Developing a domain specific style sheet language for building haptic soundscape maps

Keith Albin

December 7, 2007

Presented to the Department of
Computer and Information Science
at the University of Oregon
in partial fulfillment of the requirements
for the Departmental Honors Program.

Abstract

The purpose of the project described in this thesis is to design and implement extensions and improvements to the usability of the style sheet component of the Haptic Soundscape map system, part of the ongoing University of Oregon Haptic Soundscape Map project. The immediate goal of the overall project is to develop an electronic map for use by students with visual impairments. A longer-range goal is to develop a map building tool that can be used to study the user interface issues involved in the production of electronic maps for visually-impaired users. The style sheet component allows a non-programmer to configure a variety of map attributes, accelerating map development and rapid construction of variations for experimental evolution.

I would like to thank my advisor, Professor Michal Young, for his guidance and assistance throughout this project and the production of this thesis. I would further like to express my appreciation to Professors Lobben and Young for the opportunity to be a part of the University of Oregon Haptic Soundscape map project.

Contents

Introduction

The purpose of the project described in this thesis is to design and implement extensions and improvements to the usability of the style sheet component of the Haptic Soundscape map system, part of the ongoing University of Oregon Haptic Soundscape Map project. The immediate goal of the overall project is to develop an electronic map for use by students with visual impairments. A longer-range goal is to develop a map building tool that can be used to study the user interface issues involved in the production of electronic maps for visually-impaired users. The style sheet component allows a non-programmer to configure a variety of map attributes, accelerating map development and rapid construction of variations for experimental evolution.

The initial development of the haptic soundscape map was done by an inter-disciplinary team composed of undergraduate and graduate computer science students and graduate geography students during Spring term of 2007 at the University of Oregon. The studio class was led by Professor Michal Young of the Computer and Information Science Department and Professor Amy Lobben of the Geography department. Professor Lobben and her students were all specialists in the field of cartography. This inter-disciplinary group developed a successful map

prototype that was later reviewed by students and adults at the Oregon School for the Blind.

The concept of a haptic soundscape map is that the user of the map receives haptic feedback (vibrations) indicating different surfaces or objects from the mouse he is using to navigate the on-screen map. Additionally, the visual features of the map are enhanced by sounds, both iconic and descriptive. For example, there is a specific haptic and sound effect for entering a building. The user is also able to click the mouse button while over the building to hear its name pronounced.

The following objectives were achieved by the initial prototype:

- The map was capable of being generated directly from campus GIS (Geographic Information System) data. No graphical editing of map information in an illustration or photo editing program was necessary.
- The map utilized Adobe Flash as the medium for displaying the map from a web server and enabling the special effects through a web browser.
- The map utilized both speech and non-speech sounds. The non-speech sounds were fixed (a sound indicating that the mouse pointer is over a specific object type) and parametric (sounds that indicate general directional position on the map).
- The map provided textural feedback through a haptic-enabled mouse, indicating surfaces on the map.

- The map effects could be customized though the use of a style sheet. The style sheet allowed experimenters to bind settings for the visual, haptic, and sound features used in the map to objects on the map.

This last item, the haptic soundscape map style sheet, is the focus of the project described in this thesis. The purpose of this project is to link the style sheet interpretation more closely to the map generation and to increase the usability of the style sheet, making the map-building software easier for experimenters without a computer science background to configure.

Background

My experience with the University of Oregon Haptic Soundscape Map project began in the Spring term of 2007. Along with a number of other undergraduate and graduate computer science students, I had enrolled in Professor Michal Young's Advanced Software Methodology class, conceived as an experience in cross-disciplinary studio learning. We collaborated with Professor Amy Lobben and a group of her graduate geography students to design and create an initial version of a haptic soundscape map of the University of Oregon campus.

The experience of working closely with other students with different educational backgrounds was both challenging and rewarding. We met regularly as a large group to refine design goals and worked in groups of 3-5 persons to build sections of the project. Some of the computer science students had studied user interface design for those with disabilities, but none had previous experience with cartography. The large meetings became intense learning experiences as we learned about map design for sighted persons as well as those with visual impairments. I like to think that the geography students also learned a bit about the software development process.

By the end of the term, we had created an electronic map of the University campus that utilized a haptic mouse for tactile feedback representing various surfaces and sound generation for navigational cues, object identification, building names, and user assistance. The development of this project required us to learn how to utilize data from the University GIS database and convert it into a format usable for the map; how to install, configure, and utilize an interface to the haptic mouse device; how to record and synthesize sounds for use in the map; how to use Adobe Flash software and compose instructions using the ActionScript language to build the map; how to design a style sheet language interpreter capable of defining the interaction of elements in the map; and how to integrate these components into the finished product.

2.1 Maps for Those with Visual Impairments

Maps for the sighted user operate at two levels – as a product of pleasing visual design and as a way for the user to incorporate the map information into his internal sense of spatial orientation. While research has been done to quantify the usability of maps for the visual user [Golledge and Stimson, 1997], little work has been done to study how maps can satisfy the needs of those with vision impairments.

The need for map information by blind and vision-impaired people is probably even greater than for sighted people, since they are unable to use signs and visual landmarks to navigate an area. To fulfill this need, two different mapping strategies are currently being investigated by cartographers.

The first of these is a tactile map, where the visual representation of objects on the map are replaced by tactile surfaces. The visually impaired user can then explore the map using his fingertips, much as is done with braille text. However, the amount of information that can be conveyed in this manner is constrained by the limited sensitivity of the nerves endings in human fingertips.

The second approach is to employ computer technology to create sound and haptic soundscape maps. This is the approach undertaken by this project. Research into haptic soundscape mapping is currently being conducted at a few other locations in addition to the current project at the University of Oregon. However, this is still new territory for researchers and there is a great deal of work yet to be done to evaluate the effectiveness of various haptic and sound strategies with the end users. This process is additionally hampered by the difficulty of developing and editing these software projects during the testing phases of the research. The goal of the ongoing project at the University of Oregon is to automate the map building process in order to enable experimenters to easily modify special effects without resorting to elaborate computer programming skills.

2.2 The Style Sheet as a Domain Specific Language

Viewing the Haptic Soundscape map as a tool for researchers, the style sheet becomes an important component to the finished project. The style sheet functions as the principal user interface to the map-building software and allows researchers to easily edit the sound and haptic effects used in the map. This enables a rapid, iterative design process for working interactively with users with vision disabilities.

The history of style sheets dates back to the early years of computer science. Some of the most well-known examples are FOSIs (Formatting Output Specification Instances), developed by the Department of Defense for standardizing documents; DSSSL (Document Style Semantics and Specification Language), a scheme-like presentation language; CSS (Cascading Style Sheets), designed for HTML presentation; and XSL (Extensible Style Language), a standard for display and translation of XML (Extensible Markup Language) documents [Walsh and Muellner, 1999].

More properly, though, style sheets are members of the classification referred to as domain specific languages (DSL). A DSL is defined as "a small, usually declarative, language that offers expressive power focused on a particular problem domain." [van Deursen et al., 2000]. Duersen, Klint, and Visser characterize the difference between a DSL and a general purpose programming language as being the difference between approaches that are *generic* and those that are *specific*. The specific approach used in DSLs offers a better solution for a smaller set of problems than the generic approach as used in a general purpose programming language.

One way to compare different domain specific languages is in terms of their executability. [Mernik et al., 2005]. Some DSLs, such as a configuration file for an application, may be only declarative – a set of keys and values to record application settings. Others, such as the macro language of a spreadsheet, approach the broad domain of a general purpose programming language.

The Haptic Soundscape map style sheet lies somewhere between these extremes. Its main function is to generate map configuration attributes, but certain programming language features, such as variable definition, string concatenation, and macro

execution have been added to increase its functionality.

These extensions were possible to implement with relative ease because the tools chosen to implement the interpreter are based on the widely-used general purpose compiler construction tools, lex and yacc [Provins, 1998]. Actually, these types of compiler construction tools are themselves examples of the use of domain specific languages. The lexical properties of the style sheet are defined as regular expressions, a DSL, and the syntactic properties are defined in a form of BNF grammar, another DSL. The compiler construction tools use these definitions to generate the desired interpreter, which is then able to read and interpret the style sheet language.

An additional consideration in the design of a DSL is the effort that will be required from the user to learn the language [Mernik et al., 2005]. To make this problem more manageable, it is advisable to base the new language on an existing language that users may already know. For that reason, much of the syntax of our style sheet language is based on CSS, the style sheet language used to style HTML pages on the World Wide Web.

The domain specific language is, in its way, an example of the computer science principle of abstraction. When programmers construct software, they use a general programming language, created to support many types of projects. This generality is good, but to avoid getting lost in the coding details, the programmer must create abstractions of lower level logic in order to express the higher level "business" logic of the finished software. In this manner of thinking, the DSL is the "ultimate abstraction" [Hudak, 1996].

Guy Steele, a computer scientist well known for his work in language design has been quoted as saying:

> A good programmer in these times does not just write programs. [...] a good programmer does language design, though not from scratch, but building on the frame of a base language [Steele, 1998].

When we, as programmers, create a new software project, we are creating a new language and by using a domain specific language, we are able to extract these new interfaces and conventions and make them more easily available to the user [Freeman and Pryce, 2006].

Approach

3.1 Haptic Soundscape Map-building architecture

The goal of the original haptic soundscape map development project was to automate the map-building process as much as possible. This would free experimenters from the necessity of manually creating map components and special effects when necessary for conducting research into the usefulness of different techniques.

3.1.1 Target Platform

The map is served from a standard web server as an Adobe Flash object component in a Hypertext Markup Language (HTML) web page. There is one Flash swf file representing the map controls and a set of additional Flash files, one for each layer of the map. Each layer corresponds to one of the object class names listed in Appendix B. The data for each layer file is generated automatically by the map-building software and saved in Flash ActionScript format. Each ActionScript layer file is then opened in the Flash program and manually saved in Flash swf format.

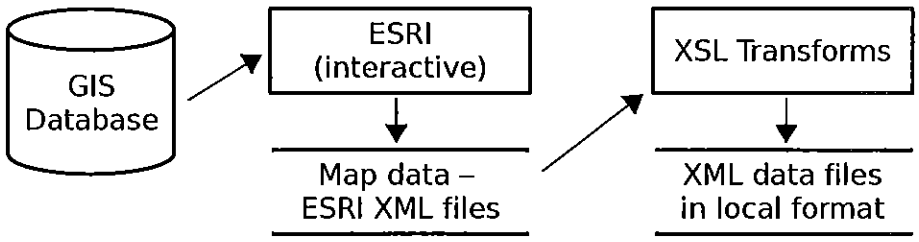


Figure 3.1: Conversion of ESRI GIS XML data files to Haptic Soundscape XML format.

Also present on the web server are the sound and haptic effect files, which are accessed by their URL (Uniform Resource Locator) addresses.

3.1.2 Map Source Data

The object data for the map originates from the University of Oregon GIS database. Using the ESRI[1] interface software, the data for constructing maps can be exported in an ESRI XML format. This is a manual operation, but is only necessary when existing map outline information has become outdated.

