SAGA: A Story Scripting Tool for Video Game Development

Lucas Beyak
Department of Computing and Software
McMaster University

April 21, 2011

Abstract

Video game development is currently a very labour-intensive endeavour. Furthermore it involves multi-disciplinary teams of artistic content creators and programmers, whose typical working patterns are not easily meshed. Usually a domain expert has to communicate their ideas to a programmer in order to realize their designs into a final software product. This process of transferring information may introduce error or ambiguity, while also hampering productivity.

Domain-specific languages (DSLs) attempt to increase development productivity by allowing domain experts to express ideas in a customary manner, while also providing a mechanism to directly translate these ideas into usable code.

The SAGA tool, which includes a DSL and compiler program, uses these methods in an attempt to increase developer productivity. Various domains from video games were considered but the area of story design was chosen.

The story design DSL developed employs a text-based, natural language style in order to be less daunting to a non-programmer. A story is modeled as a transition system through some notions of story states and transitions that can progress the story according to important events.

From the DSL source code the compiler program produces a usable story manager module that interacts with game code to supervise the progression of story. This story manager module can be rendered in C#, C++, or Java. The compiler also creates a visual output of the story graph.

A video game developer can use the SAGA tool to easily model their story and employ a story management system in their products.

Contents

Chapter 1

Introduction

Recently the video game industry has become a mainstream phenomenon. Video games are now as ubiquitous as movies, books, and other forms of popular culture. The industry's growth might even be out pacing that of the more classical art forms [5, 28]. Much of the growing popularity of video games is due to advances in technology, such as alternate input devices and improved visuals. Revolutionary devices such as the Nintendo Wii and Microsoft Kinect are enticing new users to the industry who felt like they could not get involved in video games using traditional control schemes.

Since video games are becoming more popular, developers continue to produce more complex systems to appease the masses. Eventually the methods utilized to create these interactive media must evolve in order to keep development productive [7]. Although video game development has come a long way since it's infancy, the ones involved must still make advances in this sector.

In order to understand the problem that pertains to this project, one must have a cursory understanding of how a medium to large-sized game development studio functions. Since there is such a broad spectrum of expertise required to produce a complex software like a video game, there will most likely be employees that have a deep understanding of a certain aspect of development but know little about other areas.

At a high level the common departments that would be present in a devel-

opment studio would be: art, music, programming, game design, and management [11]. These broad departments might be partitioned into smaller sections depending on the size of the studio. There might be dedicated staff for world and level design, music and sound effects, user interface design, or story design to name a few. These employees are domain experts in their respective fields, but they may know very little of how the other departments operate.

Since all developed material must eventually be realized in a final software product, the people like the level and story designers must express their designs and ideas to the programmers. This can take much effort and time, and it can also be easy for the programmers to misunderstand the true intent of the designers. This situation can hamper productivity.

One attempt to increase productivity is to eliminate the requirement to transfer information from the designer to the programmer. It would be much easier if the domain experts could simply express their ideas in their usual way and have their designs be used directly in the game software.

A way of accomplishing this is by utilizing a domain-specific language. The domain expert can use a language that they understand and continue working at their full potential. There will be some automated mechanism that translates the expert's ideas into a usable form [6]. Since the language is designed for use in a specific problem domain it can accurately express the information a designer wishes to convey.

This report will explain the technical background of domain-specific languages as well as the craft of storytelling and how stories can be represented. The syntax and semantics of the constructed domain-specific language will be described along with the program to process this language. Finally, the method of using the output of this tool in a system will be revealed.

Chapter 2

Requirements

The goal of this project is to create a tool that will aid in video game development for a particular domain. Such a tool will attempt to increase the productivity of an expert in the chosen domain. This tool must satisfy a number of requirements and properties.

The tool should provide some interface for a domain expert (non programmer) to describe their ideas in a standard way. It should be fairly natural for the expert to express themselves through this interface.

The information gathered from the expert must be somehow transformed into a usable form for an existing game framework. This output will be in the form of compilable code, which can be processed by a standard compiler and linked into the game executable.

The computational overhead for interactions between the game engine and generated code should be minimized. Since video games are real-time systems that require many calculations per second, the available resources cannot be squandered.

It would be attractive if the tool could generate code in multiple output languages. C++ is by far the most common for serious game development, but other fairly common languages are C, C#, and Java. This should not be too problematic since the syntax for these languages are quite similar.

Additionally, the system should be well-designed according to software quality metrics such as understandability, conciseness, and maintainability.

Chapter 3

Domain-Specific Languages

3.1 What Is a Domain-Specific Language?

A domain-specific language (or DSL) is a small language that is designed to express a specific portion of a software application [13]. This specific aspect, or domain, can be very efficiently and accurately defined through the custom language. A DSL is not usually Turing complete, and therefore is unlikely to be used in the same way as a general-purpose language to define an entire system.

In order to design a DSL for a domain, that domain must be well understood. The reason for this is that there must be an automated mechanism that unambiguously translates a DSL program into some desired form [6]. In addition, if domain experts are going to be using this language to describe their ideas, there should be more or less a standard representation of problems in that domain.

Programming in a DSL should be much easier than writing code for the same purpose in a general-purpose language. This means that productivity should be increased by utilizing a DSL. Unfortunately this is not always the case. Even if productivity is not increased, however, other benefits may be achieved. Some of these other advantages might be attributes of software quality like reliability, understandability, maintainability, testability, and efficiency [14]. Since the DSL can accurately express the domain it is designed

for, this means that domain experts, who may or may not be programmers, will be able to use it as well to communicate their ideas to the rest of the programming team.

DSLs can differ in the programming style that they employ. Some might be "solution-oriented", meaning that the actual form of the solution is described in algorithmic details. Others might be "problem-oriented", where the code describes what the program should do but not how it is accomplished. This is analogous to the differences between general purpose imperative and declarative programming languages. Most DSLs utilize a declarative programming paradigm, and so are oriented to describe the problem to be solved. DSLs are presumed to be easier to use, and usually the declarative style lends itself to this purpose.

The goal of such a language would be twofold. Firstly, such a language should make it quite easy to express common tasks performed in a conventional language (in the DSL's domain of course). Secondly, tasks that would be very difficult to express in a general-purpose programming language should be able to be performed with the DSL with an acceptable difficulty.

3.2 Internal vs. External DSLs

A distinction can be made between so called internal (or embedded) and external domain-specific languages [13]. There are reasons for and against the usage of each kind.

An internal DSL will be embedded in some general purpose host language. This means code is written in the host language, but some additional constructions are made so the DSL code looks like a "new" customized language. This type of language can also be referred to as a fluent interface [13].

It is easier to quickly create a DSL of this type because the host language's compiler is used to process the code. Also, all the advanced features of the host language are available to the DSL developer. One reason why developing an internal DSL might be difficult is sometimes the embedded

language's expressiveness is reduced, as the code must follow the rules of the host parser in order to be well-formed. This loss of DSL expressiveness varies greatly from host language to host language.

General purpose languages can be quite divergent, and so creating an internal DSL in one might be much easier than creating a DSL in another. Another problem in writing internal DSL code is the domain expert is programming in a language that they may not fully comprehend. This may become confusing if they are not familiar with the host language.

An external DSL is not embedded in some host language and does not use that host's compiler. This type of DSL can have a completely custom syntax and be made very expressive for the domain being considered. It should be more natural to communicate ideas in this way. It can be more difficult to implement such a DSL as a compiler must be produced to process the custom DSL code. This is not a trivial feat, as the compiler has to parse the DSL code, perhaps make some intermediate representation of the code, and then perform code generation to produce the end result.

