Dynamic Game Language, A Rapid Game Prototyping System

by

Travis McPhail

A Thesis Submitted
in Partial Fulfillment of the
Requirements for the Degree

Master of Science

Approved, Thesis Committee:

Joe Warren, Professor
Computer Science

Ron Goldman, Professor
Computer Science

Devika Subramanian, Professor
Computer Science

Houston, Texas

May, 2007
INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.
Dynamic Game Language, A Rapid Game Prototyping System

Travis McPhail

Abstract

The computer game industry is an ever growing component in today's society. With the increased success of the industry comes increased demands on the quality of the games. This high-quality standard coupled with cutthroat time constraints has placed limitations on the level of creativity that we see in computer games. We wish to increase programmer productivity by providing tools that rapidly prototype games. In this thesis, we present a system for prototyping arbitrarily complex 2D card games. This system is packaged with a card game programming language, Dynamic Game Language (DGL). This framework takes game descriptions which focus solely on game logic and infers user interfaces and networking engines in order to provide quick feedback on the playability of game ideas.
Acknowledgements

I would like to thank God for giving me the resources and perseverance to complete this thesis topic. I would like to thank Joe Warren, my advisor, for all of his support through this endeavor. I would also like to thank my committee members for their helpful comments and questions. Finally, I would like to thank my parents who prepared me to get to this point. This thesis is dedicated to my Lord and Savior, Jesus Christ.
5.2.3 Future Work ........................................... 56

Bibliography ........................................... 58
Illustrations

1.1 SpaceWar! released in 1962. .................................................. 2
1.2 Super Mario Brothers, released on the Nintendo Entertainment
    System, NES (left). Donkey Kong, also released on the Nintendo
    Entertainment System. .................................................. 3
1.3 Super Mario 64, released for the Nintendo 64 (left). Final Fantasy
    VII, released on the Playstation (right). Notice the graphical
    improvements compared to the games depicted in figure 1.2 ......... 4
1.4 Metal Gear Solid II, released for the Playstation 2 (left). Halo 2,
    released for the XBox (right). ........................................ 5
1.5 The first version of John Madden Football released in 1990 for the
    Super Nintendo (left). A 2001 version of John Madden Football on
    the Nintendo 64 (right). ............................................. 6
1.6 The 2006 versions of John Madden Football for the Playstation 2
    (left) and the XBox 360 (right). ..................................... 6
1.7 Visual layout of the prototype development process. ................. 7
1.8 Screenshot from one of Warcraft 3’s most popular mods, DotA. ..... 11
1.9 A screenshot of Poker produced from a game description in EGGG. . 14

2.1 A phase transition diagram of Texas Hold’em (left) and Magic: The
    Gathering (right). This maps the progression of the game. The
    orange outline represents a possible winning phase. ................ 20
2.2 Example cards from Texas Hold’em (left) and Magic: The Gathering (right). The circled items are the attributes of the cards. 23

2.3 Board game that uses cards mercenary and modifier cards in its progression. 24

3.1 DGL game description file for Speed (left). Corresponding actions (right). 30

3.2 Example of object declaration in DGL. 31

3.3 Example of player and zone declarations in DGL. 32

3.4 Example of timer and game event declaration in DGL. 32

3.5 The initialization phase for the DGL version of Speed. 33

3.6 The main phase for the DGL version of Speed. 33

3.7 The flip rules for the DGL version of Speed. 34

3.8 The play event along with the deadlock and play rules for Speed. 34

3.9 The draw event and draw rules for Speed. 35

3.10 Example of 5 listeners and the order in which they appear in the priority queue given the order that they are introduced. The sequential ordering is from oldest to most recent. Events are passed to the start of the queues located at the bottom. 38

3.11 Example of adding and removing listeners from an event type. 39

3.12 Example of dynamic rules in a game. The modifiers have labels that tell the order in which they appeared. The attacker cards have their raw power circle followed by their first modification (red), second modification (green), and third modification (blue). 41
4.1 Screenshot from DGL version of Texas Hold’em. This shows the
        general layout of the user interface provided by the system. A subset
        of the visible zones are outlined in blue. A subset of visible objects
        are outlined in red. All of the player events are outlined in yellow. . . 44

4.2 Sample event declarations in DGL . . . . . . . . . . . . . . . . . . . . . . . . . . . 48
Chapter 1

Introduction

1.1 Overview of Computer Games

Computer games are ubiquitous in today’s society. Adults and children alike flock to stores like GameStop, BestBuy, and EB Games and wait in lines for days in anticipation of receiving the latest game system when it’s released. People pre-order new games months before they debut. Movies and cartoons such as Tomb Raider and Resident Evil have been spawned from computer games. Parents can find clothing and backpacks for their children with computer game paraphernalia at the neighborhood Walmart. There are even TV channels dedicated to computer gaming. Society is captivated by games and has been more than two decades [4, 9].

But before we can investigate computer gaming, we need to define the concept of a game. In this document, a game is a simulation of various activities (primarily recreational) that is defined by a set of rules. This activity involves people interacting with one another as well as other possible objects based on the rules. The constraints place by these rules create the challenge and appeal of the game. What is a computer game? A computer game is a computer-controlled game with some visual display for human feedback and some type of input device(s) (i.e. joystick(s), controller(s), keyboard(s), etc.) Note that in this thesis, unless a game is explicitly described as a card game or a classic game, it is a computer game.
1.2 The Game Industry

1.2.1 The Beginning

The computer game business is a fast-paced and growing industry. From its humble beginnings, the game industry has developed into a multi-billion dollar powerhouse. Before the computer game era, there were primarily two classes of classical games: indoor games such as board/card games and outdoor games. There was a wide range of things to do, but games were limited to people's physical and mental capabilities. Then in the 1950s, the first computer games were introduced. These games were created for mainframes and oscilloscopes [2]. The earliest of these computer games was a missile simulation created in 1947 by Thomas T. Goldsmith and Estle Ray Mann. The game employed cathode-ray tubes and was called the "Cathode-Ray Tube Amusement Device". Before this game, simulating missile fire without the actual physical devastation was not deemed possible. This game led to the later development of other games such as Tennis for Two and Noughts and Crosses. Each of these games simulated a new experience, which in turn, sparked their appeal. Soon afterwards, Steve Russel developed Spacewar! [4] (see figure 1.1).

![SpaceWar!]  
Figure 1.1 : SpaceWar! released in 1962.

Spacewar! is a two-player game, where each player controls a spaceship and their
goal is to destroy one another. Spacewar! was played across the country and was the first game to become truly popular. These games paved the way for creating machines for the purpose of playing games. Ralph Baer, the Father of Video Games, designed the first video game console, *Television Gaming and Training Apparatus*, in 1967. Baer released the first actual game console, with Magnavox, entitled *Magnavox Odyssey* in 1972 [4].

1.2.2 The Boom

The rapid rise of the computer gaming industry began in the 1980s with the release of Atari’s *Atari 2600*, Nintendo’s *Nintendo Entertainment System*, and Sega’s *Genesis*. A surge of independent game developers arrived on the scene during this period, creating some timeless games and characters such as Mario Brothers and Donkey Kong (see figure 1.2).

![Figure 1.2: Super Mario Brothers, released on the Nintendo Entertainment System, NES (left). Donkey Kong, also released on the Nintendo Entertainment System.](image)

In comparison to today’s standards, these games systems are simple, primarily because they are two-dimensional [9]. This relatively easy programming environment gave game developers freedom to create new and innovative game ideas. New genres of games emerged: 2D role playing games, fixed perspective sports games, side-scrolling adventure/fighting games, puzzle games, etc. All of these types of games had their
own appeal and at the time, they were novel ideas. The creativity of the game developers was a major contributor to industry's growing success. Then a new era came into fruition, the 3D age [9, 2].

Figure 1.3: Super Mario 64, released for the Nintendo 64 (left). Final Fantasy VII, released on the Playstation (right). Notice the graphical improvements compared to the games depicted in figure 1.2

In 1994-96, three new 3D game platforms came onto the scene: the Sega Saturn, the Nintendo 64, and the Sony Playstation. These new systems provided increased flexibility in comparison to the previous generation of game consoles, the main improvement being 3D visualization. Games such as Virtual Fighter 2, Super Mario 64, and Final Fantasy VII became huge successes (see figure 1.3). People were immersed in more realistic environments and captivated by the smooth motion of game characters. New genres emerged such as 3D fighter games and 3D adventure games. The already existing genre, first-person shooter, became extremely popular. The experiences that these new types of games offered were graphically revolutionary. Game developers needed to come up with 3D environments that were more detailed than their 2D predecessors. In contrast to previous years, companies needed to pay greater attention to small visual details such as: shadows, transparency, character animation, and physics interactions. These extra needs spawned thousands of jobs for artists and game designers to push the envelope on the look of the games being developed. New types of technology were developed through research in order to drastically reduce
the amount of time needed to provide these superior graphical experiences. Due to the extra visual complexities of 3D games, the instruction sets for these systems were more complex, so the focus began to shift from making new types of games to making better looking games [4, 9, 2].

