play if it is dead. Writing these operations for every single card dealing damage would result in redundancy and a lot of unmaintainable code.

For that reason, an approach based on card fragments was appropriated. A large number of basic blocks are implemented one for all in separated classes that can be instantiated to be used by multiple cards. Complex cards can then be built by combining basic blocks together. For example, the card “Deal 2 damage to a chosen target.” is composed of the three basic blocks “Deal damage”, “2” (the amount of damage dealt), and “a chosen target”. They are respectively called the ***Effect***, the ***Parameter***, and the ***Target***.

Effects

The effect determines the nature of the alteration. There is a large variety of effects implemented (Table 7), such as gaining resources, drawing a card, summoning a minion, moving a creature, modifying a creature’s attack points, or changing the game time.

An effect is created by extending the abstract *Effect* class, and implementing a *trigger*() method that define the set of operations performed by that effect. Some also implement a *cancel*() method to revoke that effect, whereas others are irreversible. Each card is associated to an instance of *Effect*. When a card is played, the associated *Effect* is triggered.

Particular effects			
And	Triggers multiple effects at the same time.		
Repeat	Triggers an effect a specified number of time.		
PlaySound	Plays a sound from the specified audio file.		
Atomic effects			
Name	**Target**	**Param.**	**Description**
AddToHand	Player	Card	Adds the specified card to target player's hand.
Buff	Creature	Integer	Increases target creature's attack and health.
DealXDamage	Creature	Integer	Deals the specified amount of damage to target creature.
Destroy	Creature	-	Removes target creature from the battlefield.
DestroyEvent	-	-	Removes the current event card from the battlefield.
Discard	Card	-	Removes target card from its owner's hand.
GainTrait	Creature	Trait	Adds the specified trait to target creature.
GainType	Creature	Type	Adds the specified type to target creature.
GainXAttack	Creature	Integer	Increases target creature's attack by the specified amount
GainXDevotion	Player	Integer	Gives the specified amount of devotion to target player.
GainXHealth	Creature	Integer	Increases target's current health and maximum health.
GainXMana	Player	Integer	Gives the specified amount of mana to target player.
Move	Creature	Location	Moves target creature to the specified location.
ReduceCost	Card	Integer	Reduces target card's cost by the specified amount.
Replace	Creature	Creature	Destroys target creature and summon the specified one.
RestoreXHealth	Creature	Integer	Restores the specified amount of health to target creature.
RestoreFullHealth	Creature	-	Restores target creature to full health.
Reveal	Card	-	Returns target card face up.
SetNight	-	-	Sets game time to night.
SetTime	-	Integer	Sets game time to the specified one.
Summon	Location	Creature	Puts a creature into the battlefield on target location.

Table 7: List of atomic effects

Parameters

A parameter is an additional piece of information specific to an effect. For example, when dealing damage, one must specify the amount of damage dealt.

Parameters are represented by the abstract *Parameter<T>* class, that possesses a *get()* method returning an object of type *T*. When an effect triggers, it invokes the *get()* method of its parameter, and resolves with the returned value. Parameters are specific to each effect, and are determined by the effect's constructor. The *DealXDamage* effect, for instance, requires a *Parameter<Integer>*, i.e. a parameter with a *get()* method that returns an integer.

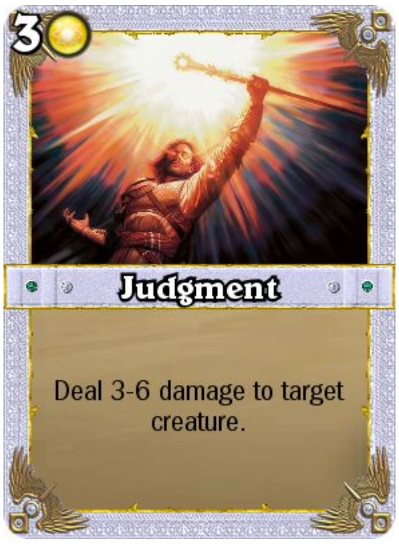
The interest of representing a parameter by an object is the ability for its *get()* method to return a different value each time it is invoked. For example, the card *Judgment* (Figure 19a) is implemented as

```
Effect e = new DealXDamage(new Range(3,6), target);
```

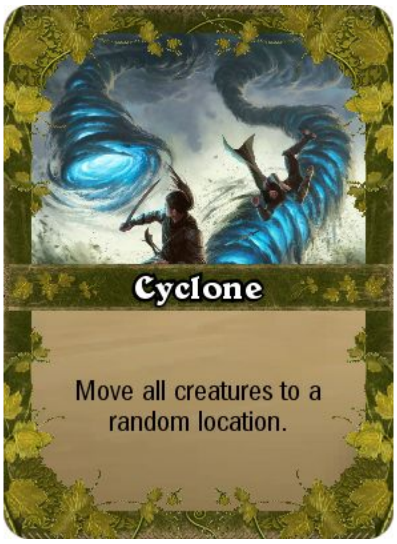
where *Range* is a *Parameter<Integer>* whose *get()* method returns a random integer between two specified bounds. Since the parameter's *get()* method is invoked each time the effect triggers, the value returned may differ from a call to another.

A parameter may also represent a value that is not known in advance, because it will depend on the game state at the moment the effect triggers. Consider for instance the card *"Deal damage equal to the number of cards in your hand."*. When that card will be played, the parameter will dynamically determine the amount of damage based on the current game state, and return that value to the *Effect* object.

Since the *Parameter<T>* class is generic, a parameter can represent any type of object (not necessarily an integer). For example, the *Move* effect (Figure 19b) takes a *Parameter<Location>* to determine where a creature is moved to.



(a) *Parameter<Integer>*



(b) *Parameter<Location>*

Figure 19: Parameters examples

Targets

The target determines the object(s) on which an effect is applied. It is, in fact, a particular case of parameter.

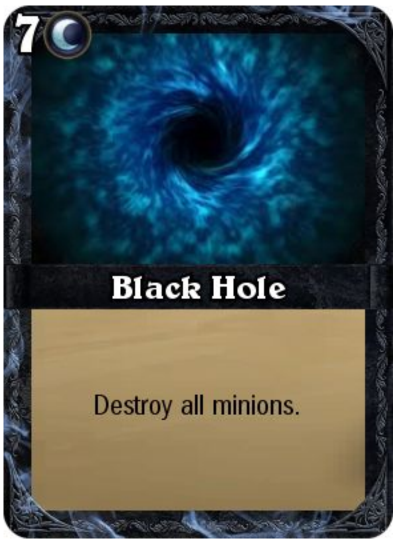
A target is represented by the *Target<T>* class, which is a specialisation of *Parameter<List<T>>*. Whereas regular parameters return a single *T*, a target returns a list of *T*. When an effect triggers, it invokes its target's *get()* method, and successively resolves for each value in the list returned.

Consider for example the card *Black Hole* (Figure 20a). It is implemented in the following way:

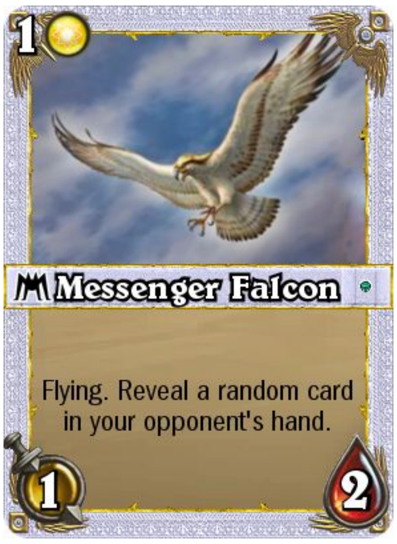
```
Effect e = new Destroy(new AllMinions());
```

In this trivial example, *AllMinions* is a *Target<Creature>* whose *get()* method returns the list of all minions currently on the board (i.e. creatures that are not heroes). When *Black Hole* is played, the *Destroy* effect triggers, invokes the *get()* method of *AllMinions,* and resolves on each element of the list returned, eventually destroying each minion in play.

A target can of course represent a single object. In that case, the list will contain only one element. Similarly to parameters, the genericity of targets allow them to represent any type of object (not necessarily a creature). For example, the *Reveal* effect (Figure 20b), an effect that turns a card face up, takes a *Target<Card>* to determine which card(s) to reveal. Table 8 gives additional examples of targets.



(a) *Target<Creature>*



(b) *Target<Card>*

Figure 20: Targets examples

Target<T>	
Card.target	The object on which the card is played.
Random	A random value among the list of another target.
Target<Player>	
Player	A specific player.
EachPlayer	Both players.
Target<Card>	
AllCards	All cards in the game.
AllCreatureCards	All creature cards in the game.
AllMinionCards	All minion cards in the game (i.e. creatures that are not heroes).
AllSpellCards	All spell cards in the game.
CardsInHand	All cards in a player’s hand.
Clone	A clone of a card (conserving instance values).
NewInstance	A new copy of a card (initialised via a constructor).
RandomCardInHand	A random card in a player’s hand.
RandomMinion	A random minion card.
RandomSpell	A random spell card.
TopDeckCard	The card at the top of a player’s deck.
Target<Creature>	
AllCreatures	All creatures in play.
AllMinions	All minions in play (i.e. creatures that are not heroes).
AllOtherCreatures	All creatures in play except the specified one.
AllOtherMinions	All minions in play except the specified one.
AnotherRandomCreature	A random creature in play, excluding the specified one.
RandomCreature	A random creature in play.
RandomMinion	A random minion in play.
Target<Location>	
AllLocations	All locations of both players.
EmptyLocations	All empty locations.
FrontLocations	The three locations at the front of each side.
RandomEmptyLocation	A random location that is empty.

Table 8: List of targets.

Filters

Filters are a particular type of card fragment that allow to refine the targets subject to an effect based on a specific condition.

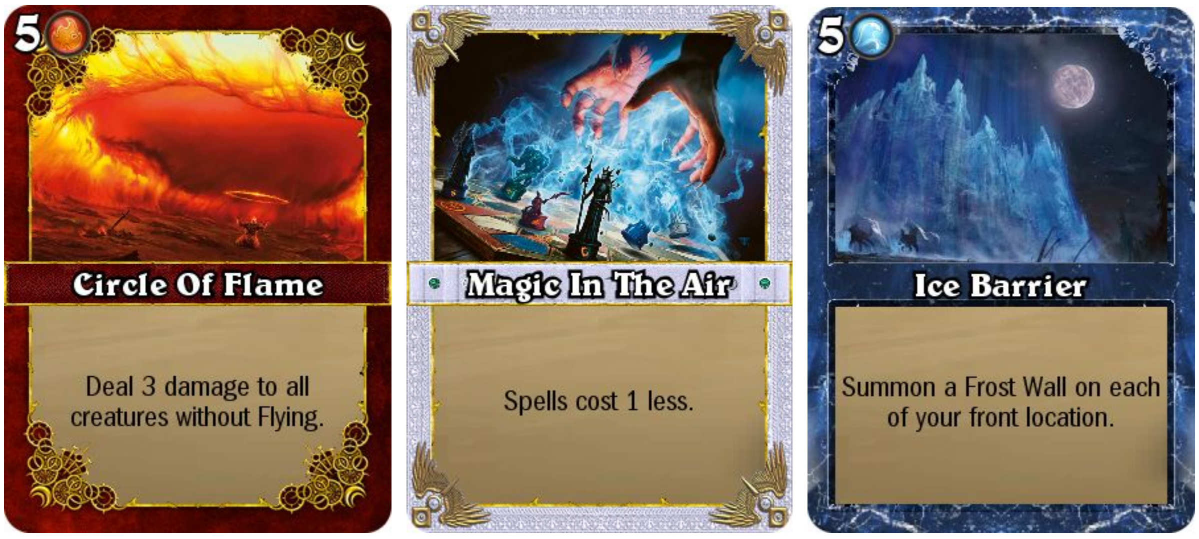
A filter is implemented through the ***Filter<T>*** interface. This interface contains a *is(T)* method that returns true if the given *T* meets the filter's specific condition. When a *Target* is initialised with a filter, the objects that do not meet the filter's requirement (i.e. *is(T)* returned false) are removed from the target list. Complex targets can thus be created by combining the right filters.

Consider for example the card *Circle of Flame* (Figure 21a) that does not target flying creatures. It is implemented by combining a *AllCreatures* target with a *Trait* filter (in this case, the *Flying* trait):

```
Effect e = new DealXDamage(3, new AllCreatures(new Not<>(Trait.Flying)));
```

In this example, *Trait* is an enumeration (of every existing trait) that implement the *Filter<Creature>* interface. *Trait.is(Creature)* returns true for the creatures owning that particular trait. The filter *Trait.Flying* therefore only conserves flying creatures. However, in this particular case, we negated the filtering (to exclude the flying creatures instead) by introducing an additional *Not<T>* filter between the target and the *Trait* filter (without the *Not*, only flying creatures would be targeted).

Since the *Filter<T>* interface is generic, it can be used to filter any type of objects. Figure 21 gives some examples of filters. The card *Magic in the Air* (Figure 21b) uses a *Filter<Object>* based on the objects' classes to discriminate spell cards from others. The card *Ice Barrier* (Figure 21c) uses a *Filter<Location>* to select locations at a specific position. Table 9 gives additional examples of implemented filters.


5
Circle Of Flame
Deal 3 damage to all creatures without Flying.
Magic In The Air
Spells cost 1 less.
5
Ice Barrier
Summon a Frost Wall on each of your front location.

(a) *Not<>(Trait.Flying)* (b) *ClassFilter(Spell.class)* (c) *PositionFilter(3, 4, 5)*

Figure 21: Cards using filters

Filter<T>	
And(Filter<T>, Filter<T>)	Combine two filters to form a logical AND.
Not(Filter<T>)	The inverse of another filter.
Filter<Object>	
ClassFilter	Objects of a specific class.
EqualsFilter	Objects equal to another specific object (via *Object.equals(...)*).
Filter<Card>	
Player	Cards belonging to a specific player.
School	Cards belonging to a specific school of magic.
Rarity	Cards of a specific rarity.
Filter<Creature>	
Type	Creatures of a specific type.
Trait	Creatures owning a specific trait.
DamagedFilter	Creatures that are damaged.
Filter<Location>	
EmptyFilter	Locations that are empty.
PositionFilter	Locations at specific position(s).

Table 9: List of filters

Conditional Effects

The effects introduced so far are fairly limited. Indeed, they can only trigger a predetermined action, while ideally a card's effect should be able to adapt depending on the game state or particular events. To provide such behaviors, we introduced effects that manipulate other effects by controlling the way they trigger.

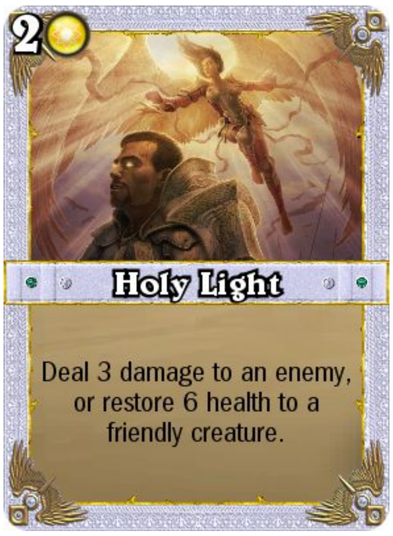
A conditional effect is one that triggers only if a specific condition is met, and optionally triggers another effect in the other case. It is equivalent to a logical *if then else*. The effect is implemented by the ***If<T>*** class that takes an object of type *T*, a condition in the form of a *Filter<T>*, and one or two effects. When a conditional effect triggers, the specified filter is evaluated on the given *T*. If the condition is true, the first given effect is triggered. Otherwise, if a second effect was specified, that other effect is triggered instead.

An example is given by Figure 22a. The effect of the card *Holy Light* is implemented as

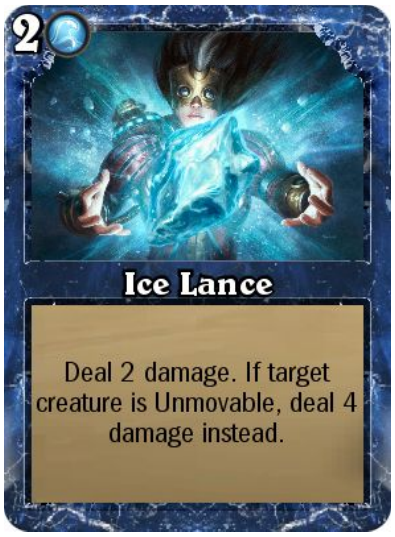
```
new If<>(target, opponent, new DealXDamage(3, target), new RestoreXHealth(6, target));
```

The first argument is the object on which the condition is evaluated. In this case, "target" refers to the object on which the card is used. The second parameter is the condition to evaluate. In this case, "opponent" refers to a *Filter<Card>* able to discriminate cards belonging to a specific player. The third and fourth arguments are the effects to trigger if the condition is respectively true or false.

The overall effect can be red as "If target is an opponent, deal 3 damage to target. Else, restore 6 health to target.". Note, however, that the logical description of a conditional effect is usually rewritten in an intuitive human-friendly language.


2
Holy Light
Deal 3 damage to an enemy, or restore 6 health to a friendly creature.

(a) Using a *Player* filter


2
Ice Lance
Deal 2 damage. If target creature is Unmovable, deal 4 damage instead.

(b) Using a *Trait* filter

Figure 22: Examples of conditional effects

Triggered Effects

A triggered effect is a card that has a trigger condition and an effect. Whenever that trigger condition is met, the associated effect is executed.

Triggered effects are implemented by the abstract ***Whenever<T, X>*** effect, that takes another effect as argument. When such a card is played, the *Whenever* object starts listening to the game via the *GameListener* controller. Each time the event of interest is raised by the *GameListener*, the *Whenever* object triggers its inner effect. When the card is destroyed, the *Whenever* object stop listening to the game to no longer trigger the effect. The event in question is specific to each implementation of *Whenever*. A list is given in Table 10.