3.3 Domain Selection

It can be seen from the title of the paper that the selected domain was story design. It was not, however, an obvious selection. There is a plethora of different areas in game design in which a DSL could be utilized. The three domains considered for selection during this project were: character status representation, real-time strategy game rules, and story design.

3.3.1 Character Status Representation

In some games the player is mainly concerned about building up the character they are controlling throughout their play. This avatar is very important as it represents the player in the immersive gameworld in which they are participating. The player will progress through the game, all the while gaining power and abilities for their character. This is mostly common in role-playing games, or RPGs. There are numerous attributes that keep track

of the ever-changing status of a character, and so the rules governing these attributes can become complicated. A DSL for this domain would be able to simply define attributes and how these attributes affect the abilities of such a character.

A classic example of this is Dragon Quest, which is a 1986 RPG game released in Japan and later in North America [20]. This was one of the first games to introduce concepts such as attributes for a character and different skills to utilize, both inside and outside of combat.

More recent examples of games that would greatly benefit from a DSL for the character representation domain would be Final Fantasy XIII or the insanely popular World of Warcraft. Both these games have a heavy emphasis on building and leveling up a character throughout the game.

3.3.2 Real-Time Strategy Game Rules

Real-time strategy (RTS) games are known for having multiple factions with which you can do battle. Each faction is unique with regards to either the buildings and units that can be created, or the mechanics or rules that the faction follows.

A perfect example of this is Starcraft or Starcraft 2 developed by Blizzard Entertainment. There are three factions (completely distinct races in this case) that abide by unique rules [26]. The Terran race, humans, can make most buildings take flight and must construct buildings on regular land. The Zerg, a primitive alien race, can only create structures on "creep" (with some exceptions), which is a living foundation that spreads across the ground near their buildings. The Zerg also make all units from a single building, where the other races have specific buildings that create specific units. The advanced alien race called the Protoss must create buildings within a certain radius of a "Pylon" structure, and can create buildings very efficiently because no "worker" unit is required to be present to actually build the structure. Instead, the structure "warps" in unattended and the player need not concern themselves with it.

One of the first modern RTS games was Dune II: The Battle for Arrakis.

It was released in 1992, and all following RTS games utilized the same concepts introduced with this game [27]. Such concepts included harvesting resources, building units and buildings, and selecting, moving, and attacking with units.

A recent game within the RTS genre is R.U.S.E. [9]. There are six factions in the game, each having their own strengths and weaknesses. There are many aspects of this game that could be described by a DSL for this RTS game rules domain.

3.3.3 Story Design

The story of a game is important because it immerses the player [2]. Immersion is a very powerful tool to engage the player in the game world, and make the game more important or exciting by making it seem real. Most games, however, have a linear story, which does not offer the player much agency during their gameplay.

Agency in a video game is the idea that the player has some control over their surroundings and that their actions have an effect on the game world [19]. A player who feels like they are actually interacting with a game's environment and are able to change it will become immersed more easily. Agency and immersion in a game will lead to a more involved player experience, and usually leads to more enjoyment.

Like in the movies, if a film portrays strong character development or an engaging storyline, this can drive a very powerful experience for the viewer. Since video games are a highly interactive media, this experience can be compounded to be even more immersive.

The reason why games commonly adopt a simple story is because for every choice the player can make, there is a number of different directions that the story can then take. Therefore, many choices for the player leads to a combinatorial explosion of story "states". This adds much complexity to the design of the game, and requires much time, effort, and money spent on the development of assets that will not always be appreciated.

Myst is a classic example of a game that makes use of a story that can

be changed by the player, although only at the end of the game [25]. The player can make one of four choices, and based on what the player decides, different ending cinematics and dialogue are introduced. Myst also has an "open world" style of gameplay, where the player is free to roam throughout the game world as they wish, which again creates immersion as the player is never presented with a "you can't do that now" moment. A moment like this can quickly jar the player out of deep immersion and cause great annoyance [2].

There are a few excellent examples of very recent games that make great use of interactive storytelling. Most notable are the titles Heavy Rain and the Mass Effect series (Mass Effect 1 and Mass Effect 2). These games not only let the player make decisions in the game, but these decisions have important consequences on the game's story. In Heavy Rain, there are four main characters, but depending on the path that the player takes, some main characters can even be killed and will not return to the game for the remainder of play [23]. An interesting aspect of the Mass Effect series is that the save data from Mass Effect 1 can be used to start a game of Mass Effect 2 [22]. This keeps the story consistent based on the decisions that were made by the player during the play of Mass Effect 1.

3.4 But Why Choose Story?

Not all players are created equal. There are many aspects to a video game and some players enjoy a particular aspect or aspects to a greater degree. Some players love fancy graphics while others enjoy the music and sound effects of a game. Players expect a more engaging experience, and desire a game experience that can be tailored to their play style.

An engaging story can fulfill this requirement, and has much room to grow over it's current condition in the video game industry. The idea of interactive storytelling will only continue to increase as the industry matures over time.

A video game story can be unambiguously defined using mathematics, specifically the area of graph theory. A story can be represented mathe-

matically by a directed acyclic graph (DAG) [16]. Since the theory used to model this problem is well understood and can be easily represented, it is a good choice to explore when considering the possibilities of a DSL in video game development.

Since the domain of this project is the story of a video game, elements correlated with this aspect will be discussed in more detail.

Chapter 4

The Video Game Story

4.1 Story Structure

The story of a video game can be structured in the same ways that a story is constructed in movies or books. The only difference is that instead of passively watching characters make decisions like in a film or book, the player of the video game is the one making decisions for the characters. The overall structure of a story in a game can be either linear or non-linear, as described in this chapter.

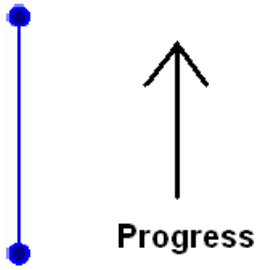
4.1.1 Linear

The most simple structure for a game's story is the linear structure [3]. A linear structure does not offer the player much choice in the way of interactive story. The player will begin the game and be directed to make "choices", which are not actually choices at all because the player cannot decide to take alternative paths through the game.

Games containing linear stories are predictable, which may lead to a low replay value since the player already knows exactly what is going to happen the second time through. On the plus side, linear stories are easier to design than their non-linear counterparts because of their low complexity. The developer knows that all the content that they create for their game will be seen every time the game is played, and so no effort is wasted on content

that might never see the light of day. Figure 4.1 below was originally taken from [3].

Figure 4.1: A linear story graph.



4.1.2 Non-Linear

The other style of story structure is one that is non-linear. At certain points in the game the player may be presented with a real choice of what to do next, and based on this choice the story of the game may change. This is very interesting as the player now feels like they can affect their game environment. This is known as agency and was introduced earlier in this writing. Unlike linear stories, non-linear story structure can be represented in multiple ways.

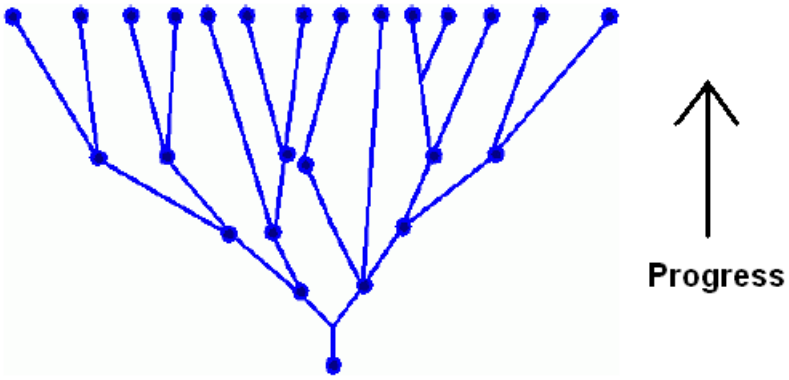
Branching

A branching story structure [3] is the standard interpretation of a non-linear story. In a branching story a player makes many choices throughout the game, where every choice will lead the player in a completely new direction. This is very good, and the player will be very excited about how the story may progress. However, this is unfortunately also a large problem.