Figure 1.4: Metal Gear Solid II, released for the Playstation 2 (left). Halo 2, released for the XBox (right).

The next generation of consoles came out during 1999-2001. They included the likes of Sega’s Dreamcast, Sony’s Playstation 2, and Microsoft’s XBox. Like their predecessors, these consoles offered more than the previous generation, creating even more complicated environments to develop software titles. With the increase in the capabilities of the new hardware consoles, the visual demands for games increased. Titles such as Metal Gear Solid II and mega-hit Halo became the new industry standards (see figure 1.4). The industry had now become a well-oiled machine. Startup companies were now multi-million dollar players in a business with seemingly endless possibilities.

1.3 Rapid Prototyping

Now with the aversion to risk of the industry and the high expectations on visual quality, a trend towards developing sequels has become prominent. Due to the shift towards creating aesthetically more pleasing games, game play has suffered. For
example, let’s look at Electronic Art’s *Madden NFL* franchise. The first console version for this franchise appeared on the Sega Genesis in 1989 followed by the 1990 release on the Super Nintendo (see figure 1.5, left).

The game play elements from the first version of *Madden NFL* and the most recent version released 17 years afterwards for the XBox 360 (see figure 1.5, right), are essentially the same. But, there are notable differences in the look of the game. The new hardware technologies have provided many visual facelifts to the game. However, the way in which humans interact with the game has only received minute touch-ups. The *Madden* franchise is one of the most glaring examples of the sequel
phenomenon, yet franchises such as *Grand Theft Auto*, *Final Fantasy*, and *Halo* also suffer from this defect.

This continual develop of game sequels lead us to the following questions: Why are new game ideas not being created? Have we reached the peak of our creativity? Are there no more new games to create? It is hard to believe that new types of games with new experiences for people cannot be created. So let’s ask a better question: Why has the development of sequels become prevalent? Let’s take a look at the game development cycle of the computer gaming industry. In today’s market, games require a substantial amount of resources. Game companies divide these resources into 3 major areas: game design - the overall idea for a game, game art - the visual appeal of the game, game engines - the underlying intelligence of the game. At the top-level, a team of designers interact with one another to create new game ideas. After the game concept is created, designers delegate tasks to company’s artist and programmers in order to build working a prototype, or first-cut, of the game [12, 3].

Figure 1.7 : Visual layout of the prototype development process.
The artists look at the designers idea and try to create an environment that reflects the major themes of the game. For example if game designers were developing an archeological game, artists would create environments that reflect archeological dig sites or ancient ruins. The artists begin this process by creating mock levels, characters, and user interfaces. These preliminary items serve as the visual crux for the prototypes. On the other end of the development spectrum, programmers create the systems that include all of the desired game elements. They work on different engines in order to get the infrastructure ready so that their efforts can be integrated with those of the artists. This integration process seems smooth, but the amount of time and man power needed to construct these demos can be substantial [12, 3].

The prototyping process has several inefficiencies. One major flaw is the belief that artists and programmers can successfully work in parallel even though they depend on another’s efforts. Programmers do as much as they can without the content provided by the artists, but eventually their work is stalled by the productivity of the artists and vice versa. There are complicated physics engine which govern the interactions between the models and the environment. There are rendering engines that integrate a number of special effects and animation/lighting features. There can be networking engines to handle communication between different users on varied platforms. User interfaces engines can become complex depending on the needs of the game. Together, these systems have hundreds of thousands of lines of code. It takes artists a great deal of time to come up with interesting level concepts and take advantage of the game play elements for the current project. If this is not bad enough, the game design usually changes during the development process. Due to this backtracking and the possible new technologies and art needed for these changes, prototyping is slowed down even more.

Like the automotive industry, millions of dollars worth of time and manpower are spent on this prototyping cycle alone. What happens if the costly demo is not fun to play? A company has wasted an enormous amount of time and money, which
is not a profitable strategy. It is far easier to use previous game ideas (along with their code base) as a framework for new development. Designers add a few new game play elements. Programmers tweak some of the functionality of their game engines and use some extra features to improve in-game rendering. Artists add new art, and voila, a new game is created. But even this set of changes has a life cycle that spans months, because there needs to be enough new epsilon changes to entice people to purchase the game. Now if companies have to decide between spending months on a set of epsilon changes or risking loss developing a completely innovative game, they will undoubtedly go with the first option. Companies are focused on producing revenue. In order to persuade the industry to promote new ideas, the duration of the development cycle must be reduced.

The main goal of this thesis is to show how to prototype game ideas quickly so that companies have more incentive to create new games. The game industry cannot survive on new impressive visualization technologies alone. Innovative game ideas must come to the forefront for the industry to stay prosperous. In 10 years, the same Halo-esque games will not sustain the XBox franchise. This thesis is not a complete solution to this problem; it is a step towards this goal.

1.4 Previous Work

In this section, we will discuss similar approaches to this problem or related problems. We will compare and contrast the goals and successes of other individuals to our own approach. Primarily there have been two major approaches: game modding (or game modification) and game prototyping (or generating) scripting languages or systems. The first approach has a great deal of support and many games have been created with it; the second is not as popular.
1.4.1 The Mods

Game mods are a popular avenue for creating new types of games. A game mod is simply a modification of an existing game. The modification can be small, i.e. changing the look of characters in a game. However, the change can be substantial, virtually creating a new game [5]. But for the most part, mods add new game content such as weapons, characters, game modes, levels, and story lines. So why has this approached received attention? Game mods:

- Encourage gamer creativity
- Distribute easily across the internet
- Extend replay value of existing games

Allowing gamers to create new content in an existing game, relaxes the pressures of creating new game ideas for developers. Mods allows the gamers to do the work for the developing companies. Many modern games like Half-Life and Warcraft 3 were designed so that they could be modified without a great deal of difficulty. Game expansions such as Counter-Strike for Half-Life and DotA for Warcraft 3 (see figure 1.8) have received positive reviews. Sometimes, the modified game outshines the original. Counter-Strike is an example of such a game mod.

Mod-friendly games usually come with a suite of functions provided in their game engines that people can either override or implement to create new game experiences. This extensibility creates new experiences for all users, but it places heavy constraints on the types of games that can be built. A mod can only stray so far from its ancestor. The majority of the code used for the new game is already present in the original version. This reusable code is the appeal of modding games. A non-professional programmer can change the manner in which they interact with a game with a relatively small amount of effort. However, if someone wants to create a completely different experience, he/she must either build a new game engine or spend
a great deal of time adjusting the low-level functionality of the existing engine (if it is possible to edit the engine). It would take a great deal of work to create a basketball game from the game engine used in Warcraft 3. The code for Warcraft 3 is written to create a professional quality product even with outside user modification. But this professional product has to stay in the realm of Warcraft 3.

1.4.2 Game Generation Tools

The other approach to developing games quickly is to create tools to abstract game play concepts and enable people to create games on a higher level than programming languages such as C++ and Java. Game generation tools are broken into two groups: text adventure game tools and prototyping/archival tools.
Text Adventure Games

Text adventure games are games that are controlled by textual input. These games are visualized either by textual output or graphical output. Text adventure games have an underlying decision tree structure which corresponds to all possible decision paths. Depending on the choices a player makes, the story changes course. The crux of a text adventure game is the story progression that is generated by a player's choices. A great deal of effort is placed in creating the different possible scenarios, keeping the participants intrigued as if they are reading a novel.

The textual input for these games are written in languages comparable to English. Some types of inputs are strict subsets of the English language. This type of input is impressive because there are inherent ambiguities in human language that need to be explained to computers. The text-based game does not know the meaning of: "Go into the next room". A programmer provides that meaning. Scripting languages have been developed to speed up prototyping for these games. Two popular tools are SAGE (the Scripting Adventure Game Engine) and Inform [5, 10]. If someone wishes to create a game, they provide the different scenes, or rooms. If the game has a graphical component, the developer must supply the images as well. Now, items are placed in the scene, and action/outcome pairs are defined for the different rooms. A nice feature of this system is the user-friendliness of the language. The language is remarkably similar to English, but restrictive in the type of commands that can be issued.