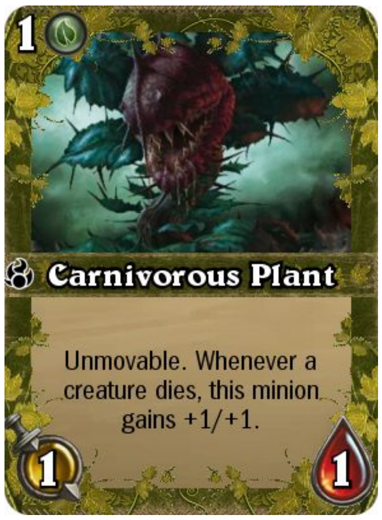
As an example, *Carnivorous Plant* (Figure 23a) is implemented in the following way:

```
Effect e = new WheneverACreatureDies(new Buff(this, 1, 1));
```

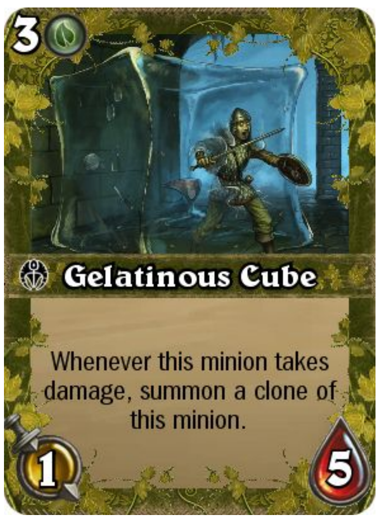
In this example, `WheneverACreatureDies` is a specialisation of *Whenever<T, X>* that overrides the *creatureDead()* method of *GameListener* to trigger the specified effect (in this case, a *Buff* effect).

A *Whenever* can be associated to any number of *Effects*. When the event occurs, every effect is triggered in the order they were given.

Triggered effects are compatible with filters, that can be used to refine the condition upon which the effect triggers. A *Whenever<T, X>* effect can take up to two filters, one for the target of the condition (a *Filter<T>*) and one for the parameter (a *Filter<X>*). For instance, the *Gelatinous Cube* (Figure 23b) uses a *Filter<Creature>* to restrict the triggering to "this minion". Without the filter, the effect would trigger when any creature takes damage.


1
Carnivorous Plant
Unmovable. Whenever a creature dies, this minion gains +1/+1.
1
1

(a) No filter


3
Gelatinous Cube
Whenever this minion takes damage, summon a clone of this minion.
1
5

(b) With a filter ("this minion")

Figure 23: Examples of triggered effects

Name	Filter 1	Filter 2
CreaturesAttackedBy	Creature	Creature
CreaturesDamagedBy	Creature	Ability
WheneverAbilityCast	Creature	Ability
WheneverACreatureDies	Creature	Location
WheneverACreatureMoves	Creature	Location
WheneverACreatureTakesDamage	Creature	Integer
WheneverDayStarts	-	-
WheneverNightStarts	-	-
WheneverYAttacks	Creature	-
WheneverYDrawACard	Player	Card
WheneverYPlayACard	Player	Card
WheneverYSpendMana	Player	School

Table 10: List of triggered effects

Ongoing Effects

An ongoing effect is an effect that persists as long as the source card remains in play. Whereas regular effects trigger only once (when the card is played) and remain indefinitely, ongoing ones also affect targets entering play afterwards, and vanish as soon as the source card leaves play.

Ongoing effects are implemented by the ***OngoingEffect*** class that takes another effect as argument, the one to dispense continuously. The implementation is based on multiple *Whenever*:

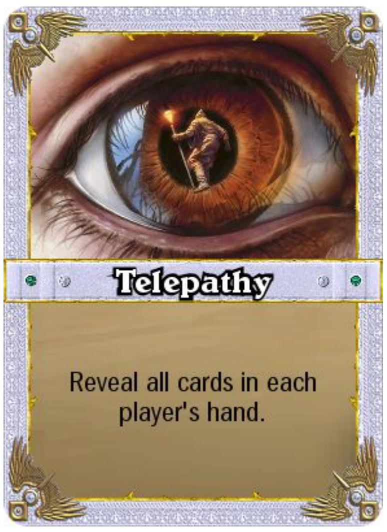
1. When the source card is played, *trigger*() the effect on every valid target in play.
2. *Whenever* a valid target enters play, *trigger*() the effect on that target.
3. *Whenever* a valid target leaves play, *cancel*() the effect on that target.
4. When the source card leaves play, *cancel*() the effect on every remaining valid target.

As an example, the card *Telepathy* (Figure 24a) is defined in the following way:

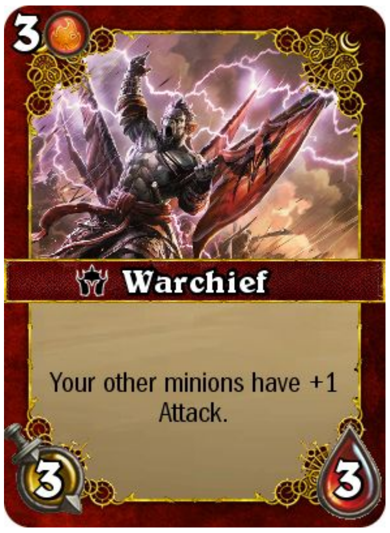
```
Effect e = new OngoingEffect(new Reveal(new CardsInHand()));
```

In this example, the card continuously reveals players hands. The effect thus *trigger()* on each card entering a player's hand and *cancel()* on each card leaving a hand.

An ongoing effect can also target creatures in play, such as the *Warchief* (Figure 24b). In that case, the effect triggers whenever a creature is summoned, and is canceled whenever a creature dies. Again, filters can be used to refine the objects affected by a continuous effect. For example, the *Warchief* uses a *Filter<Creature>* to affect exclusively friendly minions.



(a) Effect on cards in hand



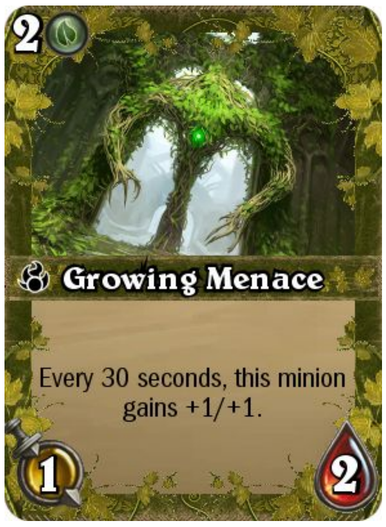
(b) Effect on creatures

Figure 24: Examples of ongoing effects

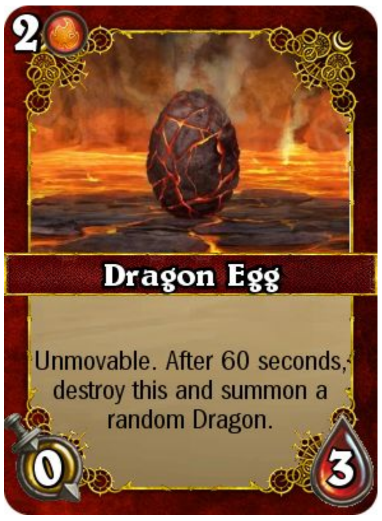
Temporal Effects

A temporal effect triggers at regular time intervals. They can be seen as the real-time equivalent of “*Each turn*” or “*At the start of your turn*” that can be found in traditional turn-based card games.

Temporal effects are implemented by listening to the *Time* controller. The ***EveryXSeconds*** class (Figure 25a) takes an effect as argument and triggers it every time a specified delay has elapsed. A slight variation, the ***AfterXSeconds*** class (Figure 25b), triggers the inner effect only once, by unregistering from the time controller the first time it is notified.



(a) Every *x* seconds


2
Dragon Egg
Unmovable. After 60 seconds, destroy this and summon a random Dragon.
0
3

(b) After *x* seconds

Figure 25: Examples of temporal effects

A summary of the logical layer is provided by the class diagram of Figure 26.

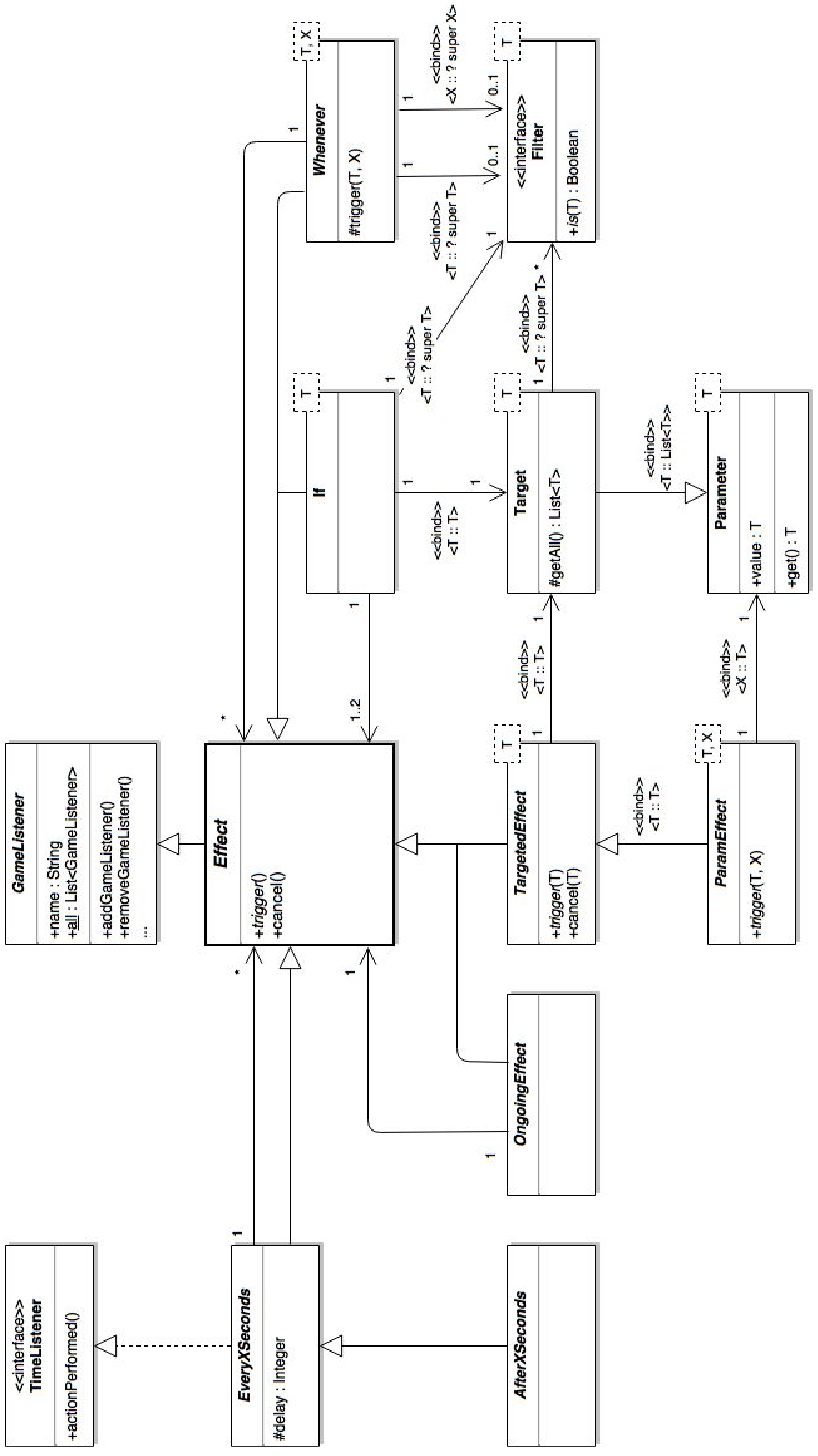


Figure 26: Class diagram of the main logical components

3.4 The Model Layer (src/mana/cards)

The model layer represents the game content, which is mainly constituted of the list of all cards.

Cards

A card is represented by the abstract ***Card<T>*** class, whose parameterized type *T* is the type of object on which the card can be used. Each card is associated to an *Effect*, that is triggered when the card is played. If a card is dropped on an object that is not of type *T*, that card automatically returns to where it came from, most of the time in the player's hand (though a few cards may be used from the graveyard).

The abstract *Card<T>* class implements the general behavior that is common to every card. It is extended by abstract card types (i.e. ***Creature***, ***Spell***, and ***Event***) to implement specificities of each type. Card types are further extended to implement concrete cards.

The list of all cards is built automatically at game launch, based on the class files in *\bin\mana\cards*. As a consequence, cards like *Wild Magic* (Figure 27) exploiting directly this list may generate newly added cards without requiring any modification.

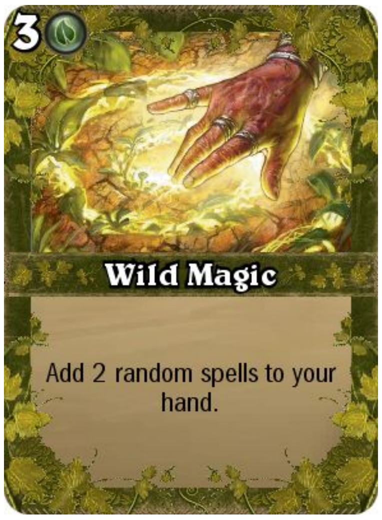


Figure 27: Wild Magic

The name of a card is derived from its class name (e.g. *WildMagic.java*), and the art with an identical name is associated to the card. If such a file does not exist, a default art is displayed instead.

Cards descriptions are generated automatically. This is achieved by invoking *toString()* recursively on the basic blocks constituting a card. By default, the *toString()* method returns the class name of an effect (that's why effect classes names are such explicit). When an effect owns a parameter, the 'X' occurring in its class name is replaced by the textual representation of that parameter. For instance, the *DealXDamage* effect generates the description of Figure 28.

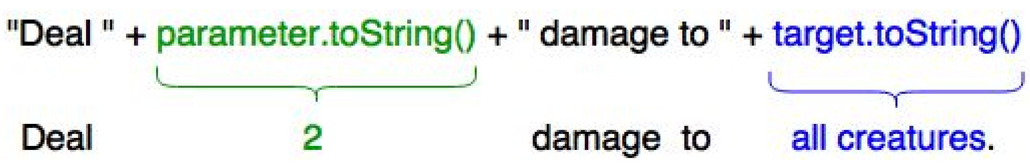

"Deal " + parameter.toString() + " damage to " + target.toString()
Deal
2
damage to
all creatures.

Figure 28: Illustration of *DealXDamage.toString()*

It is still possible to customize a card's description by overriding its *Card.description()* method. That can be exploited to increase a card's clarity, as well as adding flavor or fancy text. However, automated descriptions ensure consistency among the hundreds of cards and save a considerable amount of time. They also avoid two cards with identical effects from being written in two distinct ways, as it occasionally occurs in *Hearthstone* [38], [39]. When a card is modified, its description automatically adapts to its new effect.

Since each card is represented by a concrete class, it is possible to implement features specific to a card. For instance, a double-headed giant could display two health gauges by overriding the method that renders a card.

Creatures

Creature cards are played on a location of the battlefield. Hence, the ***Creature*** class is an extension of *Card<Location>*.

When a creature is played, the *Creature* class triggers an additional ***Summon*** effect that places the creature on the board. Afterward, the card's specific effect triggers.

A creature may own one or several abilities. An ***Ability*** is an action that may be triggered by the creature's owner. The most common ability of a creature is its ability to attack. When a creature attacks another, it deals damage equal to its attack points to the target and takes damage equal to the other creature's attack points. The ***Attack*** ability is thus naturally implemented by the following effect:

```
Effect e = new And(new DealXDamage(owner.atk, target),
                   new DealXDamage(target.atk, owner));
```

Heroes

Heroes are defined as a particular type of creatures starting with 0 attack points. The ***Hero*** class overrides the card's cost to make heroes playable without paying resources. In addition to each hero specific power, every hero regularly produces resources by listening to the *Time* controller. Other creature features stay unchanged. Heroes are thus able to move and attack as a creature does (although attacking requires to give them attack points).

Spells

Spells can be used on a large variety of targets, including creatures on the board, empty locations, player hands, decks, or graveyards. Some spells even do not require any target to be specified. As a consequence, the abstract ***Spell<T>*** class leaves the responsibility to each concrete spell to specify the target type *T*. Spells that do not require a target are declared as *Spell<Void>*.

When a spell is played, the *Spell* class pushes that card at the top of its owner’s graveyard after the spell’s specific effect has resolved.

Events

Event cards can only be played on a particular area of the board called the ***EventBox***. Hence, The ***Event*** class is an extension of *Card<EventBox>*.

Since only one event is allowed in play at any time, when an event card is played, the *Event* class first triggers a ***DestroyEvent*** effect whose purpose is to put the current event card in play (if any) in its owner’s graveyard. Afterwards, the new event is placed on the *EventBox* and its effect triggers.

Decks

Decks are defined by a hero and a list of cards. A deck can include multiple copies of the same card. In that case, the card is simply instantiated several times.

The deck list is automatically retrieved based on the class files in the *\bin\mana\cards\decks* folder. As a consequence, creating a new deck in that folder automatically adds it to the deck selection menu.

A summary of the model layer is provided by the class diagram of Figure 29.