3.1.3 Map Data Conversion

As a first step in the map-building process, the raw ESRI XML object outline files are processed into a simplified XML format by an XSL transformation. Figure 3.1 illustrates this process. The purpose of the transformation is to identify the layers

[1]ESRI GIS and Mapping Software. Available from: (http://www.esri.com)

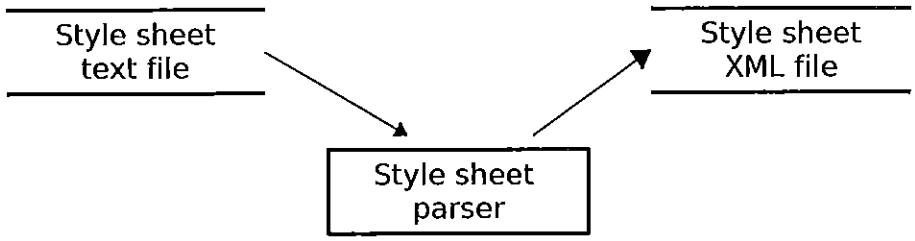


Figure 3.2: In the initial project design, the style sheet is converted to XML format by the style sheet parser and saved to an XML file. For revised design, see figure 4.4 on page 35.

by class name and, if appropriate, the individual objects by object id. Each object in the XML file is described by a series of points and line or arc commands. In the transformation, the point values used are rounded and translated to smaller values relative to the extent of the map area used in the project.

3.1.4 Style Sheet Data

The other data source used to produce the Haptic Soundscape map is the style sheet. Figure 3.2 illustrates how the style sheet text file is processed. The style sheet contains class name and object identifiers and links user actions with specific resources or settings. The style sheet also provides drawing information like color for specific objects or layers. In the original version of the map-building software, the style sheet was parsed and saved as an XML file.

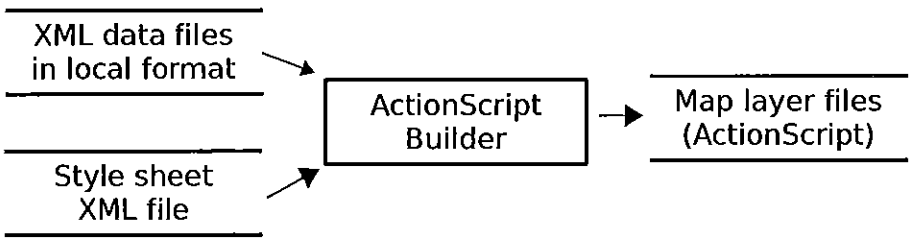


Figure 3.3: Style sheet and map data are combined to generate ActionScript files.

3.1.5 ActionScript Builder

This component of the map-building software contains several classes written in Java language which combine to build the destination ActionScript files from XML object outline files and style sheet XML. Figure 3.3 illustrates the data path for this operation. The main file is GisXmlReader, which first reads the XML object data files and saves the outline information in a list of GeoShape objects. Next, it calls the StyleSheetReader class to read the style sheet XML and apply the style attributes to the appropriate GeoShape objects. Finally, it passes the list of GeoShape objects to the ActionScriptBuilder class to generate the complete ActionScript files, one for each map object layer.

3.2 The Style Sheet

From the initial planning stages of the project, the style sheet was considered an important component of the map-building software project. The style sheet con-

tains many of the variables that researchers would wish to change and is designed to be easily accessible and editable by non-programmers. The style sheet is essentially the user interface for the haptic soundscape map researchers.

The motivation for the creation and use of a style sheet in general is to separate text and data from the information related to its display or publication. Currently, probably the most widely used style sheet language is the Cascading Style Sheet (CSS) used to control the display of web pages. Rather than embedding attributes such as type size and color in the HTML document, that information is located in the CSS file. This provides for better style consistency throughout documents and easier changes to these style attributes.

Initial Version

The style sheet parser and lexer are implemented in the Java programming language using the JFlex[2] and Java CUP[3] tools. To make learning easier, the style sheet follows many of the conventions used by Cascading Style Sheets [Meyer, 2004] for HTML. Using Java tools would make the style sheet parser easier to integrate with the other Java classes in the map-building project.

The style sheet supported:

- class selectors with class name, optional object id, and optional modifier.
- style attributes blocks with key and value pairs.
- variable definitions saved in a hash map table and usable later in the style sheet.

[2]JFlex - The Fast Scanner Generator for Java. Available from: (http://www.jflex.de)
[3]CUP LALR Parser for Java. Available from: (http://www2.cs.tum.edu/projects/cup)

- very basic error messages.
- single line comments using "//" to precede the comment.
- extra returns and white space in definitions were ignored.

Upon interpretation of the style sheet, variables were replaced with their definitions. Then style attributes were stored in a hash map of class names to StyleClass objects. Each StyleClass object contains a hash map of object IDs to another hash map of style attribute keys and values.

When interpretation of the style sheet was complete, the data structure was "dumped" to print its contents in XML format, which was saved to a file to be used in the ActionScript generation.

3.3 Plans for Style Sheet Improvements and Extensions

3.3.1 Integration

An immediate improvement in usability would be to integrate the style sheet interpretation directly with the process of generating the ActionScript layer files. At the initial phase of the project, this was a two step process – first parsing the style sheet and saving it to an XML format and then parsing the resulting XML file to generate the ActionScript layer files. This process is illustrated in figures 3.2 on page 16 and 3.3 on page 17. While the XML object outline layer files from GIS are

unlikely to change often, the style sheet is subject to frequent change and a more direct path for integration into the building process would be desirable.

There were also issues of efficiency to be overcome in the StyleSheetReader class. Upon parsing the style sheet XML output, the class saved it into a DOM tree and then traversed the tree searching for matching attributes for each object encountered in the list of GeoShape objects. It would be much more efficient if the StyleSheetReader were to access the style attribute data from the tables where the parser stores that information during the processing of the original style sheet.

3.3.2 Style Sheet Language Extensions

String Concatenation

While the style sheet did support variable definitions, these were not used well. The style sheet contained many lines like the following:

```
//Deady
building#005{
    on_click_sound:"http://www.cs.uoregon.edu/map/mp3s/005.mp3";}

//Willamette
building#046{
    on_click_sound:"http://www.cs.uoregon.edu/map/mp3s/046.mp3";}
```

This is in stark violation of what Hunt and Thomas [Hunt and Thomas, 1999] call the DRY principle of programming – "Don't Repeat Yourself." A resource name like the directory where sound files are stored is repeated many times in the style sheet which can lead to errors that are difficult to find. When the resource

location is changed – a likely occurrence – all the style attribute values need to be changed.

By implementing string concatenation, we should be able to define such resources once in a configuration variable and use that to construct the resource URLs.

Color Names

When variable definitions were used in the original style sheet, they did not contribute to the readability of the style sheet. For example:

```
building_color = "#00FF00";

...

building{
    color: building_color;
}
```

It would be more intuitive if the color name of "Lime" was used in the color definition instead of "building_color." A set of named colors would make color definition easier for those who might not understand the hexadecimal red-green-blue color model.

Variable Substitution Macros

In the case where there are many style attribute definitions that vary only by the name of the particular object, the style sheet could be compacted and made easier

to manage by a system of defining substitution macros. For example, when concatenation is in place, building name sound resources become nearly repetitious definitions:

```
//Deady
building#005{
    on_click_sound: buildingSoundPath + "005.mp3";
}

//Willamette
building#046{
    on_click_sound: buildingSoundPath + "046.mp3";
}
```

Using a system of variable substitution, all such definitions could be condensed into one and expressed as:

```
building [%n]{
    on_click_sound: buildingSoundPath + "[%n].mp3";}
```

Additional Style Sheet Imports

It would be useful to allow one style sheet to import another. This import should be processed as if it were an "@import" statement in CSS or an "#include" statement in the C/C++ programming language. The effect should be that the statements and definitions in the imported style sheet are interpreted in the place where the import occurs.

Import file statements should be able to use a file path relative to the style sheet from which it is imported or an absolute path if that is desired.

Default Class Definitions

There should be a "default" class definition that would apply attributes to all classes. Also, class name and object id should each be optional parameters in defining a style block. A user should be able to define a class name with no id, meaning an attribute that applies to every member of a class; or an id with no class name, meaning an attribute that is applied to every matching id, regardless of the class name.

Attribute Key Format

In keeping with the format used in HTML Cascading Style sheet, the key of a style attribute should be defined as "on-enter-sound", not "on_enter_sound," using underscore characters instead of the hyphens. For compatibility, the style sheet language could accept either underscores or hyphens.

Multi-line Comments

Multi-line comments should be allowed using the CSS convention of "/*" to open the comment block and "*/" to close the comment block.

Color Definitions

In the initial style sheet implementation, only object color could be defined and it was applied to both fill and outline. It would be desirable to have separate

definitions for fill-color, line-color, and line-width. The "color" definition would become the default if others are not defined.

Error Messages

There are three types of errors that might be generated on interpretation of a style sheet – lexical errors, syntactical errors, and undefined variable errors. In the first version, these errors were causing the parser to terminate and issue stack dump messages, but very little useful information about the cause of the error. Error messages should contain the at minimum, the line where the error occurred. Undefined variable definition errors should print the name of the undefined variable. When multiple style sheets are allowed, the file name where the error originated should be printed. All errors should be caught and should allow the programs to terminate gracefully without printing stack dumps.

Allowing file import will also create a possible error for style sheet files that cannot be found or read.

Result

4.1 Introduction – Software Tools Used

Since most of the haptic soundscape map project was developed using the Java language, Java compiler tools were selected to construct the style sheet interpreter. The tools selected are called JFlex and Java CUP. These are compiler construction tools based on the traditional Unix LEX and YACC tools [Brown et al., 1995]. To use these tools, a developer defines the lexical, syntactical, and symantic rules of the language and then the software uses this description to write the programs that become the compiler or interpreter. To interpret the language used in the style sheets, two distinct tasks must take place. First, style sheet input is broken into predefined units, called tokens. Then a relationship is established between these tokens to determine their meaning.

The first step of this process is called the lexical analysis and is done by the Java class file generated by JFlex. The lexical units of the language, which may be anything from individual punctuation characters to longer patterns are described using a special language called regular expressions. A regular expression is a pow-

erful and flexible way to describe a block of text. JFlex supports a large set of regular expression operators [Friedl, 1997]. JFlex will search for a match between input characters and the regular expressions defined for the lexical structure of the language. The lexer will locate the longest match and then execute the code associated with that expression. In most cases, that block of text and other relevant information will become part of a token object that is passed on to the parser unit. In other cases, as in the case of a character that can not be used for any match, the associated action may be to issue an error message or to take other actions.

The second step of the process of interpreting the style sheet language uses a Java class generated by the Java CUP (Constructor of Useful Parsers) software. Java CUP is a system for generating a LALR parser based on the grammar of the target language, in this case the map-building style sheet [Aho et al., 1986]. The language description used by Java CUP resembles the BNF (Backus-Naur form) grammar used to define the style sheet language in Appendix A of this paper [Pagan, 1981, Louden, 2003]. However, each grammar definition also includes Java language code indicating actions to be performed upon assembling a complete syntatic phrase from the incoming tokens.

In the initial version of the Haptic Soundscape map, the parser interpreted the style sheet, wrote that information to a file in XML (Extensible Markup Language) format, and then exited. This XML style file was then read by software designed to generate the map layer files.