Prototyping and Archival Tools

Creating tools that can aid developers in building and archiving games is a difficult problem. One has to find commonalities between games and be able to exploit these items to provide tools that allow users to think on higher-levels, the levels that contain these commonalities. At the same time however, one wishes to allow for expressiveness of new ideas. One problem with creating these types of tools is finding
the appropriate balance between convenience and expressiveness. As a game becomes more complicated, there are fewer commonalities between it and other games. This lack of commonality forces tools to provide support for handling more general classes of games. The wider this support base, the less effective the tool. On the other hand, as games become similar, more abstractions emerge which can be exploited to allow for tools that are extremely convenient. The fundamental question then becomes: Should the tool be convenient and only cover a small class of games, or expressive and cover a wide variety of games? Due to the novelty of this area of research, this approach has received little attention. The major contributor in this area has been Jon Orwant.

EGGG: The Extensible Graphical Game Generator

The first true attempt to look at a set of games and find the commonalities between them was EGGG, developed by Jon Orwant in February of 2000 [11]. Before Orwant created EGGG, he was affiliated with the Programmer’s Apprentice Project at MIT Media Laboratory. The Apprentice project focused on making programs that aided programmers in creating, editing, and analyzing programs. The class of programs was wide and diverse targeting seasoned programmers. Orwant’s main goals for EGGG were to [11]:

- Develop a system for creating synchronous 2D games (primarily card games) using concise representations
- Provide a convenient language that is similar to English
- Build a system that infers a user-interface from a game description

Orwant succeeded and this thesis is influenced greatly by his work. EGGG is a system, based in the Pearl scripting language, that reads relatively short descriptions of the rules of games and infers a playable version for the user. His system is very
compact and convenient for describing a certain class of games. Orwant's poker
description is only 19 lines of code. Overall EGGG does a decent job at handling his
primary concerns; however, the class of games that can be represented compactly is
small.

Figure 1.9 : A screenshot of Poker produced from a game description in EGGG.

1.5 Our Contribution

Where can we go from this point? How can we improve upon the progress of EGGG?
We wish to create a system for prototyping arbitrarily complex card games quickly
and describing existing card games concisely. This involves:

- Compactly describing both asynchronous as well as synchronous games
- Describing games in which their set of rules can change during execution
• Inferring the visualization of games where the physical layout changes frequently

To better understand our goals, let's contrast our approach with that of EGGG. EGGG is good at describing games of a certain level of complexity such as: Tic Tac Toe, Chess, Poker, Crosswords, Tetris, Rock Paper Scissors, Deducto, and Mammon. These games are described in Orwant's thesis. The rules are simple and Orwant demonstrates that a great deal can be inferred about their structure. But this level of complexity is not high enough for our particular needs. Do not misunderstand what is being said here. EGGG can describe arbitrarily complex games, but in order to do so, one must begin writing in the lower-level Perl scripting language. DGL desires to raise this bar. We wish to chip away at some of the conveniences that EGGG provides in order to concisely represent a larger class of games while making it easier to represent arbitrarily complex games. In short, we are not trying to force the amount of text for game instructions written in English and game descriptions written in DGL to be equivalent. We want the English instructions and the game descriptions to be comparable. Another one of our major goals is to represent games in which the set of rules can change dynamically as the game progresses. So as new entities are introduced into a game or new events occur during the game, the rules can be altered. Why is dynamic rule modification desirable? As stated before, in order to make the development of new game ideas attractive to industry, the prototyping life span needs to be drastically shortened. Allowing individuals to change the rules of a game during execution is a large step towards tackling this problem.

We also desire to represent both synchronous and asynchronous games. We didn't wish to limit ourselves to turn-based games, and when we discuss our taxonomy in Chapter Two, we will show that DGL naturally classifies asynchronous games. Synchronous games can be described as asynchronous games with rules that enforce turn-based play. Finally, we want to create a system that infers all of the visualization needs of described games. Because of the previous goal of modeling games with modified rules, entities in games have the potential to move frequently and the layout
of the game can potentially change radically during its execution.

1.6 The Focus, Card Games

In this section, we discuss the class of games we chose to focus on, card games. We give the reason for this choice then we discuss the history of card games. These topics will lay the foundation for our later discussion on the commonalities between card games in chapter two.

1.6.1 Why Card Games?

There are more games out there than 2D card games. To name a few, there are:

- Action/adventure games
- Fighting games
- Puzzle games
- Role playing games
- Sports games
- Strategy games

In order to create a system that can describe all of these games, we need to produce the minimum set of distinct game elements that, when abstracted out, allow us to describe these types of games with a small to moderate amount of effort. So let’s begin.

For shooter games there need to be entities that represent weapons. The weapons should have some adjustable parameters that allow them to modify sets of attributes from other entities based upon their respective positions. Shooter games also need a notion of continuous motion for the characters, and if we are dealing with 3D shooters, a physics engine needs to be added.
Let's now look at adventure games. We need some type of dependency graph that allows us to encode all of the significant events or goals in the game as nodes and their dependencies as edges. In order for a player to accomplish a goal, previous dependent goals must be met as well. This is similar to the notion of backward chaining in artificial intelligence [7]. Adventure games as well as role-playing games need some type of text or voice based system so that information is relayed to the player in some meaningful way. Fighting games need a means to encapsulate a finite set of moves which are effective only in certain character configurations. For this type of game, the physics engine would have to be more sophisticated in order to handle these complex needs. We could continue along this train of thought, but the point is clear. To successfully establish commonalities and provide the intelligence to infer these abstract concepts is quite cumbersome and ambitious. Attempting to accomplish a general purpose game prototyping system is beyond the scope of a master's thesis.

So why card games? Card games:

- Have well-defined structures
- Can incorporate dynamic rules
- Have concise taxonomy allowing for meaningful abstractions

### 1.6.2 History of Card Games

To begin to understand the commonality in card games, let us explore their origins. As we look at their history and see the different types of games that have been created through time, we can begin to understand the similarities existing between card games. With this information, we will be better prepared to provide a tool for prototyping new types of card games as well as archiving established games.

The first known card games originated in Central Asia in the 10th century. These games used paper pieces that were the basis for what we know today as dominoes. The pieces were derived from the cubic dice and each piece represented one of the
twenty-one distinct results from a roll of two die. One half of each piece was set with the pips from one die and the other half was set with the pips from the second die. There were a number of games that used these pieces and have spawned games that we still play today such as: Mah Jong, Pai Gow, and Tien Gow [6].

The standard card decks that we know today originated in the Islamic culture. The Islamic culture was the first to introduce four-suited cards with a special class of "court" cards (King, Viceroy, and Second Deputy). This convention was later imported by Europeans in the late 1300s. The French set the four modern suits, spades, hearts, clubs, and diamonds, and also introduced the royal court: Ace, King, Queen, and Jack. The games that were derived from the card decks of this time have had a great deal of social significance. Shedding games such as Bartok granted an enormous enthusiasm (partially due to the drinks that were consumed during them). Gambling games such as Blackjack and Poker have enticed people by intermingling the elements of skill and chance [6]. On the other hand, Go Fish was created as a means to bring pagan children to God. It was based upon the story of Jesus instructing Peter to "go fish" a gold coin from a fish's mouth.

Today, there is a large collection of card games that are no longer bound by the constraints of the standard 52 card deck and a fixed rule set [6]. Collectible card games such as Flux, Pokemon, and Magic: The Gathering have sparked new interests in card games. The constant creation of new cards and game elements has attracted millions.
Chapter 2

Game Generator Theory

2.1 Card Game Taxonomy

With such a large class of card games, one must ask: How does one design a game generator that will not only describe these games compactly, but will also express new, innovative game ideas in a concise manner? Before one can tackle this question in its entirety, meaningful abstractions must be discovered to express card games. Is Speed in an similar class of games as Spades? How is Solitaire related to Magic: The Gathering?

We will begin by classifying the known card games. We can place card games into two categories: games with a fixed set of rules and games with a changing set of rules. The first group of games come with a set of instructions that cannot be modified during the execution of a game. The second group also starts with a rule set; however as the game progresses, new game entities can add, remove, or modify the existing set of rules.

