Advl: A Dynamic Visualization Language

by

Tolga Cerrah

A Project Report Submitted
in
Partial Fulfillment of the
Requirements for the Degree of
Master of Science
in
Computer Science

Supervised by

Dr. Hans-Peter Bischof

Department of Computer Science

B. Thomas Golisano College of Computing and Information Sciences
Rochester Institute of Technology
Rochester, New York

May 2016
Abstract

Advl: A Dynamic Visualization Language

Tolga Cerrah

Supervising Professor: Dr. Hans-Peter Bischof

Visualization is a process used by scientists to help the understanding of the data. During this process, the scientific data is turned into an image or a combination of images that we call a movie. Movies give a visual aspect to the data, making it easier to comprehend and analyze. During a movie, there might be points of interest that a scientist needs to put the viewer's attention to. To achieve this, visual aspects of the visualization can be changed when a point of interest is reached. Time is usually the value that determines these points of interests and can be used to determine the values for directing. This is similar to directing a real movie. A director may want the camera to move closer to an object at a scene at a specific time to show the audience the importance of that object. Dynamic visualization is a process that allows changing the visual aspects during the visualization. The Spiegel framework contains a directing language that can be used to achieve this task. Advl is a new language that adds more functionality by allowing the use of functions, control statements, variables, and mathematical expressions. Advl also contains additional features that ease the coding for the user. These features increase the control over the visualization process, and also give more power to the user. This report focuses on the design and implementation of Advl, and also discusses a use case.
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Abstract</strong></td>
<td></td>
<td>ii</td>
</tr>
<tr>
<td><strong>1 Introduction</strong></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>1.1</td>
<td>Project Description and Motivation</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Related Work and Background Information</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Overview of Spiegel Framework</td>
<td>3</td>
</tr>
<tr>
<td>1.4</td>
<td>Approach and Challenges</td>
<td>3</td>
</tr>
<tr>
<td><strong>2 Design</strong></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Types</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>Advl</td>
<td>6</td>
</tr>
<tr>
<td>2.2.1</td>
<td>Variables</td>
<td>6</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Streams</td>
<td>6</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Anchor Points</td>
<td>7</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Statements</td>
<td>7</td>
</tr>
<tr>
<td>2.2.5</td>
<td>Expressions</td>
<td>8</td>
</tr>
<tr>
<td>2.2.6</td>
<td>Functions</td>
<td>10</td>
</tr>
<tr>
<td>2.3</td>
<td>Lexer, Parser, and Tools for the Language</td>
<td>10</td>
</tr>
<tr>
<td>2.4</td>
<td>PT Walking and Code Generation</td>
<td>11</td>
</tr>
<tr>
<td><strong>3 Implementation</strong></td>
<td></td>
<td>13</td>
</tr>
<tr>
<td>3.1</td>
<td>Lexer and Parser</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>PT Walking and Code Generation</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Variables and Scope</td>
<td>14</td>
</tr>
<tr>
<td>3.4</td>
<td>Values</td>
<td>15</td>
</tr>
<tr>
<td>3.5</td>
<td>Expressions</td>
<td>16</td>
</tr>
<tr>
<td>3.6</td>
<td>Functions</td>
<td>16</td>
</tr>
<tr>
<td>3.7</td>
<td>Control Statements</td>
<td>17</td>
</tr>
<tr>
<td>3.8</td>
<td>Streams and Anchor Points</td>
<td>18</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

This chapter will focus on the project description and motivation, related work, background information that is necessary to understand the project, the proposed solution, and some of the challenges I faced working on this project.