4.2 String Concatenation

After the initial development of the Haptic Soundscape map, the style sheet did incorporate variable definition, but this capability was not yet being applied to improving the usability of the style sheet. Values could be assigned to identifiers as in the following example:

```
building_color = "#00FF00";
bicycle_color = "#990000";
callbox_color = "#FF0000";
```

The style sheet uses the notation of the equal sign (=) for variable assignment and the colon (:) for style attribute assignment to differentiate the two. Once a variable has been defined, the parser then saves the identifier and its value to a hash table for look-up later in the processing of the style sheet. However, one of the major problems in the first version of the style sheet was duplication of information. Thus, the style sheet contained hundreds of lines like the following:

```
//Deady
building#005{
    on_click_sound: "http://www.cs.uoregon.edu/map/mp3s/005.mp3";
}

//Willamette
building#046{
     on_click_sound: "http://www.cs.uoregon.edu/map/mp3s/046.mp3";
}
```

The style sheet had the ability to define the path and assign that to an identifier, but didn't yet have the ability to combine that with other variables or string constants to build longer strings. To solve this problem, the style sheet parser needed to be extended to gain the ability to concatenate string values.

Since CSS does not support string concatenation, we had to look elsewhere to find a convention for the notation. The decision was made to use the plus (+) sign to join strings as is done in the Java language. Concatenation should be valid in any right-hand side value of either type of assignment statement. Secondly, it should be possible to concatenate any number and combination of variable identifiers and string literals. To accomplish this, it was necessary to modify both the lexer and parser instructions. To the JFlex instructions, the following was added to recognize the plus sign as a valid token:

```
"+"  { return symbol(tokens.CONCAT, yytext()); }
```

This instruction tells Jflex to identify an instance of the plus sign and insert it into a symbol object with the label of "CONCAT." Then the instructions to the Java CUP parser generator needed to be modified to perform the correct action when encountering this symbol. First, a new nonterminal declaration was added:

```
non terminal  strcat;
```

Next, it was necessary to create a recursive definition for the string concatenation process to ensure that any number of string objects could be put together. The language definition already had a "value" nonterminal to define the right-hand side of either assignment type, so this was where the modification occurred.

```
value ::= str:i {: RESULT = i; :}
        | NUMBER:n  {: RESULT = n; :}
        | value:v strcat:s {: RESULT = v + s; :}
        | IDENT:i
            {:
              /* Variable look-up and error handling goes here */
```

```
            :} ;

strcat ::= CONCAT str:s {: RESULT = s; :}
         | CONCAT IDENT:i
            {:
              /* Variable look-up and error handling goes here */
            :} ;
```

Variable look-up code has been omitted for clarity. In the first definition, a value is defined to be either a quoted string constant (handled by the "str" nonterminal), a "NUMBER" terminal, an "IDENT" (variable identifier) terminal that is then located in the variable definition table, or another "value" followed by a "strcat" concatenation nonterminal. It is this recursive definition that allows concatenations of any length. The "strcat" nonterminal then defines the value to be concatenated as either a "str" (string literal) or "IDENT," a variable identifier to be retrieved from the table. With these additions, the parser could now interpret a style sheet organized as follows:

```
// default path definitions
soundPath = "http://www.cs.uoregon.edu/map/mp3s/";

//Deady
building#005{
    on_click_sound: soundPath + "005.mp3";}

//Willamette
building#046{
    on_click_sound: soundPath + "046.mp3";}
```

This enables the definition of constants in one convenient place, like the beginning of the style sheet, and makes the style sheet attributes shorter and easier to read. If the path changes for a different server location, it needs be changed in one

place only. This makes the style sheet more convenient to use and eliminates the possibility of errors in the duplication of the path information.

4.3 Variable Substitution Macros

Adding concatenation improved the style sheet format, but there was still a problem with readability. There were hundreds of (shorter) style attribute definitions for assigning building name pronunciations – each differing only in object ID and sound file name. In fact, all the sound files had been named based on the building ID. The next logical improvement in the style sheet should be to express this relationship in some sort of variable replacement language that could be expanded as needed. Then, in the interpretation of the style attributes, these substitutions could be used unless overridden by a more specific specification.

First, the decision was made to use square brackets with the percent sign (%) and a letter variable to designate the expression marker. Here is an example of that notation:

```
building #[%n] {
    on-click-sound: buildingSoundPath + "[%n].mp3";
}
```

This is interpreted to mean that the object ID marker of "#[%n]" stands for any object name in the class of "buildings." The value of that ID (after the pound sign) is to be applied as a text substitution wherever it is used in the following lines. For example, if the building ID is "#005," the file name becomes "005.mp3" and

this string is concatenated to the "buildingSoundPath" to create the style attribute value string.

To allow this expression, it was necessary to modify the definition of an object ID in the JFlex lexer instructions to include the square bracket and percent characters. At this point, the regular expressions for identifiers were also refined to clarify their function.

```
/* class and definition identifiers must start with letter */
[a-zA-Z_][-a-zA-Z_0-9]* { return symbol(tokens.IDENT, yytext()); }

/* object identifiers may be any combination of letters and numbers,
 beginning with a hash sign */
"#"[-a-zA-Z_0-9\[\]%]+ { return symbol(tokens.ID, yytext()); }

/* number, possibly negative, int or float */
[-]?([0-9]+|([0-9]+"."[0-9]*)|([0-9]*"."[0-9]+))
   { return symbol(tokens.NUMBER, yytext()); }
```

Previously, there had been one definition for an identifier, used for classes, variable identifiers, and object IDs. This had been defined as consisting of any combination of letters, numbers, and underscore characters. This was redefined into three types: an IDENT token, for class, style, and variable names; an ID token for object IDs; and a NUMBER token for numerical data. Using the regular expressions shown above, an IDENT is defined as a string starting with a character followed by any number of letters or digits. An ID is the pound sign (#) followed by any string of letters and digits, including the ones used in the new macro definition. A NUMBER is defined as having an optional negative sign followed by digits with an optional decimal point.

Now a new object ID non-terminal was created for the Java CUP portion of the style sheet interpreter.

```
obj_id ::= ID:i {: RESULT = i.substring(1, i.length()); :} ;
```

The above expression removes the pound sign from the object ID and returns the remaining string value.

Since the style sheet was still being saved as an XML file at this point in the evolution of the project, the interpretation of the macro was temporarily added to the StyleSheetReader class that processed the resulting XML file. Later, after the style sheet parser was integrated into the main map project classes, a method was added to the parser to interpret the macro language. The macro definition is basic, defined to handle its projected usage in the Haptic Soundscape map project, but it could be easily extended if a more sophisticated need were to become apparent.

4.4 Style Sheet and Map Integration

One of the main problem areas with the initial version of the Haptic Soundscape Map project was the extra manual step necessary for making changes to style attributes. The style sheet needed to first be parsed and the results saved to an XML file. Then the main GisXMLReader class had to be executed to open the XML file, read the attributes into a data structure, and extract matching style attributes for each GeoShape object, the class that represents objects on the map. Besides being inconvenient to researchers, there were obvious efficiency concerns with this process. As the style sheet was being parsed, the data was saved to a hash table

structure, which was later traversed to print the XML file. Then the XML structure was read into a DOM tree which was traversed each time an object was created for drawing the map.

To eliminate this duplication of effort, it would be desirable for the StyleSheetReader class, called by GisXMLReader, to directly call the style sheet interpreter and utilize the efficient look-up characteristics of the hash table data to add style attributes to the GeoShape objects. The first step to realizing this goal was to streamline the data storage tables in the Java CUP parser. A new class, called StyleAttribute, was created to store the three components of a style attribute – a key, a value, and a possible modifier.[4] Two hash table were created – one for regular style attributes and one for macro definitions – each with keys representing object classes and values composed of the StyleClass objects used previously. Each StyleClass object contains a hash table with object IDs as the keys and a list of StyleAttribute classes as the value associated with each.

Next, public data access methods – getAttr() and getMacro() – were created for external classes to read directly from the parser data structure. In the case of the table for macros, the macro interpretation logic was encapsulated in its access method. Each method returns a list of StyleAttribute objects to the caller. Using this strategy, it became possible to eliminate most of the code in the StyleSheetReader class. Now the class needed only to initialize the parser and call the appropriate data access methods for each map object. The order in which each of these style attributes is read determines the priority for applying style attributes to objects. A new class name of "default" was created, resulting in five levels of

[4]See Appendix A for definitions of these components.

attribute priority from least to most specific style specifications.[5]

This simplification of the StyleSheetReader class also prompted a reorganization of the functionality in the main map creating classes. In the initial version of the map, the StyleSheetReader class had been assigning styles to constants defined in the GeoShape class using a series of if-else statements. For example, a style sheet attribute key of "on_click_sound" was being matched to the GeoShape constant value of "ON_CLICK_SOUND." This is a function that should be encapsulated in the GeoShape class. To streamline this process, the GeoShape class was modified. The constants contained in the initial GeoShape class were instead implemented as keys in a Java enumeration object. Each of these enumeration constants references a matching style sheet key string. On construction of a GeoShape object, the style sheet key strings are used as keys to create a hash table for storing the attribute values. This results in a constant time look-up of style attribute values by either their constant value, used by the ActionScriptBuilder class; or by their style sheet key, used by StyleSheetReader. Then methods were created in the GeoShape class to set and read the attributes. An abbreviated portion of this enumeration is shown below:

```
public enum Func {

...

    ON_CLICK_SOUND("on-click-sound"),
    ON_CLICK_ZOOMED_IN_SOUND("on-click-sound:zoomin"),
    ON_CLICK_ZOOMED_OUT_SOUND("on-click-sound:zoomout");
...

    private String style;
```

[5]See "Precedence of Style Attribute Assignment" in Appendix B

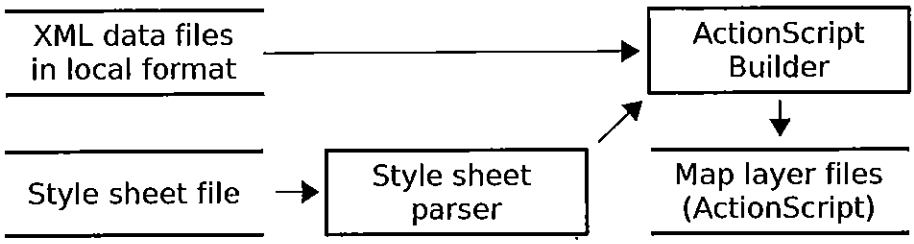


Figure 4.4: Style sheet parser integrated with map-building process for generating ActionScript files. For comparison to original design, see figures 3.2 on page 16 and 3.3 on page 17.

```
    Func(String style){
        this.style = style;
    }
}
```

After this reorganization, it was possible to read and interpret the style sheets from within the map-building process, eliminating one of the most problematic manual steps in the map creation process. This new organization of the map-building classes is shown in fugure 4.4.

4.5 Style Attribute Name Additions and Improvements

While adding new functionality to the style sheet language, several attribute naming issues became apparent. Therefore, the style sheet attribute naming conventions and their interpretation by the Haptic Soundscape map-building software

were adjusted in an attempt to solve these problems.

4.5.1 Key Names

In the initial version of the Haptic Soundscape map, style attribute key names used the underscore character as a word separator. For example:

```
on_enter_sound: classSoundPath + "911dtmf.mp3";
```

The CSS standard that we are following when possible uses the hyphen as a word separator:

```
on-enter-sound: classSoundPath + "911dtmf.mp3";
```

Since this form is closer to the established standard and probably easier for those not accustomed to using the underscore character, the style sheet was changed to use hyphens in style attribute keys. The method in the GeoShape class that provides for setting attributes was edited to accept either separator character.