For every choice the player makes there is a new unique path through the story. The reason why these many paths produce a problem is that for each new story "state" content must be produced by the game developers. In addition, since the player makes these choices and goes through a single path through the game, much of the content created for the game goes unseen.

This branching structure creates so many states that it is simply impractical to make a game with a story structured in this way. A branching story graph is shown below in Figure 4.2, which was taken from [3] and modified slightly.

Figure 4.2: A branching story graph.



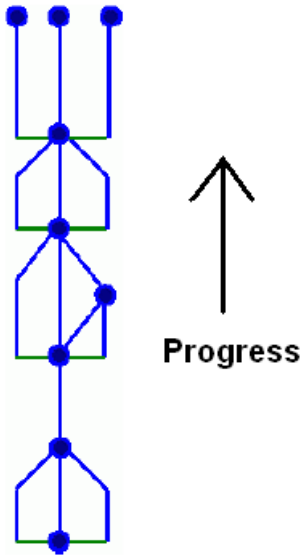
Parallel Paths (Foldback)

The parallel paths [4] structure is a compromise of a linear and branching structure. The player is still given choices that affect the story of the game, but at a certain point the story paths converge, fold back on themselves, to a central point. This central point will always be reached by the player regardless of what path they choose to take. After this central point, the player will be given choices again that branch out for a short time before once again folding back.

This structure gives a good illusion of choice to the player. The player does have choices that matter but not all the time. Only after playing the game another time it is possible that a player will realize that there are some inevitable events that cannot be altered.

This structure offers freedom to the player while not containing a prohibitive amount of story content to create. Because of this reason it is the standard selection of story structure in modern games [2]. The foldback graph that was taken from [3] is shown below.

Figure 4.3: A foldback story graph.



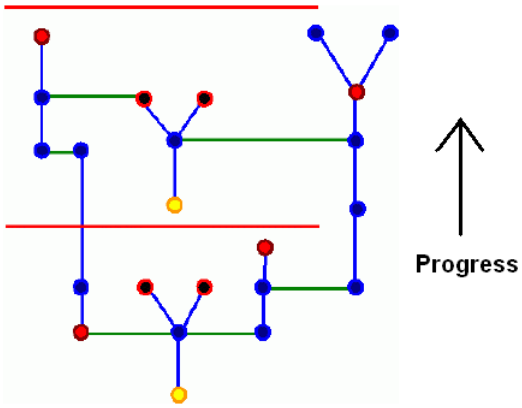
Threaded

The threaded structure [4] is quite different from the ones shown so far. In this structure there are multiple independent paths that develop on their own, regardless of what else is happening in the story of the game. When the game is nearing the end of the story, these paths usually converge to form the final events. These are called "threads", hence the name of the structure, and it is a similar technique to what authors or filmmakers use to craft an intriguing story.

Unlike in the previous story structures, the plot does not progress along a single path. Figure 4.4 below shows a possible story structure for a game. This story graph was taken from [4], and is supposed to be an approximation of the first two acts of the game Discworld Noir. The yellow dots represent events that start an act. An act is just some structure to break up the main events of the story. The red horizontal lines separate two acts shown in the figure. The blue dots are optional story events, while the red dots are mandatory. Red dots with black inside are story events that will make the next act available. At least one of these special red dots must occur in

each act. The green lines show the areas where threads interact with one another.

Figure 4.4: A threaded story graph.



This system allows the player free-movement throughout their experience, which is a rarity in today's games. Keeping track of the changes in the story graph can be difficult, and makes implementation of this method challenging. As well as the technical implementation challenges, the effort required in the quality assurance (QA) process greatly increases due to the fact that players have this increased freedom to explore.

Much of the time the player will simply be oblivious to the fact that this type of freedom exists. However, as it is often the case with design of system elements (such as a fluid and natural user interface), they should be transparent and not intrude on the user experience. In this case of non-linear storyline, the user simply notices that many of his/her desired actions are not limited, making them fully immersed and content.

Dynamic Hierarchical

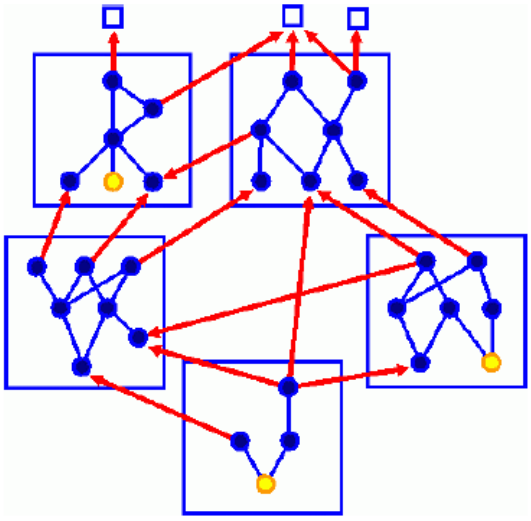
The given name for this structure in [4] is dynamic object-oriented. The term object-oriented is not being used in the correct way here. This structure actually is hierarchical, and uses abstractions to represent story. This method can be even more expressive than the Threaded structure, but still

be as manageable as the Parallel Paths (foldback) structure.

The idea is that a very complicated structure can be simplified greatly when wrapping smaller units in other larger representations. In this way you can ultimately think of your story as a simple graph of transitions between nodes. The complexity of each of these nodes, which could contain a whole complex graph of its own is hidden by abstraction. This nesting of complex nodes repeats indefinitely until the highest level structure can be easily designed and managed by a human.

Having many paths through the story means that there must be a huge amount of content that must be created for the game like game levels and characters. To avoid this asset creation problem, the same locations and characters can be reused many times but with different settings, like lighting, colours, and dialogue. This style of structure is shown in an example below in Figure 4.5. The graph was taken from [4].

Figure 4.5: A hierarchical story graph.



4.2 How is Story Affected

Story progression in a video game is usually represented by some type of pre-recorded video during the game, whether they may be in-game cut-scenes,

or completely rendered cinematics. A story will progress forward when some important event or events take place. The most important idea when speaking about a game's story is the fact that once the story has progressed in some way, this is a permanent change and the player cannot "undo" this progression. This is congruent with the earlier statement that a story is represented by a DAG.

There may be many events that the player can trigger during a game, such as acquiring some new item or skill. It is not necessarily the case that all events will actually affect the state of the game's story. It might not matter what kind of sword the player has attained throughout their adventures. It is important not to confuse the idea of an "open-world" style game with a non-linear storyline. An open-world game means that the player has the freedom to freely explore a large part of the game world [17]. Some games may give the illusion of a complex, non-linear story by allowing the player to experience a wide variety of areas, which will customize the player's experience.

When present in a game, both of these concepts will increase the agency of the player. This does not mean that these ideas are the same. An open-world game may have an extremely linear story, while a more closed-world game can have a non-linear storyline. This being said, it is probably more common for an open-world game to have a non-linear story than a linear one. In an open-world game the player will have access to a large number of areas in the world. If the player can only further the storyline by doing a specific event in a specific area of the world, this may become irritating for some players. It may be difficult to find this area to continue on with the game's story, or the player may realize that there is not much point exploring around the world because they cannot access the story content.

Chapter 5

The Story as an Acyclic Graph Assembly (SAGA) DSL

5.1 Syntax

Like most DSLs, the one designed for this project is text-based [13]. The language is quite small, meaning there are very few keywords present in the grammar. A story can be programmed through these keywords, and the story description is meant to be similar to natural language.