Let us start by focusing on the games with a fixed rule set. The set consists of trick-based, matching, gambling, single player, and shedding games. Trick-based games, like Hearts and Spades, focus on dividing rounds into units called "tricks". Matching games such as Gin Rummy and Go Fish focus on grouping cards based on some type of pattern recognition. Gambling games like Poker deal with both cards and money. Single player games such as Solitaire involve attempting to rearrange cards from an initial configuration to a winning configuration. Shedding games such as Speed and Crazy Eights focus on players emptying their hands as quickly as possible [6].

Well this is great: these games have been classified and one can create libraries
that abstract the common elements of these game types. However, focusing on merely classifying this set of games does not give us everything we need to prototype arbitrary card games efficiently. Classifying card games with dynamically changing rules allows us to prototype general card games quickly. If it were possible to create a taxonomy of card games with dynamic rule sets, then the task of creating new games is reduced to introducing entities with attached rules into existing games.

2.2 Card Game Commonalities

Up until this point, we have focused on games with fixed rule sets. Let us take a look at the relationship between games such as Texas Hold'em and Magic: The Gathering, MTG.

![Figure 2.1: A phase transition diagram of Texas Hold'em (left) and Magic: The Gathering (right). This maps the progression of the game. The orange outline represents a possible winning phase.](image)

Texas Hold'em is a variant of Poker. All players have some amount of money. Ignoring the blinds rule, the game proceeds with rounds that are separated into the following phases: the deal, the bet, the flop, the bet, the turn, the bet, the river, the bet, then the end of the round. The dealer deals two cards to every player. All
players then look at their hands and either: make a bet, call the current bet, raise the current bet, or fold. The dealer discards a card and reveals the flop which consists of three cards. Players go through the betting phase once again. The dealer proceeds to discard another card and reveal the turn card. Players analyze their hands and make the appropriate bets once more. The dealer then discards another card then reveals the river card. A final betting round takes place and the player(s) with the best 5 card hand out of the seven available to them wins the collection of money that has been accumulated. Magic: The Gathering is more complicated.

MTG is a complex card game in which new cards that come into play alter the set of rules for the game. This game has many different entities with a variety of win conditions. There are 6 main objects in MTG: land cards, creature cards, artifact cards, enchantment cards, instant cards, and sorcery cards. Magic is a turn-based game. Each player’s turn consists of several phases: beginning phase, main phase, combat phase, main phase, and end phase (see figure 2.1). There are several areas where cards can be located during the course of a game: deck, hand, graveyard, in play, out-of-game. Each player starts the game with 20 life points. In the base set of rules, the game is over when one player has zero life points or when a player is forced to draw a card from their deck but is unable to do so. The simplest method of winning a game is to have creatures you control attack your opponent. MTG will not be explained further; the game changes substantially as time progresses due to hundreds of new cards produced annually.

Restating our goals, we would like to concisely represent Texas Hold-em, but at the same time we wish to have enough abstractions so that modeling games as complex as MTG can be done in a reasonable amount of time. These two games are very different in structure on the surface, but they share common elements. As mentioned, you can see a high-level progression of the game through the phases depicted in figure 2.1. The primary objects in the games are cards even though they have different types of information on them (see figure 2.2). Both games are based off of decks, hands, and
other areas to lay out cards. In short, there is a set of common elements for these games that can be abstracted. These entities are: attributes, objects, zones, phases, events, and listeners.

2.2.1 Attributes

Attributes are the simplest entities of a game. They are the smallest quantity that define the current state of the game. Objects, players, and games consist of an assortment of attributes. The set of valid attributes in a game are numbers, strings, timers, and rules (the representation of rules will be discussed later). In Texas Hold’em, the pot, or total amount of money that all players have bet, is a number attribute of the game. The amount of money that each player has is also an attribute of the respective player. In MTG, the number of life points for a player is a number attribute of that player. It should be noted that all attributes are not necessarily visible to everyone in the game. For example, if two individuals, A and B, are playing a game that forces both players to have hands only visible to themselves (assume the game is based on the standard 52 card deck). Player A would not be able to see the suit and rank attributes on the cards in player B’s hand. One can think of this visibility relation as a function of the form:

Attr × Obj × Player → visible|invisible

Here Attr is the attribute whose visibility needs to be determined, Obj is the player/game/object that the attribute is attached to, and Player is the player who wishes to view the attribute. This function allows for different (and possibly complex) classes of visibility and is crucial in the functionality of certain games. Visibility is a concept which will be explained in detail in the chapter 3.
2.2.2 Objects

Objects are entities with a set of attributes attached to them. Note that by this definition, objects are not equivalent to cards. Cards are a strict subset of objects. In Texas Hold’em, a standard card is an object with: string attribute - suit and number attribute - rank. In MTG, a creature card is an object with: number attribute - power, number attribute - toughness, string attribute - name, etc. An artifact card is an object with an attached set of rule attributes. The individual attributes of objects can have different visibilities as described earlier, but the object itself can also have visibility functions which can supersede that of individual attribute visibilities. The form of the visibility function for objects is:

\[ \text{Obj} \times \text{Player} \rightarrow \text{visible/invisible} \]

Here \( \text{Obj} \) is the object whose visibility is in question and \( \text{Player} \) is the player who wishes to view the object. Objects also have an associated type system. In games such as Crazy 8’s, the standard 52 card deck is used; however, cards with rank 8 have special abilities and can be treated as a different type of card. Another example of this
type system is the game Condottiere, a board game that uses mercenary cards and modifier cards which alter the attributes of the mercenary cards or the progression of the game. Within mercenary cards there are normal mercenaries and the heroine (which has the special ability of being unaffected by modifiers – see figure 2.3). So in Condottiere, there are two main types of cards: mercenaries and modifiers. Within mercenaries there are two sub classes: normal mercenaries and heroines.

Figure 2.3 : Board game that uses cards mercenary and modifier cards in its progression.

This game illustrates a simple example of a multiple inheritance structure with objects. For general card games, multiple inheritance is important and will be modeled in the next chapter.
2.2.3 Zones

Zones are the areas where objects can be placed. Zones dictate the physical layout of a game to the players. Examples in Texas Hold'em would be: the deck, player hands, the discard pile, the flop, the turn, and the river. If chips are added to the game, each player's chip stack and the common pot could also be considered zones. In Magic: The Gathering the zones consist of: the deck, the hand, the graveyard, the in-play zone, the out-of-game zone. In card games, there are two types of zones: List zones (i.e. hands) and Stack zones (i.e. decks). If one were to play these games one would notice that these zones are typically viewed differently. The view details for these zones will be discussed later.

2.2.4 Patterns

Patterns are relations defined between objects. A great deal of logic is used to determine whether certain patterns exists within sets of objects. One can see pattern recognition in Poker, detecting a straight, three of a kind, a flush, a full-house, four of a kind, or a straight flush. One can view Gin-Rummy as a game that tries to find partitions of hands into different sets with run and set patterns defined on the individual subsets in the partition. An even simpler example is detecting an ascending pattern between a card in a player's hand and a card on top of one of the game piles in Speed. For certain classes of games, a significant amount of game logic is determined through the use of patterns.

2.2.5 Events

So far we have discussed the physical entities of a game, the physical layout, and pattern recognition. All of these constructs do little for us if there is no action. Events are the catalyst for actions to take place in the game. In Texas Hold'em, there are betting events with an associated player and monetary amount. There are also call events, raise events, and fold events. In Magic: The Gathering there are a
number of events including: tapping land to add resources for players, and attacking opponents with creatures. Since we are interested in creating networked games with our system, events can be considered the message conduit from players to the actual game, or from client to server.

2.2.6 Listeners

Another extremely important piece of a game are the actual rules. A game would not be a great deal of fun if players tried to interact with a game and could not do anything. Listeners enforce the rules for a particular game. They come with code that represents a set of rules. Listeners wait for particular events. Once the specific event occurs, the listener executes its code. Listener execution is an atomic action in most cases; however, the listener execution can be yielded by the phase construct discussed next. In Texas Hold’em when a person has either made a bet, raised the bet, called, or folded, a rule must be executed that passes priority to the next player in round-robin order.

At this time, it is important to understand the synchronization features of this system. Events can be triggered by players at any time, which implies an asynchronous nature to the games that can be modeled. However, synchronous, or turn-based, games are implemented through the constraints placed by the listeners in the game. So our system is not limited only to turn-based games like our EGGG counterpart.