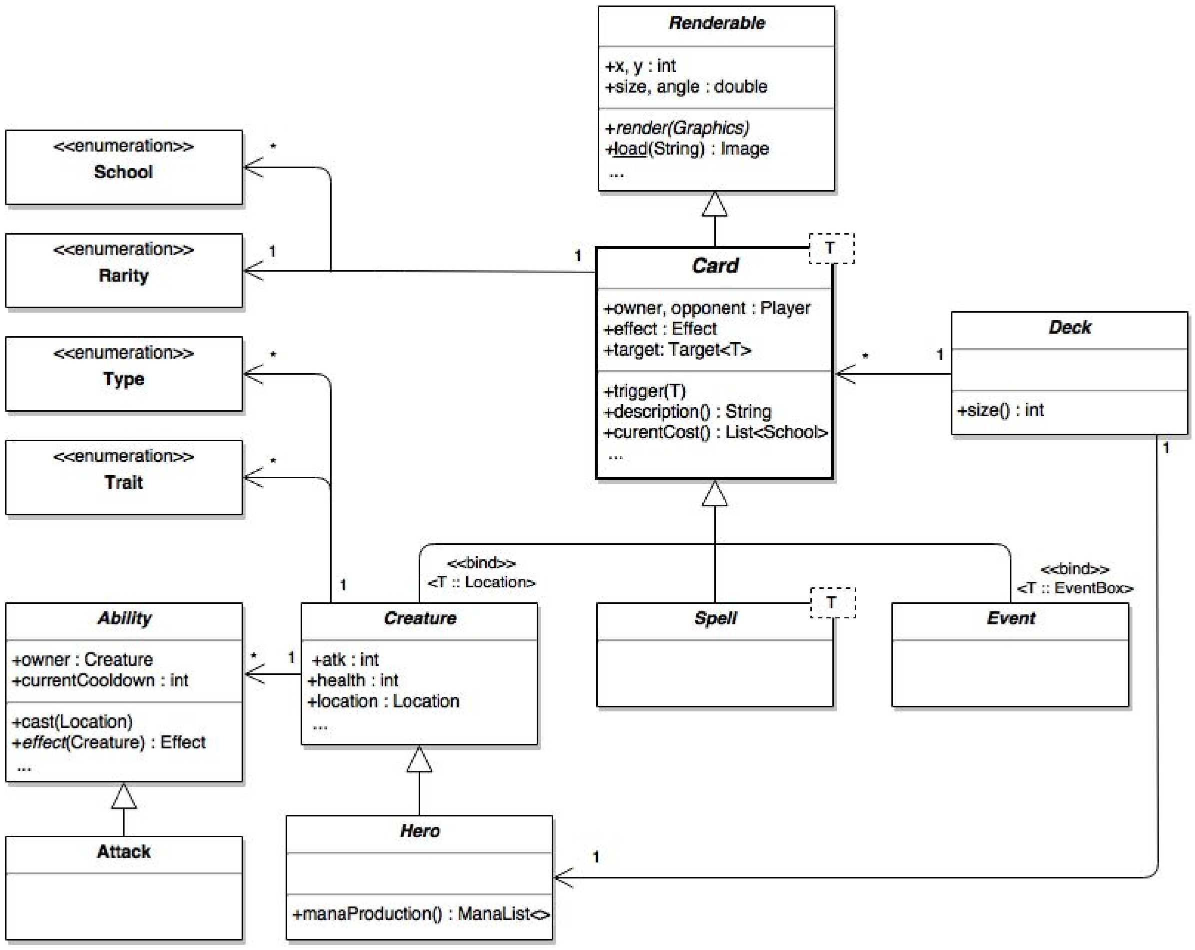


Figure 29 : Class diagram of the model layer

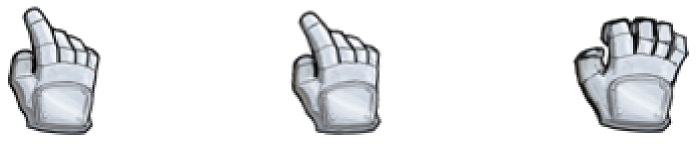
3.5 The Graphical User Interface Layer (src/mana/gui)

The GUI layer displays the game objects according to the current game state.

The game is displayed in a ***Window*** (a *JFrame*) that contains a ***Screen*** (a *JPanel*). Two screens are currently implemented: the deck selection screen and the main board screen. They are both described in details below.

A *Screen* is composed of a list of ***Renderable*** objects. A *Renderable* possesses 2D coordinates, a size, and an angle that can change with time. Any *Renderable* must define a *render()* method that render the object at its current coordinates, size, and angle. A Screen is therefore displayed by rendering all the objects composing it.

A particular instance of *Renderable* is the ***Mouse***, which continuously displays a cursor at the mouse coordinates. The cursor's appearance changes to indicate interactions with game objects (Figure 30). It is possible to attach an additional *Renderable* to the *Mouse* such that its coordinates follow the cursor, for example to drag a card.



(a) Default (b) Pressing an object (c) Dragging an object

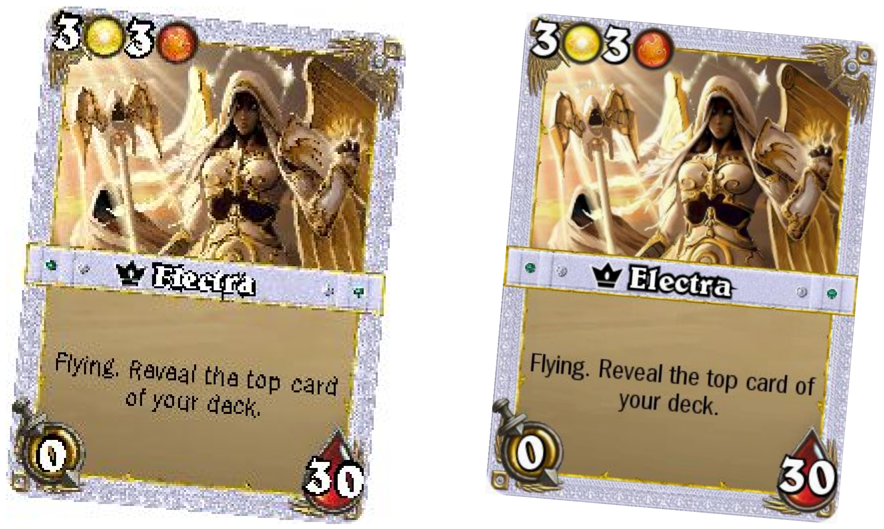
Figure 30: Cursor's appearances (source: *Hearthstone*)

A *Renderable* can gradually move to a given position, size, and angle, by using a ***MoveAnimation*** object. The animation is performed by updating at regular time intervals the object's properties toward the targeted values in order to produce a smooth movement. This is used, for example, to automatically move a destroyed minion on top of its owner's graveyard.

Cards

A ***Card*** is a *Renderable* built in three layers. Firstly, the card art is scaled to fit in the dedicated area and rendered. Since the original picture's ratio is conserved, and usually does not correspond to the area's dimensions, the scaled art often exceeds the available space in height or width. Secondly, the card frame is rendered with rounded corners above the art in order to hide the frequent exceeding portion. Thirdly, the card's cost, type, name, description, and stats are added.

The resulting card is considered as a single component that may be subject to rotation, scaling, or shearing transformations. However, such transformations highly impact the quality of rendering (Figure 31). In particular, when rotating a card, the pixel coordinates of the rotated image may lie between the integer coordinates of the original image's pixels, producing artifacts at the edges of shapes. That phenomenon was minimized by using anti aliasing and bilinear interpolation to blend the pixels at the edges of shapes.



(a) Default java rendering (b) With anti aliasing and interpolation

Figure 31: Rendering of a card scaled (0.62) and rotated (3°)

Deck Selection Screen

The ***DeckSelection*** screen (Figure 32) is the first screen appearing when launching the game. It displays the available decks, which are represented by the hero card associated to them, and allow the user to select one of them.

Decks are forming a 3-dimensional circle, with the currently selected deck at the center of the screen. The user can turn that circle to choose another deck by rolling the mouse wheel. The circle is animated using a *MoveAnimation* on every deck. Although decks only store 2D coordinates, an impression of depth is created by adapting the size of each deck to its position in the circle, and by rendering closest decks above farthest ones.

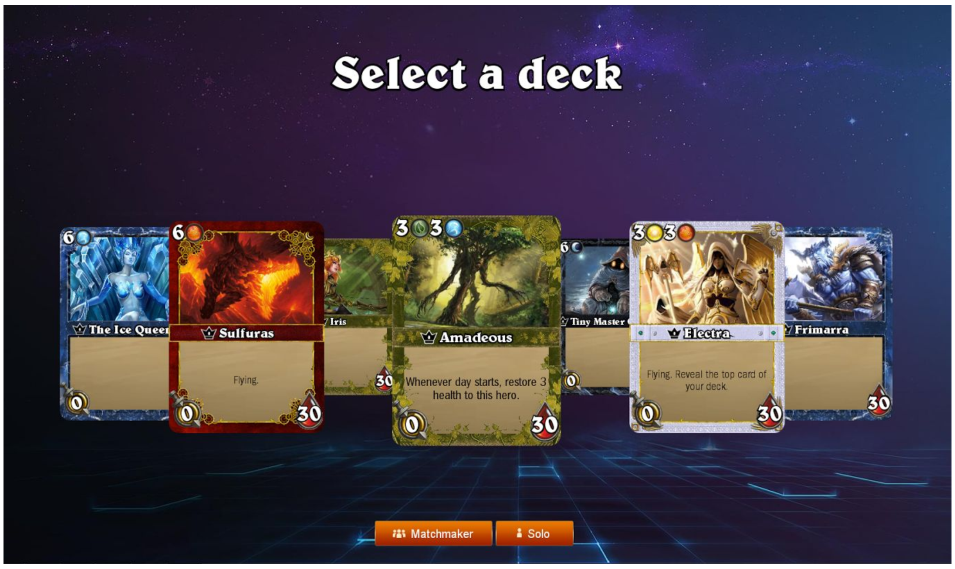
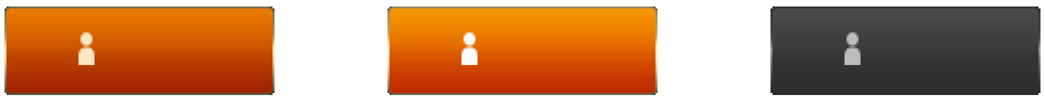


Figure 32: The deck selection screen

The user can start a new game with the selected deck by clicking on one of the two game modes, i.e. matchmaker or solo. Buttons are responsive and change their appearance to indicate the availability of the corresponding game mode (Figure 33). If the connection to the server could not be established, the "Matchmaker" button is disabled, but the user can still play in solo against the artificial player.



(a) Default (b) Mouse over (c) Disabled

Figure 33: Button states

When the user joins the matchmaker, the string "Waiting for another player…" is displayed until another player is found by the server. Afterwards, the window proceeds to the board screen and the game starts.

Board Screen

The board screen (Figure 34) is the main screen in which the game takes place. It is horizontally symmetric, with one player on the left and the other on the right of the screen.

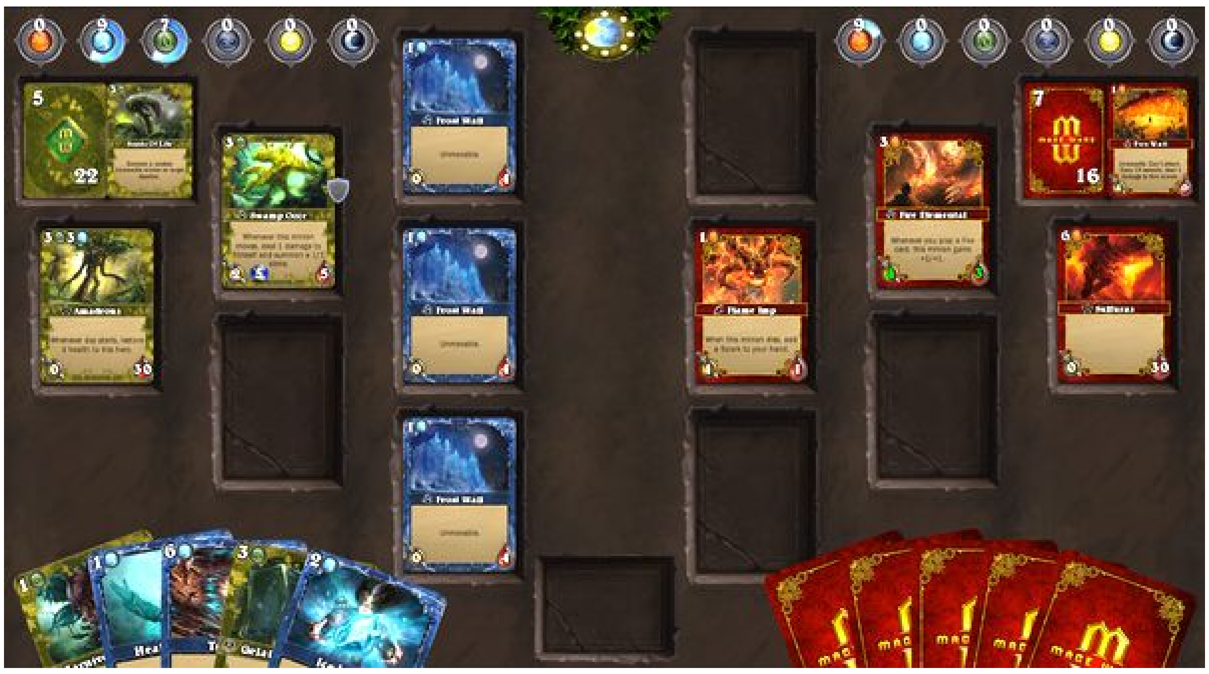


Figure 34: The game screen

The ***ManaBar*** (Figure 35) indicates the current amount of resources available to the player. It is composed of six instances of the ***ManaToken*** class, each one corresponding to a distinct school of magic. The blue fluid inside a token represents the production progression of the next mana point.

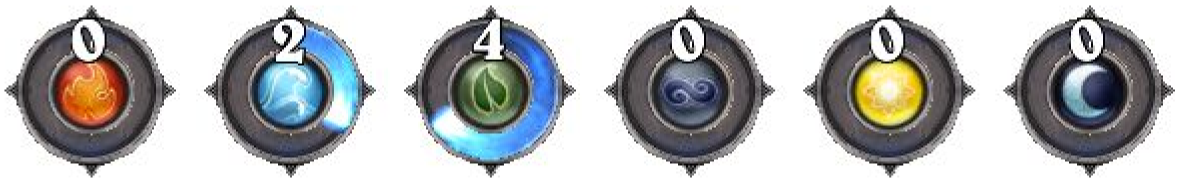


Figure 35 : Mana bar

The player's ***Deck*** and ***Graveyard*** are positioned under the mana bar. The deck, on the left, is the pile from where cards are drawn. On the right, the graveyard contains cards already used . Cards in graveyards are visible for both players, while those in decks stay unknown. Both piles are rendered with decreased size so as not to be confused with cards in play. The user can click on his deck to draw a card and add it to his hand.

The ***Hand*** (Figure 36), at the bottom of the screen, holds the cards available to the player. They form a sector of a circle to imitate physical cards. Cards in the opponent's hand are turned face down to stay secret. The user can move the cursor over a card in his hand to zoom in and, if the required resources are available, drag it into play. When a card is played on the board, its size is slightly reduced to give the impression cards are lifted and dropped on the board.

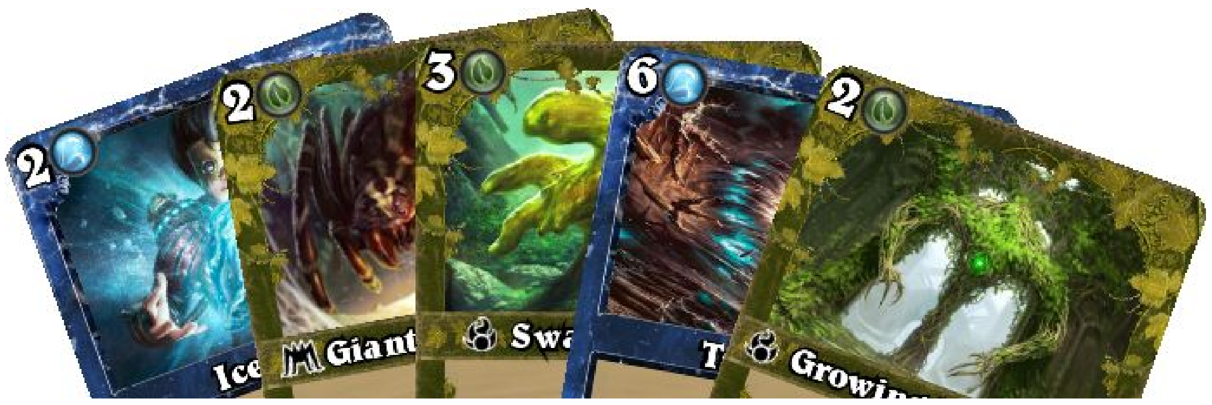
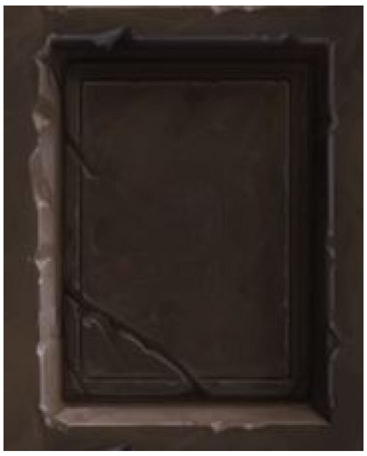


Figure 36 : Hand

The board is constituted of two ***Battlefields***, one for each player, composed of six ***Location*** instances. A location is an emplacement that can contain a creature card. When a card is dragged by the user, valid locations on which that card can be played are highlighted (Figure 37b). By listening to the *Mouse* controller, a location catches any valid creature that is dropped on top of it.


(a) Empty

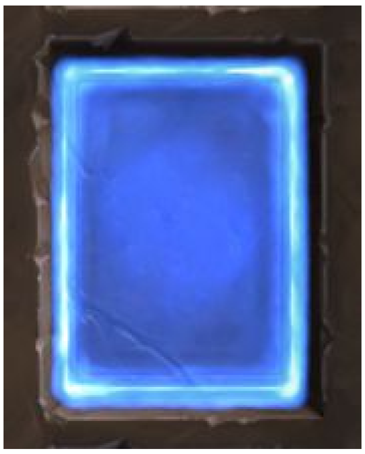

(b) Highlighted

Figure 37 : Battlefield location

While a creature is on the board, its abilities (such as attacking) can be used by holding the right click. The class ***TargetArrow*** displays a red arrow following the cursor to indicate which creature is going to be attacked (Figure 38). Releasing the mouse click over a valid target confirms the attack. If damages are dealt, the ***Damage*** class displays the amount over the hurted creature for a limited time.