5.1.1 Grammar Definition using EBNF

EBNF (Extended BackusNaur Form) is a notation to formally define the syntax of a language. It is based on BNF, which was originally used to describe the Algol 60 programming language [24]. The EBNF grammar for the SAGA DSL is shown below in Listing 5.1. There exists many variants of the EBNF that utilize different syntactical conventions, but the one used here is outlined by an ISO standard [18].

Listing 5.1: EBNF Grammar of the SAGA DSL

```
story = opt_whitespace , story_name , whitespace, start_node ,
        whitespace , sections , whitespace , 'WHERE' ,
        trans_list , opt_whitespace ;

story_name = 'STORY' , whitespace , label ;

start_node = 'INITIAL' , whitespace , label ;

sections = section , { whitespace , section } ;
section = section_name , whitespace , '{' , whitespace ,
          trans_list , whitespace , '}' ;
section_name = 'SECTION' , whitespace , label ;

trans_list = trans , { opt_whitespace , "," , opt_whitespace ,
             trans } ;
trans = pre_nodes , whitespace , 'GOES' , whitespace , label ,
        whitespace , 'WHEN' , whitespace , events ;
sect_trans_list = [ trans_list ] ;

pre_nodes = label , { 'OR' , label } ;
events = label , { 'AND' , label } ;

label = word , { whitespace , word } ;
word = char , { char } ;

whitespace = whitespace_char , { whitespace_char } ;
whitespace_char = ? any white space character ? ;
opt_whitespace = [ whitespace ] ;

char = ? any visible ASCII printable character ? - invalid_char ;
invalid_char = ',' ;
```

5.2 Semantics

For this language there is some notions of Nodes, Sections, Transitions, and Events. As explained earlier in this paper, the structure and progression of a story can be represented as a directed acyclic graph. The nodes are states of

the story, and the transitions are directed edges connecting the nodes. These transitions describe how the story will progress. The transitions require that some event or events have occurred in order to change the state of the story.

The domain expert designing the story will most likely not think about the story in this way. They will simply write the story just as how a writer writes a book. They will think about the characters, what happens to them, and what the characters will do in order to overcome their obstacles. The designer will need to be able to express their story into the discrete form that the language requires.

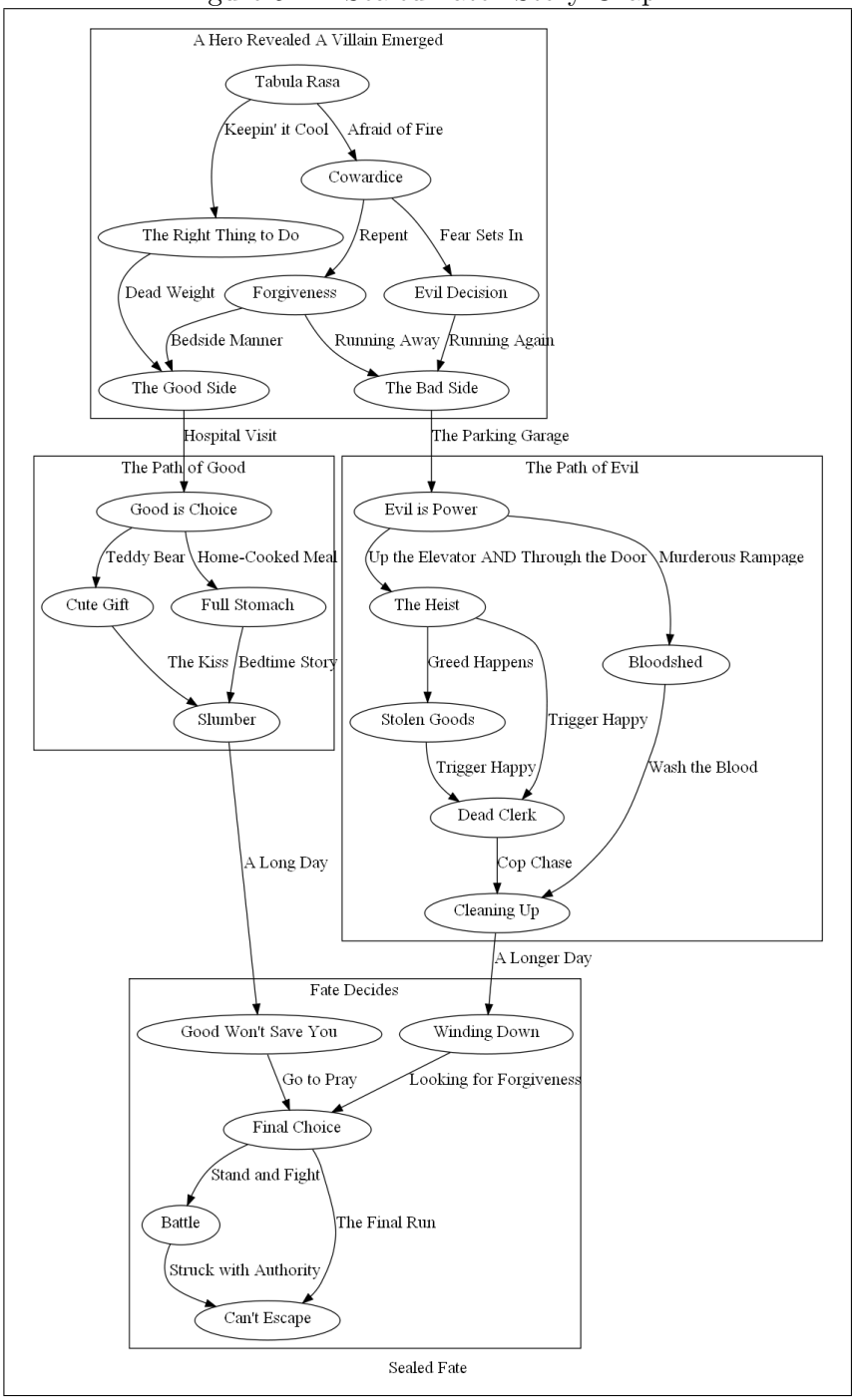
Before explaining the meaning of the syntax with respect to the story elements, an example will illustrate how a story description is transformed to a story graph.

Figure 5.1: "Sealed Fate" Story Description

```
STORY Sealed Fate

    INITIAL Tabula Rasa

    SECTION A Hero Revealed A Villain Emerged {
        Tabula Rasa GOES The Right Thing to Do WHEN Keepin' it Cool,
        Tabula Rasa GOES Cowardice WHEN Afraid of Fire,
        The Right Thing to Do GOES The Good Side WHEN Dead Weight,
        Cowardice GOES Forgiveness WHEN Repent,
        Cowardice GOES Evil Decision WHEN Fear Sets In,
        Forgiveness GOES The Good Side WHEN Bedside Manner,
        Forgiveness GOES The Bad Side WHEN Running Away,
        Evil Decision GOES The Bad Side WHEN Running Again
    }

    SECTION The Path of Good {
        Good is Choice GOES Cute Gift WHEN Teddy Bear,
        Good is Choice GOES Full Stomach WHEN Home-Cooked Meal,
        Cute Gift GOES Slumber WHEN The Kiss,
        Full Stomach GOES Slumber WHEN Bedtime Story
    }

    SECTION The Path of Evil {
        Evil is Power GOES The Heist WHEN Up the Elevator AND Through the Door,
        Evil is Power GOES Bloodshed WHEN Murderous Rampage,
        The Heist GOES Stolen Goods WHEN Greed Happens,
        The Heist OR Stolen Goods GOES Dead Clerk WHEN Trigger Happy,
        Dead Clerk GOES Cleaning Up WHEN Cop Chase,
        Bloodshed GOES Cleaning Up WHEN Wash the Blood
    }

    SECTION Fate Decides {
        Good Won't Save You GOES Final Choice WHEN Go to Pray,
        Winding Down GOES Final Choice WHEN Looking for Forgiveness,
        Final Choice GOES Battle WHEN Stand and Fight,
        Final Choice GOES Can't Escape WHEN The Final Run,
        Battle GOES Can't Escape WHEN Struck with Authority
    }

    WHERE
        The Good Side GOES Good is Choice WHEN Hospital Visit,
        The Bad Side GOES Evil is Power WHEN The Parking Garage,
        Slumber GOES Good Won't Save You WHEN A Long Day,
        Cleaning Up GOES Winding Down WHEN A Longer Day
```

The story description in Figure 5.1 produces a story graph as seen below. Make note of how the sections are contained within labeled boxes, as well as the effects of using the "AND" and "OR" keywords. They are both being used in the "The Path of Evil" section.