2.2.7 Phases

Phases are synonyms for game states. Phases are high level constructs that give a road map for the progression of a game. Phases are triggered and terminated by the execution of listener code. Phases are unique in the sense that they only can create/remove/modifier game types, enable/disable events and their listeners, fire events, and return a value. After a phase has executed it’s code, it waits until a series of events causes a listener to end it with a return value. In Texas Hold’em the last
card dealt to the players in the game triggers a rule to start the betting phase. During the betting phase, the system yields in execution while the active player decides to bet, call, raise, or fold. Once the last bet, call, or fold event marks the end of the phase and starts another phase. Within a phase, some subset of the possible actions in a game can take place. For example, a dealer cannot burn a card and reveal the flop in Texas Hold’em while the players are betting. Analyzing both of our example games, we see the following phase structure. In Texas Hold’em we have the dealing phase, betting phase, the flop phase, betting phase, the turn phase, betting phase, the river phase, betting phase, and end of turn phase. In MTG, we have, beginning phase, main phase, combat phase, main phase, and end phase.

2.2.8 Players

The most important entity is the actual player. Players are the driving force of the game, spawning events which govern the game flow. Players also own zones, like a player owning his/her hand. Decision making separates players from the other previously described entities. Despite this uniqueness, players are similar to objects in the sense that they can have attributes (i.e. a player’s money stack in poker).

Players conclude the discussion of the major components that have been discovered in card games. These abstractions will lead to an even-driven system that can represent arbitrarily complex card games. But, how do you design such a system? What is the syntax? How do you visualize games? All of these questions will be addressed in the next chapter.
Chapter 3

DGL: Under the Hood

Now that the structure of card games is understood, we can move towards our game generator and understand how the generator brings these games to life. In this chapter we will discuss how DGL transforms high-level game descriptions into playable games. Our game prototyping system is a networked application that takes game descriptions and produces a visual multi-player experience. In this chapter we will focus on the production of the game itself, not the networking and the automatic visualization. These topics will discussed in the next chapter.

3.1 The Language

In the previous chapter, we discussed a taxonomy of games which allowed us to think of abstract concepts that are common amongst the class of card games. How do we use these concepts to model a programmable system to describe card games?

Let’s start with the language itself. We will give brief overviews of the major abstractions discussed in the previous chapter. These overviews will not serve as a complete manual on the capabilities of DGL. Rather, these explanations will be a discussion of the major themes of the language and how these can be seen in a real example.

If we look at generic description, we will see that the description is a collection of different declarations:

- Player declarations
- Object declarations
- Attribute declarations
- Zone declarations
- Event listener declarations
- Pattern declarations
- Phase declarations
- Function declarations
- import declarations

Attribute declarations tell the system how to build the structure of different object/player types and the game type. The player and object declarations are similar in form: both have a name that identifies the type of object or player followed by an optional list of names that represent object/player types that the newly declared object/player type extends. The actual implementation of inheritance will be discussed in the next section. There is an optional block of code which defines the attribute declarations with possible default values following the object/player declaration.

Zone declarations tell the system the name of the new type of zone in the game. These declarations give the type of the zone, list or stack, and they give the type of object that is contained in the zone. Event declarations give the system information about the data type of the parameters and the type of the return value associated with an event name. The different types of listeners verify the return type of the event they are associated with and contain the block of executable code to run upon the triggering of an event.

Together, these make up the primary declarations in our language. But seeing is believing, so we shall provide an example description of the asynchronous game, Speed (see figure 3.1). We will then proceed to walk through the major components of the description.
3.1.1 Speed

Let us take a look at the game description for Speed (see figure 3.1):
The program is just a series of declarations. Note that the order of these declarations does not matter. The games developed by this system are event driven. As long as the a declaration has been introduced into the system by the time the system requests for that declaration, the game will run smoothly. The user always has the option of placing declarations in sequential order (for better code organization), but this ordering is not necessary. From this description we are able to create the following game playable version of speed.

Let's carefully examine the description. The first declaration imports a pre-existing card library. For the benefit of the reader, several lines from card.dgl will be provided and explained. Note that these lines are not included in the Speed description, but it is beneficial to see how object and player types are defined.

![Figure 3.2: Example of object declaration in DGL.](image)

The first line in figure 3.2 tells the system that there is a new object type called card. The next two lines tell the system that cards have a string attribute called suit and integer attribute called rank. Both of these attributes are enumerable. The suit attribute can only have 4 distinct values and the rank attribute can only have 13 distinct values. From now on, if an object of type card is created, it will automatically contain these two attributes, unless the attributes are removed from the object type during execution.

The next few lines give the following information:

- There will be a new player type called cardplayer.
- There will three types of zones all containing cards (piles, hands, and decks).
Piles and decks are stack zones and hands are list zones. Note that up until this point, none of the lines of code are present in the actual Speed implementation. So these lines do not add bulk to the Speed description. We now can proceed to "Speed-specific" game elements and rules.

The first line in figure 3.4 tells the system that the game has a deadlock timer. The following line establishes a new Flip event that can be spawned by the game. In this version of Speed, the global deadlock timer fires whenever all players are idle for some period of time, and the flip event moves a card from each player’s deck to that player’s respective pile. The manner in which this occurs is discussed later on.

Next we come to the overall progression of Speed. There are two main stages to Speed: initialization (see figure 3.5) and the frenzy of players emptying their hands (see figure 3.6).

Every description in DGL will either override or use the default Init phase functionality. In this description, the init block instructs the system to add a player to the game for each person connected to a game. The description also adds the same
Figure 3.5: The initialization phase for the DGL version of Speed.

Figure 3.6: The main phase for the DGL version of Speed.

number of piles to the game. All of the players in the game receive an empty deck which is immediately populated by all 52 cards from the standard deck. (NOTE: Populate is a function that looks at all of the enumerable attributes of an object type and creates an instance of it for each unique enumerable attribute combination.) After the decks have been populated, seven cards are dealt to each player’s hand. Then the speed phase begins.

The speed phase consists of adding the flip listener to the registered priority queue of listeners for flip events. The same is done for the play listener and play events. The lock listener is assigned to trigger on the deadlock timer and the play event is enabled. After this code is executed, a flip event is triggered.
One can see the basic structure of the game from the previous declarations. The two phases tell the user what events are valid, set up the listeners for these events, and wait for human interaction. But in order for the game to actually react to players, rules need to be provided (see figures 3.7, 3.8, and 3.9).

![Image]

Figure 3.7: The flip rules for the DGL version of Speed.

In figure 3.8, we see another example creating an event type called Play. When a cardplayer spawns this event, that player needs to pick a card pile and a card to play.

![Image]

Figure 3.8: The play event along with the deadlock and play rules for Speed.

Now we come to the interesting part, the rules of the game, also known as the
listeners. In the previous section of code there are two listener declarations: the LockListener and the FlipListener. In this example, the LockListener is set equal to the FlipListener. All this means is that the body of code from the FlipListener is copied to the LockListener. We make a distinction between them by the type of event they trigger on. Looking at the FlipListener, we can see that upon a flip event:

1. If there exists a player with no cards in the deck, then all players lose

2. Otherwise, move the top card in each player’s deck and put it on top of the respective player’s pile.

The next portion of code declares the listener for Play events and tells the system what to do when someone desires to play a card on a pile. Let $A$ represent the card being played and $B$ represent the top card of the desired pile. First there is a check to see if the rank on $A$ and the rank on $B$ differ by one in a cyclic fashion. This simply means that either $A$ immediate follows $B$ or vice versa. If the person played a valid card, then the deadlock timer is reset and the card is moved from the player’s hand to the player’s corresponding pile. Afterwards the player that triggered this event draws another card and continues play.

![Figure 3.9: The draw event and draw rules for Speed](image)

This last code snippet creates a new draw event type for the game (see figure 3.9). This event requires a cardplayer to be passed with it. The listener which enforces
rules for this type of event is declared afterwards. The draw listener gives one of the end conditions for the game. If the player that is passed with the event has an empty hand, that player wins the game. Otherwise a card is moved to that player’s hand from his/her corresponding deck.

This example code is *ALL* that is needed for creating a 1 to 6 player game of Speed. This shows the ease of describing card games. The six player cap on our games is due to our visualization system which will be discussed in the next chapter.

### 3.2 The Game Engine

In the last section, we showed how a sample game description communicates to the system to recreate the experience for players. This breakdown showed the capabilities of the language; now we focus on the inner workings of our system, or engine. The design of the game engine is an extremely important step in the creation of this system. We can make very few assumptions about the complexity of the future games produced with this system aside from the following: they are 2D and discrete. For example, objects could be defined to have certain properties at one instance in the game, but have different sets of attributes as the game progresses. The system should be able to make considerable changes to many entities quickly. With these considerations in mind, let’s create the engine.