4.5.2 Color Names

In the initial version of the style sheet some colors were defined, but not in a manner that made the colors easy to visualize. For example, the following were used in the style sheet:

```
parking_color = "#0000CD";
entrance_color = "#FFFF00";
```

```
...

car_parking{
    color: parking_color;
}
entrance:zoomin{
    color: entrance_color;
}
```

Even to those experienced with the use of the hexadecimal RGB color model, it is not a simple task to recognize the color for parking as medium blue or the color for entrances as yellow. Therefore colors were defined by their names, as designated in the X11 color model. This color model was chosen because it is a widely used standard and its colors are supported by most web browsers. The replacement definitions appear as follows:

```
MediumBlue = "#0000CD";
Yellow = "#FFFF00";

...

car_parking {
    color: MediumBlue;
}
entrance: zoomin {
    color: Yellow;
}
```

This makes color assignments more intuitive for anyone. The remaining problem is that any colors used must be listed at the beginning of the style sheet. This issue was later resolved by enabling the import of multiple style sheets. See section 4.7 "Style Sheet File Import" for more detail.

4.5.3 Color Definitions

In the initial version of the Haptic Soundscape map, only the "color" style attribute was allowed and this was applied to both object fill and outline. To add functionality to the style sheet and flexibility to the visual map image, the color style attributes were separated into "fill-color" and "line-color." Additionally a "line-width" attribute was added. These were implemented in the GeoShape and ActionScriptBuilder classes. The "color" style attribute was preserved and an hierarchy was established in the GeoShape class to assign color attributes. If either "fill-color" or "line-color" is not assigned for an object, the style defaults to the "color" setting. If no color attribute is defined, the object uses a color predefined as the default in the GeoShape class.

4.6 Multi-line Comments

The initial version of the style sheet language supported single line comments in the form used by the C++ and Java languages. All characters from double slash comment marker (//) to the end of the current line are considered comments and discarded. The comment marker can occur anyplace on a line.

CSS supports multi-line comments using the opening marker of a slash and star (/*) and a closing marker of the opposite (*/). Adding this type of comment would support the CSS standard for comments and also offer the a more convenient method to comment out whole blocks of code. Allowing comments to be nested would allow a block of code to be temporarily commented out, even if it

contains comments.

Comments are processed by the JFlex lexical analyzer portion of the style sheet parser. Comment characters are discarded until the comment pattern ends and then the next token is analyzed as normal. To address nested comments, it is necessary to take advantage of JFlex support for multiple lexical states.

Lexical analysis always begins in the default state of "YYINITIAL" shown below, with the first few token pattern matches as well as the starting marker for multi-line comments:

```
<YYINITIAL> {

  ":"         { return symbol(tokens.COLON, yytext()); }
  ";"         { return symbol(tokens.SEMI, yytext()); }
  "{"         { return symbol(tokens.LBRACK, yytext()); }
  "}"         { return symbol(tokens.RBRACK, yytext()); }
  "="         { return symbol(tokens.EQUAL, yytext()); }
  "+"         { return symbol(tokens.CONCAT, yytext()); }

  "/*" {  yybegin(COMMENT);
          commentCount++;
          if (commentCount == 1) {
              startingCommentLine = yyline;
              startingCommentColumn = yycolumn;
          }
       }
```

Lexical expressions are often regular expressions, but can also be simple strings as in the examples above. In the action associated with the multi-line comment beginning, several variables are set and analysis moves to the COMMENT state. The variable commentCount is used to track nested multi-line comments and the other variables record the position of the opening comment marker in case of error. Below is the COMMENT state:

```
<COMMENT> {

  "//"{SINGLE_LINE_COMMENT}*  { }

  "/*" { commentCount++; }

  "*/" { if (--commentCount == 0) yybegin(YYINITIAL); }

  {COMMENT_TEXT} { }

  {NEWLINE} { }
}
```

The upper-case constants are regular expression defined elsewhere in the lexer pattern file. SINGLE_LINE_COMMENT is defined as any character except the end-of-line (NEWLINE) character. The empty braces mean that no action is taken – the characters are discarded. While in the COMMENT state, any single line comment skips to the end of the line. Another multi-line beginning marker increments the commentCount and a closing multi-line marker decrements the same variable. When the commentCount reaches zero, we jump back to the YYINITIAL state. Finally, any NEWLINES are discarded.

COMMENT_TEXT is defined by a complex regular expression:

```
COMMENT_TEXT =
 ([^*/\n]|[^*/\n]"/"[^*/\n]|[^/\n]"*"[^/\n]|"*"[^/\n]|"/"[^*/\n])*
```

This negated regular expression discards all characters encountered, stopping only at instances of "/*", "*/", "//", or a newline. When any of these character combinations are found, the pattern matching returns to the beginning of the COMMENT state. Processing multi-line comments one line at a time avoids the possibility of overflowing the buffer with an extremely large comment block.

These code blocks enable multi-line comments, possibly containing other single- or multi-line comments. This feature also opens up new opportunities for difficult-to-diagnose style sheet errors. To help pinpoint these problems, special error handing for comments blocks was implemented. This is described in section 4.8 "Error Handling."

4.7 Style Sheet File Import

The final style sheet enhancement specified for this project was the ability of one style sheet to import another. This would, for example, allow all the 140 color definitions in the X11 color model to be made available to the user without cluttering the main style sheet with the definitions.

Although we refer to this action as file *import*, it should not be confused with the import statement in Java and other object-oriented languages. The action resembles the "@import" statement in CSS or the "#include" statement in C/C++, where the included file is, in effect, inserted at the location of the command.

For syntax, we chose to again follow the CSS precedent. Differing from CSS, the style sheet import statement does not need to appear as the first item in the style sheet. A style sheet import statement appears as follows:

```
@import "colors.ss";
```

The entire file import process is handled from within the lexical analyzer. The first step in enabling file import is to switch the skeleton file used in compiling the lexical analyzer from "skeleton-default" to "skeleton-nested." This skeleton

file provides for maintaining a stack of file buffer objects, popping each after it has been read, and continuing analysis on the next buffer in the stack. The new skeleton file is specified by changing the JFlex compilation command in the parser make file.

```
PROJHOME = ../../../trunk
JFLEX = $(PROJHOME)/tools/jflex/bin/jflex
...
$(JFLEX) stylesheet.lex -skel $(PROJHOME)/tools/jflex/src/skeleton.nested
```

The first two lines define the relative addresses of the project root and the jflex executable file. During the compilation phase, JFlex reads the lexical instructions in the "stylesheet.lex" file and writes a Java file to be compiled to form the finished style sheet lexical analyzer. Next the file import statement was added to the lexical definitions file as a complete lexical token:

```
"@import"[ \t\r\n]+"\""{FILENAME}"\""[ \t\r\n]*";"
{
    String fname = getFileName(yytext());

    FileReader stream = getFileStream(fname);
    if (stream != null) {
        yypushStream(stream);
    }
}
```

This regular expression defines the import statement as being the literal string "@import", whitespace, the file name surrounded by double quotes, possibly additional whitespace, and a closing semi-colon. Since the lexical analyzer always searches for the longest possible token match, it will accept the entire import line and not attempt to separate it into either string or semi-colon tokens.

The FILENAME regular expression match is defined elsewhere as any combination of upper and lowercase letters, numbers, hyphens, underscores, or slashes. While many operating systems might allow additional characters or spaces in file names, the file name specification for style sheets has been kept relatively simple. The file name can be either an absolute or relative address. The new file name is first passed to the getFileName function, which checks for file existence and returns a modified path to the file (more on this later). Then a new FileReader object is created and pushed onto the stack of file streams using a method from the "skeleton-nested" code skeleton file. JFlex automatically resets the file line and column counters used for error messages and preserves this information by using special file stream objects on the stack.

Upon impoting a second file, the lexical analyzer switches its context to reading the new file. When the end of the new file stream is reached, it is necessary to handle the end-of-file character in a different manner to return to the previous style sheet context at the point where the new file had been imported. To facilitate this, JFlex provides a special end-of-file state. When this state is defined, JFlex will switch to the EOF state instead of returning an end-of-file token to the parser. Here is the initial code for the EOF state:

```
<<EOF>>   { if (yymoreStreams()) yypopStream();
            else return EOF;
          }
```

When reaching the end of the imported file, JFlex calls the pre-defined function yymoreStreams() to determine if there are more file stream objects on the stack. If so, it pops the top stream and begins processing the next. If the stack has no other

stream objects, the EOF token is returned to the parser and style sheet processing is done.

The issue encountered at this point is the addressing of the imported style sheets. The use of relative file addressing is desirable because it allows for easy relocation of project files. However, at this stage of file import development, all addresses are relative to the parser itself. This is counter-intuitive; it would be much clearer for the user if the import file address were relative to the file into which it is being imported. This required several changes to the lexical analyzer code.

In its default configuration, the lexical analyzer is constructed with a FileReader object passed to it as the initial input. Therefore, the lexical analyzer has no way of knowing the actual address or name of the main style sheet file. To solve this problem, an alternate constructor was created for the lexical analyzer. This constructor allows for a file name to be passed to the scanner instead of an opened file stream.

```
scanner(String fileName) throws FileNotFoundException {
    this(new FileReader(fileName));
    fileNames.push(new File(fileName).getPath());
}
```

Next, a second stack was created to store the file names. This would also prove to be useful for error messages.

```
private Stack<String> fileNames = new Stack<String>();
```

Whenever a new file stream object is pushed or popped from its stack, an equivalent action is preformed to the file name stack. Having access to the current and

any previous file names, it now becomes possible to load an imported style sheet based on an address relative to the path of the style sheet from which it is imported. The Java code to handle that file address transformation in a platform independent way is shown below:

```
/* create File object from new file name */
File newImport = new File(filename);

    if (newImport.isAbsolute()) {
        newImportName = newImport.getPath();
    }
    else {   /* relative path  */
        File curFile = new File(fileNames.peek());
        newImport = new File(curFile.getParent(), newImport.getPath());
        try {
            newImportName = newImport.getCanonicalPath();
            }
        catch (IOException e) {
            System.err.println("Unable to locate import file: " +
                               newImportName +
                               "\n(No such file or directory)");
        System.exit(1);
    }
```

If the new file address is absolute, no path resolution need be done. If the new path is relative, the parent path of the current file is combined with the relative path. The method getCanonicalPath() combines the parent and relative path, eliminates unnecessary path directives like "../", and confirms the existence of the resulting absolute file path.

Maintaining parallel data structures for the file name and file stream objects is not optimal from a software architectural perspective, but was used in this case to keep modifications in the application code area. Modifying the skeleton library file might cause problems if the JFlex library is updated with a new version.

With these modifications, style sheet imports can now be handled in a transparent manner. Additional error handling considerations are discussed in the following section.

4.8 Error Handling

P.J. Brown of the University of Kent describes the implications of developing good error messages:

> One of the most important yet most neglected aspects of the human/machine interface is the quality of the error messages produced by the machine when the human makes a mistake. [...] The user learns whether his system is a friend or a foe when he makes errors. A friendly system should give an informative message and provide help in correcting the error [Brown, 1983].

Brown would probably not consider the initial version of the Haptic Soundscape map style sheet a friend of the user. The error messages provided little information about the type of error or its location. Furthermore, features added in this version create new circumstances and require additional error messages to help users locate a problem.