Figure 5.2: “Sealed Fate” Story Graph



As seen in the example description and graph above, a story must be given a name using the STORY keyword. The story is labeled at the bottom of the story graph, and is present so a reader can identify the story they are inspecting. An initial story state is specified using the INITIAL keyword. If the story graph is examined it can be seen that the only node that does not contain an incoming transition is the initial node. The story will always begin here and then branch out. The sections are defined using the SECTION keyword. The name of the section is specified, and the description of the contained nodes and transitions are defined inside curly braces.

The well-formedness of a transition is defined in the EBNF grammar, and the meaning of these definitions will be described now. There will be a label of a node preceding the GOES keyword, and following GOES there will be another label of a node. In graph theory terms the first node will be the tail while the second node will be the head of the arc that this syntax describes. There is also an OR keyword that can be used to shorten story description code in the following way. Before the GOES keyword there can be more than one label for a story node separated using OR. This is simply a short form that allows the description of multiple transitions with a single definition. For every tail node appearing before GOES there will be a separate transition created. The head node of these transition arcs will always be the same.

There is still the end part of a transition definition to be explained. After the head node is labeled this is followed by the WHEN keyword. Following this a list of event labels must be provided, separated by the AND keyword. These labels identify important events that must have occurred before the transition is allowed to be taken. Since these events are separated by AND, this shows that all of the events must be satisfied before using the transition.

When examining the sample story description another part must be specified. The sections are described already, but the transitions between these sections must also be specified. The keyword WHERE is used and this is followed by a list of transition definitions. Even if there is only one section defined in the story description, the WHERE keyword must still be provided.

5.3 Possible Extensions

For most people it is easier to comprehend a story through some visuals. Although the current language is simple and some structure can easily be seen from the text, there is still much room for improvement. A front-end visual interface for the language could be a step in the right direction. There would be some canvas area where the user could place story elements using some tools provided through the interface. It would most likely be organized similarly to a digital photo editing program.

Some of the tools that would be available would be: Place Node, Make Transition, Group Nodes into Section, Set as Initial Node, and Assign Events. Of course there would also be standard features like moving elements and panning and zooming the view of the canvas. In addition this front-end to the language could also include functionality to describe the desired configuration settings to give to the language's compiler.

There might be issues of scaling with this visual description of a story. If the story is complex and contains a large number of nodes then this might make it difficult to navigate while designing or viewing. One aspect of the language's design may be able to assist in this matter to some extent. There are concepts of Sections and Nodes for a story. A Section is a collection of Nodes. In order to view the story graph more easily there could be a tool that would collapse the Sections on the canvas into a small visual area. The Section can then be thought of as a simple Node even though it is known to contain some structure within. If the language could support the nesting of Sections, which it currently does not, then this can be repeated to collapse a complex portion of the graph into a smaller area. This concept was explained earlier when speaking about dynamic hierarchical story structure.

Most likely the text-based language would be preserved and the front-end interface would be able to transform the picture representation into a story description file like is used currently.

Chapter 6

The SAGA Program

6.1 The Big Picture

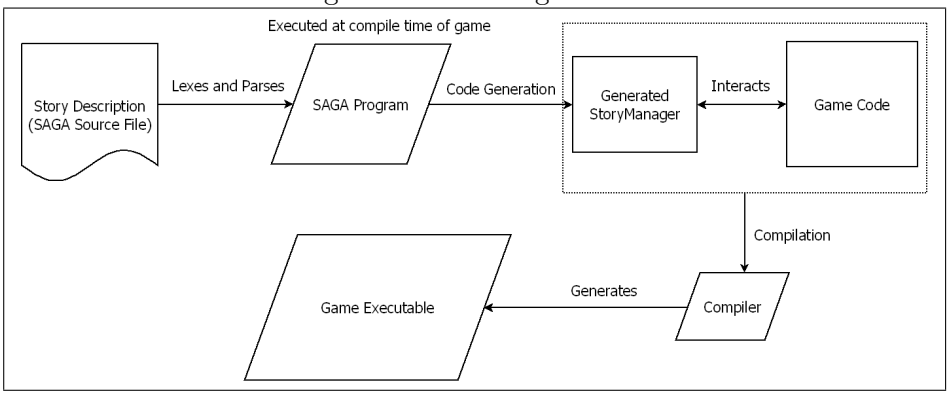
The SAGA program is a compiler for the SAGA DSL. It parses a story description file as well as a configuration file, and creates some internal data structures based on these. Once the program processes the input and creates some representation, the next step is to prepare the solution code, but in an abstract way. Doing this makes the generation of code for multiple languages an easier process. Once this abstract code is created it can be passed to a certain rendering mechanism of a particular language, which will generate the final usable code.

The program is implemented in Haskell, a purely functional language, for a number of reasons. Haskell allows for higher-order functions to be created in a more concise manner than a language like C++. In C++ it is somewhat clumsy to use function pointers to specify a function as an input parameter to a higher-order function. Haskell has a lightweight syntax that promotes readability [8]. Constructs like folding and functional composition allow a programmer to produce powerful code in very little actual lines of code. Once a developer overcomes the initial functional paradigm learning curve, clean, concise code can be written very quickly to accomplish any task.

Figure 6.1 below shows a general picture of how the story description

DSL and compiler will interact with the game compilation and execution.

Figure 6.1: The Big Picture



6.2 Parsers

As mentioned earlier, the SAGA program is a compiler. This means the first action that the program has to perform is lexical analysis and parsing of a source [10].

For these scanning and parsing duties there exists the parser combinator library Parsec. Parsec is simple to use, well documented, and provides good error reporting messages that specify position, unexpected input and expected productions. Many of the advanced features of Parsec were not required to build a parser for the DSL, but it still allowed for an easy creation. Once the BNF grammar of a language is known it is relatively simple to implement the required parser in Parsec.

There are various design decisions to be made when making a parser in Parsec. Things must be considered like case sensitivity, whitespace requirements, valid identifiers, comment style, and reserved operations and keywords.

6.2.1 Story Parser

Since a story description looks mostly like natural language, the keywords must be capitalized in order to be recognized. This makes them stand out

from the labels used to denote names of sections, nodes, and events. The whitespace layout of the language is completely irrelevant, except that at least some amount is required between keywords. This means the person writing a description can align their code however they like.

Comma punctuation marks are used to separate transition definitions. These are required to delimit the label of the events of a transition and the following transition's tail node label. The parser also supports C-style comments, both line and block varieties.

6.2.2 Configuration Parser

The required configuration file utilizes a parser as well. Clearly by the EBNF definition below it is very simple and is currently used to choose the desired language for code generation. If the tool were to be developed further there could be many other configuration options added.