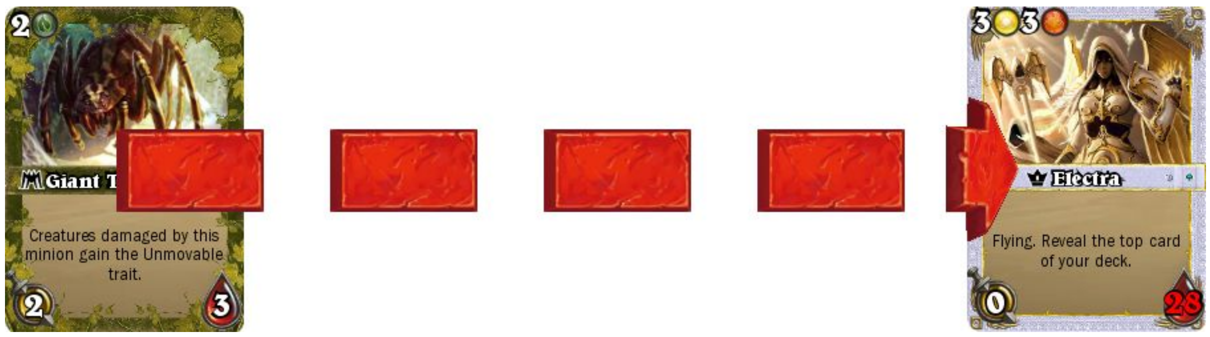

Giant T
Creatures damaged by this minion gain the Unmovable trait.
Electra
Flying. Reveal the top card of your deck.

Figure 38 : Target arrow (source: *Hearthstone*)

The smaller emplacement at the bottom of the screen is where event cards are played. It is implemented by the ***EventBox*** class that acts like a *Location*, with the two exceptions that (a) only event cards can be played in it and (b) it allows a new card to be played even if there is already a card placed in it (the latter card then replaces the former one). Event cards in play only display their art, but their description can be red by moving the cursor over the event box.

At the top of the screen, the ***Clock*** shows the current game time. The number of light dots indicates the progression of the day or night (Figure 39). Some cards are affected by the game time (e.g. by triggering two distinct effects during day and night) or may control the clock.

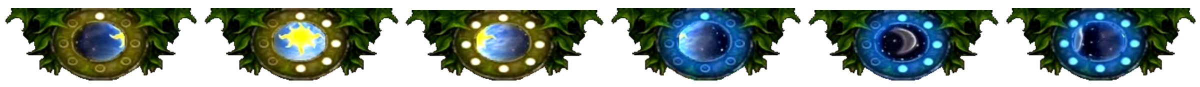
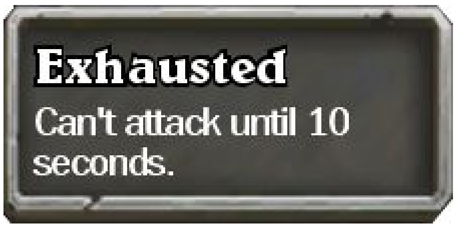


Figure 39 : Clock (source: *Warcraft 3*)

Lastly, the user can position the cursor over various interface elements to display an informative ***Tooltip*** (Figure 40).



Exhausted
Can't attack until 10 seconds.

Figure 40: Tooltip (source: *Hearthstone*)

Part 2
Creation of an Artificial Player

A major problem inherent to the creation of video games is the conception of an artificial player that offers to the user a real challenge, and at the same time a feeling of intelligent behavior.

Unsurprisingly, Computer Games is the field in which artificial intelligence is the most used [40]. The reason is that the association of artificial intelligence and games is advantageous for both game developers and researchers. On the one hand, video games publishers can efficiently improve their products by simulating intelligent behaviors. On the other hand, the ever-increasing realism of video games offer complex, but still fully controlled, environments for testing purpose.

This part details the creation of an artificial player for the card game developed in section 1. It consists in the following chapters. Chapter 1 introduces the foundations of techniques used in artificial intelligences playing games. Chapter 2 defines the problem statement, which is then solved in two distinct manners in Chapter 3. Lastly, Chapter 4 presents the results and gives some possible future research topics.

Chapter 1
Artificial Intelligence in Games

“ Imagination is the essence of discovery. ” - Winston (Overwatch)

1.1 Tree Representation

Games are traditionally represented by a tree [41], whose nodes are the possible states the game can take, and whose edges are the possible moves available to the player in a given state. In this tree, the root node is the initial game state, and the leaves represent the different outcomes in which the game can terminate. Each leaf is associated to a result, generally *win*, *loss*, or *draw*.

With this representation, the problem of playing a game consists in determining the move from the current node that maximises the probability to end on a winning leaf. This is achieved by backing the leaves’ values up to the root node, assuming the player is acting optimally (i.e. win moves are preferred to loss moves).

In a single-player deterministic game, this is straightforward as the player’s choices fully determine the game outcome. If at least one winning scenario exists, that scenario will naturally be chosen and moved up to the root node.

1.2 Two-player Games

In two-player games, however, the game outcome is also affected by the opposing player’s choices, that are not known in advance. Such games thus need to simulate the opponent as well. This is achieved by making the assumption that the opposing player is acting optimally, i.e. with the objective of maximizing its own probability to win. If the two players pursue conflicting goals, it is equivalent to minimising the other player’s probability to win.

The game tree is then supplemented with additional nodes reflecting the opponent’s decisions. In sequential games, since players are acting one after the other, even depths correspond to the current player and odd depths correspond to the opponent. Considering both players are acting optimally, a move’s outcome can be determined by alternatively maximising and minimising at each level the root player’s probability to win. When every move from the root node have been examined, the one with the highest probability to win is selected. This decision rule is called *minimax* [42].

An example of minimax tree for Tic-tac-toe is given in Figure 41. In this example, *win*, *loss*, and *draw* are respectively assigned to values 1, -1, and 0. The tree was expanded in depth-first order, with edges numbered in the order they were explored. As it can be seen in the figure, on even depths a win or a draw is selected over a loss (e.g. edge 1 is preferred over edge 6), whereas on odd depths losses are selected over a win or a draw (e.g. edge 7 is preferred over edge 8). From the root, the best move is the one corresponding to edge 1.

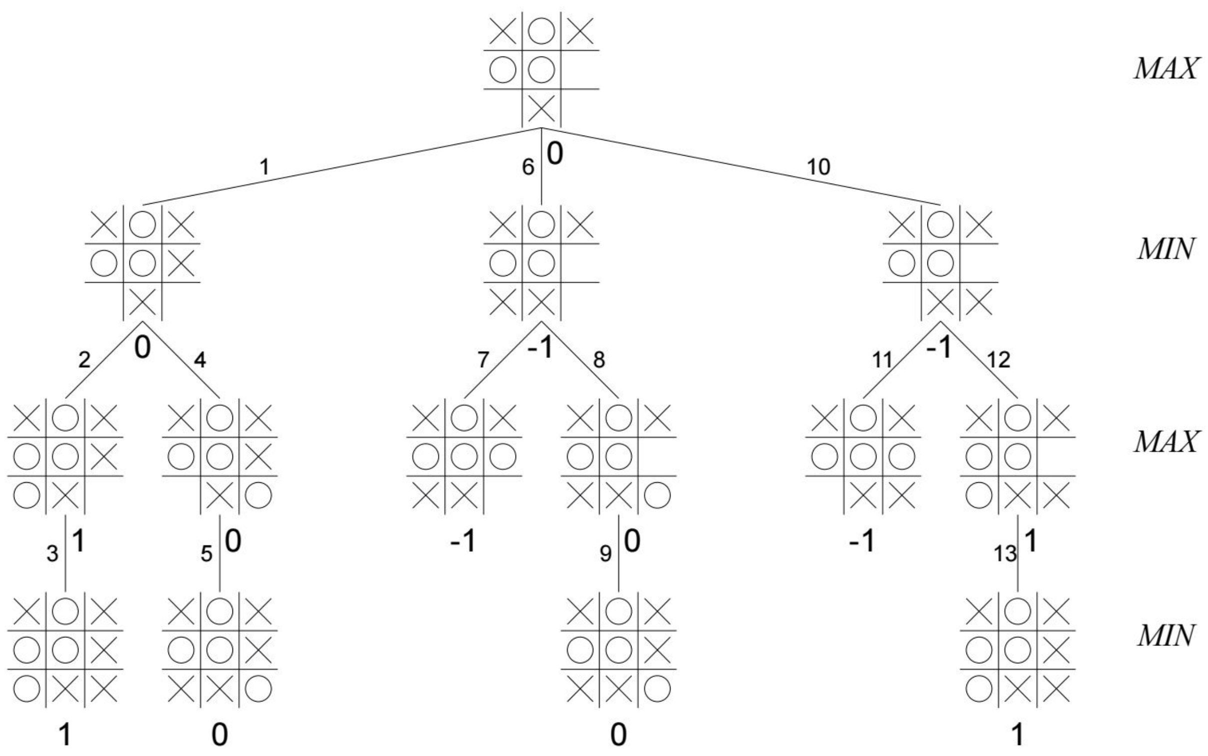


Figure 41: An example minimax tree (source: [43])

1.3 Pruning

When generating the game tree, some subtrees have no way to improve the optimal move given the already discovered results. The tree may thus be pruned in order to speed up the building process. A widely used pruning technique is named *αβ pruning* [44]. In αβ pruning, a threshold of the anticipated minimax value is maintained. If the current node's value lies outside that threshold (i.e. is lower than *α* or greater than *β*), the current subtree can safely be pruned without modifying the optimal move.

Figure 42 shows the game tree obtained by applying αβ pruning on the tree of Figure 40. After investigating the left subtree, it appears that the first move would lead to a draw. Since the root player maximises his score, he will prefer that move to any other one leading to a loss, and is thus already guaranteed to get a score of at least 0. The lower bound *α* of the root node is then 0.

While investigating the second move, node E ends on a loss (-1). Since the opposing player is playing at node C, the score at that node is minimised, and the value of node C will necessarily be smaller or equal to -1. Because -1 is lower than 0 (the lower bound *α* of node A), the second root move will never be chosen regardless of the remaining unexplored children of C. As a consequence, the exploration of that subtree can be omitted without impacting the root node. A similar reasoning can be applied on the third move that necessarily leads to a loss.

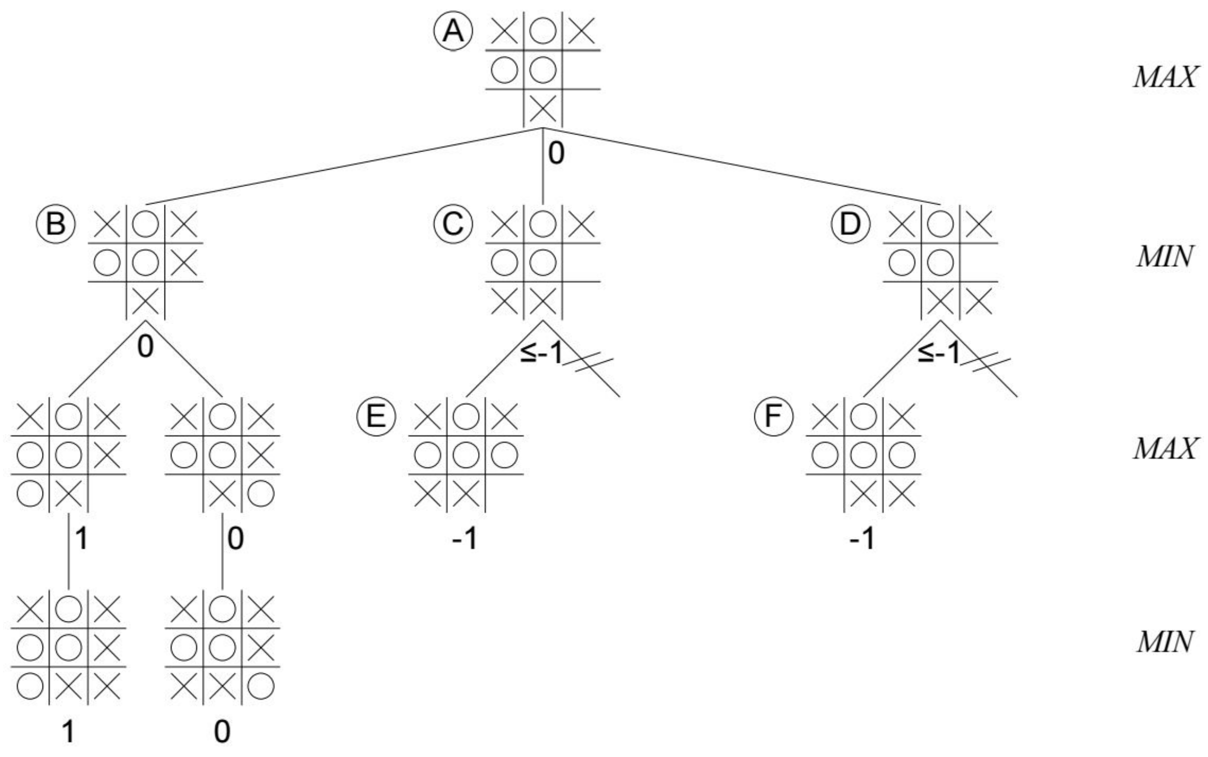


Figure 42: An example αβ pruning (source: [43])

1.4 Randomness

In non-deterministic games, such as card games, the game outcome may also be influenced by random events independent of players choices. For example, this occurs when rolling a die, or drawing a card from a shuffled deck. In such games, the game tree is required to take that randomness into account.

Expectimax search [45] is an adaptation of minimax to non-deterministic games. With expectimax, the game tree is supplemented with additional chance nodes reflecting random events. Edges emanating from a chance node correspond to the possible outcomes of the random event, and the value of a chance node is given by the weighted average over every possible outcome. Expectimax may also be used to represent the revealing of hidden information.

Figure 43 shows an example of expectimax tree. In this example, chance nodes are positioned between each player's move (but in practice, it depends on the game rules). The value of chance node A is calculated based on its possible outcomes (i.e. 6, 4, and 5) and their respective probability (i.e. 0.5, 0.2, and 0.3), giving the weighted sum $0.5 \times 6 + 0.2 \times 4 + 0.3 \times 5 = 5.3$. Similarly, the value of chance node B is computed as $0.5 \times 0 + 0.5 \times 9 = 4.5$. Because the root player maximises his probability to win, in this case, he will naturally choose the move leading to node A.

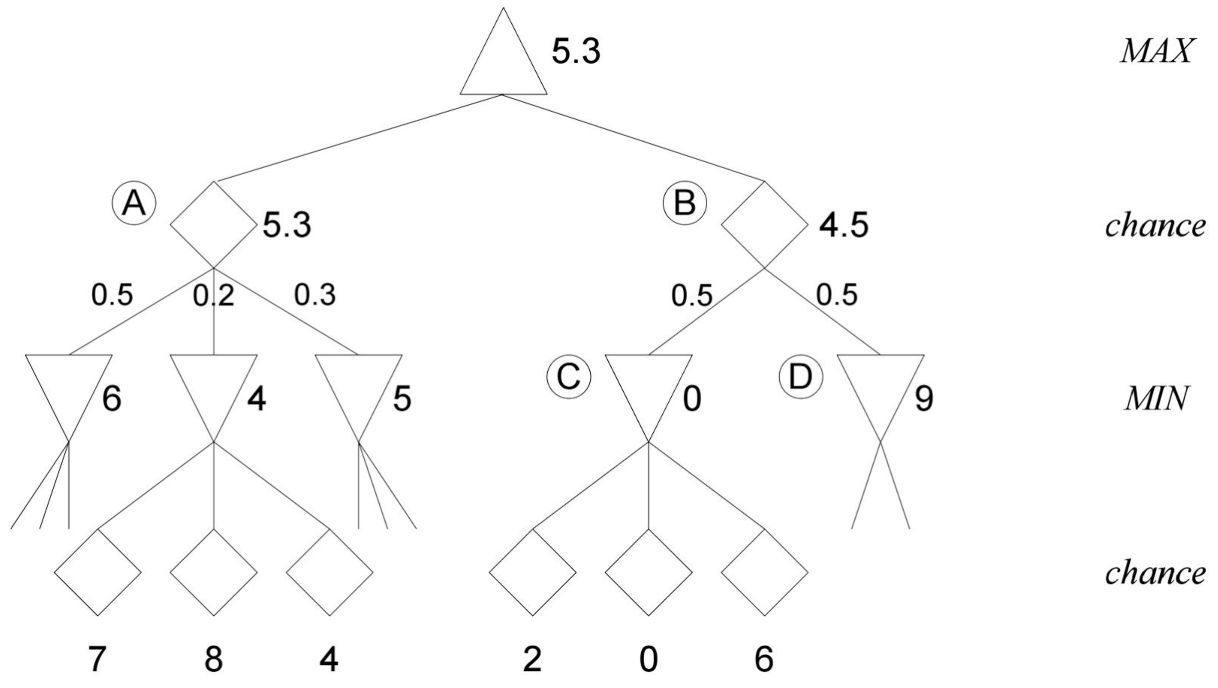


Figure 43: An example expectimax tree (source: [43])

1.5 Tree Search

So far, we considered the complete game tree was expanded until a terminal game state (i.e. *win*, *loss*, or *draw*) was reached for all possible scenarios. A game “whose outcome can be correctly predicted from any position, assuming that all players play optimally” is said to be solved [46]. For example, checkers [47] and *Connect Four* [48] have been solved.

In practice, for most games, this is not achievable. For example, *Shanon* showed there are roughly 10^{123} ways to play a game of chess, and around 10^{47} possible chess board states [49]. With that number of nodes, the game tree can’t be fully explored in a reasonable amount of time. As a consequence, it is only developed up to a limited depth, or until a time constraint has elapsed. The choice of a move must then be performed with partial knowledge of the game tree.