1.1 Project Description and Motivation

Visualization systems give the representation of the data in a more comprehensive way. The data that is visualized has some parts that are not of interest, and there are some that are important for the experiment. The main goal of a visualization is to be able to emphasize on the parts of the data that are point of interests to the scientist. The data is usually based on a scalar value; time. The visualization framework that Advl is aimed to be a part of, Spiegel[3], uses time as to create those anchor points and direct the visualization using them. Spiegel contains a directing language to achieve this task[4]. This directing language has certain keywords that can be used by the programmer. There are streams, and anchor points. The input to the directing program is time. Time creates the anchor points. The streams have to be declared first and then assigned new values inside those anchor points. At the end, the values for other components are determined by those streams depending on the viewer time. This allows the directing of a visualization to be just like a real movie. Depending on the time, we can change the value of a camera component so that, we can move the camera in a certain way. Another application would be to change the color or
transparency of an object to direct the attention of the viewer into that specific object. Values for the streams are set inside this directing language, but the functionality is limited. The language doesn’t allow you to have control statements, functions, variables, or expressions. Adding those in to the language lets the user to do scientific calculations and have a better control over the values of the streams. Adding those features will also diminish some of the other problems such as the repetition of code. Advl is aimed to address these problems and have the features listed above in addition to the current functionalities.

1.2 Related Work and Background Information

Most of the visualization systems that I was able to find, focused more on variety of different aspects of a visualization other than dynamic visualization. In many cases, dynamic visualization was achievable with different approaches but difficult to do so. For this project, I also did a research on camera motion and cinematography to get a better understanding of the subject.

Camera motion and cinematography was the starting point of my research. As much as they helped me to understand the creation of movies in the world, it also showed that cinematography is different than visualization. Problems in the cinematography [8] gave me an insight on what problems are encountered in cinematography and how they are solved by the directors. This was a good source to realize the differences between real world cinematography and visualization. A multi-criteria approach to camera motion [7] focused on camera motions and path generation. The authors propose a way to find paths between point of interests for camera motion. They also discuss where the camera should look at during those transactions. Real-Time Cinematography for Games [6] provided detailed information on cinematography in games. I learned a great deal about some techniques used in real life cinematography and how they can be incorporated in a visual world. Along with some detailed information there were examples of how a camera should move, how it should be positioned, and the importance of perspective in a shot. All these resources had a different context compared to scientific simulation or the type of visualizations I was
aiming to find, but they were extra knowledge about camera motion which was useful for my understanding of the subject.

Forbes [5] focuses on behaviorism; a visualization framework proposed by authors to address some of the issues in visualization. It explored different domains in visualization. The proposed framework was structured by three connected graphs. The timing graph was used for animation. It is an interesting approach but it wasn’t a feasible approach for my project. VisIt [1] is one of the most popular tool used in visualization of scientific data. Even though it contained an extensive guide for all the features it has, there wasn’t any information on dynamic visualization. Animations were created using keyframes, and/or Python scripts. I couldn’t find an approach that handles dynamic visualization in a simple way to let the user direct the visualization.

1.3 Overview of Spiegel Framework

Spiegel[3] is a visualization framework that lets the user create specialized visualization systems. It is written in Java. It consists of components that are connected as a directed graph. Each of these components have a number of input and output channels. Spiegel makes sure that the communication between these components is established and the data is transmitted from the sending component to the receiving component. These components can be given different arguments such as color or transparency to fine-tune them. These arguments can be modified during the visualization to achieve dynamic visualization.

1.4 Approach and Challenges

I faced a lot of challenges during this project. As a result, I made some mistakes and learned from them. The first step I took was to search for related work, and to discover the current approaches to dynamic visualization. There were a lot of visualization systems, but while dynamic visualization was possible, none of them really addressed a simple solution to it. A lot of these visualization systems were focused on other aspects of a visualization like
quality. Many of them were specialized in a certain type of visualization. As suggested by my advisor, I started thinking about creating a new language to handle some of the problems at hand. I first created the grammar for the language. The challenge I faced there was to figure out what I can and can not have in the grammar. It turned out to be an ongoing process, and the grammar kept changing until the last week. I decided to use the Antlr4 tool to create a lexer and a parser for the language. The parser created by the tool was able to take source code for the language and create a parse-tree from it. Rest of the work was to figure out how to walk this parse-tree, evaluate everything and generate code that can be recognized by Spiegel. I first started using a listener type of approach to walk the parse-tree. The main problem with that was the structure of a language. I started annotating the parse-tree nodes to store information like values of expressions on that node. It started to become really complex and almost unmanageable. After working on it for a while, I had to make a change to an approach that uses visitors. In the evaluation phase, one of the biggest challenge I had was handling the functions. Functions had variety of problems in the implementation. Some of these problems included the matching of parameters and arguments, scope inside the function, and recursion. Variables were also somewhat tricky to handle but I was able to do it by creating a hierarchy of scopes that contain the values of that variable inside that specific scope. The last phase I handled was code generation, which computed the values of streams inside different anchor points, and output them in a format that is currently recognized by Spiegel.
Chapter 2