Listing 6.1: EBNF Grammar of the Configuration File

```
config = opt_whitespace , 'Generation' , whitespace ,
         'Language' , whitespace , '=' , whitespace ,
         language , opt_whitespace ;

language = 'C#' | 'C++' | 'Java' ;

whitespace = whitespace_char , { whitespace_char } ;
whitespace_char = ? any white space character ? ;
opt_whitespace = [ whitespace ] ;
```

6.3 Code Generation

The purpose of the SAGA tool is to generate a story manager to be used in a video game. In order to accomplish this the tool must produce compilable code in the same language in which the game is developed. The modules responsible for code generation make up the majority of the SAGA program.

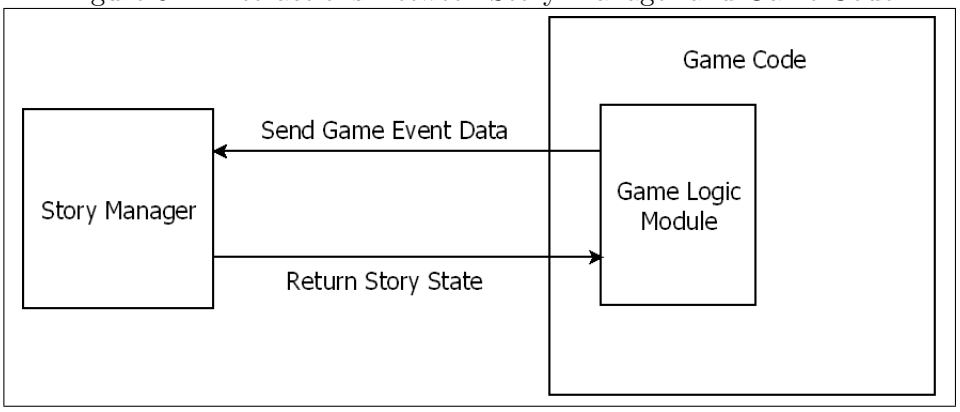
6.3.1 Story Manager Details

The overall picture of how the SAGA tool is used for game development was shown in Figure 6.1. The generated story manager module has some interaction with the rest of the game code. These interactions will now be explained in more detail.

When a story is specified using the SAGA DSL there are many events that describe how the story may progress through transitions. These critical story events will be arranged into a list, and will become part of the state of the story manager. These events can occur at some time during gameplay. At the beginning of execution of the game none of the events will have occurred yet. There will be a list of boolean values that corresponds to the list of event labels, indicating whether each event has occurred yet or not. The order of these events will be the order in which the labels occurred in the SAGA specification.

Figure 6.2 shows the information that is passed between the game code and the story manager. The game logic will keep track of the important story events in a list of boolean values having the same length as the number of story events of which the story manager is aware. At the start of game execution all of the values will be initialized as false as no events have occurred yet. As events are triggered by the player or environment the corresponding boolean values will be assigned true.

Figure 6.2: Interactions Between Story Manager and Game Code



Now there is the matter of communicating this data to the story manager to be analyzed. The story manager has a function that will take the event data from the game in some standard container form depending on the implementation language. Depending on the truth of these event values the story manager can determine if the story has progressed. The function will return the current story state as a story Node. The game logic can then check the label of this Node and decide how this affects the game. If the story has changed state then perhaps the game will display a cinematic, interface element, or perform any one of various responses.

6.3.2 Abstract Code - An Intermediate Representation

When using the SAGA language to describe a story, some problem information is supplied to the SAGA compiler. The SAGA source code itself provides no indication of how this problem information is transformed into a solution. There is a mechanism provided in the SAGA compiler that performs this transformation.

The SAGA program contains a module responsible for structuring a generic solution to the story manager creation problem. This generic solution consists of a set of abstract modules. These modules, when instantiated with problem data, represent the various object-oriented classes used in the generated code. In order to represent these somewhat complex modules, many Haskell data structures have been defined. The abstract code structure is a many-layered data structure that is built up from structures of packages, modules, state variables, and functions. These structures also require some notions of state types, function types, and parameters in order to be constructed.

The abstract code should be designed in such a way such that it is language independent. The meaning of this is that the abstract code representation should be able to be processed into code of any language. This is of course difficult because of the vast differences that exist between programming languages. During the design process of the abstract code it was desired to be able to represent code of both imperative and functional

paradigms. This was attempted at first, but it was realized that this goal was ambitious and outside the scope of this particular project. It was decided that the abstract code would be designed with object-oriented languages in mind, and in particular "C Family Languages" like C#, C++, and Java [1].

A comparison was discussed earlier about external and internal DSLs. The SAGA language is an external DSL, as it uses a stand-alone compiler to produce code. The abstract code representation can be thought of as an internal DSL. It is present inside the Haskell host language and is meant to look like a new language. Many assistive functions have been created in order to help the abstract code definitions to be visually similar to actual object-oriented code. Haskell provides means to easily define new operators and their characteristics such as right/left association and precedence.

The important module modeled using abstract code language is the story manager. This manager will contain a story as part of it's state, and also functionality to interact with game code. In order to construct this story state many other modules are required. These include modules to represent nodes in a story, the larger collections of story nodes referred to as sections, and also transitions between nodes and transitions between sections. These four modules together are utilized in a story module, which can then construct a story object for the story manager.

The problem information provided through the SAGA source is implanted into some initial data structures. Then this story-specific information is passed through a function that instantiates this data into the generic story manager solution. However, not all of the story manager logic is dependent on the problem data. There contains functionality that is already defined by the generic solution. This logic is "hard-coded" into the abstract code. The actual functions that interact with the story data, such as setting and getting functions for node and transition modules, will be defined in this way. Now that a fully defined solution has been created it is ready to be transformed into it's final form through a rendering process.

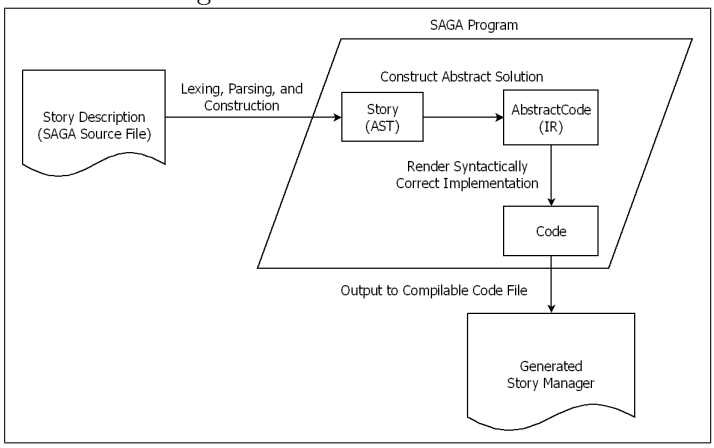
Chapter 7

SAGA Program Output and Code Interactivity

7.1 Rendering Usable Code

All of the logic needed for the actual generated code is present in the intermediate representation of the abstract code. All of this information is represented as Haskell data structures, and so another step is required to translate to the final generated code. See Figure 7.1 below for a general control flow through the SAGA compiler.

Figure 7.1: SAGA Data Flow



Since the intermediate representation must be rendered into different languages there must be a renderer for each one. That is not to say that all rendering is language-specific. Because the syntax of the three currently renderable languages is very similar there are quite a few functions that are usable across languages. These functions do however take a parameter indicating the desired rendering language, and so are language-parametric functions.

Each language's renderer is responsible for producing the correct syntax. For instance, while C# and Java use the keyword "Boolean" for a boolean variable, C++ uses "bool" instead. All the various syntactical differences such as these must be accounted for in the various renderers. Other considerations include syntax for standard container usage, as well as the functions that operate on them.