Because terminal game states are not necessarily reached, the exact outcome of each move is generally unknown. In order to assign a value to non-terminal leaves, an evaluation function can be used. An evaluation function is a heuristic based on expert knowledge that associates to any state an estimation of its profitability for the root player. After building the partial tree, the move leading to the most favorable game state is chosen. However, in complex games, an evaluation function is hard to determine, and is often highly arbitrary.

The limited tree exploration also raises the question of which nodes to explore in priority. Since the portion explored influences the quality of the final move, the exploration strategy is of crucial importance. For that reason, the development of efficient search techniques able to orient the search toward most promising nodes is required. As an example, Monte Carlo Tree Search is a notable search technique used in game play.

1.6 Monte Carlo Tree Search

Monte Carlo Tree Search (MCTS) [50] is a search technique that has been successfully applied to artificial intelligences playing games. Instead of a heuristic, this technique evaluates non-terminal leaves by simulating moves until the end of the game is reached.

The algorithm is composed of four steps: selection, expansion, playout, and backpropagation. These steps are applied repeatedly to construct a search tree until a stopping criteria is reached (such as a time constraint). Figure 44 illustrates the basic concept behind Monte Carlo Tree Search.

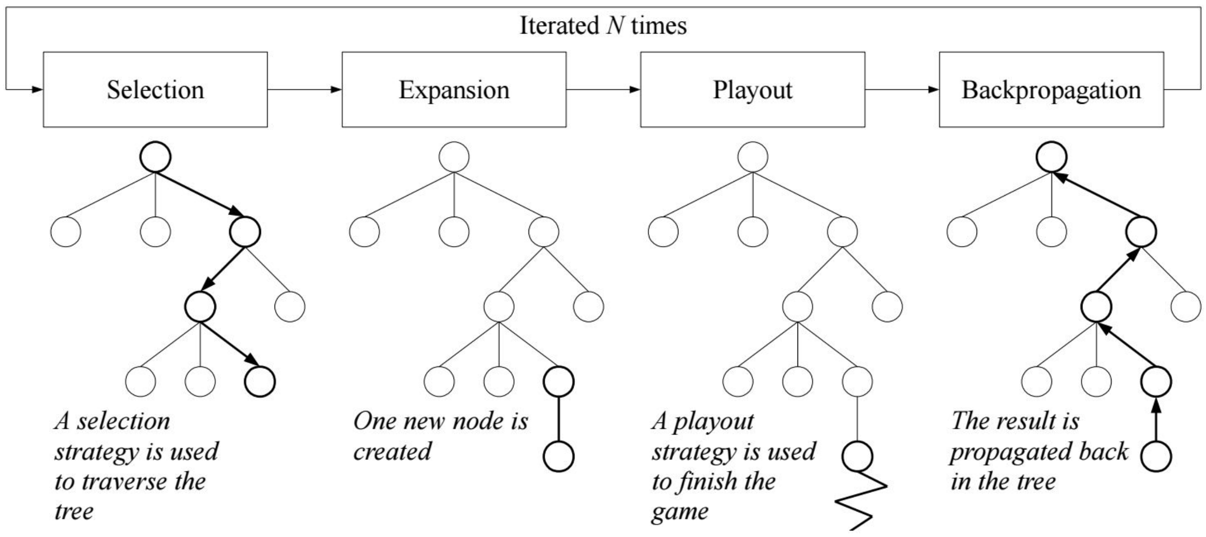

Iterated *N* times
Selection
Expansion
Playout
Backpropagation
A selection strategy is used to traverse the tree
One new node is created
A playout strategy is used to finish the game
The result is propagated back in the tree

Figure 44: Monte Carlo Tree Search (source: [43])

In the selection step, a selection strategy chooses the next node to extend in the tree being constructed. A typical selection strategy defines a trade off between *exploitation* and *exploration*. Exploitation involves selecting nodes with highest win rates (obtained via simulations) in order to converge toward most promising moves. By contrast, exploration involves selecting nodes with a low number of simulations, with the objective of uncovering new favorable moves (they may not have been visited enough for revealing their true potential).

In the expansion step, one or several child nodes are added to the selected node. These children represent the next game states obtained by continuing the game from the parent node.

In the playout step, the long-term game outcome is simulated from the newly added node by playing moves until a terminal game state is reached. Whereas simplest playout strategies select random moves, realistic simulations can be performed by using a more sophisticated strategy. For example, moves can be influenced by game-specific knowledge to improve their quality.

Lastly, in the backpropagation step, the simulation result is propagated back to the root node. Each node maintain internal statistics, including the number of visits and the number of wins obtained by traversing that node. When a result is propagated, all nodes encountered along the path are successively updated. These statistics will then influence the further selection steps of next iterations.

The above four steps are repeated a predetermined number of times, or until a time constraint has been reached. Afterward, one of the root move is selected as the best move based on the gathered statistics. As one would expect, the final move quality increases with time.

Monte Carlo Tree Search has been successfully applied to artificial intelligences playing various games, including non-deterministic games (e.g. *Settlers of Catan* [51]) and real-time games (e.g. *Pac-Man* [52]). In particular, it contributed to considerable improvements in the game *Go* over the last decade. Recently, the *AlphaGo* [53] program defeated a professional Go player of 9 dan rank by combining MCTS with deep learning [54].

Regarding card games, an attempt to applied MCTS to *Magic: The Gathering* was made [55]. However, the game had to be simplified to make the artificial player able to move after a reasonable amount of time. Still, due to the exponential branching factor inherent to card games, the results were not satisfactory: “Our MCTS AI with its current limitations has the propensity to play only marginally better than a random player due to the vast number of possible outcomes that can result given any move made” [55]. The authors suggested to use a heuristically guided search to reduce the branching factor and orient the simulations toward realistic moves.

Chapter 2
Problem Statement

“ Creativity is intelligence having fun. ” - Albert Einstein

This chapter defines the tasks an artificial player should solve in order to play the game. Section 2.1 first underlines the properties specific to the game. Afterward, section 2.2 formally decomposes the game into subproblems.

2.1 Game Properties

Games are generally categorized based on several properties. This classification is crucial because different techniques are applied to different categories of games. Hence, analysing a game’s properties is the first step to identify the AI techniques that are relevant to its resolution.

For example, we distinguish single-player games from two-player and multi-player games, perfect-information games from partially observable games, deterministic games from non-deterministic ones, and sequential games from real time games. This section identifies the properties related to the game of interest.

Two-player

Firstly, the game is a *two-player* game, not to be confused with *multi-player* games. In the field of artificial intelligence, *multi-player* refers to a game that can be played by more than two players. More precisely, our game is a *zero-sum* game, in the sense that players compete against each other with symmetrical goals. The goal for both players is to reduce the opposing hero’s health to zero.

Partially Observable

Secondly, the game is *partially observable*. This means that, as opposed to a perfect-information game, all information about the game state is not known by the players. For example, players do not know what cards are composing their opponent’s hand, nor the next card they will draw from their shuffled deck. As a consequence, the opponent’s available actions are not completely known, and predicting them remain even more complex.

Non-deterministic

Thirdly, the game is *non-deterministic*. It includes some random card effects with multiple potential outcomes over whom neither player has control. For example, a card may deal a random amount of damage, or summon an absolutely random minion on the board. In such situations, the profitability of a move can still be estimated based on all its possible outcomes, but chance will always have the last word.

Real-time

Lastly, our game is a *real-time* game. As opposed to turn-based games, players are acting simultaneously. This introduces additional time-management constraints. In particular, when simulating the game's outcome based on its current state, the opponent may invalidate at any time the simulation by simply modifying the game state.

Real-time games constitute one the most challenging form of video games for artificial intelligence [56]. Until now, most researches have focused on sequential games [57].

2.2 Game Decomposition

The ultimate goal of playing a card game is a complex task that can be divided into particular subproblems. For the game of interest, we identified three subproblems: (a) attacking with the creatures on the board, (b) positioning the creatures on the board, and (c) choosing which cards to play among those available in the player's hand.

The overall task of playing the game can then be completed by solving every particular subproblem individually. Each problem is described separately in the further sections.

Attacking

The first subproblem is the attacking strategy of the artificial intelligence.

Each creature features an attack value and a health value. The attack represents the amount of damage the creature deals to its target when attacking. Health represents the amount of damage the creature can take before dying. When a creature attacks another, it deals damage equal to its attack points to the target and takes damage equal to the other creature's attack points.

For example, a *Lava Elemental* has 6 attack points and 4 health points, and an *Armored Griffin* has 3 attack points and 4 health points. If the *Lava Elemental* attacks the *Armored Griffin* (Figure 45), the former loses 3 health points, while the latter loses 6 health points.

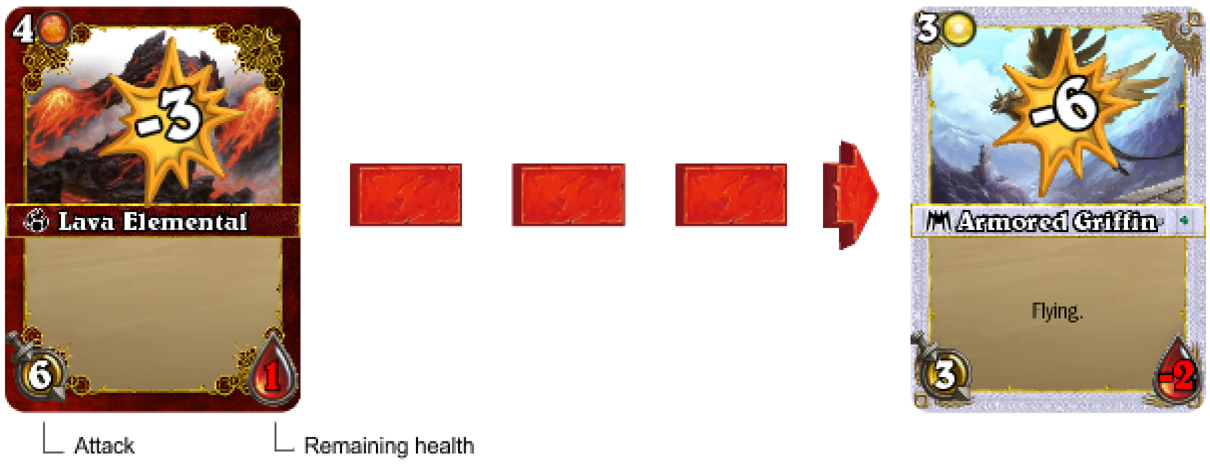


Figure 45: A 6/4 Lava Elemental attacking a 3/4 Armored Griffin

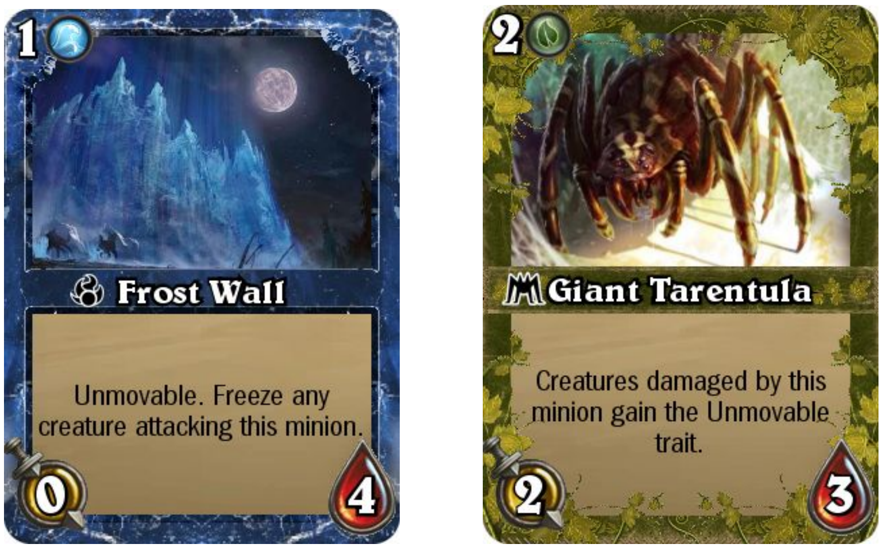
After an attack, any creature with a remaining health less or equal to zero is removed from the board. In the above example, *Armored Griffin* would leave play. It is worth mentioning that dead creatures cannot attack anymore.

In order to attack, a creature must first be played on the board. The choice of which creatures to play and when to play them is not part of this subproblem. It is delegated to the problem of playing cards (see “playing cards” below).

When a creature is played, it cannot attack immediately. In card games, this is traditionally referred to as *summoning sickness*. The time a creature must wait before attacking is determined by its attack speed. Most creatures have an attack speed of 30, meaning they can attack every 30 seconds. However, some creatures are faster or slower. After a creature attacked, it is *exhausted*, meaning that it must wait again a delay equal to its speed before attacking again.

During a match, the board is generally populated by a group of creatures on both sides, each one owning different attack, health, and speed values. The problem of attacking consists in distributing allies attacks efficiently toward enemy targets with the objective of maximising the profitability of the current player. This typically involves eliminating enemy threats by losing the least possible allies.

In addition to the basic concepts described earlier, each card may add, break, or modify a rule. Attacks must then be dispensed by taking into account these particular changes. For example, some cards alter the attack, health, or speed of a creature. Furthermore, attacking may potentially trigger another card’s effect and produce indirect consequences. Figure 46 gives two examples of such cards that can interfere with attacks.


1
Frost Wall
Unmovable. Freeze any creature attacking this minion.
0
4
2
Giant Tarentula
Creatures damaged by this minion gain the Unmovable trait.
2
3

(a) Frost Wall (b) Giant Tarentula

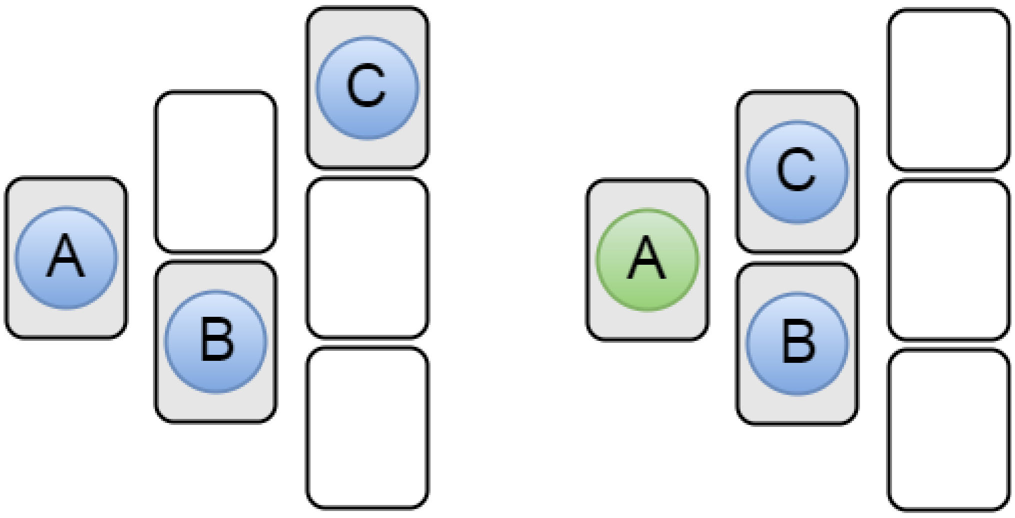
Figure 46: Cards interfering with attacks.

Positioning

The second subproblem involves arranging allied creatures on the board.

Creatures are positioned in a triangular formation (Figure 47). Each player's side is composed of 6 slots in which creatures can be played. When a creature is on the board, it can move to another location at any moment, and without paying a cost. Two creatures can also interchange their position with each other, unless at least one of them is subject to a particular rule prohibiting his movements.

When two creatures are positioned in front of a third, that third one is *protected*. As a consequence, it can't be targeted by the opposing player. Figure 47b shows an example of configuration protecting a creature.



(a) None of the creatures are protected.

(b) Creature A is protected by B and C.

Figure 47: Triangular formation

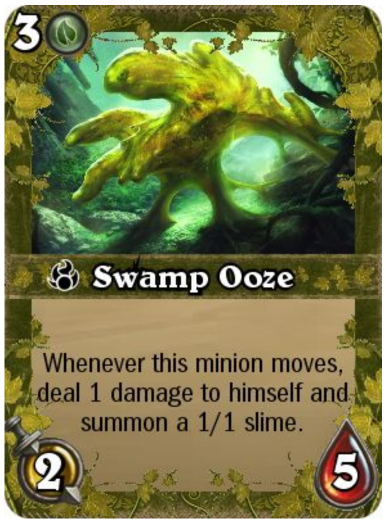
In this example, creatures B and C can themselves be protected by placing in the same way two additional creatures in front of them. Creatures in the three front locations are never protected. Consequently, at most three creatures can be protected simultaneously.

The position of a creature doesn't affect its own capacity to attack the opponents, and is thus unrelated to the previous problem of attacking.

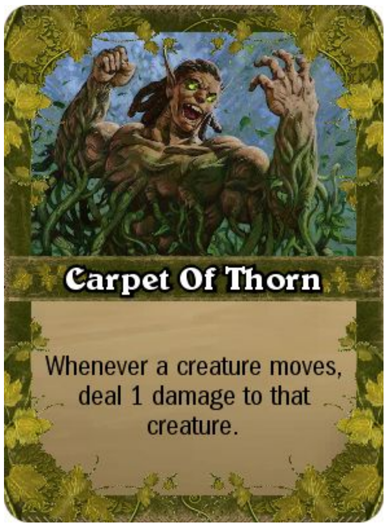
The problem of positioning consists in assigning each allied creature in play to a particular location, given all the creatures in play and their respective characteristics, to prevent the opponent from attacking most valuable allies. In particular, this requires to determine what makes an ally more vulnerable or valuable than others.

Advanced rules may interfere with a creature's ability to move or be protected. For example, a creature owning the *Unmovable* keyword cannot be moved by its owner. A *Flying* creature can attack any opponent, regardless of that opponent's position. Abilities and spells may also allow a player to move the other player's creatures. In particular, an efficient strategy involves moving the opponent's *Unmovable* creatures because their owner can't move them back to their original position.