Design

Advl supports variables, functions, statements and expressions in addition to streams and anchor points that are used to generate the output. Appendix A shows the grammar of Advl. A program contains a variable block, followed by streams, followed by functions and finally followed by anchor points. This was done to ensure that it is easy to read and understand the code. Also, streams have to be defined before and can be used globally for every anchor point. The process of creating a compiler is not an easy task. After writing the grammar, I started searching on tools that can help me and I came across several. Antlr4 is a tool that is commonly used and eases the creation of a lexer, a parser, and the application code afterwards. Antlr4 is my tool of choice for this project.

This chapter includes detail on the elements of the language, and also the design of the compiler.

2.1 Types

There are four regular types used in the language; int, double, point, and boolean. There is also an additional return type for the functions; void. All the variables declared inside the language can use those four types. Types for streams are stored with the corresponding stream information such as name and interpolator. Types of variables are stored when they are first declared and checked when they are assigned to a new value. Functions checks what they return with their return type and they are also responsible to check the argument types with the parameter types. Mathematical expressions can use both the int and double
type, however when both are combined, integer is casted into a double. The boolean type is used inside control statements.

2.2 Advl

Advl is created so that it is simple to learn, easy to read, and helpful to reinforce the usage of streams and anchor points. This section includes brief explanation on the parts of the language.

2.2.1 Variables

Variables can be used throughout the program. The variable block in the beginning of the program contains the declaration for the global variables. There the variables can be defined, and values can be assigned to them as an option. Those variables can be accessed from anywhere in the code. Variables can also be declared locally inside any block of statements. The scope of these variables are only limited to the given block, and can not be used outside of it. They can also be used as parameters, control variables inside control statements like a for loop, and can be accessed by the local block. When a variable is declared, a type has to be given to it. It can also be flagged as a constant with a ’const’ in front of it. These variables can not be assigned a new value. There are also special operators ’+=’, ’=’, ’++’, ’–’ that lets the user assign values to the variables easier.

2.2.2 Streams

Streams have to be declared in the stream block in the beginning of the program. Name of the streams and the type has to be given. Assigning the stream an interpolator is optional, if there isn’t any, a default interpolator is assigned. Streams are stored throughout the program. Also, whenever their values are changed, the change is stored with the corresponding time.
2.2.3 Anchor Points

Anchor points include the time followed by a block of statements. The time can be defined in several different ways. If the keyword 'time' is used, it means that the latest time will be used as a value. This is followed by a double value. Identifiers can also be used here. For example, the expression 'time + deltaTime' for the time value would be used to telling the language to take the anchor point time as the last used time added by the value of deltaTime which is set to be a double value inside the variable block. The block of statements can contain assignments to the streams. These assignments are tracked by the language to be used as a part of output at the end of execution. Anchor points can also have loops inside them that will increase the time by a delta value that is defined by the control statement. Anchor points time values always have to increase. An anchor point with a lower double value of a time can not follow another anchor point.

2.2.4 Statements

Statements are used inside a block in the program. There are five general types of statements; if statement, while statement, for statement, function call, and variable declarations and assignments.

- If Statement:

  If statements’ structure is the same as the ones in Java. An if statement is composed of an if block, 0 or more else if blocks and an optional else block. The program goes through each of these from the start and stops if one of the expression turns out to be true. Also, the only time a block of statements are evaluated is when the corresponding boolean expression returns a true value.

- While Statement:

  While statements’ structure are also similar to the ones in Java. The boolean expression inside the while loop is evaluated before each iteration and if it is true the block of statements are visited again for evaluation.
• For Statement:

For statements’ structure is a little bit different. For the loops you declare an integer by giving it a identifier, a starting value and an ending value along with a delta. This delta controls the process of going from the start to the end. Also, when a for loop is used inside an anchor point, the current time used as the anchor points will be incremented by the delta value of the for loop. Each iteration can change the stream causing the program to store the streams in different anchor points. The addition of the for loop eliminates the repetition in the code which was one of the problems that was aimed to be addressed.