4.8.1 Interpreter Errors

The basic style sheet interpreter errors can be classified as being of three types: lexical, syntactical, and semantic. An example of a lexical error is a character in the style sheet that doesn't match to any of the token specifications. A syntactical error could be something like a malformed statement block or an improperly terminated line. An example of a semantic error would be the use of an undefined variable.

Lexical errors are caught by the lexical analyzer. If no match can be made to any of the token definitions, a default error handler catches the first illegal character and calls the fatal error method.

```
[^] { reportFatalError("Illegal character \"" + yytext() + "\""); }
```

This match calls the fatal error method and passes it a message that includes the type of error and the text that triggered the error. The error method combines this with the line and column position where the error was located and prints this information for the user:

```
private void reportFatalError(String msg) {
    reportFatalErrorPos(msg, yyline, yycolumn);
}

private void reportFatalErrorPos(String msg, int line, int column) {

    System.err.println("Fatal error: " + msg );
    System.err.println("in line " + (line + 1) + ", column " +
                       (column + 1) + ".");
    System.exit(1);
}
```

These overloaded methods provide for calling the error message with the current line and column positions or for specifying other saved locations. The later form is used for certain comment block errors (described in next section).

The current line and column positions are saved into the token object passed to the parser component, so syntactic error handling in the parser is very similar to the example above. The major difference is that syntactic errors are caught by the Java CUP runtime code, so the default error handler is overridden to halt the program and print the error message with line and column location information.

Semantic errors, like the use of an undefined variable must be caught by the parser code:

```
if (var_table.containsKey(i)) {
  RESULT = var_table.get(i);
}
else {
  parser.report_fatal_error("Undefined variable in stylesheet: " + i,
  (java_cup.runtime.Symbol)CUP$parser$stack.elementAt(CUP$parser$top));
}
```

This example illustrates the action necessary to issue a undefined variable error. If the variable name is not found in the look-up table, the token symbol object must be retrieved from the runtime code to call the error handling method with proper line and column location information. That object on the top of the parser stack is the second parameter passed to the report_fatal_error method.

4.8.2 Comment Block Errors

Extending the comment syntax to include multi-line, nested comments increases the usability of the style sheet, but it also increases the number and complexity of possible errors. For example, it is necessary to define the meaning of the following comment block:

```
/**
* ------------------------------------------------
*  Comment block may span multiple lines
*  and contain other embedded comments.
*           // single line comment */
*  /* another embedded comment
*  */
* ------------------------------------------------
*/
```

In line 5 of the example, the closing comment marker (*/) is correctly ignored because it is part of a single line comment. The single line comment marker (//) interprets everything from that point to the end of the line as comment text, even if it includes a multi-line comment marker. However, this makes structural errors easy to commit and, unless special checking is enabled, difficult to diagnose. For example, suppose a user marks his comment block with slashes:

```
/*
// ------------------------------------------------
//  Comment block may span multiple lines
//  and contain other embedded comments.
    /*
    embedded comment
    */
// ------------------------------------------------
// */
```

This is an error because the lexical analyzer will not recognize the final closing multi-line marker – part of the single line comment that begins that line 9 (//). Without enhanced error checking, the lexical analyzer will discard all the text from this point forward as part of a comment block without generating any errors. Therefore, a check was added to the EOF block of the lexer. If the count of comment markers is not 0 at the end of processing a file, an error should be reported to indicate the problem. However, this error will be of little use in locating the point of the problem, because the line and column numbers will point to the last characters in the file.

When processing nested comments, it is impossible to unambiguously match the starting markers with ending markers as intended by the user, but it is possible to save the position of the first opening marker. This is more likely to point out the approximate point of the error. First, the position of the first opening comment marker in each block is saved when encountered:

```
"/*" {  yybegin(COMMENT);
        commentCount++;
        if (commentCount == 1) {
            startingCommentLine = yyline;
            startingCommentColumn = yycolumn;
        }
     }
```

Then, inside of the EOF block, the comment marker count is checked when exiting a style sheet file:

```
<<EOF>>  { if (commentCount > 0) {
               reportFatalErrorPos(
                    "Unmatched comment markers (/* */). " +
    "Opening marker ", startingCommentLine,
```

```
            startingCommentColumn);
        }
        if (!fileNames.empty()) fileNames.pop();
        if (yymoreStreams()) yypopStream();
        else return EOF;
    }
```

With properly formed comment blocks, the commentCount should always be 0 at end-of-file. Anything else can safely be assumed to be an error. The fatal error method is called with the position of the opening marker in the most recent blodk of multi-line comments.

Additionally, a pattern match was added to catch an unmatched ending comment marker:

```
"*/"   { reportFatalError("Unmatched closing comment marker (*/)"); }
```

This condition would have generated an illegal character error without the additional check, but this error message describes the problem more accurately.

4.8.3 Special File Import Error Handling

All errors now print the type of the error, the actual token causing the error, and relatively accurate line and column positioning for each error. However, with multiple input style sheets, we still need to know in which file the error occurred. Therefore, all error messages need to include the name of current style sheet when an error is reported. In the lexical analyzer, this information is easily available by "peeking" at the top of the style sheet name stack. Here is an example of the fatal error message with the new information added:

```
private void reportFatalErrorPos(String msg, int line, int column) {
    System.err.println("Fatal error: " + msg );
    System.err.println("in line " + (line + 1) + ", column " +
                       (column + 1) + ".");

    String fname = "style sheet file";
    if (!fileNames.empty()) {
          fname = fileNames.peek();
    }
    System.err.println("Error in file " + fname);
    System.exit(1);
}
```

Transferring this new information to the parser, however, poses a different problem. The default Java CUP Symbol object used for encapsulating the token data does not include a field for file name. Therefore, it was necessary to extend the class as shown below:

```
public class StyleSymbol extends java_cup.runtime.Symbol {

    public String fileName = "";

    StyleSymbol(int type, int yyline,
                int yycolumn, String file) {
        super(type, yyline, yycolumn);
        fileName = file;
    }

    StyleSymbol(int type, int yyline, int yycolumn,
                Object value, String file) {
        super(type, yyline, yycolumn, value);
        fileName = file;
    }
}
```

The two class constructors shown correspond to the default constructors of the Java Cup Symbol class, but add the additional string file name field to the class.

Since the new class extends java_cup.runtime.Symbol, the lexical analyzer can replace the default class with the StyleSymbol class when passing token information to the parser. The parser error handling method checks that the proper type of object has been passed to it and extracts the file name for use in the error reporting message. That portion of the parser error handler is shown below:

```
if (info instanceof StyleSymbol) {
    System.err.println("Error in file " +
                       ((StyleSymbol)info).fileName);
}
```

The error messages generated by these techniques provide more accurate information to the user about the cause and location of possible errors.

Conclusion

I have appreciated this opportunity to make my contributions to the University of Oregon Haptic Soundscape map project. Research into making computer technology more accessible and useful to those with disabilities is an area where there is much work yet to be done.

At the conclusion of my work extending the map-building style sheet, a number of usability and efficiency milestones have been met. At the same time, new goals have come into view. I'll discuss some of those new ideas in "Future Directions" on page 58.

5.1 Style Sheet Improvements

The integration of the style sheet interpreter into the main map-building software has resulted in greatly increased efficiency of operations for both the user and the computer hardware.

Previously, it was necessary to interpret the style sheet and store the attribute settings into one data structure. This data structure was traversed to print an XML

style sheet output file. Next, in a separate manual operation, the map-building software was initiated in order to read the style sheet XML file and built a DOM tree from its contents. This data structure was traversed to search for matching attributes for each object encountered in the map-building process. Finally, each matching attribute was applied to the map object by traversing all possible attributes of the object to find the correct match.

With the project enhancements in place, the style sheet interpreter is initiated by the map-building software. The style sheet is interpreted and and its attributes are stored in constant time into a single data structure. The map-building software accesses matching attributes for each object directly from that data structure, again in constant time. These attributes are assigned to the map object, using a constant time table in the map object.

The extensions to the style sheet language have greatly increased the readability of the style sheet itself. After adding new attributes, the ability to call the parser directly, more efficient handling of attributes, and five levels of priorities for applying attributes, the size of the map-building classes have increased by less than three percent, due to code optimizations. Extended functionality, command set, and error checking has increased the size of the parser code by about fifty percent.

The most dramatic change brought about by this phase of the project has been the readability and usability of the style sheet itself. The original style sheet contained 714 lines. The new style sheet, including additional instructional comments, has a length of only 162 lines. The elimination of repetitious attribute assignment has resulted in short, succinct style blocks. The file size of the new style sheet is

about 25% of the original.

5.2 Style Sheet Usability

Modifying the style sheet should prove much easier and less intimidating for researchers. The style sheet user now has access to a large set of named colors. New visual style attributes are available. The ability to add and remove attributes by use of comment blocks should simplify experimentation. Finally, attributes can be applied directly to map-building files with no need to resort to manual interpretation steps.

New error checking and messages should provide better user support by pointing out the cause and location of style sheet errors. Errors caught by the interpreter stop all processes and inform the user of the problem and its location. Unrecognized style attributes cause warnings to be issued during map-building.

5.3 Benefits of using a domain specific language

The style sheet as a domain specific language for the Haptic Soundscape map is an essential component of the project. The map could be more easily built by computer scientists without the inclusion of a DSL, but the finished product would be a static application. Any modifications determined necessary by researchers would require more computer programming to implement. By using the style sheet DSL, anyone with moderate computer skills can reconfigure the map and

generate the new model in a matter of minutes.

Domain Specific Languages, also called "little languages," serve other purposes as well. They evolve naturally from observations gained while working in the problem area of the main project. As abstractions of underlying processes, they enable users and designers to focus on *what* is to be done, not *how* it is done. Functioning at this higher level, they are less prone to error than the general purpose programming language used to develop the main application. The concept of developing a domain specific language is one that programmers should embrace for most larger projects. Jon Bentley has this to say about little languages:

> Languages surround programmers, yet many programmers don't exploit linguistic insights. Examining programs under a linguistic light can give you a better understanding of the tools you now use, and can teach you design principles for building elegant interfaces to your future programs [Bentley, 1986].

Compiler construction tools have been known to computer scientists for years, and are available now for use in most language and system environments. These tools, mostly based on the standards of the original LEX and YACC tools, provide a relatively easy and powerful method to develop domain specific applications. The map-building project has benefited greatly from the choice of JFlex and Java CUP tools for implementing the style sheet from the beginning of the project. With this foundation in place, modifying and "growing" the style sheet language has been an uncomplicated process. Future modifications to the style sheet language should also pose no major problems for developers.

Future Directions

As the current stage of the style sheet enhancements and improvements to the general map-building process has been completed, new ideas for the future development of the Haptic Soundscape map have become apparent. Some of these are:

- Finding functional and affordable haptic-enabled input hardware has been problematic from the beginning of the map project. The device chosen for the initial development was a haptic mouse originally intended for use in computer games. Its older style proved to be bulky and not as responsive as would be desired. The software support also limited experimenters to using only Internet Explorer web browser on the Microsoft Windows operating system. By the end of the initial phase of the project, even this mouse model had been discontinued by its manufacturer. In addition, more recent trials have shed doubt on the use of a computer mouse as an accessible pointing device for visually-impaired users. A more ideal interface would probably be some sort of haptic-enabled tablet, if such hardware were available. That would provide the user with a better spatial representation of the onscreen area.