On a high-level we wish for the game engine to merely execute code when it receives events and reflect the necessary game changes to users. When there are no events, the system should merely be idle. The first component needed for the game engine, is an event processor. The job of this component is to merely notify the system kernel of received events and have the ability to enqueue events from either the kernel or the networking layer (discussed in the next chapter) and dequeue events and pass them to the kernel.

The kernel is the controlling unit for the game engine. It keeps track of a current runtime (or phase) stack [8]. At any given time the top-most phase is governing the
execution of the system. Whenever a listener ends a phase with a value, that value is passed as the return value of that phase.

The kernel needs storage for the zones and references to their content. The kernel also needs to store the current state of attributes for players, objects, and the game. With the class of possible generated games, the only static content is the initial number of players connected to the game. Whenever rules are introduced or removed from the system, there is a possibility that some attributes will change. These changes need to be calculated and sent to the players of the simulated game. For fast update of game content, attribute information needs to be stored by their respective types in either hashtables or databases. For the sake of efficiency the simulating engine only needs to ship game changes to players who can view this information. Let us look at the following scenario: Let there be three players in a game (A, B, and C). The only visible content for a player is his/her respective hand. If the cards in player A’s hand change or are updated, these changes should only be sent to player A. Our framework should be able to handle this as well, so the attribute information also needs to be hashed by the player who can view the changes as well. Finally the kernel will need a storage for the actual rules of the game for event processing.

3.2.1 Event/Listener Interaction

Since the rules of a game make up it's identity, it is important to explain the process by which the rules are executed within our engine. There are events that are spawned and acted upon by certain listeners (Note that events are synonymous with triggers and listeners are synonymous with rules). Here is the basic overview of events and listeners. For each event type, there is a priority queue of registered listeners. Within the queue, the listeners are ordered by increasing priority followed by decreasing lifespan (see figure 3.10). Rule code can enqueue or dequeue listeners onto listener queues at runtime. Rule code can also enable and disable listeners. This functionality points toward the dynamic nature of the rules within this prototyping framework, but
Figure 3.10: Example of 5 listeners and the order in which they appear in the priority queue given the order that they are introduced. The sequential ordering is from oldest to most recent. Events are passed to the start of the queues located at the bottom.

the discussion of dynamic rule changing features will be deferred to the next section. In the end, the queue ordering makes up the core functionality of a game.

Now, there are two possible ways an event is spawned. Either the game or a player explicitly fires an event or when the game queries for one of the attributes of an object, player, or the game. Thus there are two distinct classes of events: active and passive. The only events that the registered event listeners act upon are active events. So what happens with passive events?

There is one additional feature of attributes: they do not have to be static data. The attribute can refer to code that constructs an attribute when requested, much like a function. This feature was added to aid in localizing rule code to the object(s) it affects. If the attribute refers to code, then that attribute is an attribute rule. The attribute rules wait for a request and generates a response. Upon a passive request for an attribute, an active event is spawned with either the raw simple attribute or the result from the respective attribute rule passed along with it.

Once an active event is fired, the event is shipped to the correct priority queue of event listeners. Then each rule in the queue behaves in order as followed:
∀r ∈ R, arg = r(arg)

Here let R be the queue of rules for an event and arg be a possibly void argument passed to the rule. The code for R is executed with arg in the code’s local scope. After the code is executed, a return value of the same type as arg is returned. This return value is set to the current value of arg and either returns arg as the result of the processed event or passes arg to the next listener in the queue.

3.2.2 Dynamic Rule Changing

For a subset of the class of card games that the prototyping framework needs to describe, dynamic rule changing is a necessity. The queue ordering is key to the functionality of the game. The ordering based on decreasing lifespan within the priority queue gives newly introduced rules precedence over their older counterparts. Depending on the order in which listeners are registered to event types, the resulting actions can be drastically different. As a side note, the registering and removal of listeners to events is as simple as the following lines of code in figure 3.11. (Assume that Play is an event type and PlayListener, are listeners that can register with a Play event).

Figure 3.11: Example of adding and removing listeners from an event type.

The first three statements add rules to the registered priority queue of listeners for Play events. The next statement removes the topmost occurrence of the PlayerListener0. The last statement clears all the registered listeners for Play events.
By definition, games with dynamic rules introduce new rules as objects that are either created or moved around in the game at runtime. The dynamic modification implies that the new rules should be attached and completely contained by objects. With this logic, the generic event listener is a valid attribute for an object, player, or game. (In fact, the default behavior of adding an event listener is adding it as an attribute to the game.) If a card has an event listener attached to it. The listener is not automatically registered to an appropriate queue of listeners. Rule code is responsible for registering. The dynamic rule modification feature allows for complex behavior to be simulated.

To give a better understanding of this dynamic rule changing interaction, we shall look at a simple game. This game consists of two types of cards: attacker cards and modifier cards. Attacker cards have two attributes: color and power. The color attribute can either be Red or Blue and the power is an integer from 0 to 10. Modifier cards alter the powers and colors of attacker cards or removes other modifier cards. When a player plays an attacker card, the opponent is dealt damage equal to the attacker's power. The attacker is then discarded. Whenever a modifier card is played, the card stays in play altering attributes of attacker cards.

In this example, modifiers are objects with listeners that work whenever requests are made for the power of attacker cards. In addition, players spawn attacking events with attacker cards. Once the attack is triggered, there is an attacker listener that looks up the power of the attacking card, which spawns a passive event that looks up the power of a card. Now the raw power of the card is sent to the queue of power listeners. The listeners potentially change the value and return this value to the attacker listener which assigns the appropriate damage.

We show a visual 5 card example and show how the order in which the modifiers are introduced alters the effects of attacking (see figure 3.12).

This functionality allows the event/listener system to cover a wide range of complexity. This complexity will be discussed further in the concluding chapter.
Figure 3.12: Example of dynamic rules in a game. The modifiers have labels that tell the order in which they appeared. The attacker cards have their raw power circle followed by their first modification (red), second modification (green), and third modification (blue).
Chapter 4

DGL Networking and Visualization

In the previous chapter, we discussed how our system deciphers game logic from a text description. These constructs are fine, but the experience is incomplete if the users cannot interact with the game. Somehow DGL needs to give players the experience of the game. We want players to be able to compete or cooperate with others and to have some idea of how the game looks without adding a great deal of complexity to the existing description. For games that use the standard 52 card deck, we want NO extra complexity. In this chapter, we will discuss how our prototyping engine provides this experience for players. We will describe the networking and visualization inferences that the system provides. We will also describe the visualization and networking inferences that the system provides.

4.1 Visualization

When people experience a game, the most important element is the aesthetics of the game. Aesthetics are how players perceive the game. Our prototyping system needs to provide a user interface that is reasonable and can accommodate the needs of the games described in DGL. We are not attempting to make a high-end graphical interface that is of professional quality. We are not trying to create games and sell them immediately with the user interface provided by our engine. The goals of the system are to provide a graphical interface that is functional and provides an enjoyable prototyping experience for those involved. The visualization algorithms that are used are good for testing the functionality of games. However, there are games that can be described such as Solitaire that could have a significantly better user interface. In
this section we describe the algorithms used to infer a visual experience of a game from the game’s description written in DGL.

4.1.1 Game Layout

The flow of the explanation of the visualization algorithms will be from the highest level, downward. We shall begin with the division of the window into different regions for the game. It was mentioned earlier that the games this system simulates have a cap of 6 players. Why is there a limit? We chose to have players view the game in one window. We also chose to ensure that all of the pertinent game information was visible to all players at all times. Due to these constraints, the amount of visual real estate for the game is limited. We divide the screen into three panels. The bottom 25% of the screen is devoted to game messages and player event buttons, the game messages spanning the left third and the player event buttons spanning the rest of the region. The next 25% of the screen is dedicated to local player information which will be discussed in the next subsection. The upper 50% of the screen is dedicated for the common game area and opponent/ally information. This space is evenly divided into two rows of 3 rectangular sub-panels. The middle section on the bottom row is dedicated to the game and all of its zones and attributes. This location was chosen because it is in the middle of the screen much like attributes associated with a physical game are in the middle of the players. The remaining 5 sections are allocated for at most 5 other players. These features are where the 6 player cap on the games is introduced. We felt that with one window, 6 players was the maximum number that would lead to a pleasurable visual experience (see figure 4.1).