• Function call:

Function calls are in the form of an identifier with the arguments inside the parenthesis. These arguments have to be expressions and will be evaluated before invoking the function. To invoke the function, program finds the function signature by using the identifier of the function and the types of the arguments.

• Variable declaration and assignment:

Variable declarations are in the form of a type that has to be one from the four types described above, the identifier, and an optional expression. There is also an option to use a keyword 'const' to make sure the variable can not be assigned any other value. If the optional expression is there, the program evaluates it first and then defines the variable with the evaluated value. Every variable is declared inside a scope. Assignments are similar to the declarations but they don’t contain their type, and they have to have an expression which they will be assigned to. If the types don’t match, or the identifier can not be found in the scope, an error message is sent to the user.

2.2.5 Expressions

• Binary Expressions:
Binary expressions are in the form of an expression followed by an operator and another expression. One type of binary expression are the boolean expressions. Boolean expressions take two expressions that can be compared to each other and apply the operator on the operands. The operators for boolean expressions include equality and comparisons such as greater than or less than or equal to. Another type of binary expressions are the arithmetic operations. These include addition, subtraction, multiplication, and division. There are also other mathematical operations that are supported by the language such as taking the power of an expression or modulo operation. For every type of operation, the type checking occurs to ensure that the expressions can have the corresponding operator apply to them.

- Unary Expressions

Unary expressions are in the form of an operator followed by an expression. The unary operators supported by the language include absolute value, square root, and negation. Expressions have to be in the type supported by the operator.

- Trigonometric Expressions

Trigonometric functions are in the form of unary expressions. Like Java, the types of expressions passed into the trigonometric operation should have the supported type. The supported operations are sin, cos, and tan.

- Atomic Values and Identifiers

Lastly, expressions can be atomic values that are either values of the type int, double, point or boolean or they can be identifiers which have to be checked in the given scope. If the identifier is not found, the program will tell the user that the identifier is not declared.

- Function Calls

Function calls for expressions are the same as the ones for statements. It is in the form of an identifier followed by a list of expressions inside parenthesis that are going to
be passed as arguments to invoke the function.

2.2.6 Functions

Functions are in the form of a return type followed by an identifier that is the name of the function which is followed by a list of parameters and a function body. The parameters have to be in the form of the type of the parameter followed by an identifier. Function body contains a block of statements that may or may not have a return statement. Return statement doesn’t have to be there if the function is set to have void return type. Block of statements is only evaluated when the function is invoked.

2.3 Lexer, Parser, and Tools for the Language

After the grammar was created for the language, the next steps were creating the lexer which is responsible to go through the source code and change the input stream into a set of tokens. Later these tokens are picked up by the parser, and are used to generate the parse-tree. I wanted to use a tool to achieve this and decided to use Antlr4. Antlr4 has many features that are extremely useful for compiler generation. It has a similar syntax to BNF form that I wrote my grammar in. After defining the grammar inside Anlr4, there are several files that are automatically generated. Figure 2.1 shows the files generated. Lexer and Parser are generated without a need in implementation. Parser generates a parse-tree from a source code. It is also possible to get a visual representation of the parse-tree that is generated by a single command. This is extremely useful for testing the grammar and making the tweaks that are necessary. Another type of file that is generated are the context objects. For every rule that the grammar has, a context object is created. This is in correlation with the nodes of the parse tree. Context objects are extremely useful going through the source code. We can access every element of an expression, or simply get the line number of where a statement is by using simple function calls to the context object of that node. Another type of file that is automatically created is the a listener class that contain empty methods for every node and can be implemented by the programmer to walk
the parse-tree created by the parser. With a special command, Antlr4 is also capable of creating a visitor class instead of the listener which is what I used.