- The original map was developed using version 2 of the ActionScript language for creating onscreen objects and actions for the Adobe Flash platform. One issue that remains in the functionality of the map as a research tool is that the automatically generated ActionScript files must be manually opened in the Flash application to generate the final file format. By converting the language used to ActionScript version 3, the map-building process could take advantage of a newly available command-line compiler and the entire process could be automated.

- It should be possible to construct the Haptic Soundscape map for display systems other than the web server/Adobe Flash/web browser interface explored during initial development. The ActionScriptBuilder class was designed to be an interchangeable interface and could be replaced with a class designed for a different target platform. One project currently underway is attempting to do this using a Java desktop application as the new target. This promises better control over system features not available through a web browser.

A logical extension of the concept of the style sheet as the project user interface would be to expose more application functionality through the style sheet. There are a number of mapping features such as overall map size, zoom levels, grids, and navigational sounds that are still hard-coded into the main project files. Having access to more mapping functionality would provide researchers with a more flexible tool.

Appendix A

Style Sheet Language Reference

A.1 Introduction

This document is a reference for the style sheet language created for the Haptic Soundscape Map project, developed as a collaboration between the Computer Information Science and Geography departments at the University of Oregon in 2007. The map itself is generated from a GIS (Geographic Information System) information database at the University. The purpose of the style sheet component of the project is to make the finished map easily configurable by researchers without needing to enlist the help of others with experience in computer programming.

Our design for the map-building style sheet is modeled on the Cascading Style Sheet specification used with Hypertext Markup Language (HTML). The project style sheet is a plain text document that can be easily edited to set haptic, sound, and visual properties of the various elements displayed on the map.

A.2 BNF Grammar

This document uses an extended version of Backus-Naur form (BNF) notation for specifying lexical and syntactic conventions used in the style sheet language. Here is an example of the notation that will be used.

⟨*identifier*⟩ ::= ⟨*letter*⟩ (⟨*letter*⟩ | ⟨*digit*⟩ | [_] | [-])*
⟨*letter*⟩ ::= ⟨*uppercase*⟩ | ⟨*lowercase*⟩
⟨*uppercase*⟩ ::= [A] ... [Z]
⟨*lowercase*⟩ ::= [a] ... [z]
⟨*digit*⟩ ::= [0] ... [9]

The first line of the preceding example means that an identifier begins with a letter character and then is composed of any number of letters, digits, underscores, or hyphens. Parentheses are used for grouping expressions. The vertical bar (|) indicates a choice between two or more alternatives. Literal characters, like the hyphen and underscore above, are enclosed by boxes. Immediately after an item, a star (*) means zero or more repetitions of the item are allowed by the definition.

The second line indicates that a letter can be either uppercase or lowercase, which are each defined in following lines. The use of an ellipsis (three dots) between literal characters indicates a series. Thus, a digit may be any of the characters from 0 to 9, inclusive.

Examples of identifiers, as defined by the above rule, include "a", "a99", "A_9_c-", and "a-pretty-good-name", but not "99" or "9abc" (because they don't begin with a letter).

⟨*number*⟩ ::= [[-]] (⟨*integer*⟩ | ⟨*float*⟩)
⟨*integer*⟩ ::= ⟨*digit*⟩+
⟨*float*⟩ ::= (⟨*digit*⟩* [.] ⟨*digit*⟩+) | (⟨*digit*⟩+ [.] ⟨*digit*⟩*)
⟨*digit*⟩ ::= [0] ... [9]

An optional item is surrounded by square brackets [ ] as shown in the optional minus sign in the first line of the preceding example. Plus (+) means one or more repetitions, as illustrated in the definition of an integer a being composed of one or more digits. The definition of a float (floating point number) can be interpreted as defining that symbol as zero or more digits followed by a decimal point followed by one or more digits; or one or more digits followed by a decimal followed by zero or more digits. In other words, a float must have at least digit either before or after the required decimal point.

Examples of integers include "99", "009", and "-34". Examples of floats include "99.7", "0.0", and ".09".

⟨*string_literal*⟩ ::= ["] ⟨*stringchar*⟩* ["]
⟨*stringchar*⟩ ::= ⟨any character except double quote⟩

The preceding example demonstrates how a character class may be described by rules. Text within angle brackets ⟨ ⟩ is a rule to define a particular category of characters allowed. A string literal is defined as two quotation marks surrounding any number of characters that may consist of anything except quotation marks.

A.3 Lexical Structure

This section defines the individual building blocks, called lexical elements or tokens, that are used by the the style sheet language. A style sheet is first broken down into these basic blocks and irrelevant characters, such as spaces and comment lines, are discarded. Then the tokens are assembled into logical statements and their meaning is interpreted.

For example, in the University of Oregon haptic soundscape map, the style sheet block below associates a special sound file with Deady Hall on the campus:

```
//Deady Hall
building#005 {
  on_click_sound: buildingSoundPath + "005a.mp3";
}
```

If we break this statement into tokens, we have:

```
building
#005
{
on_click_sound
:
buildingSoundPath
+
"005a.mp3"
;
}
```

These are the elements that are interpreted to assign attributes to parts of a map. Note that the comment line has been discarded.

A.3.1 Whitespace and Indentation

The term whitespace refers to the empty space that is used around items to visually separate them. The set of whitespace computer characters consists of the SPACE, TAB, CR(carriage return), and LF(line feed) ASCII characters. When two tokens appear on the same line, it is often necessary to separate them by the use of one or more whitespace characters. Multiple whitespace characters are allowed and the unnecessary characters are discarded.

In general, end-of-line characters are regarded as whitespace and allowed to occur any place where one might place whitespace (space or tab) characters.

```
car_parking{color: blue;}
```

As an example, the above statement contains no internal spaces or line breaks and the following example contains a number of spaces, tabs, and line breaks. Both are interpreted exactly the same.

```
car_parking
{
          color     :       blue;
}
```

Whitespace and indentation options are left entirely to the style sheet author. Tab or space characters appearing before token strings have no effect on the interpretation of the style sheet. The user is encouraged to adopt a consistant layout style for statements in the style sheet, using spaces and line breaks as appropriate to enhance readability.

A.3.2 Comments

This style sheet specification supports both the multi-line type of comments used in CSS documents and the single-line comments supported by the C++ and Java programming languages. All comments are ignored by the style sheet interpreter.

For single line comments, the comment block begins with two forward slash characters (//) and continues to the end of the current line. Eveything on the rest of the line is ignored by the style sheet interpreter. The beginning of the comment block is allowed to occur anywhere in the line. A line may also consist of a style sheet token followed by a comment.

Multi-line comments begin with a forward slash followed by a star (/*) and end with a star followed by the forward slash (*/). Multi-line comments may also begin and end anywhere in a line. Multi-line comments may be nested – one comment block may contain another inside. However, the beginning and terminating tokens must be properly matched and nested to be interpreted correctly.

Examples:

```
/**
* ------------------------------------------------------
*  This is a legal multi-line comment
*
*  /* Another comment may be nested
*     inside the first  */
*
*  // This is also correct because everything on these
*  // lines after the double slashes is ignored. */
*
* ------------------------------------------------------
*/

// This is a single line comment.
```

```
@import "colors.ss"; // Comments may appear beside tokens

// Comments may appear as part of required whitespace.

building{            // default building colors
  fill-color: green;  // fill color
  line-color: black;  // outline color
}
```

While it is possible to create complex, nested comments as illustrated above, this practice is not recommended. If the starting and ending tokens are not properly nested, the interpreter could consider the rest of the style sheet a comment or register an error. This feature, however, is useful when temporarily commenting out a block of code containing other comments.

The one exception to this rule is that the parts of the file import statement cannot be interrupted by comments. This statement is considered as one compound token.

A.3.3 Blank Lines

Blank lines, possibly containing whitespace or end-of-line characters are ignored by the interpreter and may be placed wherever desired to increase the readability of the style sheet.

A.3.4 Other Tokens

Aside from the tokens already discussed, the style sheet can contain the following classes of tokens: @import statements, class and definition identifiers, object identifiers, numbers, string literals, color constants, operators, and delimiters.

The longest possible stream of characters will always be combined to form tokens in the style sheet language. Thus, classSoundPath will be interpreted as one identifier, not as "class" followed by "Sound" followed by "Path."

All identifiers and keywords are case-sensitive. This means that "classSoundPath" is not the same identifier as "classSoundpath."

A.3.5 Import Statements

An import statement is used to import a second style sheet into a style sheet currently being processed by the interpreter. The imported style sheet file is read at that point before continuing to read the rest of the original file. It is as if the "@import" statment were replaced by the contents of the imported file. Style sheet imports may be nested to any depth.

⟨*import-statement*⟩ ::= `@import` ⟨*filename*⟩ `;`
⟨*filename*⟩ ::= (⟨*letter*⟩ | ⟨*digit*⟩ | `_` | `-` | `.` | `/`)+

File paths must use the forward slash (/) character as a directory separator. An example of an import statement is shown in the following example along with a preceding comment line:

```
// import color definitions
@import "colors.ss";
```

This use of an imported style sheet can make a large number of definations available for use in the main style sheet without cluttering that file. The above example is used to import color definitions into the style sheet environment.

A.3.6 Identifiers

Identifiers are used for class identification on the map (such as buildings or walkways) and for definitions used within the style sheet.

⟨*identifier*⟩ ::= ⟨*letter*⟩ (⟨*letter*⟩ | ⟨*digit*⟩ | _ | -)*

Identifiers must always begin with an upper- or lower-case letter character, followed by any combination of upper- or lower-case letters, digits, underscore characters, or hyphens.

The example below contains 3 identifiers: car_parking, color, and blue.

```
car_parking{
  color: blue;
}
```

A.3.7 Object Identifiers

Object identifiers are used to specify a particular object within a class of objects.

⟨*object_id*⟩ ::= [#] (⟨*letter*⟩ | ⟨*digit*⟩ | [_] | [-] | [(] | [)] | [%])+

Below are two examples of object identifiers. Within the University of Oregon Haptic Soundscape map, "#044" is used to identify the object Deschutes Hall within the class of "building." "#NC895e13" is id of the UO Bookstore.

```
building#044
building#NC895e13
```

A macro definition is a special type of object identifier. This allows an action to be performed on every object within a class by substituting the object identifier in the action.

An example of a macro substitution object id is "#[%n]".

A.3.8 Numbers

Numbers may be positive or negative integers or floating point numbers using a decimal point.

⟨*number*⟩ ::= [[-]] (⟨*integer*⟩ | ⟨*float*⟩)
⟨*integer*⟩ ::= ⟨*digit*⟩+
⟨*float*⟩ ::= (⟨*digit*⟩* [.] ⟨*digit*⟩+) | (⟨*digit*⟩+ [.] ⟨*digit*⟩*)
⟨*digit*⟩ ::= [0] ... [9]

The following example illustrates usage of an integral number:

```
building {
  line-width: 1;
}
```

A.3.9 String Literals

⟨*string_literal*⟩ ::= " ⟨*stringchar*⟩* "

⟨*stringchar*⟩ ::= ⟨any character except double quote⟩

Strings are used to define resources and colors for the map production. The quoted characters below are examples of string literals.

```
LightPink = "#FFB6C1";
on-hover-sound: "stairways.mp3";
```

A.3.10 Color Constants

Color constants are a special type of string literal used to define colors for the map.