Lastly, some cards effect may be triggered by moving a creature. Figure 48 gives examples of such cards that may influence moving decisions.


3
Swamp Ooze
Whenever this minion moves, deal 1 damage to himself and summon a 1/1 slime.
2
5

(a) Swamp Ooze



(b) Carpet of Thorn

Figure 48: Cards interfering with positioning

Playing Cards

The third subproblem is inherent to every card games and involves choosing which cards to play among those in hand.

Players start the game by drawing six cards. During a match, subsequent cards are drawn at regular intervals. Cards require the player to spend a magical resource called "mana", that is produced continually at a fixed rate throughout the match. There are six types of mana, and the mana types being produced depend on the hero incarnated by the player.

Each card has a mana cost indicated in its top-left corner, that must be paid in order to play the card. The number represents the amount of mana, while the symbol represents the mana type. A cost may combine several mana types. When a card is played, the corresponding mana is consumed and the card is removed from its owner's hand, preventing him from using it again. The loss of the card itself can also be considered as part of the cost, since the number of cards drawn from the deck is limited.

Cards have various effects on the game state that are described in their text pane. There are three types of cards: *Minions*, *Spells*, and *Events*.

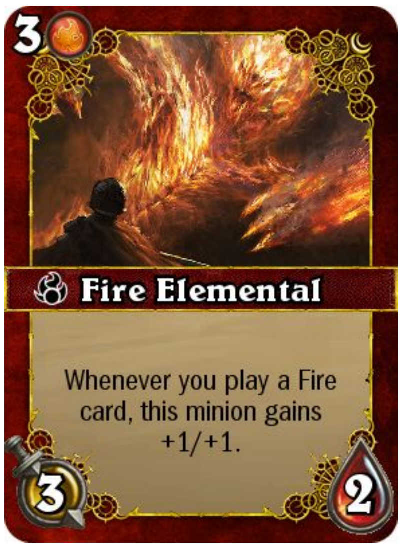
Minions are the primary card type and represent entities evolving on the battlefield. While a minion is on the board, it can repeatedly attack (as described earlier in the problem of attacking) until it is destroyed. Minions must be played on a chosen location, but this decision is delegated to the problem of positioning.

Spells trigger one time effects to alter the game state. There is a large variety of spell effects, from dealing damage to drawing cards. Conversely to minions, a spell card doesn't enter play, but is removed from the game immediately after its effect has resolved. Spell cards may be used on a variety of targets, including minions, heroes, decks, graveyards, and player hands. Often, with the decision of playing a spell may also come the choice of a target.

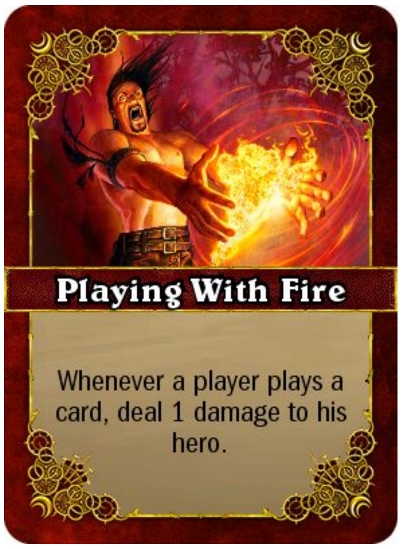
Events add or modify a game rule equally for both players. They are played on the board and their effect remains as long as the card stays on it. An event card generally costs no mana to be played, but only one event is allowed in play at a time. As a consequence, playing an event card destroys the current event already in play, if any.

The problem of playing cards involves deciding which cards to play among those in hand, given the current game state and the available resources. In addition, when playing a spell, a valid target must be chosen among the available ones in play. A player also has the option of discarding a card. This is useful, for example, when the player cannot draw a card because his hand is already full, but doesn't want to play any of them.

Cards should be played by taking into account that the game state may influence their profitability. In particular, the cost of a card can be temporarily altered by another one in play. Furthermore, playing a card may also potentially trigger another card's effect, as those shown by Figure 49.



(a) Fire Elemental



(b) Playing with Fire

Figure 49: Cards interfering with playing cards

Chapter 3
Solutions

“ With great power, comes great responsibility. ” - Peter Parker (Spiderman)

This chapter describes the solutions designed to play the game. Two artificial players were implemented. Section 3.1 presents a heuristic player that evaluates moves by quantifying the amount of resources it gains (or loses) versus the amount of resources the opponent gains (or loses). Section 3.2 presents a more advanced player that simulates the long term consequences of a move to predict its outcome.

3.1 The Heuristic Player

The heuristic player makes decisions by quantifying the raw advantage each move would grant. Coupled with expert knowledge enhancements, its choices are reflecting an intermediary level of play. It was created for testing purpose of more elaborated artificial players.

Most heuristically guided techniques in games tend to evaluate board states. When every potential next state has been evaluated, the artificial player naturally chooses the move leading to the most profitable state. However, due to the complexity of interactions inherent to card games, board states cannot be accurately rated by taking into account each card’s specific effect.

For that reason, this player does not evaluate game states directly, but instead evaluates the impact of each individual move. Because this card game is a zero-sum game, the advantage a player gains is exactly what the other loses. Following this principle, we can predict if a given move improves or deteriorates the current game state, by comparing the amount of resources the player gains (or loses) versus the amount of resources his opponent gains (or loses). The heuristic player can then select the move that would generate the greatest raw advantage.

Attacking

In order to evaluate attacking decisions, the player is required to quantify the advantage a creature provides to its owner. Such a measure could use the creature’s cost as a reference, assuming it accurately reflects a card’s strength. However, a creature’s attack and health may vary during its lifetime, altering its strength while maintaining the same cost. A reliable measure should thus reflect these variations by taking into account a creature’s current attack and health values.

The value of a creature is estimated as the sum of its attack and its health. There are two exceptions. Firstly, a dead minion (i.e. whose health ≤ 0) is no longer valuable, regardless of its attack value.

Secondly, the value of a hero is set to infinite because its destruction is the winning condition of the game. In this way, the artificial player will prevent as much as possible the loss of his hero who is the most valuable creature on the board. Symmetrically, killing the enemy hero is infinitely rewarded such that any move leading to its direct elimination is always chosen.

If C is a creature, its value is defined as

$$value(C) := \begin{cases} +\infty & if\ C\ is\ a\ hero \\ C_{attack} + C_{health} & if\ C_{health} > 0 \\ 0 & otherwise \end{cases}$$

Since attack points are never negative, $value(C) \in [0\,;+\infty[$ for every creature C, with creatures closer to 0 being less valuable. This value reflects the tactical advantage a player gains by summoning a creature on the battlefield, and the advantage he loses when that creature leaves play.

When a creature takes damage, it can either die or survive the attack. If it dies, its owner loses an advantage equal to that creature's value. On the opposite, a creature surviving to an attack is still valuable because it is guaranteed to impact the game in the future, by being involved in another combat and dealing damage again. In that case, its owner only loses a fraction of its value, proportional to the damage taken (i.e. equal to the attacker's attack value).

If the attacking creature A deals damage to the defending creature D, the advantage gained by A's controller is quantified as

$$Damage(A, D) := \begin{cases} value(D) & if A_{attack} \geq D_{health} \\ A_{attack} & otherwise \end{cases}$$

In practice, when two creatures enter combat, they both deal damage to each other. Consequently, both players may lose resources in the process. In card games, this exchange of advantage resulting from a combat is traditionally referred to as a *trade* [58]. The profitability of a trade is evaluated by comparing the advantage the attacking player gains (or loses) versus the advantage the defending player gains (or loses):

$$Trade(C, T) = Damage(C, T) - Damage(T, C)$$

If $Trade(C,T) > 0$, the trade is profitable for C's owner. On the contrary, if $Trade(C,T) < 0$, the trade is considered unprofitable for C's owner.

When an allied creature C is ready to attack, all possible $Trade(C,T)$ are successively evaluated for each target $T \epsilon \tau$ (the set of enemy creatures). The best target is naturally defined as the one resulting in the most profitable trade:

$$BestTarget(C) = arg\,max_{T \epsilon \tau}\ Trade(C, T)$$

If none of the available trades are profitable (i.e. $Trade(C,T) < 0 \; \forall \; T \in \mathcal{T}$), the creature should simply not attack at all.

Positioning

The problem of positioning was solved similarly by evaluating the possible trades, but from the point of view of the opponent. The solution consists in analysing the opponent's available trades, and to position allies in order to prevent the opponent from performing his best trades.

For each ally, all possible trades in which that ally is involved are successively evaluated. The vulnerability of an allied creature C is then defined as the worst trade in which that creature is attacked:

$$Vulnerability(C) := max \; Trade(T, C), \;\; T \epsilon \mathcal{T}$$

These are the most profitable trades for the opponent and, assuming he is playing optimally, the ones he will most probably want to perform. As a consequence, the artificial player will position allied creatures in order to prevent him from performing those trades.

This is achieved by sorting allies in decreasing vulnerability order, and positioning most vulnerable ones at the back, as shown in Figure 50. That positioning strategy ensures that the largest possible number of allies are protected, given the total number of allies, and that most vulnerable ones are protected before others (i.e. worst trades are prevented).

If the artificial player owns less than six creatures, the locations with highest numbers will stay empty. The positioning of friendly creatures is updated whenever a game event may change the opponent's available trades.

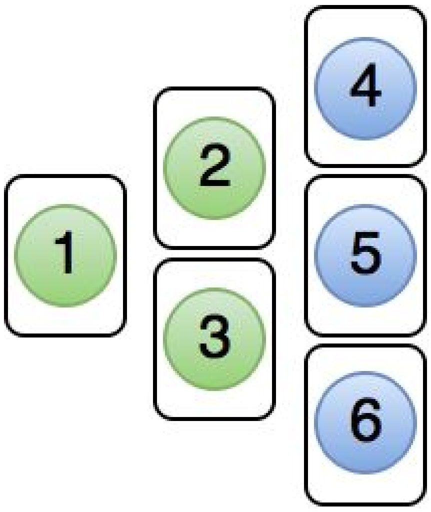


Figure 50: Positioning strategy (1 is the most vulnerable ally, 6 is the less vulnerable)

Playing Cards

Playing cards is the most challenging subproblem to solve heuristically. The reason is that a card can break or modify a rule in any imaginable way, making its profitability hard to evaluate. Considering the game features hundreds of different cards, it is not feasible to treat each of them individually. Furthermore, additional card sets must be added to the game without requiring adaptations of the artificial player.

In a turn-based card game, the problem of playing cards can be solved by determining the best way to spend the fixed amount of resources that is available each turn. This can be achieved by evaluating the advantage each card would grant (similarly to the problem of attacking) and choosing the combination of cards that is the most profitable. Since unused resources are lost from a turn to the next, the player does not have the option of accumulating them with the objective of playing a more expensive card on the next turn.

By contrast, this real-time card game slowly accumulates resources with time, raising the question of *when* to play cards. On the one hand, spending resources as soon as possible impacts immediately the game and puts the greatest pressure on the opponent. On the other hand, by accumulating resources, a player may have more options to deal with the opponent's subsequent actions, and make more profits of cards by playing them at the best moments.

This heuristic player is too simplistic to evaluate the advantage (or disadvantage) of conserving resources given the current board state. Consequently, it plays favorable cards as soon as the required resources are available. More precisely, minion cards are played by increasing cost order, and spells are played as soon as they are more profitable than every minion in hand. Since event cards cost no mana, they are played immediately when they are drawn.

By playing cheapest minions first, the heuristic player takes control of the board early game and is able to maintain his advantage thanks to favorable trades with the opponent's minions.

3.2 The Simulating Player

The simulating player is an enhancement of the heuristic player that simulates the long-term consequences of each action in order to make decisions.

This player was created with the objective of understanding what a card does (or more generally, an action) without relying entirely on game-dependent knowledge such as the heuristic player. The challenging task of predicting a card's effect is solved by playing that card on a copy of the game, simulating it until the end of the game, and observing the final result. This provided to the player anticipation skills with, in particular, the ability to accurately predict a game's outcome before it ends.

Representing Real-time

Most researches have focused on sequential games, using a game tree to represent states and transitions between states, and well-known search methods (such as *Alpha Beta Minimax search* and *MCTS*) to explore that tree. This section discusses a way to adapt that tree representation to the real-time space.

There are two major differences between a turn-based game and a real-time game. Firstly, in turn-based games, players are forced to act one after the other. As a result, the player who is acting can be deduced from a node's depth: even depths correspond to the current player (the one that maximises the game's value) and odd depths to the opposing player (the one that minimise it). The player who is acting is not explicitly stored in the tree.

By contrast, in real-time games, the same player can execute several moves in a row. Consequently, the player who is acting can no longer be deduced from a node's depth. A real-time game tree is thus required to store, for each move, the player who is performing that move.

The second difference is that, in real-time games, moves can be performed at any moment. This temporality must be taken into account, because the same move performed at different moments can lead to different game states. One way to adapt the game tree is to add time data on edges reflecting the delay before a transition. With this solution, the time needed to reach a particular state can be deduced by summing all delays along the path up to the root.

However, the introduction of continuous time data makes the tree infinite. Indeed, each moment a transition can be performed gives rise to a distinct branch. To make the tree finite, a solution could be to regroup transitions performed at different moments but leading to a similar result. These branches can be represented as a single edge labelled with the time interval in which that move would lead to the same result. By slicing the continuous time into intervals, we end up with a finite number of edges.

Interestingly, with this representation, a sequential game appears as a particular case of real-time game whose edges have the same length (there is exactly one turn between each transition).

Simulating Real-time

The simplest way to simulate a real-time game is to run an accelerated clone on the background. However, this card game uses *Swing* timers that are known for erroneous delays at the scale of tens of milliseconds [59]. At normal game speed, such timing errors are unoticable, but with a speed hundred times faster, they start to interfere with the game flow. In addition, a substantial amount of processing time would be wasted for simulating the game time elapsing, while most of the time nothing happens.

Instead, the game is simulated by sorting actions in the order they are expected to occur, and successively executing them without any delay in between. The game is, in a sense, reduced to a sequential game.

A simulation is performed by cloning every creature on the board, and ordering them in increasing *exhausted time* (i.e. the time remaining before a creature can attack). Let's consider, for instance, the simulation represented by the timeline of Figure 51. In this example, the board is composed of the two heroes (denoted by ♛) and three minions, with two of them owned by the green player and the third one owned by the red player. The timer below each creature is the delay before that creature can attack. Creatures that cannot attack, such as heroes (as long as they have no attack points), are labelled with an infinite timer.

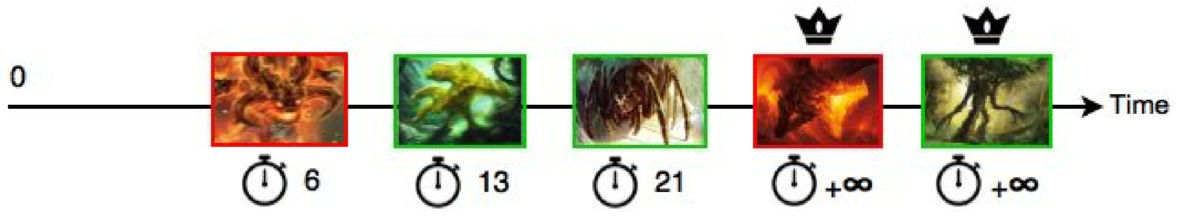


Figure 51a : Initial state

During a simulation, both players are assumed to play optimally. Due to the nature of the game, there is usually no advantage in waiting before attacking, unless all possible trades are unprofitable. As a result, we assume that a creature attacks as soons as it is allowed to do so, if there is at least one favorable trade. In this case, the red minion on the left is expected to be the first to attack, since its exhausted time is the smallest (6 seconds).

The simulation starts by jumping to the first anticipated attack (Figure 51b). The game time itself is not simulated, and timers are only variables used to order the flow of actions.

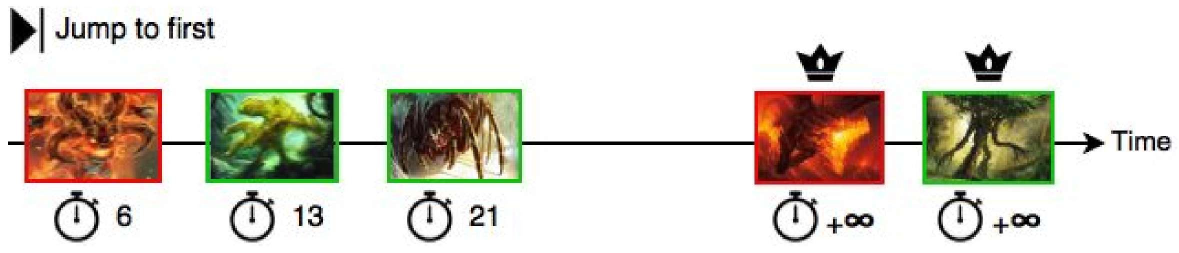


Figure 51b : First action

In order for that red minion to attack, a realistic target must be chosen. This is achieved by evaluating every possible trade similarly to the heuristic player (see 3.1). Furthermore, the artificial player also guesses the positioning of creatures. In this example, the green player owns three creatures. By assuming he is playing optimally, we infer that one of them should be protected, probably the most vulnerable one. Instead of attacking the best target, that is presumed protected, the red minion will thus follow its second best profitable trade (if any). In this case, it attacks the green spider (Figure 51c).