The rendering of C# and Java into code files was very similar. Each abstract module is rendered into a file having the same name. Rendering the C++ code is different since a header file is used. The rendering modules are designed to allow the code output to be put into multiple files.

7.2 Other Output

7.2.1 Story Graph

In addition to generating compilable code for a game, the program also produces a story graph for easy inspection. An example of this type of graph was shown in Figure 5.2.

This visualization is performed via the DOT language, which is a part of Graphviz, a graph visualization software [15]. As described earlier, after the story description is parsed some data structures are constructed that represent the story. There is a module in the program that handles DOT code generation from these data structures.

During the execution of the program, the "dot" command is invoked to produce a digital image of the story graph. This means that the Graphviz software must be installed, and that the required binary "dot" is recognized

at the command-line. Of course if the "dot" executable is not recognized the program will continue to perform it's functions, but will not produce a story graph.

7.3 Interaction Timing

One might begin to wonder about the timing of these interactions between the story manager and game logic. A video game is a real-time system after all, and events can occur very quickly and in succession depending on the style of game. How often must be story manager be called upon to check the state of the story?

The critical story events that the manager is considering will most likely be few and far between. These junctions in the game's story will be significant, and will probably not occur in a short amount of time. For this reason the interactions between the story manager and game need not be frequent. There are some other considerations when speaking about the timing of these interactions, namely game performance and human reaction times.

Performance is a hugely important issue in video games. There are countless operations that must be performed every second to provide a seamless experience for the player. People desire to have the highest frame rates and smoothest gameplay experience. Because of this, calculations spent on determining the story state should be minimized. Also, the number of calculations performed by the story manager for a single check is low.

The manager only has to check the events relevant to all outgoing transitions from the current story node. This can vary depending on how complex the story is, but typically there should only be a few available important story choices to make. Any available processing resources would be spent more wisely on other aspects of the video game than checking the story progression. The slowest possible story state checking would be optimal, as long as it does not affect the game experience.

Perhaps the most crucial consideration to determine the timing of the interactions can be decided by human nature. It is widely accepted that the

mean visual and aural reaction times for college-aged people are 190 ms and 160 ms respectively [21]. If a player cannot even physically react to more than four to six stimuli per second then it is pointless to update the story state more than this amount.

For the multiple reasons outlined above, the highest rate of story state checking should be about six times per second. If a certain game is on a tight computational budget then two or even one check per second would probably be acceptable and not disturb the user experience.

Rather than utilizing this polling architecture, another more elegant solution exists. Control inversion [12] can be utilized through an event-driven approach. This method makes more sense because then only necessary calculations need to be performed. Instead of asking the story manager at regular intervals whether something important has happened, the game engine will tell the story manager when something has occurred.

The story manager will register the critical events with the game engine, and so will be notified if one of these story-changing events occur. In this way the story manager will only be called upon when required, and will notify the game what effect this event has on the state of the story. The game engine can then decide on it's own schedule how to deal with this new information it has received from the story manager.

Chapter 8

Conclusions

The results of this project show the power of utilizing DSLs, and how they can improve the current state of video game development. Perhaps this method is a step in the right direction of advancing game development, and improving or sustaining time to market as game complexity increases.

In fact, it was not a very difficult task to create a human-readable language for story management. Technology such as standard parsing libraries and powerful programming languages allows the straightforward creation of a tool such as SAGA.

The transformation from abstract code to compilable object-oriented code was not an insurmountable feat. It does take some amount of effort and time to implement this feature of course, but it is not a major predicament that is difficult to solve. With an expressive enough intermediate representation, and general rendering framework it would be possible to "drop in" new output languages with minimal effort.

This being said, the component of the tool that required the most strenuous design effort was the development of the aforementioned abstract code. It was designed to be as general as possible in order to account for multiple output languages. Clearly some intelligent design had to be put forth to accomplish the task. Fortunately this investment of strong design awarded dividends. It is impressive that three output languages are available with this tool, even if these languages are quite similar.

An attractive and ambitious end-goal would be the existence of some standard video game development framework that makes use of multiple DSLs. There could be custom languages to describe most of the features of games. Employees could specialize in one or many of these languages, perform very productively, and achieve results quickly. Of course this would be good for development but possibly detrimental to competitiveness if this framework was available to everyone.

The SAGA tool is not likely to be appropriate for use in a real-world video game today. The tool has only been tested for simple proof of concept examples. In order to assess it's suitability for real game development the tool must be applied to more practical situations. Further development must be carried out to determine the capacity of a tool such as SAGA in the video game industry.

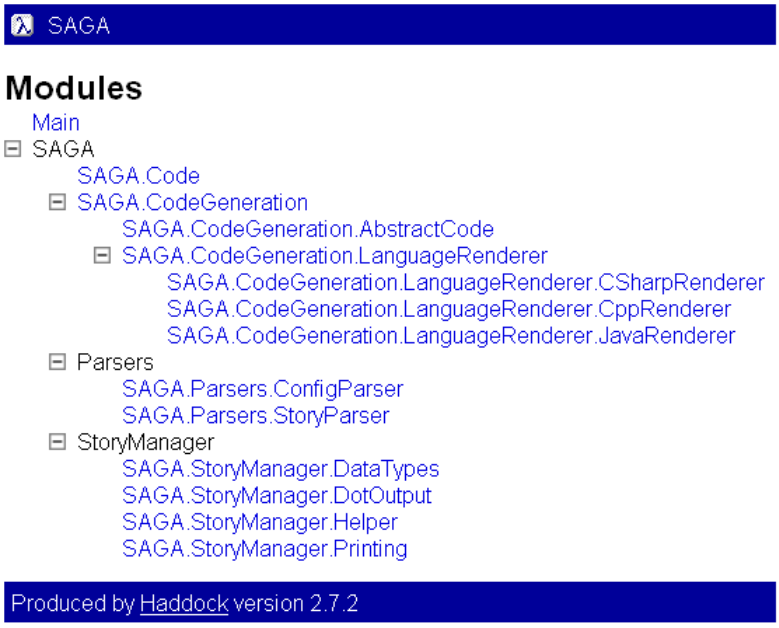
Appendix A

Module Information

A.1 Module Hierarchy

Haddock, a tool for generating documentation for Haskell code, produced the following module hierarchy output.

Figure A.1: SAGA Module Hierarchy



A.2 High-level Module Descriptions

A.2.1 Main

Contains the entry point for the SAGA program. This module holds the logic of organizating operation of various I/O operations including: initiate parsing, construction of required data types, and producing code files.

A.2.2 SAGA.Code

SAGA.Code

Contains the structure of the Code data type.

A.2.3 SAGA.CodeGeneration

SAGA.CodeGeneration

Contains the logic to construct Code data types from an AbstractCode, as well as the logic to produce the actual generated code files.

SAGA.CodeGeneration.AbstractCode

Contains the structure of the AbstractCode data type, which is some intermediate representation (IR) of the desired code. Specifically, an AbstractCode will be a logical structure of the solution that is desired to be generated.

SAGA.CodeGeneration.LanguageRenderer

Contains the framework for a class of renderers. This module has functions that define syntactical elements of the output languages. Some functions are general across the output languages, and they simply accept a parameter to denote language.

SAGA.CodeGeneration.LanguageRenderer.CSharpRenderer

Contains the logic to render compilable C# code from an AbstractCode.

SAGA.CodeGeneration.LanguageRenderer.CppRenderer

Contains the logic to render compilable C++ code from an AbstractCode.

SAGA.CodeGeneration.LanguageRenderer.JavaRenderer

Contains the logic to render compilable Java code from an AbstractCode.

A.2.4 SAGA.Parsers

SAGA.Parsers.ConfigParser

Contains the parsing logic for the SAGA configuration file.

SAGA.Parsers.StoryParser

Contains the parsing logic for the SAGA story description file.