⟨*color_constant*⟩ ::= # ⟨*RRGGBB*⟩

⟨*R*⟩ ::= ⟨*digit*⟩ | ⟨*hex_letter*⟩

⟨*G*⟩ ::= ⟨*digit*⟩ | ⟨*hex_letter*⟩

⟨*B*⟩ ::= ⟨*digit*⟩ | ⟨*hex_letter*⟩

⟨*hex_letter*⟩ ::= A | B | C | D | F | a | b | c | d | f

⟨*digit*⟩ ::= 0 ... 9

A color constant takes the form: #FF3300. It consists of a hash(#) followed by exactly 6 hexadecimal digits [Meyer, 2004, p. 68]. Uppercase hex letters are

recommended, but either uppercase or lowercase will function properly. Below are examples of color constants used in defining color names.

```
Plum = "#DDA0DD";
Violet = "#EE82EE";
Magenta = "#FF00FF";
```

Note that you can import the color list in Appendix C and then refer to colors by name only.

A.3.11 Operators

The style sheet language uses the following three operators:

: = +

The colon is used to define object style attributes. The equal sign defines variables for use in the interpretation of the style sheet. The plus sign indicates string concatenation.

A.3.12 Delimiters

The style sheet language uses the following delimiters:

{} [] "" ;

Braces are used to group style attribute definitions. Square brackets are used to denote macro substitution components. Double quotes are used to delimit strings. The semi-colon functions as an end of logical line character.

A.4 Statement Syntax

This section describes the syntax of the style sheet. In this language, each import, assignment, or attribute statement is contained in a logical line terminated by a semi-colon (;) character. This logical line may span any number of physical lines.

A.4.1 Import Statement

This is a special statement that is used to import additional style sheets into the one presently being processed by the lexical analyzer.

⟨*import-statement*⟩ ::= `@import` ⟨*filename*⟩ `;`
⟨*filename*⟩ ::= (⟨*letter*⟩ | ⟨*digit*⟩ | `_` | `-` | `.` | `/`)+

An example of this syntax is:

```
@import "../colors.ss";
```

This would import a file named colors.ss from the parent directory with respect to the location of the current style sheet being read. The file name may include an absolute or relative path statement. A relative path statement begins with the name of a sub-directory or ".." which refers to the parent directory. All directory separators should use the forward slash (/) convention, regardless of the operating system being used.

An example of where a file import would be useful is to list all color definitions in a separate file and import this into the top of the main style sheet. In this man-

ner, statements in the main style sheet may refer to defined color names such as "LightBlue" instead of "#ADD8E6".

If the import file cannot be found or cannot be read, the interpreter will report an error and exit.

Since this statement is actually processed in the lexical analyzer instead of the parser, comments are not allowed within the import statement. Any whitespace character, including line breaks, are acceptable where whitespace can be used to separate tokens within the statement.

A.4.2 Assignment Statement

Assignment statements are used to assign string or numeric argument values to an identifier.

⟨*assignment*⟩ ::= ⟨*identifier*⟩ [=] (⟨*string_literal*⟩ | ⟨*identifier*⟩) [;]

Examples of an assignment statements are:

```
classSoundPath = "./classSounds/";
Plum = "#DDA0DD";
```

The first creates a variable identified by classSoundPath which has a value of "./classSounds/". The second example assigns a hexadecimal RGB color value to the name Plum. The variables can then be used in the right side of any assignment or class attribute statement.

A.4.3 Style Attributes

Style attributes are assigned by using the colon (:) character.

⟨*style_attributes*⟩ ::= ⟨*identifier*⟩ [:] ⟨*value*⟩ [;]
⟨*value*⟩ ::= ⟨*string_literal*⟩ | ⟨*variable_identifier*⟩
⟨*variable_identifier*⟩ ::= ⟨*identifier*⟩

Examples of style attribute assignment are:

```
fill-color : "#8B0000";
fill-color : DarkRed;
```

The first assigns the value of "#8B0000" to the "fill-color" attribute key. The second assigns the predefined color name of DarkRed. The two statements are actually equivalent, since DarkRed is defined as "#8B0000" in the X11 color model.[6]

A.4.4 Concatenation

The plus (+) sign is used in the style sheet for string concatenation – combining two or more strings to form one.

⟨*concatenation*⟩ ::= ⟨*item*⟩ ([+] ⟨*item*⟩)+
⟨*item*⟩ ::= ⟨*identifier*⟩ | ⟨*string_literal*⟩

[6]See "Color Constant Names" in Appendix C.

An example of usage for concatenation is:

```
classSoundPath = "./classSounds/";
on-enter-sound: classSoundPath + "bikebell.mp3";
```

This would result in the style attribute "on-enter-sound" being assigned the concatenation of the value of classSoundPath and the literal string "bikebell.mp3", resulting in the complete file path "./classSounds/bikebell.mp3".

Any combination of variables and string literals may be concatenated on the right side of any assignment or style attribute statement. The plus sign signifies string concatenation only. Thus, the statement:

```
3 + 3
```

produces:

```
33
```

A.4.5 Style Selector

The style selector specifies what object or class of objects a style attribute or set of attributes should be applied to.

⟨*style_selector*⟩ ::= (⟨*class_name*⟩ [⟨*object_id*⟩] [⟨*modifier*⟩])
| ([⟨*class_name*⟩] ⟨*object_id*⟩ [⟨*modifier*⟩])

⟨*class_name*⟩ ::= ⟨*identifier*⟩

⟨*object_id*⟩ ::= [#] ⟨*id*⟩

⟨*modifier*⟩ ::= [:] ⟨*identifier*⟩

Acceptable characters for an object identifier or id are defined in lexical analysis section. A style selector may contain a class name, object id and a modifier, which specifies a particular object from a particular class under the condition specified by the modifier. Here is an example:

```
building#005 :zoomin
```

A style selector may also contain only the class name, which applies the style to every object in that class, or only the id, which applies to style to a particular id in any class where it is found. The modifier restricts the setting to a particular mode of operation. In this case the selector refers to the object when the map is in "zoomed in" mode.

A.4.6 Style Block

A style block contains a style selector and one or more style attribute assignments.

⟨*style_block*⟩ ::= ⟨*style_selector*⟩ [{] ⟨*style_attribute_set*⟩ [}]
⟨*style_attribute_set*⟩ ::= ⟨*style_attribute*⟩+

The set of style attributes contained in the curly braces are applied to the object and condition specified in style selector. An example is:

```
bicycle_parking: zoomin{
    color: dark-red;
    on-enter-sound: classSoundPath + "bikebell.mp3";
}
```

This block defines the color for every object in the class bicycle_parking when the map is in "zoomed-in" mode. It also specifies the sound to be played when the mouse pointer enters the object.

A.4.7 Substitution Macro Definition

This is a special type of style block used as shorthand for attributes that are repeated a number of times using a particular pattern.

⟨*style_selector*⟩ ::= ⟨*class_name*⟩ ⟨*substitution_id*⟩ [⟨*modifier*⟩]
⟨*class_name*⟩ ::= ⟨*identifier*⟩
⟨*substitution_id*⟩ ::= `[%` ⟨*letter*⟩ `]`
⟨*modifier*⟩ ::= `:` ⟨*identifier*⟩

The object id is then substituted for the marker wherever it is encountered in the style attributes. An example is:

```
building #[%n] {
    on-click-sound: "[%n].mp3";
}
```

In this example, if the building id is #105, the string "105" is substituted in the attribute line. In this case the attribute becomes

```
on-click-sound: "105.mp3";
```

A.5 Data Model and Access Methods

As the parser interprets a style sheet, it saves the attribute settings in a data structure for retrieval by the client program. This data structure consists of two Java hash map objects. One called "classes" for regular attributes and the other called "macros" for the substitution macro attributes.

Each of these data structures maps a class name to a StyleClass object. The StyleClass object contains a hash map of ids for that class mapped to a list of StyleAttribute objects. A StyleAttribute object contains the style key, value, and modifier.

Either a class name or object id may be represented by the empty string (" "). If the object id is blank, the attributes apply to every object within a class. If the class attribute is blank, the attributes apply to every object with that id in any class.

Access to the set of attributes for a particular combination of class name and object id is accomplished via the public methods:

```
parser.getAttr(class, id) { ... }

parser.getMacro(class, id) { ... }
```

Each returns the appropriate list of StyleAttributes. The macro substitutions are expanded within the getMacro function, based on the id passed into the function by the caller and returned as part of a list of style attributes.

Appendix B

Style Sheet Interpretation in the Haptic Soundscape Map

B.1 Introduction

While the style sheet itself has been created for general usage, the interpretation of its generated values by the University of Oregon Haptic Soundscape map imposes limits on the available options. The purpose of this document is to describe those limits and other options taken in the interpretation of the style sheet.

B.2 Precedence of Style Attribute Assignment

The map applies style attributes on a priority basis from the most general to the most specific. This means that a more specific style attribute setting will always override a more general setting.

1. The most specific style attribute is considered to be a particular class name with a specific object id. This receives the highest priority.

2. The next style attribute to be considered is one for which an id has been assigned, but no class name. This will be applied to any matching object within any class.

3. The next style attribute to be considered is one with a class name, but no object id specified. This will be applied to all objects within the specified class.

4. The next style attribute to be considered is one resulting from a macro substitution pattern. This style attribute will be applied appropriately to each object within the specified class.

5. The lowest priority style attribute uses the class name of "default". This is applied to all objects in all classes that have no instances of higher priority settings.

B.3 Class Names

The class names are derived from GIS layers used to generate the University of Oregon map. The acceptable names for this mapping application are listed below.

athletic_surfaces	emergency_callboxes	stairways
bicycle_parking	hydrology	streets
buildings	parking	walks
bus_stops	public_entrances	walk_underpasses

There is one additional class name recognized – the default class. If an attribute is listed under the class name of "default", it is applied to all classes.

B.4 Modifiers

This map application uses two modifiers: "zoomin" and "zoomout." Style attributes having the "zoomin" modifier are applied only when the map is in "zoomed in" mode. Likewise style attributes modified by "zoomout" are applied only when the map is in "zoomed out" or normal mode. Style attributes with no modifier are applied to both modes of the map.

B.5 Keys

B.5.1 Drawing Attributes

In the haptic soundscape map, attributes settings for object outlines and fill apply to all map zoom modes for that object and any modifiers (zoomin or zoomout) are ignored.

Line width is set using the key "line-width" and a numeric value. In the Flash/Actionscript version of the map, a line width of "1" is considered the default value for drawing objects. A value setting of "0" will result in the object being rendered smaller than expected. Other rendering systems may consider "0" to be the default.

Color is set either by defining a value for the "color" key, or for the "fill-color" and "line-color" keys. The "fill-color" and "line-color" settings will take precedence in determining object color. If either "fill-color" or "line-color" has not been defined, the rendering operation will default to the "color" setting. If none of the color keys has been specified, a default color of Oregon Green (#006633) will be used.

B.5.2 Actions

Attribute action keys define the interaction between the map and the motions and clicks of the user's mouse. All actions may be modified by "zoomin" and "zoomout", resulting in three possible settings for each. Keys are defined for sound and texture effects.

Sound

- on-enter-sound
- on-exit-sound
- on-hover-sound
- on-click-sound

Texture

- on-enter-texture
- on-exit-texture
- on-hover-texture
- on-click-texture

These settings correspond to the mouse pointer entering or exiting the area occupied by an object on the monitor, the mouse pointer pausing over an object's area (hover), or the primary mouse button being clicked over an object.