4.1.2 Player Layout

Now we will take a look at how a player’s space is partitioned. At the top of a player region, there is a strip where the player’s name is displayed along with all of his/her visible non-timer attributes. The rest of the space is allocated for the player’s timer
attributes or the controlled zones. Whenever a player’s timer starts, if it is visible, a vertical timer progress bar appears at the left-most side of the player panel. This timer forces all other timers and zones to resize themselves and move to the right. As long as the presence of a timer does not force a zone panel to horizontally re-adjust less than its minimal size, the timer will be visible. When the timer is finished, the vertical rectangular panel disappears. When a player owns a zone, a black panel is added to that player’s layout. Each player’s panel is initially empty because the player does not start with any zone types. As the game progresses and players receive zone types, zone panels are added to their respective layouts.

Figure 4.1 : Screenshot from DGL version of Texas Hold’em. This shows the general layout of the user interface provided by the system. A subset of the visible zones are outlined in blue. A subset of visible objects are outlined in red. All of the player events are outlined in yellow.
4.1.3 Zone Panel Layout

As zone types are added to a player, new zone panels are created for the player. These panels are empty at first, however, our system still allocates space for that zone panel. Each zone panel takes up the entire vertical space for a player panel. The only varying dimension is the width. The layout of a zone panel is completely dynamic. When the contents of these zone panels change, the layout is recomputed.

The amount of horizontal space that the $i^{th}$ zone panel uses is:

$$h_{\text{space}_i} = h_{\text{min}_i} + \left( \frac{w_i}{\sum_k w_k} \cdot F \right)$$

Here $h_{\text{min}_i}$ is the minimum space for the $i^{th}$ zone panel, $w_i$ is the weight of the $i^{th}$ zone panel, $\sum_k w_k$ is the sum of each weight for all of a given player's zone panels, and $F$ is the amount of horizontal space that is left after the timers have been laid out.

The minimum space for a zone panel is governed by:

$$h_{\text{min}_i} = \max(|\text{zone}_i\text{type}|, 10)$$

Here $|\text{zone}_i\text{type}|$ is the number of characters in the name of the panel's type. This constraint ensures that minimally one sees the name of the zone type that a panel corresponds to (for type names less than or equal to 10).

The weight for a stack zone panel is calculated as follows:

$$w_i = |S| + 0.4 \cdot |N|$$

Here $S$ is the set of non-empty zones and $N$ is the set of empty zones.

The weight for a list zone panel is:

$$w_i = \sum |v_k| + |NVL| + 0.4 \cdot |NVE|$$

Here $NVL$ is the set of non-visible, non-empty list zones and $NVE$ is the set of non-visible, empty zones.
This weighting system gives precedence to list zone panels because list zones usually have more content to display than stack zones. It also gives precedence to zone panels that contain multiple zones.

4.1.4 Zone Layout

Now that we have laid out the zone panels and we have calculated their sizes, we can turn towards laying out the individual zones within the panels. If a zone is a list zone that is visible to the viewing player, then all of the objects within the zone will be visible. If the zone is a stack zone that is visible to the viewing player, then only the top card will be visible and a stack of cards will be rendered underneath. Otherwise, a face down stack of cards will be displayed with the size of the stack printed on top.

Now when the system renders a zone panel, the system calculates the maximum scale factor to minimize the number of rows needed to draw the zones’ contents. This calculation is performed as follows:

- For stack zones

\[
\frac{\alpha \cdot |S|}{n_{\text{rows}} \cdot \text{panel}_{\text{width}}} = \frac{n_{\text{rows}} \cdot \text{max}_{\text{height}}}{\text{panel}_{\text{height}}}
\]

Here \(S\) is the set of stack zones for the zone panel.

- For list zones

\[
\frac{\sum_k \alpha(L_k)}{n_{\text{rows}} \cdot \text{panel}_{\text{width}}} = \frac{n_{\text{rows}} \cdot \text{max}_{\text{height}}}{\text{panel}_{\text{height}}}
\]

Here \(\sum_k \alpha(L_k)\) is the sum of a function applied to each list zone for the zone panel. \(\alpha(\cdot)\) is of the form:

\[
\alpha(L_k) = \begin{cases} 
0 & |L_k| = 0 \\
\hat{w}d & |L_k| = 1 \\
\hat{w}d \cdot (1 + \frac{|L_k| - 1}{3}) & \text{otherwise}
\end{cases}
\]

Where \(\hat{w}d\) is the average width of objects in the zone, \(L_k\), and \(|L_k|\) is the number of objects in zone, \(L_k\).
Once we have the global scale factor for the zone panel, we calculate how many objects we can fit on each row (ensuring that objects from different zones do not overlap). Once the number of objects has been computed for each row, we then calculate a display scale factor for each row. This scale factor has an upper bound of 1, so we do not enlarge and possibly distort the images for objects. For list zones, we use the display scale factor to determine whether we calculate a horizontal offset to be used after each object is rendered in a zone. Note that negative offsets mean that cards overlap. We continue with this algorithm for the remaining rows. After this we have successfully laid out an individual zone panel.

4.1.5 Object Layout

We finally turn our attention to the lowest visualization component in our graphics engine, the object. There is not a great deal to consider when rendering individual objects. The scale factor for the associated image file is passed from the containing zone panel. The system simply renders the scaled image at the location specified by the zone panel. The system then renders object timers on the left side of the object. As objects move from one zone to another, there are animations to depict this movement. After the move is performed, the zone layouts are recomputed. There is also a quick, smooth animation to depict the zone panel relayout.

There are ways to add visualization information to the description which do not greatly affect the complexity of the description. Every object has an implicit attribute called *image*. This *image* is just a string that can be set to the filename of the associated texture image file. One can also have *image* be the name of a Photoshop document that can be parsed to have the number and string attributes laid out on the object in different ways, but this detail is beyond the scope of this thesis.
4.1.6 Visualization and Events

Before we end the discussion on the visualization for this engine, we need to discuss how events are visualized on the screen. While interpreting a game description, if the system encounters a player event declaration, the engine knows to ship that event type to the respective clients, which place a button in the event panel discussed in the Game Layout section. Now whenever this event is enabled for a player, that player may press the button to initiate creating an event or clicking on the correct entity to spawn an event through event inferencing, discussed later. After instantiating the desired event, the necessary arguments need to be supplied. For most of the arguments that correspond to objects, zones, and players, the user can simply click on one of the desired items. However, for other arguments such as numbers and number/string enumerations, new sub-windows appear to facilitate this selection.

As players began using this system, we noticed that the number of clicks used to play these games became cumbersome. To reduce this effect, we wanted to reduce the number of mouse clicks needed to initiate events. As a result, the concept of event inferencing was introduced. There may exists events that can be completely inferred based on their arguments. To demonstrate, let's look at the following example (see figure 4.2):

![Sample event declarations in DGL](image)

Figure 4.2: Sample event declarations in DGL.

From this example, we can completely infer the last two events. If the player selects a pile first, the system knows that the player is attempting to create a play event. So DGL knows to expect a card from the player's hand next. Likewise, if the
player selects a deck then a draw event should be created. One might argue that the Flip event can be inferred as well, but since the Flip event takes no arguments, the player only needs to click the Flip event button.

There is a simple algorithm for determining inferencing. Before describing the algorithm, we need to define a few concepts and define the inferrable relation between two event types. A *path* in an event type is the sequential ordering of arguments supplied to the event, where the beginning of the path is the leftmost argument and the end is the rightmost argument. A *prefix* is a sub-path that originates from an event type’s starting argument. Two event types are *inferrable* if and only if the types do not share a non-empty prefix in their paths. Events that do not have any arguments are cannot be inferred.

Once a player receives event types, the system will check to see if the incoming type is inferrable with the current set of event types. If so, the incoming type will be added to the special group of inferrable events. As new event types are added to a client, event types may be removed from the inferrable events set. Likewise as event types are removed from a client, an event type may be added to the inferrable set of event types. This special set is recognized by our system and allows users to simply supply arguments for these events without worrying about superfluous button clicks.

This ends the discussion of the visualization capabilities of the prototyping engine. DGL provides a functional user interface for games that can be inferred with little to no extra information, aside from the rules of the game.