2.4 PT Walking and Code Generation

The last step of this project is to walk the parse-tree that is generated, evaluate the program, and create output that can be recognized by the Spiegel framework. The overall pipeline
can be seen in figure 2.2. Application code written for it in Antlr4 is in Java. For parse-tree walking the visitor pattern is selected for this project. After using both the listener, and the visitor approach it became apparent that the visitor pattern was the natural way of evaluation. Listeners have to go through every node in the parse-tree. This becomes redundant, as some of them are not needed to be visited especially with control statements. Also, there is a lack of control on the walk. This is a problem for many parts of the language such as the control statements. A visitor approach addresses those problems. A visit function for every node is created automatically by Antlr4 inside the BaseVisitor class. This class is implemented to fill those empty methods. Every visit call has to have a return value of the same type. This was one of the main limitations of the approach. A wrapper class is formed to handle every type of value that is returned by the visit calls. Also, the node’s context object is passed inside the visit method to handle the information necessary for the implementation. There are two passes done to the parse-tree to ensure that all the functions are recorded in the first pass and the whole program is evaluated in the second pass with the defined functions. During the evaluation the streams and the anchor points are recorded. Each time a stream changes a value in a specific time this is also recorded to later use to write on the target code. The evaluation is responsible for generating a map for streams that are declared in the stream block and a map of timeframes that contain the values of the streams in that specific time. A writer class is created to take these maps as an input and output them to the target code. The created file has the extension ‘.movie’ added to the input name.

![Figure 2.2: Pipeline of Antlr4](image-url)
Chapter 3

Implementation

The implementation of the language was mainly done in Java as it was the supported language for application code in Antlr4. The implementation phase can be divided into three main stages; creating the lexer and parser, evaluation and application code, and converting the results into target language. Evaluation and application code is the most complex part of the implementation, and majority of the sections below are used to explain this phase.

3.1 Lexer and Parser

The implementation of the lexer and the parser was automatically handled by Antlr4. The main work that is done for this stage was the creation of the grammar for the language and also defining it correctly in the Antlr4 syntax. It was important how the lexical tokens were defined. Some rules of the grammar were labeled, so that they can be later used by the parse-tree walker. For example, statements have a rule defining it as either an if statement, a while statement, a for statement, a variable declaration or assignment statement, or a function call. These options are all labeled properly, so that when a statement is visited during evaluation the correct visit function is called. The context object for the following labeled rule contains information on every token on that line. Functions to retrieve this information are automatically created after running Antlr4. If an optional token is not used a call to the function will return null, and this is used in the application to perform certain operations. An example would be a variable declaration. During a declaration it is checked to see if the variable is marked as a constant. Context object created during this
initial stage can be called on the const value, and if it returns null that means the variable is not a constant. Another example is checking the variable declarations to see if there is an assignment. If the context object for an expression returns null, that means that there wasn’t an assignment. If it doesn’t the expression is returned for evaluation. After the evaluation the value that is returned is assigned to the identifier in the current scope.

3.2 PT Walking and Code Generation

A new class called AdvlVisitorMain is created to handle the whole process. The process starts from taking the source code as an input and goes all the way to writing a file that has the output. The main method takes the source code from input, calls the lexer that is created from the previous stage to get the tokens. After that, the tokens are passed to the parser and in return a parse-tree is generated. The main class also contains two classes inside that are used to walk the parse-tree. First one is only responsible for walking the function declarations and store the corresponding information in a map. This information is used by the main evaluator class when a function is invoked. The second class is the main evaluator class. This class implements the AdvlBaseVisitor that is automatically generated by Antlr, and implements every visit method for every node of the parse-tree. After the evaluation is done, two separate maps are recorded. First one is the map that contains all the streams that are declared. Second one is the map that contains all the stream values in different anchor points. When the evaluation is done, and these two maps are set, this information is sent to a writer object. The writer class is implemented in such a way that it takes these maps, and when the write method is called, it iterates through the streams and the anchor points and prints them in order that is recognizable by the Spiegel directing language.