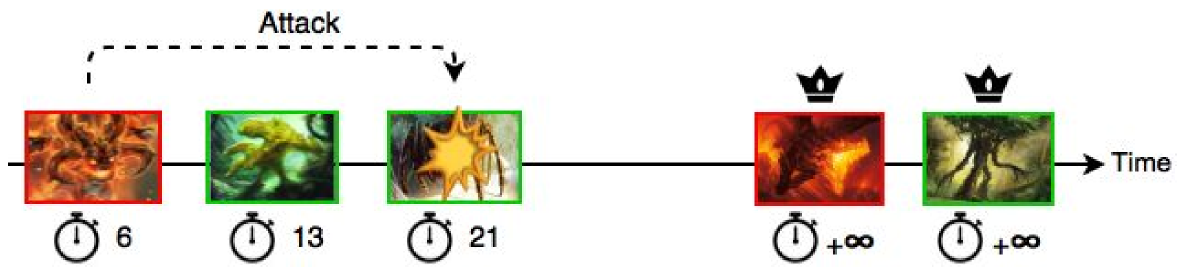


Figure 51c : Attacking

If any creature dies due to combat damage, it is removed from the timeline such that (a) it cannot attack anymore and (b) it can no longer be targeted by subsequent actions (e.g. attack or card played). The creature that just attacked is now *exhausted*: it must wait before attacking again. In this example, the red minion attacked at $t = 6$ and can attack every 30 seconds, so its next attack is planified for $t = 6 + 30 = 36$. The creature's timer is updated and the actions are reordered by increasing timer value, causing the creature to move back in the timeline (Figure 51d).

In a situation where none of the possible trades are profitable, the creature is simply assumed not to attack as long as the game state doesn't change. Because the creature does not attack, it is not exhausted, and thus keeps its ability to attack at any moment. In that case, the creature is repositioned at the second position in the timeline. As a result, as soon as the game state will be modified by the next action, that creature will directly have another chance to attack in case a favorable trade now exists.

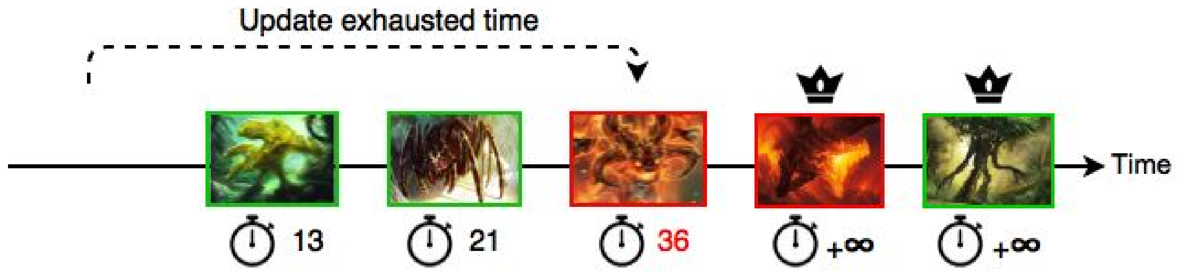


Figure 51d : Updating timeline

The simulation continues by jumping to the next action without any delay (Figure 51e), and the same process is repeated.

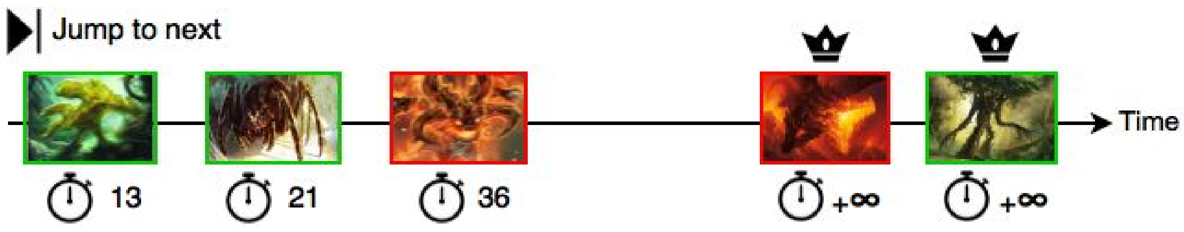


Figure 50e : Next action

The next anticipated action is the attack of a green minion. In this example, the red player has only two creatures. We can deduce that none of them would be protected in such a situation, because a total of at least three creatures are needed to protect one. The green creature is thus allowed to attack following its most favorable trade. In this example, let's consider it is profitable to attack directly the opposing hero (Figure 51f).

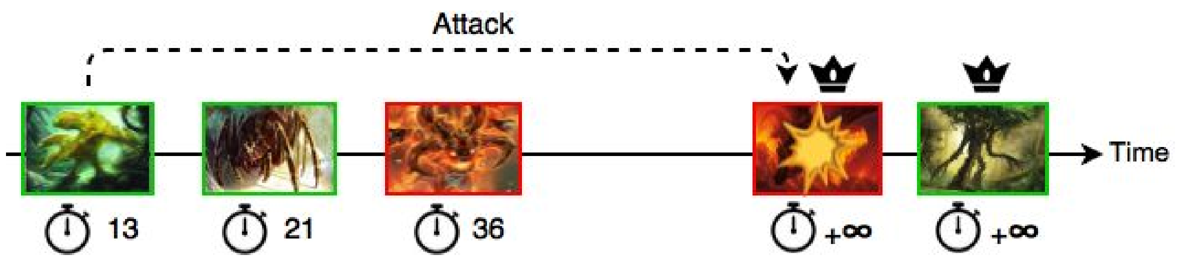


Figure 51f : Attacking

A simulation ends when one of these three conditions is reached:

1. A hero dies. In that case, the remaining player is declared as the winner;
2. The left-most creature cannot attack, because it has an infinite exhausted time. As a consequence, none of the remaining creatures can attack (since they are ordered). This can happen, for instance, when all minions destroyed each other in combat, leaving only the two heroes on the board. In that case, the simulation ends on a draw;
3. None of the remaining creatures *wants* to attack because there is no favorable trade for any of them. The simulation is stuck in an equilibrium state in which no more progress is made. The game is also considered as a draw.

In the example simulation, let's consider the green minion was able to destroy the opposing hero (Figure 51g). The game-ending condition is reached because a hero died, and the simulation stops.

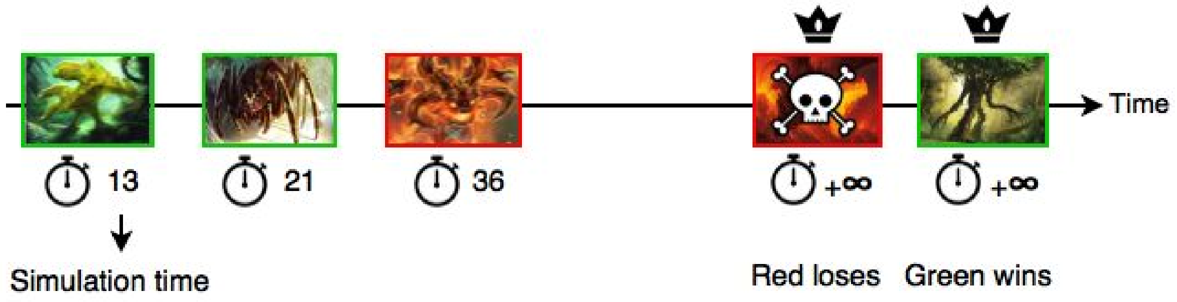


Figure 51g : Simulation end

Depending on the player that is incarnated by the artificial intelligence, the simulation returns either *Win* or *Loss*. A simulation also returns the game time *it would take* to reach the outcome that was obtained. It is simply deduced from the left-most creature's timer, since its attack corresponds to the action that ended the game.

This trivial example only considered minion attacks. In practice, the simulator is also capable to handle cards being played. Cards are added to the timeline of actions such as minions, with a timer corresponding to the time needed to produce enough resources to play the card. When the left-most action is a card, that card's effect is triggered on the virtual game state, assuming the card is played as soon as the required resources are available.

The virtual game state is then updated according to the specific card that has been played. For example, if a card destroys a minion, that minion is removed from the timeline. If a card summons a minion, that new minion is inserted into the timeline. After a card has been played, it is removed from the timeline because a card can only be used once. It is possible that the simulation ends before every cards in the queue has been played.

Choosing a Move

The ability to simulate a game until its end is a powerful tool for deciding which card to play among the possible ones. By inserting cards into the timeline, the artificial player can reliably observe their impact on the game's outcome, and select the most promising card in the real game.

Whenever the game state changes, the artificial player uses simulations to determine the best card to play given the current state. For each card in his hand, the player simulates a game starting from the current state to the end of the game, in which that card (and only that one) is played. A card is assumed to be played as soon as the required resources are available. Its position in the timeline thus depends on its cost, the resources already available, and the production rate. If a card can be played on several targets, a simulation is performed for every possible target.

Since the opponent's hand is not observable, it is not feasible to simulate the opponent playing cards. In Hearthstone, Google researcher Elie Bursztein predicted the cards a player would most probably play, by performing statistical analysis on sequences of cards played in recorded games [60]. However, in this brand new game, no prediction can be made on the opponent's deck due to the absence of *metagame* [61].

Considering the game features hundreds of different cards, simulating each of them would lead to an explosive number of possible scenarios, and simulating random cards would only interfere with the player's card being evaluated. For that reason, simulations do not consider that the opponent is playing cards. As a result, they are often too optimistic, but still allow to compare each card's impact on the current game state. In particular, cards with a stronger positive impact on the game should lead quicker to a victory.

For each card, the simulations provide a game outcome (i.e. *Win*, *Loss*, or *Draw*) and the time it would take to reach that outcome. An additional game is simulated where none of the cards are played (only minion attacks are considered). This simulation can be used as a reference to evaluate how a card improved or deteriorated the game state. It is worth mentioning that doing nothing may be the optimal move. Figure 52 is an example output produced by the simulating player.

```
Doing nothing: Defeat after 122 seconds.

Dragon Whelp: Draw after 151 seconds.
Circle of Flame: Victory after 92 seconds.

Best move: Circle of Flame
```

Figure 52: An example output produced by the simulating player

When every simulation outcome is known, the artificial player chooses the card leading to the best scenario. Victories are naturally preferred over other outcomes, but if it's not possible to win, draws are preferred over losses. If multiple victories are possible, the fastest one is selected, because cards leading quicker to a victory should have a stronger positive impact on the game. On the contrary, if every scenario leads to a loss, the slowest one is selected, to maximise the probability of drawing a card that may save the situation.

For example, Figure 53 shows the decision process with a player's hand composed of two cards. The artificial player has three options: playing none of the cards, playing the first card (C1), or playing the second card (C2). After simulating all cases, the consequences of every action is known, and the most profitable one is moved up to the root. In this case, playing the second card will lead to a victory in 92 seconds if the opponent doesn't play more cards than those already in play.

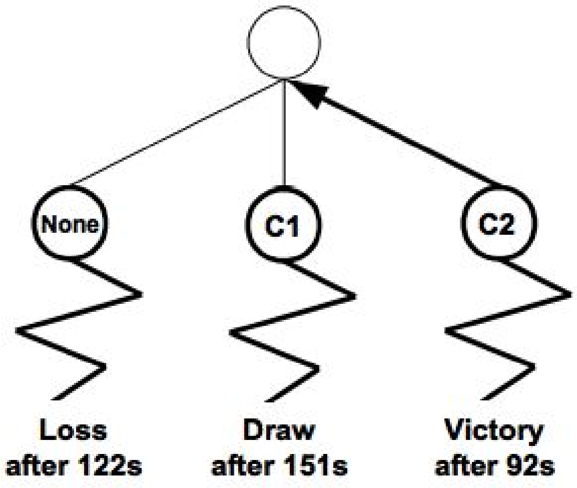


Figure 53: Decision with two cards in hand

It is possible that a simulation ends before the card's required resources were available, preventing the card from being played. In that case, the move is equivalent to playing nothing, and can be pruned without impacting the final decision.

The artificial player is also able to simulate the utilisation of two or more cards in the same simulation. However, increasing the number of simulations impacts the player's reaction time, with no guarantee to improve the move quality. In addition, in a real-time game, a decent move performed quickly can be more profitable than the optimal move performed after a longer delay. Furthermore, playing several cards in a row without the opponent playing one of its own cards would not be realistic.

In practice, new simulations are performed every time the game state is subject to modifications, including those induced by the artificial player itself. As a consequence, when the artificial player plays a card, subsequent simulations are performed on the new game state with the remaining cards in hand. If a second card combines well with the first one already played, that card might be immediately played as well.

Chapter 4
Results

“ Computers do exactly what you tell them but often surprise you in the result. ” - Richard Dawkins

The performances of the artificial players were assessed by analysing their win rates. In a first experiment, the heuristic player and the simulating player competed against each other. In a second experiment, the simulating player was confronted to an expert human player. Section 4.1 explains how to reproduce the experiments. Section 4.2 presents the results and analyses them. Lastly, Section 4.3 contains some possible future research topics.

4.1 Reproducibility

To reproduce the experiments, first launch a Server (in *mana/network/server*). Then, start two players, the ones that should compete against each others. For the heuristic player described in 3.1, launch *HeuristicPlayer.java* (from the *mana/ai* folder). For the simulating player described in 3.2, launch *SimulatingPlayer.java* (from the *mana/ai* folder). Artificial players automatically join the server’s matchmaker when they are launched.

For a human player, simply launch the *Client* class (from *mana/network/client*) as a user would do to play the game normally, and join the matchmaker from the main menu. A game is created by the server as soon as two players joined the matchmaker (regardless of the nature of those players). The first player who joined the matchmaker is on the left of the screen, while the second player is on the right.

Game Speed

The default game speed is adapted for human players. However, when two artificial players compete against each other, it can be useful to speed up the experiment. This can be accomplished by increasing the game speed via the *Time* controller (in *mana/Time*).

Deck

The deck used by artificial players is fixed by the abstract *ArtificialPlayer* class, and can be changed once to affect all concrete implementations of artificial players (including future artificial players). Human players, on the other side, can freely select a deck in the deck selection screen. In our experiments, the results were obtained by using the same deck for every player that is defined in *mana/ai/AiDeck.*

In the deck selection screen, the deck list is automatically retrieved based on the class files from *mana/cards/decks.* Because the experimental deck is not appropriate for entertainment, it is declared in the AI folder instead, such that it doesn’t appear on the game menu. In order for a human player to use the AI deck, a copy of that deck should be added to the deck folder to make it appear in the main menu. Alternatively, a custom experimental deck can be created, by extending the abstract *Deck* class. New decks are automatically added to the deck selection screen.

Coupled with the deck content, the order in which cards are shuffled into the decks definitely impacts the results. In the following experiments, players were using two decks not necessarily shuffled in the same order. As a result, one player or the other was randomly advantaged, similarly to real games. An analogue experiment could be conducted by shuffling both players' decks in the same order. However, that would only measure the players' performances on a narrowed subset of matches (mirror matches).

4.2 Results

Heuristic Versus Simulating

The heuristic player and the simulating player competed against each other in 1,000 matches. Table 12 gives the resulting win rates for both players.

Heuristic Player Win Rate	**Simulating Player** Win Rate
21,9 %	78,1 %

Table 12 : Win rates (Heuristic versus Simulating)

As it can be seen, the simulating method outperformed the heuristic one. The reason is that the heuristic player chooses the move maximising its short-term reward, but it may not correspond to the move maximising its chance to win in the long-term. As a result, it is not able to combine two cards if playing the first one has a negative impact. In particular, it never uses two weak allies to kill one strong enemy, because the first attack is evaluated as a loss of resources since it does not destroy the enemy.

On the other side, by predicting the long-term consequences of moves, the simulating player has a stronger ability to planify its strategy. Interestingly, it tends to position his hero in the front to protect his minions, while the goal of the game is specifically to defend the hero. As a result, the simulating player's hero generally takes more damage in the early game than the opposing hero. However, since minions protected by the hero cannot be attacked, they are only involved in profitable trades, and repeatedly give advantages to their owner. When the hero's health is dangerously low, it then moves to the back, but the trade advantage gained in this way is generally enough to take control of the board and compensate the hero's lost health.

In addition, the simulating player plays minions whose attack and health counter the opponent's minions, when that's possible. This is due to its capacity to predict the trades that would occur if he plays a minion. Conversely, the heuristic player plays minions based on their raw strength, regardless of the enemy threats.

Nevertheless, a notable advantage of the heuristic method compare to the simulating one is its minimised reaction time. The profitability of all trades and cards is determined in less than 1 millisecond. By contrast, the simulating player requires a significant amount of time to perform its simulations (a few milliseconds). For that reason, the heuristic player would tolerate a higher game speed than the simulating player.

Simulating Versus Human

In a second experiment, the simulating player was confronted to an expert human player in 50 matches. The reason the number of matches is low is because games had to be played in normal game speed in order to allow the human to play. Table 13 gives the resulting win rates for both players.

Simulating Player Win Rate	**Human Player** Win Rate
38 %	62 %

Table 13 : Win rates (Simulating versus Human)

Naturally, the results highly depend on the human player. In this case, the human had knowledge of the deck he shared with the opposing AI. As a result, he was able to infer the potential future cards played by the artificial player, and to adapt his play style accordingly. On the contrary, the artificial player was not trained to play nor face that specific deck, and made no assumption on the cards in the human's hand nor in its own deck. It was neither aware that his opponent was using the exact same deck. Consequently, the artificial player acted as if decks were composed of random cards.

On the other side, the real-time space disadvantaged the human player, whose performances are impacted by the game speed. Indeed, reducing the game speed may increase the human player's win rate. However, since that time constraint is imposed by the game rules themselves, we might consider it is part of the challenge of playing the game. More precisely, artificial players also suffer from an extreme game speed (simulations require a sufficient amount of time to be performed), but at a different scale than humans.

The decent results achieved by the simulating player allows an average human player to play alone against a computer-controlled opponent, and still experience a significant challenge.

4.3 Future Work

This section presents some possible future research topics.

Human-like Behavior

In this work, artificial players were designed with the objective of maximising their quality of play. In practice, artificial intelligences are generally implemented into videos games to provide entertainment to the user. Rather than trying to defeat the user, a smart artificial intelligence would behave as closely as possible to a human and give a sentiment of real intelligence.

This can be achieved, for example, by adding artificial delays in the player's actions. In addition, when all simulations result in a defeat, the artificial player could surrender as a desperate human player would do. The human-like behavior of an artificial player can be assessed using, for instance, a Turing test.