A.2.5 SAGA.StoryManager

SAGA.StoryManager.DataTypes

Contains the structure of the data types that make up a story, as well as functions to safely construct these.

SAGA.StoryManager.DotOutput

Contains the logic to render DOT code and produce a file recognized by various DOT layout schemes.

SAGA.StoryManager.Helper

Contains various useful functions for processing, including turning strings into valid variable names for the renderable output languages.

SAGA.StoryManager.Printing

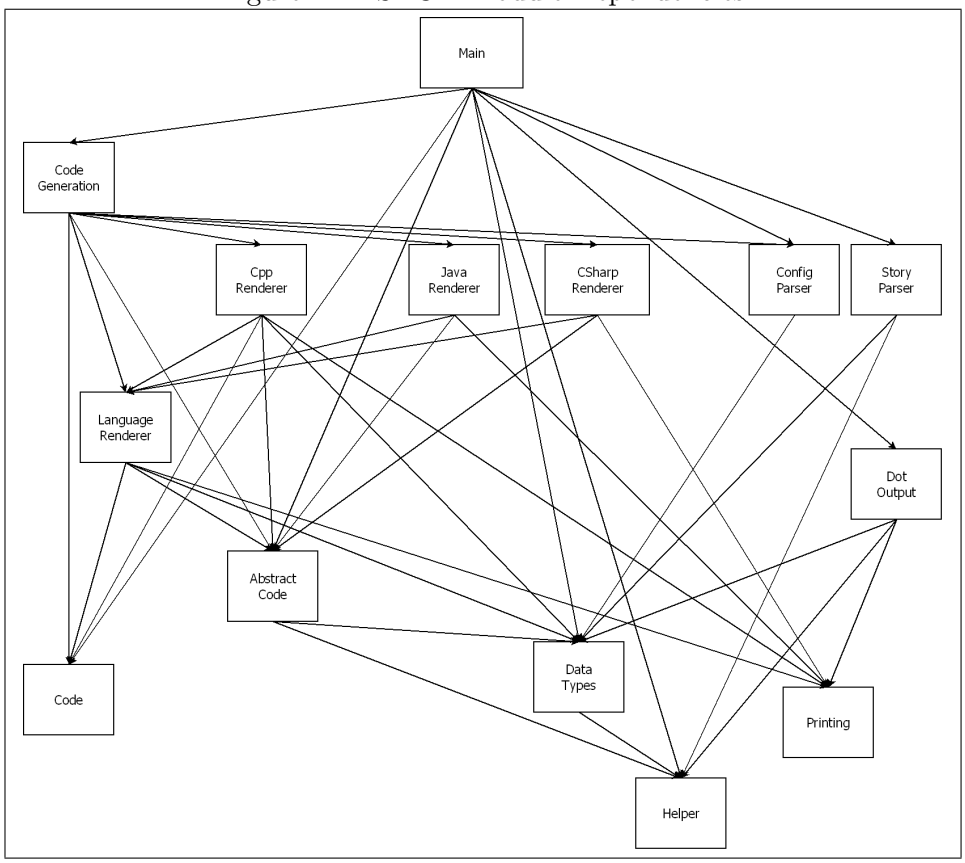
Contains useful printing functions that are not present in the Haskell pretty-printing library. The module holds logic to render the data types from the

DataTypes module into a finite form (since some of these types contain information of each other, a regular attempt to output a value will result in an infinite string).

A.3 Module Dependency Diagram

The module dependencies are shown below in Figure A.2.

Figure A.2: SAGA Module Dependencies



Bibliography

[1] The C Family of Languages: Interview with Dennis Ritchie, Bjarne Stroustrup, and James Gosling. http://www.gotw.ca/publications/c_family_interview.htm.

[2] Ernest Adams. *Fundamentals of Game Design*. Prentice Hall, 2009.

[3] International Game Developers Association. Scriptwriting for Games: Part 1: Foundations for Interactive Storytelling. http://aii.lgrace.com/documents/IDGA_Foundations_of_Interactive_Storytelling.pdf.

[4] International Game Developers Association. Scriptwriting for Games: Part 2: Advanced Plot Story Structures. http://aii.lgrace.com/documents/ScriptwritingforGames_Part_2_Parallel_OO.pdf.

[5] Eric Bangeman. Growth of gaming in 2007 far outpaces movies, music. http://biae.clemson.edu/bpc/bp/Lab/110/reaction.htm.

[6] Magnus Christerson. Intentional Software at Work. http://www.infoq.com/presentations/Intentional-Software-at-Work.

[7] Christophe de Dinechin. Concept Programming - The Art of Turning Ideas into Code. http://xlr.sourceforge.net/ConceptProgrammingPresentation.pdf.

[8] Sadek Drobi. Lennart Augustsson on DSLs Written in Haskell. http://www.infoq.com/interviews/DSL-Haskell-Lennart-Augustsson.

[9] Ubisoft Entertainment. About the Game. http://ruse.us.ubi.com/index.php?page=about.

[10] James Alan Farrell. Compiler Basics. http://pages.prodigy.net/j_alan/hitech/compiler/compmain.html.

[11] Brock Ferguson. Gaining Entry to Game Development. http://www.gamedev.net/page/reference/index.html/_/reference/110/135/advice/gaining-entry-to-game-development-r1658.

[12] Martin Fowler. InversionOfControl. http://martinfowler.com/bliki/InversionOfControl.html.

[13] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley Professional, 2010.

[14] Carlo Ghezzi, Mehdi Jazayeri, and Dino Mandrioli. *Fundamentals of Software Engineering*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2nd edition, 2002.

[15] Graphviz. Graph Visualization. http://www.graphviz.org/About.php.

[16] Frank Harary. *Graph Theory*. Westview Press, 1994.

[17] John Harris. Game Design Essentials: 20 Open World Games. http://www.gamasutra.com/view/feature/1902/game_design_essentials_20_open_.php.

[18] ISO and IEC. ISO/IEC 14977. http://www.cl.cam.ac.uk/~mgk25/iso-14977.pdf.

[19] Kristine Jørgensen. Problem Solving: The Essence of Player Action in Computer Games. In Copier Marinka and Raessens Joost, editors, *Level Up Conference Proceedings: Proceedings of the 2003 Digital Games Research Association Conference*, page CD Rom, Utrecht, November 2003. University of Utrecht.

[20] Kurt Kalata. The History of Dragon Quest. http://www.gamasutra.com/view/feature/3520/the_history_of_dragon_quest.php?print=1.

[21] Robert J. Kosinski. A Literature Review on Reaction Time. http://biae.clemson.edu/bpc/bp/Lab/110/reaction.htm.

[22] Griffin McElroy. Mass Effect 2's save game import features explained. http://www.joystiq.com/2009/12/26/mass-effect-2s-save-game-import-features-explained/.

[23] David Oso. Interactive Drama, is it really a new genre? http://gamasutra.com/blogs/DavidOso/20110417/7447/Interactive_Drama_is_it_really_a_new_genre.php.

[24] Richard E. Pattis. EBNF: A Notation to Describe Syntax. http://www.cs.cmu.edu/~pattis/misc/ebnf.pdf.

[25] Edward Rothstein. A New Art Form May Arise from the 'Myst'. *The New York Times*, December 1994.

[26] IGN Staff. Starcraft. http://pc.ign.com/articles/152/152159p1.html.

[27] TDA. The History of Real Time Strategy, Part 2.1: The Glory Years. http://www.gamereplays.org/portals.php?show=page&name=the_history_of_real_time_strategy_pt2_1.

[28] Daniel Terdiman. Video game sales explode in industry's best month ever. http://news.cnet.com/8301-13772_3-10435516-52.html.