3.3 Variables and Scope

Scope is an important part of the language to handle the variables. A scope has a reference to it’s parent scope and contains a map of variables that use the identifier as the key and a
variable object as the value. This variable object contains information on the name, type, and the assigned value of the object. There is also a boolean value that stores if the given variable is a constant or not. When a scope is created, the current scope inside the program is set as its parents. Then the current scope is set to be the newly created scope. This creates a tree-like structure for the scope, the global scope acting as a root. When the evaluator is created for the first time, a global scope is created with no parents. Whenever the program gets into a block of statements, a local scope is created. When the block is left, the current scope is set back to the parent. Functions also have their own scope. Every time there is a variable declaration, the variable is stored inside the current scope. Also, the control variables such as the ones used for a loop are stored inside the local scope of that loop, so they can be used by the statements. Functions define the arguments with their corresponding parameter identifiers inside their own scopes. Each time a function is called a new scope is created and a new instance of the evaluator is created so that the parameters that are declared inside this scope won’t have any effect in a recursive call unless they are passed as arguments to the call. When there is a variable assignment, or when an identifier is evaluated the identifier will first be checked in the current scope. If it is not found the parent will be searched for the identifier. This will continue recursively until the global scope is reached. If the global scope is reached, and the identifier is still not found in the maps, then an error is given saying that the identifier is not defined in the scope.

### 3.4 Values

The base visitor is created by Antlr4, and is implemented by the evaluator. Every visit method has to return the same type when called. This causes a problem as there are more than one type in the language and some of the nodes that are visited don’t return a value. An example to that would be visiting a block of statements. The visit method for a block just calls the corresponding visit methods of the statements after creating a new scope. To handle this issue a wrapper class is created for the values. This wrapper class contains an object value that is cast into its type when needed. This is used when the values have to be
evaluated and used as operands. The type of the object can also be checked by a method. Wrapper class can handle double, int, point, and boolean. In addition the type void is added so that the visit methods that don’t need to return a value can return void.

3.5 Expressions

There are many type of expressions in the language. They are implemented in a similar way. When an expression is being evaluated, if it contains other expressions, at first those expressions’ visit methods are called and then the given operator is applied to them. A basic example would be addition, a binary expression. When a variable is added to a double value, at first the values for the these expression are checked. The identifier for the variable gets resolved from the current scope and the value is returned. The double expression is an atomic expression and the visit call returns the value. Than those values are checked for their types, if they can be added together in the context of the operator which is addition. Finally they are added together and the value is returned the caller of the current visit method. All the operators are also supported by Java. So, when the values are checked for the operands, the operation is done using Java. Math class is used for some of the operations such as the absolute value function. Function calls and identifiers are the last remaining type of expressions. Variables are covered above, whereas functions are covered below.

3.6 Functions

Implementation of the functions have two main parts. The call to the function and it’s declaration. The main class that handles the whole process uses an evaluator for storing functions in a map. This evaluator is the first pass of the program but it only visits the function declarations. This is done separately then then rest of the evaluation to ensure that the functions are stored properly before they are called. When the walker finds a function declaration, it first collects the information of the return type, identifier, the parameters, and
the function block. The parameters are a list that contains a pair of types and identifiers. The identifier of the function and the types are used to create the function signature. The function signature is used as a key to the map that stores all the functions. All of these information including the function body is used as an argument to create a function object. Function body is not evaluated, but passed as a parse-tree block. It can not be evaluated before it is called. The function class has only one method that is used to call the function. When a function is called either as an expression or as a part of a statement, the first thing done is the evaluation of the expressions that are passed into the function call as arguments. When the values are found, they are checked for their types. This is used to figure out the function signature along with the identifier. If the function signature is found as a key in the map, the call method that was created inside the function is called with all the gathered information such as the arguments. Also, the map that contains other functions are passed as an argument to the function call, as well as the current scope at that point. The call method inside the function first creates a scope. Afterwards the arguments are matched with the parameter identifiers and are stored in the current scope which is just created. For the function body, a new evaluator is created and the current scope of the function is sent into it. This allows the usage of recursion with the functions. All the statements inside the block are visited. Blocks can have return expressions inside them. If a return expression is available it is caught by the function and returned back to the function call. This is done by making the return expression extend the RuntimeException. Also, the evaluation is placed inside a try-catch block to get this affect.