Difficulty Scaling

Another topic of interest is the ability to control the quality of play of an AI. Artificial players are commonly used in tutorials and training modes, where they face inexperienced players learning the basics of the game. In these contexts, it is crucial for players to face a fair difficulty level adapted to their skills.

The difficulty of an artificial player can be lowered by intentionally sabotaging its decisions. Instead of systematically choosing the best move, a weakened AI could occasionally take the second best option, or the third. Alternatively, reducing the time allocated to the artificial player decreases the number of simulations performed, and at the same time its accuracy.

Hearthstone AI engineer Brian Schwab pointed out that, since the opponent's hand is unobservable, a human player would not notice an AI is intentionally playing a suboptimal card [62]. By contrast, targeting decisions should be kept optimal because the human player may notice an absurd attack.

Deck Building

Beside the challenge of playing card games, also coexists the challenge of assembling custom decks. Building a deck is in fact a key strategic element of card games, as much as playing the actual game. Despite that, artificial players usually play with fixed pre-built decks, making them easily predictable. An AI that would be able to construct viable decks, with strong synergies between cards, could bring further diversity to computer-controlled opponents.

Artificial players could also be used to balance the innumerable cards and decks composing a card game. This can be achieved by comparing the win rates obtained with two identical AI using two different decks.

Meta Predictions

The weakest point of the artificial player developed in this paper is its inability to simulate the opponent playing cards. Due to the absence of metagame [61], it is forced to make the wrong assumption that the opponent's deck is made of completely random cards, and therefore unpredictable.

However, in practice, players' decks follow particular trends known for being competitive, with few or no variations at all. Based on this observation, an artificial player could try to predict the next cards its opponent will most probably play given the cards he played earlier. For example, Google researcher Elie Bursztein predicted the cards a player would most probably play in a *Hearthstone* game, by performing statistical analysis on sequences of cards played in recorded games [60]. Combined with the ability to simulate a game, such information may provide to the AI improved anticipation skills.

Conclusions

In this work, we developed a real-time strategic card game from scratch, and created an artificial intelligence to play that game.

In a first part, we analysed the major existing card games, by comparing their forces and weaknesses, and identified the specificities of the digital space. This analysis revealed that digital card games can take advantage of mechanics that are not possible in physical games to provide an improved gameplay experience.

Based on these observations, we designed a set of game rules with the objective of creating a card game that is innovative and fun. To achieve this objective, we questioned the various topics encompassing the essence of card games, and for each of them, a solution was proposed and justified.

Subsequently, the game was implemented in the java language. The source code is composed of 254 files, and uses neither a game engine nor an external library. A user interface was designed, as well as card templates and a game board. The game is playable online or offline, against other human players or artificial players, and with pre-built or custom decks. Further cards, decks, and heroes can be added without requiring any modification of the game.

In a second part, we considered the problem of creating an artificial intelligence for playing the card game. We reviewed the foundations of techniques used in artificial intelligences playing games with, notably, Monte Carlo Tree Search.

Next, we underlined the difficulties specific to the game, including the partial observability, the non-determinism, the presence of an opponent, and the real-time space. In addition, we pointed out the exponential branching factor inherent to card games. Furthermore, the game was formally decomposed into subproblems, namely (a) Attacking with creatures, (b) Positioning creatures, and (c) Playing cards.

Afterward, two artificial players were proposed for playing the card game. Firstly, a heuristic player that evaluates moves by quantifying the amount of resources it gains (or loses) versus the amount of resources its opponent gains (or loses). However, significant limitations of the heuristic method were highlighted. In particular, the move maximising the player's short-term reward may not correspond to the move maximising its chance to win in the long-term.

Secondly, we proposed a more advanced player that simulates the long-term consequences of a move to predict its outcome. The challenging task of understanding a card's effect was solved by playing that card on a copy of the game, simulating it until the end of the game, and observing the final result. This provided to the artificial player anticipation skills with, in particular, the ability to predict a game's outcome before it ends.

Lastly, the performances of the artificial players were assessed by analysing their win rates. In a first experiment, the heuristic player and the simulating player competed against each other in 1,000 matches. Results have shown the simulating player to outperform the heuristic one, with 78,1 % of matches won by the former. By predicting the long-term consequences of moves, the simulating player has a stronger ability to planify its strategy, whereas the heuristic player always chooses the move maximising its short term reward.

In a second experiment, the simulating player was confronted to an expert human player. Although the results highly depend on the human player, this experiment still allowed to roughly estimate the artificial player's quality of play. The simulating player achieved a decent 38 % win rate, which is remarkable given that the human player was an expert.

In conclusion, simulations were shown to be a reliable method for playing a strategic card game. These results suggest that simulation-based techniques may be a key feature in future artificial intelligences playing games, and in particular for playing card games.

References

[1] Activision Blizzard Inc. (2016).
Whispers of the Old Gods™ takes hold of Hearthstone® as registered players top 50 million.
http://investor.activision.com/releasedetail.cfm?releaseid=966955

[2] Statista. (2016).
Number of Hearthstone: Heroes of Warcraft players worldwide.
http://www.statista.com/statistics/323239/number-gamers-hearthstone-heroes-warcraft-worldwide/

[3] Game Zone. (2014).
Why Blizzard was able to succeed with Hearthstone when the World of Warcraft Trading Card Game didn't pan out.
http://www.gamezone.com/originals/why-blizzard-was-able-to-succeed-with-hearthstone-when-the-world-of-warcraft-trading-card-game-didn-t-pan-out

[4] Gamepedia. (2016).
Hearthstone wiki. Secret.
http://hearthstone.gamepedia.com/Secret

[5] Ben Brode, Lead Designer on Hearthstone. (2014).
Senior Game Designer Ben Brode talks Hearthstone.
http://www.gosugamers.net/hearthstone/features/38103-senior-game-designer-ben-brode-talks-hearthstone

[6] Gamepedia. (2016).
Hearthstone wiki. Minion card list.
http://hearthstone.gamepedia.com/Minion_card_list

[7] Apac Games. (2014).
China's made a real-life Hearthstone set (and now we know why Blizzard hasn't).
http://www.apacgames.com/chinas-made-a-real-life-hearthstone-set-and-now-we-know-why-blizzard-hasnt/

[8] App Annie. (2016).
Hearthstone: Heroes of Warcraft app rank history.
https://www.appannie.com/apps/ios/app/hearthstone-heroes-warcraft/

[9] SuperData. (2015).
Digital card games report.
https://www.superdataresearch.com/market-data/digital-card-games/

[10] Blizzard Entertainment. (2016).
Hearthstone: Heroes of Warcraft.
http://eu.battle.net/hearthstone/en/

[11] Guinness World Records. (2016).
First modern trading card game.
http://www.guinnessworldrecords.com/world-records/first-modern-trading-card-game

[12] Wizards of the Coast. (2015).
Magic: The Gathering.
http://magic.wizards.com/en/content/new-to-magic/community

[13] University of Washington. (1998).
Wizards of the Coast.
http://faculty.washington.edu/skotha/website/cases%20pdf/Wizards%20of%20the%20coast%201.4.pdf

[14] Wizards of the Coast. (2013).
Magic: The Gathering Basic Rulebook.
http://media.wizards.com/images/magic/resources/rules/EN_MTGM14_PrintedRulebook_LR.pdf

[15] Gamepedia. (2015).
Magic: The Gathering statistics and trivia.
http://mtgsalvation.gamepedia.com/Magic:_The_Gathering_statistics_and_trivia

[16] Wizards of the Coast. (2015).
Gatherer.
http://gatherer.wizards.com/Pages/Default.aspx

[17] Wizards of the Coast. (2016).
Magic: The Gathering Comprehensive Rules.
http://media.wizards.com/2016/docs/MagicCompRules_01162016.pdf

[18] List of Magic: The Gathering video games. (2016).
https://en.wikipedia.org/wiki/Magic:_The_Gathering_video_games

[19] SteamSpy. (2016).
Magic Duels app data.
http://steamspy.com/app/316010

[20] Nick Wilson. (2015).
How Magic Duels: Origins is adapting to the evolution of the digital CCG.
http://www.pcgamesn.com/magic-duels/how-magic-duels-origins-is-adapting-to-the-evolution-of-the-digital-ccg

[21] American Obsessions. (2015).
Magic: The Gathering - Inside the World's Most Played Trading Card Game.
https://www.youtube.com/watch?v=Plr81gaUIr0

[22] Luke Plunkett. (2011).
When Pokémon and Magic Cards Went to War.
http://kotaku.com/5859130/when-pokemon-and-magic-cards-went-to-war

[23] Brian Tinsman. (2008).
The Game Inventor's Guidebook: How to Invent and Sell Board Games, Card Games, Role-playing Games, & Everything in Between!, page 103.

[24] Kallie Plagge. (2014).
Pokémon Trading Card Game Online review.
http://www.ign.com/articles/2014/10/20/pokemon-trading-card-game-online-review

[25] Jay Hornung. (2014).
“Mind If I Roll Need?” – What PTCGO Can Learn from Hearthstone, Blizzard’s Latest Cash Cow.
http://sixprizes.com/2014/12/17/mind-if-i-roll-need-what-ptcgo-can-learn-from-hearthstone-blizzards-latest-cash-cow/

[26] Alex Langley. (2014).
Pokemon: Trading Card Game Online Review.
http://arcadesushi.com/pokemon-trading-card-game-online-review-ios/

[27] Konami Digital Entertainment. (2009).
Yu-Gi-Oh! Card Sales Set New World Record.
http://www.konami.jp/topics/2009/0807/index-e.html

[28] Guinness World Records. (2016).
Best-selling trading card game company - cumulative.
http://www.guinnessworldrecords.com/world-records/best-selling-trading-card-game

[29] List of Yu-Gi-Oh! video games. (2016).
https://en.wikipedia.org/wiki/List_of_Yu-Gi-Oh!_video_games

[30] Konami Digital Entertainment. (2015).
Konami Digital Entertainment Announces Release of New "Yu-Gi-Oh!" Content for Multiple Devices in 2016. URL
http://www.konami-digital-entertainment.co.jp/en/news/release/2015/1221/?cm_sp=01-_-release-_-20151221

[31] Eric Dodds, Game Director on Hearthstone. (2014).
GDC Vault. Hearthstone: 10 Bits of Design Wisdom.
http://www.gdcvault.com/play/1020775/Hearthstone-10-Bits-of-Design

[32] Ben Brode, Lead Designer on Hearthstone. (2013).
Ben Brode Talks 'The Coin'.
http://hearthstone.blizzpro.com/2013/09/12/ben-brode-talks-the-coin/

[33] Kyle Orland. (2014).
Do you want to go first? Balancing Hearthstone and other turn-based games.
http://arstechnica.com/gaming/2014/08/do-you-want-to-go-first-balancing-hearthstone-and-other-turn-based-games/

[34] Gamepedia. (2016).
Hearthstone wiki. Secret.
http://hearthstone.gamepedia.com/Secret

[35] Ben Brode, Lead Designer on Hearthstone. (2014).
Senior Game Designer Ben Brode talks Hearthstone.
http://www.gosugamers.net/hearthstone/features/38103-senior-game-designer-ben-brode-talks-hearthstone

[36] Ben Brode, Lead Designer on Hearthstone. (2016).
Designing Competitive Hearthstone with Ben Brode.
https://www.youtube.com/watch?v=aLcFhhwSP6U

[37] Richard Garfield , Designer of Magic: The Gathering. (2012).
Luck Versus Skill.
http://www.channelfireball.com/videos/magic-tv-extra-dr-richard-garfield-on-luck-versus-skill-magic-cruise-2012/

[38] Ben Brode, Lead Designer on Hearthstone. (2015).
Designer Insights with Ben Brode: Consistency.
https://www.youtube.com/watch?v=lfkJnKBuLSU

[39] HearthPwn. (2016).
Hearthstone database.
http://www.hearthpwn.com/cards?filter-name=injured&display=3

[40] Sachan J. (2013).
A Combination of Video Games and Artificial Intelligence.
http://ijset.in/wp-content/uploads/2014/02/IJSET_101100122013.pdf

[41] Browne, C.B., Powley, E., Whitehouse, D., Lucas, S.M., Cowling, P.I., Rohlfshagen, P., Tavener, S., Perez, D., Samothrakis, S., and Colton, S. (2012).
A Survey of Monte Carlo Tree Search Methods.
IEEE Transactions on Computational Intelligence and AI in Games, Vol. 4, No. 1, pp. 1–43.
http://repository.essex.ac.uk/4117/1/MCTS-Survey.pdf

[42] Neumann, J. von and Morgenstern, O. (1944).
Theory of Games and Economic Behavior.
Princeton University Press, Princeton, NJ, USA, second edition.

[43] Nijssen, J.A.M. and Winands, M.H.M. (2012a).
An Overview of Search Techniques in Multi-Player Games.
Computer Games Workshop at ECAI 2012, pp. 50–61, Montpellier, France.

[44] Knuth, D.E. and Moore, R.W. (1975).
An Analysis of Alpha-Beta Pruning.
Artificial Intelligence, Vol. 6, No. 4, pp. 293–326.

[45] Michie, D. (1966).
Game-Playing and Game-Learning Automata.
Advances in Programming and Non-Numerical Computation (ed. L. Fox), pp. 183–200. Pergamon Press.

[46] Wikia. (2016).
Solved Game.
http://mancala.wikia.com/wiki/Solved_game

[47] Schaeffer, J., Burch, N., Björnsson, Y., Kishimoto, A., Müller, M., Lake, R., Lu, P., and Sutphen, S. (2007).
Checkers is Solved.
Science, Vol. 317, No. 5844, pp. 1518–1522.

[48] Allis, L.V. (1988).
A Knowledge-based Approach of Connect-Four. The Game is Solved: White Wins.
M.Sc. thesis, Department of Mathematics and Computer Science, Vrije Universiteit, Amsterdam, The Netherlands.

[49] Shannon, C.E. (1950).
Programming a Computer for Playing Chess.
Philosophical Magazine, Vol. 41, No. 314, pp. 256–275.

[50] Kocsis, L., Szepesvári, C., and Willemson, J. (2006).
Improved Monte-Carlo Search.
Technical report, MTA SZTAKI, Budapest, Hungary; University of Tartu, Institute of Computer Science, Tartu, Estonia.

[51] Szita, I., Chaslot, G.M.J-B., and Spronck, P. (2010).
Monte-Carlo Tree Search in Settlers of Catan.
Advances in Computer Games (ACG 12) (eds. H.J. van den Herik and P. Spronck), Vol. 6048 of LNCS, pp. 21–32. Springer-Verlag, Berlin, Germany.
http://ticc.uvt.nl/icga/acg12/proceedings/Contribution100.pdf

[52] Pepels T. (2014).
Real-Time Monte Carlo Tree Search in Ms Pac-Man.
IEEE Transactions on Computational Intelligence and AI in Games (Volume: 6, Issue: 3).
http://ieeexplore.ieee.org/xpl/articleDetails.jsp?reload=true&arnumber=6731713

[53] David Silver & Aja Huang. (2016).
Mastering the game of Go with deep neural networks and tree search.
AlphaGo, Google DeepMind.
https://new.vk.com/doc-44016343_437229031?dl=56ce06e325d42fbc72

[54] Byford Sam. (2016).
Google vs. Go: can AI beat the ultimate board game?
http://www.theverge.com/2016/3/8/11178462/google-deepmind-go-challenge-ai-vs-lee-sedol

[55] Joe Agajanian and Taylor Brent. (2013).
Monte Carlo Tree Search Magic: The Gathering AI.
https://www.cs.hmc.edu/~jagajanian/cs151/

[56] Damien Ernst, Raphael Fonteneau, and Firas Safadi. (2011).
Artificial Intelligence Design for Real-time Strategy Games.
Decision Making with Multiple Imperfect Decision Makers.
25th Annual Conference on Neural Information Processing Systems (NIPS 2011).
Sierra Nevada, Spain.
http://www.montefiore.ulg.ac.be/~fsafadi/nips2011.pdf

[57] Nijssen, J.A.M. (2013).
Monte-Carlo Tree Search for Multi-Player Games.
Maastricht ICT Competence Centre, Maastricht University, Maastricht, The Netherlands.
https://project.dke.maastrichtuniversity.nl/games/files/phd/Nijssen_thesis.pdf

[58] Gamepedia. (2016).
Hearthstone wiki. Trade.
http://hearthstone.gamepedia.com/Trade

[59] Davison Andrew. (2005).
Killer Game Programming in Java, page 43.

[60] Elie Bursztein. (2014).
Predicting Hearthstone opponent deck using machine learning.
https://www.elie.net/blog/hearthstone/predicting-hearthstone-opponent-deck-using-machine-learning#.VAdktPldWb8

[61] Gamepedia. (2016).
Hearthstone wiki. Meta.
http://hearthstone.gamepedia.com/Meta

[62] Brian Schwab , AI Engineer on Hearthstone. (2014).
Gamasutra. Building the AI for Hearthstone.
http://www.gamasutra.com/view/news/224101/Video_Building_the_AI_for_Hearthstone.php

University of Liège

Development of a Video Game and Creation of an Artificial Player

Master Thesis Submitted for the Degree of
MSc in Computer Science

The purpose of this work is twofold. In a first part, we developed a real-time strategic card game from scratch. It included the design of game rules, a user interface, cards, decks, and the implementation of the game in java with neither a game engine nor an external library. The game is playable online or offline, against other human players or artificial players, and with pre-built or custom decks.

In a second part, two artificial players were created for playing the card game. Firstly, a heuristic player that evaluates moves by quantifying the amount of resources it gains (or loses) versus the amount of resources its opponent gains (or loses). Secondly, a more advanced player that simulates the long-term consequences of a move until the end of the game to predict its outcome.

Results have shown the simulating player to outperform the heuristic one. By predicting the long-term consequences of moves, the simulating player has a stronger ability to planify its strategy, whereas the heuristic player always chooses the move maximising its short term reward. In addition, the simulating player's performances are sufficient to challenge an expert human player.

Author:
Nicolas LORENT

Supervisor:
Prof. Damien ERNST

Academic year 2015 - 2016