4.2 Networking

4.2.1 To Send Or Not to Send

When you are playing a card game with companions, information about the other players is given primarily by the actions that they perform. If you are playing Go Fish with John and Jane, how do you know that John asks Jane off Aces without observing the action? So by the same logic, DGL knows which messages should
be sent to players via the events of the game. This idea forces events to be the communication engine for the game. For demonstration purposes, let’s go back to a snippet of the code for Speed.

We shall carefully examine these two events. The first is a flip event that only the game knows and the second is a play event spawned by a player. The flip event is an internal game event that players do not need to know. Once listeners receive this event, there may be changes to the game that need to be reflected for the players; however, the event itself is known only to the game. There is no need for network traffic for this event. On the other hand, whenever a player wishes to play a card on a pile his/her computer, a message needs to be sent to the game, which could reside on a different machine. So the game needs a network message with all of the necessary information. In short, game events do not add network traffic, but player events do.

So events possibly spawn network traffic from the players to the game, but there must be more information that needs to be transmitted in the opposite direction. Players need to know when objects are moved from one place to another. They also should be informed when attributes change on entities that they can see. To simplify these ideas, we will just list this information. Network messages need to be sent to the players whenever:

- A player/object/zone type is created, removed, or modified
- A player/object/zone is created, removed
- An attribute of a player or object is altered
- An object moves from one zone to another
• A player attempts to perform an illegal event

• The system wants to send text messages to players

4.2.2 How are Messages Sent

Now that we know when to send messages, we must mention how they are sent. During the execution of a game, the game’s notion of its state is consistent and clients, or players, reflect this state on their machines. So as the game creates new content, generating new types, objects, players, or zones, these items are serialized and shipped to the players. Serialization takes objects and converts their state information into a form that can be transported in a message [1]. When the game makes modifications to attributes of objects or players, the minimal amount of information is passed to the players to reflect these changes. If an object has 20 attributes and only one attribute changes, there is no need to serialize the entire object and ship the serialized object to a client.

In the opposite direction, entities are serialized as they are passed as events to the game. This serialization ensures that the client’s version of the game is the same as the server’s version. With the above feature, the system can perform a first level security check to ensure that there are no malicious clients attempting to perform illegal actions. There are also extra qualifiers such as fromowner’s(zonetype) that are sent along with objects which further constrains the data that a player sends. These event argument constraints aid in the enforcement of game rules.

Summarizing this section, all networking information can be parsed from the game descriptions. Most of the network traffic moves to the player (client) and is spawned as the physical state of the game is altered. There is also constrained network traffic that moves from the player to the game. But in the end, the communication needs for a game can be completely inferred from the given description.
Chapter 5

Conclusion

We have discussed our goals, a taxonomy for card games, the inner workings of the system, and the visualization and networking infrastructure for this prototyping engine. Now let's discuss what was learned from this experience and evaluate the success of this card prototyping system.

5.1 Comparisons to other approaches

When formulating the correct approach for this rapid prototyping system, a number of different solutions came to mind. What is the correct convenience/expressiveness ratio to use for such a system? Why create a stand-alone language that supports card prototyping when you can just make class libraries in an already established interpretive language such as Java or Python like the approach in EGGG?

When a person looks at game descriptions in EGGG, they can be impressed by how convenient it is to express simple games. But when new game mechanics are introduced, the descriptions quickly lose their convenience. We wanted to lessen the level of convenience in describing games, but have a language that covered a much richer range of complexity. For example, we noticed that the synchronous characteristics of the games EGGG modeled were just rules added to an asynchronous experience to incorporate time constraints to a game. So rather than go with the assumption that the majority of the games will be turn-based, we went with the more general asynchronous approach and provided libraries with extra rules to govern this behavior. We wanted the freedom to have convenient high-level control as well as convenient access to low-level mechanics.
So why do we create our own language? The answer to this question is more convenience for the programmer. We do not make the claim that the number of lines of code for a DGL implementation and an implementation from a similar framework using an interpretative language such as Python would an order of magnitude less. We actually had a student implement a version of the prototyping engine using Python as the description language. The number of lines he used to describe Hearts in Python was more than he used in DGL; however, the numbers were comparable. One of the benefits to using DGL is the self-contained, independent programming environment. Everything a programmer needs is packaged in one application, unlike the Java paradigm. However, the true benefit comes with the horizontal space that is saved with DGL. Here is an example of a segment of a game description in DGL (see figure 5.1).

Now let's try to port this description to a Java counterpart. We would need classes that corresponded to the player, object, zone, event, and listener types in the system. If there were classes for these types, the corresponding Java version would look like (see figure 5.1).

All of the singleton references correspond to the respective types in a particular game. These singletons would produce the actual players, objects, zones, events, etc. It is possible to create a direct mapping from the DGL implementation to its port in Java. This mapping shows their similarity, but when faced with a choice to program in either language for the purposes of prototyping card games, DGL is the better choice.

There are other features of DGL that reduce the size of the code such as the im-
implicit representation of all data as lists. The manner in which functions and operators act on single data types and lists of those data types is invisible to the programmer. This behavior was influenced greatly by Mathematica [13]. These factors allow for shorter code. The reduction in the code size is apparent with DGL, but the main importance is the convenience that our language provides over others.

5.2 Discoveries

5.2.1 Abstractions to Conciseness

Much like high-level programming language abstractions have made programmers' lives easier, we wanted to find abstractions within card games that would lead to more efficient, compact code. We wanted to compact visual and networking elements into abstract concepts which could be used transparently by the programmer. We succeeded in this endeavor. The first version of this system classified certain game elements: players, objects, zones, events, listeners, attributes. From these abstractions, we found ways to infer many visualization and networking infrastructure for games as well as a way to model dynamic rule changes in a concise manner. This system was purely event-driven. The state of the game was determined by the physical contents of the zones that the game and players controlled and the attributes on those objects.
There was no notion of overall state transition, or state modification. If an event triggered a state transition, it was beneficial at times to yield this processing until the game returned to the parent state. Let’s look at a game such as Texas Hold’em (see figure 2.1 (left)). One intuitive way to design this game would be to have a main phase which of the form:

In this loop, all players would start and finish betting actions in a *BetPhase*. In the early event driven version of DGL, this design would not be possible. One could not expect that processing a function call or firing an event could encapsulate a set of player actions. This lack of expressiveness led to the conception of the *phase* or state of the game. Phases led to better code organization in the game descriptions which translated to shorter code. Afterwards we found that in describing games such as Poker and Gin-Rummy, that we produced a great deal of code to provide pattern detection intelligence. This need led to the pattern abstraction. Eventually the conceptual leap to treat the physical data used by the system as lists of the respective types made descriptions even more compact. In retrospect, the lifespan of this prototyping system, the evolution of concise code was directly related to the evolution of new abstractions.
5.2.2 The Classes of Games that can be represented

So what games can our system describe? DGL comes with the standard language elements one would expect from any high-level language: functions, branching, recursion, iterative loop structure. There are also caveats for creating sub-games within games and creating code to determine the progression of the parent game based off of results from the child game. One can also change the layout of the game, remove all physical components, and create a totally new game during run-time. These features lead to a wide range of complexity. One might go as far as to say that the only card game this system cannot represent are those with persistent data (i.e. games where data would need to be stored from game to game). One can cover the standard card games that have historical merit. One can also model the newer card games that hold people's attention today. This system is an expressive fir describing and playing card games. There are no claims that this system is the solution to 2D card game prototyping, but we feel it is a small step in the right direction.

5.2.3 Future Work

DGL has been a first step in developing tools for rapid game prototyping. Card games are a small subset of the game industry. We need to continue to develop tools that allow users to prototype larger classes of games. The next natural step is to add an arbitrary graph as a zone. This abstraction will allow people to model the class of 2D board games, which completely contains the set of card games.

DLP

We are presently working on a system for archiving and prototyping synchronous and asynchronous board games. This system is the Digital Library Project, DLP. The general concept is very similar to that driving DGL. Our goals our to create abstractions to allow programmers to receive fast feedback on game ideas for board games.
We are not only developing another system. We are also creating a Board Game Editor. This editor serves as a graphical tool for developing games. With this editor, we are trying to change the paradigm for game design.

**Extensions into full simulation world**

As time progresses, we want to tackle issues for incorporating artificial intelligence, AI [7], code our set of prototyping tools. This involves both AI for computer players as well as AI for the individual pieces in the game. This would allow pieces to act without human intervention bringing us closer to prototyping simulations. Our future goals include developing tools that allow programmers to prototype 2D (and possibly 3D) discrete simulations. These goals are ambition, however, the results from DGL are encouraging.
Bibliography