3.7 Control Statements

Function calls, variable declarations and assignments are a type of statement that are already covered by previous sections. This section will cover the control statements including the if statements, while statements, and the for statements. For the if statement, the first thing that is checked is the condition expression. If this expression is a boolean and returns a true value, the visit method is called for the if block. If not, the program iterates through
the else if blocks until a condition that returns true is reached. If any of them don’t return true, finally the program checks if the optional else block is there. If it is the else block is visited for evaluation. While loop’s condition is first checked and if it is true, the identifier is defined inside the scope in every iteration. Finally, the function body is visited again to evaluate the statements. When a for statement is encountered there are three values that are evaluated first. The start value, the end value and the delta. After calculating the values they are used in a Java for loop for implementation. The identifier inside the condition is defined inside the scope in every iteration. Much like the others, at the end the block of statements are evaluated.

3.8 Streams and Anchor Points

Streams can only be declared inside a stream block in the beginning of the program. When the program evaluates the streams, three Strings are returned from the context object; the name, the type, and the interpolator. These are then used to create a Stream object, that also has an empty field called value. These newly created streams are added into the map of streams by setting the key as the name of the stream, and value as the stream itself. Streams can only be assigned values inside the anchor points. When a variable declaration occurs, the first thing that is checked is if the identifier is defined in the stream map. If it is than a function is called that handles another map called the time map. Time map uses the different anchor points as it’s key. The value is another map that uses the stream names as keys, and the streams as values. When this helper function is called, the program checks the map on the current time to see if this stream is already defined with a value. If it is then the value is changed to the evaluated value. If it is not, a new entry is created. If the time does not exist on the map, then a new entry is created by adding the time and the corresponding the stream into it.

Anchor points’ time is the first thing that is evaluated. Firstly it is checked to see if the keyword time is used. If it is, then the current time’s value is taken and added to the rest of the expression that can contain an identifier or a double value. Then the current time is set
as the newly evaluated time. Next step is to visit the block of statements inside it. Blocks inside an anchor point have a special case. When there is a for loop, for every iteration the time is moved by the delta value of the for loop. Time is not entered into the time map until an assignment for a stream occurs.
Chapter 4

An Example and Discussion

This chapter is aimed to discover a use case of the language. An example is broken down into parts to make it easier to read. After the code is briefly explained, the advantage of using this language is discussed along with how this could have been implemented without Advl.

```c
var {
    const double startTime = 10.0;
    const double deltaTime = 5.0;
    double y = 20.0;
    double z = 20.0;
    double r = 100.0;
    double midX = 10.0;
    double midY = 20.0;
    bool circular = True;
}
```

The language lets the user to declare and optionally assign global variables in the beginning of the program. Using the const keyword ensures that the value of the variable will not be allowed to change in any case. Double values startTime and deltaTime are used by the anchor points to set the time. Rest of the values are both used in the function and also
in the anchor points.

```plaintext
cameraPos {
    type point;
    interpolator TCB;
}
```

```plaintext
simTime {
    type double;
    interpolator Linear;
}
```

Streams are declared right after the variables. Streams are stored in a separate data structure and are always checked when there is an assignment. Streams can only be assigned inside anchor points. In this case, they are given an interpolator but it is optional. If there is not one specified, the default is chosen.

```plaintext
point moveCamera(double x) {
    double xComp = (x - midX) ^ 2;
    if (circular) {
        double y = sqrt(r - compX);
        y += midY;
    }
    return (x, y, z);
}
```
Functions are declared right after the streams and right before the anchor points. In the moveCam function different values of x are passed inside to calculate the y value. There is an if statement inside to check to see if a particular motion is preferred for the camera motion. The resulting y value is used to return the new point.

```
startTime {
    cameraPos = (x, y, z);
    simTime = 10.0;
}

time + deltaTime {
    double x = 5.0;
    for (l = 1 : 20 : 1) {
        cameraPos = moveCam(x + i);
    }
    simTime += deltaTime;
}
```

The final elements of an advl program are anchor points. Anchor points define the time at that point before the block of statements. This value is calculated and set as the current time. The first anchor point is set to be the start time which was declared as a constant variable. Second anchor takes advantage of the keyword time which returns the current time. The time is calculated by adding the current time to the delta time for the second anchor point. Note that, the main goal of an anchor point is to assign the values for the streams. Other than just assigning them a value straight ahead like how the first anchor point works, the second anchor takes advantage of two features of advl. A function is used to calculate the value of the stream for every iteration. Also, a for loop is used to assign
the values to the stream in increasing time values. In this example, each iteration inside the loop increases the time because of the for loop’s delta value. Finally, anchor points increase in time. An anchor point declared afterwards in the code can’t have a smaller time value.

4.1 Discussion

Advl makes the calculations easier by adding the functionality to take advantage of variables, functions, statements and expressions in the directing language. It wouldn’t be possible to change the streams in such a manner in the previous version without an external calculation. Control statements also allow the user to have more flexibility in the calculations. Some of the extra features can also be seen as really useful. Anchor points are more flexible with the use of variables. When an anchor point is changed in the middle of other anchor points that are relative to each other, having a delta variable would make it so that the change would happen for all of them. Loops greatly decrease the repetition of code. If we were to make an assignment with every iteration of the for loop in this example, we would need twenty anchor points which is not feasible at all for the same code. As the source code gets bigger, the problems also increase.
Chapter 5

Conclusions

Advl: A Dynamic Visualization Language, is able to take a source code that contains variables, functions, statements, and expressions in addition to streams and anchor points, evaluate it, and convert it into a file that can be recognized by the Spiegel framework. This basic language is easy to use and gives more power and control to the user as explained with the example. The output format is the same as the input for the director language that is being used right now. The added functionality helps the user to achieve more while it is handled as easy as before.

5.1 Future Work

There are several directions that Advl can go in the future. The language at it’s core allows the use of streams and anchors, but many other features such as control statements, functions and variables are supported. These features are thought to be useful for a directing language. As a future work, the language has to be tweaked and more features have to be added depending on the need. One need would be the addition of print statements. It is hard to debug the source code right now. A print statement added to the language along with the String type can be used to address this problem. Advl lacks proper error handling, and depends a lot on Java to generate the errors. This can become problematic as exceptions that are thrown by Java can be tough to understand. Intermediate stages can be created in the future for type checking and error handling. Another application could be to integrate Advl into other visualization frameworks.
Bibliography


Appendix A

Advl Grammar

prog: fps vars? streams func* anchor+

fps: ‘fps’ DOUBLE

vars: ‘var’ '{' varDecl* '}'

streams: ‘stream’ '{' stream+ '}'

varDecl: ‘const’? basic ID ( ‘=’ expr )? ';

stream: ID '{' 'type' basic ';'; ('interpolator' interp ';')? '}'

func: type ID '(': params? ')' block

block: '{' stmt* ('return' expr ';')? '}'

params: basic ID (',', basic ID)*

anchor: ‘time’? ‘+’? doubleValue ('+' ID)? block

stmt: ID ('=='|'+='|'--'=') expr ';
    | varDecl
    | ID ('++'|'--') ';
    | 'if' expr block ('else if' expr block)* ('else' block)?
    | ID (' args? ') ';
    | 'while' '(' expr ')’ block
    | 'for' '(' ID '=' expr ':=' expr ':' expr ')' block
expr: ID (' args? ') |
  | expr == expr |
  | expr != expr |
  | expr <= expr |
  | expr >= expr |
  | expr > expr |
  | expr < expr |
  | expr && expr |
  | expr || expr |
  | expr * expr |
  | expr / expr |
  | expr + expr |
  | expr - expr |
  | expr \% expr |
  | expr ^ expr |
  | '-' expr |
  | 'sin' expr |
  | 'cos' expr |
  | 'tan' expr |
  | 'sqrt' expr |
  | 'abs' expr |
  | '(' expr ')' |
  | INT |
  | DOUBLE |
  | point |
  | bool |
  | ID

args: expr (',', expr)*

interp: 'Linear' | 'TCB'

type: basic | 'void'

basic: 'int' | 'double' | 'point' | 'bool'

point: '(' doubleValue ',', doubleValue ',', doubleValue ')' 

doubleValue : ID | DOUBLE

bool: 'True' | 'False'