B.6 Values

Values applied to sytle attributes are string values representing either paths to haptic or sound resources, numeric values, or color values. Color values are specified in the format of "#RRGGBB" where RR, GG, and BB are hexidecimal values from 00 to FF, representing the red, green, and blue components of the color in the RGB color model [Meyer, 2004, p. 68].

B.7 Color Definitions

An auxiliary style sheet containing the entire set of color definitions for the X11 color set is provided for import into the main application style sheet. Using this set of color definitions, the style sheet author need only specify colors by name and allow the look-up mechanism make the hexadecimal substitutions. See Appendix C for a list of colors defined.

Appendix C

Style Sheet Constants Used in the Haptic Soundscape Map

C.1 Introduction

This document contains a listing of constants used in the 2007 University of Oregon Haptic Soundscape Map project.

C.2 Building Names and IDs

This is a listing of building names and matching ID values used in the Haptic Soundscape map project for the University of Oregon campus.

AAA Greenhouse – #099
AAA Woodshop – #098
Agate – #147
Agate House – #148
Allen – #017
Archaeology Research – #116

Bean – #069
Bowerman Family – #063
Canoe House – #127
Carson – #076
Cascade – #047
Cascade Annex East – #028B
Cascade Annex West – #028A
Central Power Station – #032
Central Power Station Intertie – #032x
Chapman – #006
Chiles – #002
Church Warehouse – #157
Clinical Services – #029
Collier House – #081
Columbia – #036
Computing – #039
Condon – #004
Covered Tennis Courts – #059
Deady – #005
Deschutes – #044
DPS Information Booth – #213
DPS Parking Meter Garage – #593x
Earl – #073
East Campus Graduate Village (East) – #146
East Campus Graduate Village (West) – #145
ECS (1761) – #603
ECS (1791) – #582
Education – #007
Education Addition – #041
Education Annex – #048
Erb Memorial Union (EMU) – #033
Esslinger – #023
FAS Ceramics – #125B
FAS Foundry – #125F
FAS Kiln – #125D
FAS Metalsmith & Jewelry – #125C
FAS Raku Shed – #125E
FAS Sculpture – #125A
FAS Wood Fire Kiln – #125G
Fenton – #019

Friendly – #009
Frohnmayer Music – #025
FS Adminstration Office – #136
FS Electrical Storage Quonset – #133
FS Exterior Storage Quonset 1 – #128
FS Exterior Storage Quonset 2 – #135
FS Exterior Team – #593
FS Lockshop – #121
FS Miscellaneous Quonset – #131
FS Mower Storage – #517
FS Paint Storage Quonset – #132
FS Recycling Modular – #500
FS Recycling Quonset – #134
FS Spray Booth Quonset – #129
FS Warehouse – #130
Gerlinger – #011
Gerlinger Annex – #062
Gilbert – #003
Greenhouse Preparation – #109
Hamilton – #085
Hayward East Grandstand – #012
Hayward Ticket Booth #1 – #113
Hayward Ticket Booth #2 – #114
Hayward Track Storage – #056
Hayward Weight Room – #061
Hayward West Concession Stand – #106
Hayward West Grandstand – #013
Hayward West Ticket Booth – #093L
Hendricks – #071
HEP Classroom – #536
HEP Modular Classroom – #215
HEP Office – #521
Howe Home Dugout – #053A
Howe Office/Shop – #141
Howe Press Box/Concession – #053
Howe Storage – #142
Howe Visitor Dugout – #053B
Huestis – #040
Johnson – #016
Klamath – #038

Knight Law – #050
Knight Library – #018
Landscape Ecology Lab – #117
Lawrence – #001
LERC & Military Science – #087B
Lillis – #003B
Living Learning Center North – #065
Living Learning Center South – #064
Many Nations Longhouse – #167
McArthur Court – #020
McKenzie – #030
Millrace Studio 1 – #095
Millrace Studio 2 – #096
Millrace Studio 3 – #097
MNH Archaeological Research Lab – #107
MNH Prep Lab – #115
MNH Storage – #108
Moss Street Children's Center – #168
Museum of Natural and Cultural History – #049
Old Greenhouse – #110
Olum – #165
Olum Annex – #164
Onyx Bridge – #037
Oregon – #042
Outdoor Program Barn – #052
Outdoor Tennis Courts Facility – #138
Pacific – #035
Paleoecology Research – #112
Peterson – #003A
Prince Lucien Campbell (PLC) – #008
Research Greenhouse – #111
Reske Center (Oregon Track Club Storage) – #060
Riverfront Innovation Center – #043
Riverfront Research Park (1600 Millrace) – #750L
Riverfront Research Park (1800 Millrace) – #094L
Schnitzer Museum of Art – #024
Straub – #072
Streisinger – #045
Student Recreation Center – #051
Student Tennis Courts – #139

Susan Campbell – #075
Trailer A – #055
Trailer B – #054
Trailer C – #057
Trailer D – #058
University Health and Counseling – #014
UO Annex – #082
UO Bookstore – #NC895e13
Villard – #031
Volcanology – #015
Walton – #078
Wilkinson House – #026
Willamette – #046
YWCA – #533
Zebrafish International Resource – #101

C.3 Color Constant Names and Definitions

Definitions for color variables are listed below, based on the X11 color set with HTML/CSS color definitions where there are conflicts (gray, green, maroon, and purple). New colors can be defined using a hash sign (#) followed by hex rgb color value (as used in HTML/CSS).

LightPink – #FFB6C1
Crimson – #DC143C
PaleVioletRed – #DB7093
DeepPink – #FF1493
Orchid – #DA70D6
Plum – #DDA0DD
Magenta – #FF00FF
DarkMagenta – #8B008B
MediumOrchid – #BA55D3
Pink – #FFC0CB
LavenderBlush – #FFF0F5
HotPink – #FF69B4
MediumVioletRed – #C71585
Thistle – #D8BFD8
Violet – #EE82EE
Fuchsia – #FF00FF
Purple – #800080
DarkViolet – #9400D3

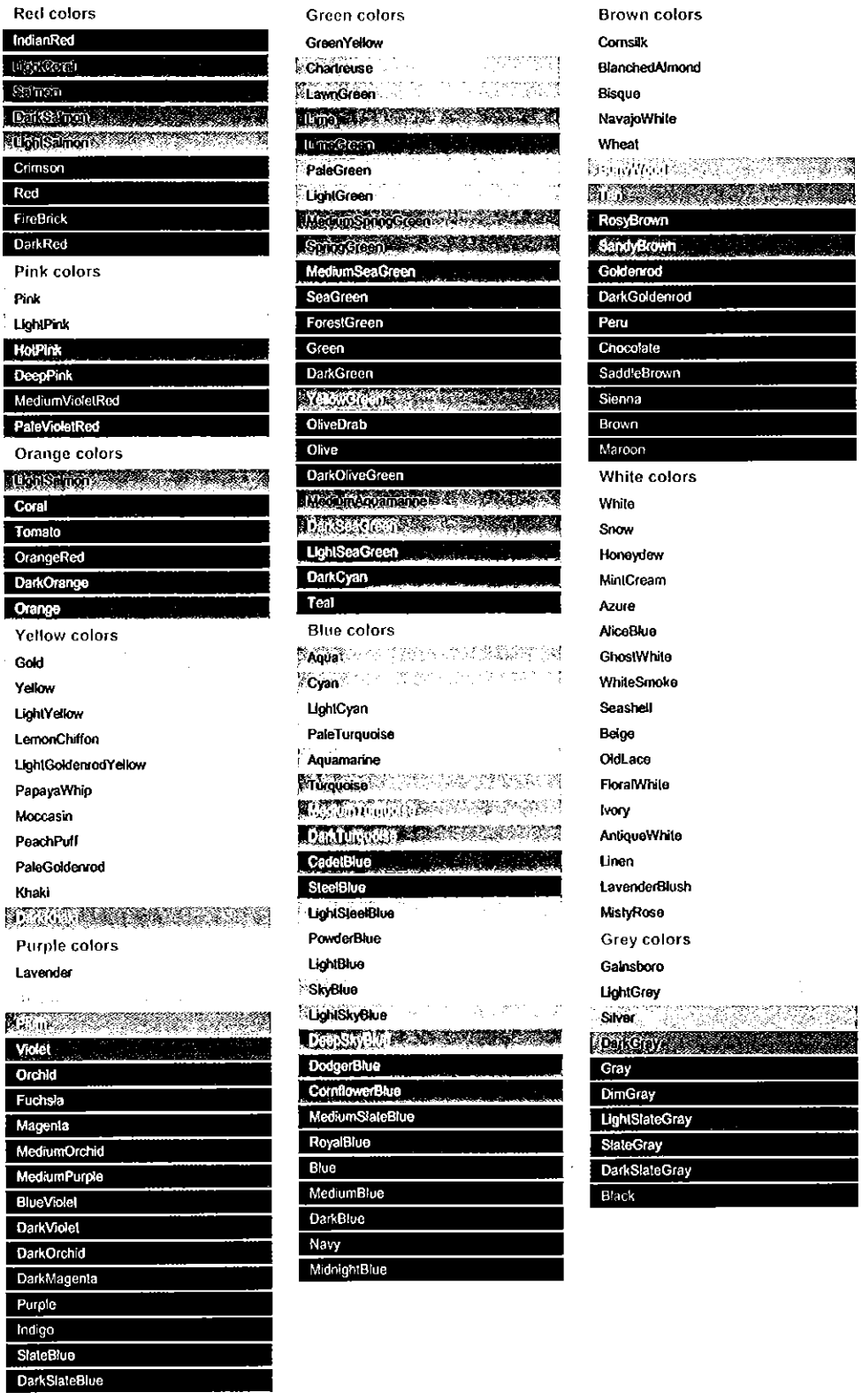
DarkOrchid – #9932CC
BlueViolet – #8A2BE2
MediumSlateBlue – #7B68EE
DarkSlateBlue – #483D8B
GhostWhite – #F8F8FF
MediumBlue – #0000CD
DarkBlue – #00008B
RoyalBlue – #4169E1
LightSteelBlue – #B0C4DE
SlateGray – #708090
AliceBlue – #F0F8FF
LightSkyBlue – #87CEFA
DeepSkyBlue – #00BFFF
PowderBlue – #B0E0E6
Azure – #F0FFFF
PaleTurquoise – #AFEEEE
Aqua – #00FFFF
DarkSlateGray – #2F4F4F
Teal – #008080
LightSeaGreen – #20B2AA
Aquamarine – #7FFFD4
MediumSpringGreen – #00FA9A
SpringGreen – #00FF7F
SeaGreen – #2E8B57
LightGreen – #90EE90
DarkSeaGreen – #8FBC8F
Lime – #00FF00
Green – #008000
Chartreuse – #7FFF00
GreenYellow – #ADFF2F
YellowGreen – #9ACD32
Beige – #F5F5DC
Ivory – #FFFFF0
Yellow – #FFFF00
DarkKhaki – #BDB76B
PaleGoldenrod – #EEE8AA
Indigo – #4B0082
MediumPurple – #9370DB
SlateBlue – #6A5ACD
Lavender – #E6E6FA
Blue – #0000FF
MidnightBlue – #191970
Navy – #000080
CornflowerBlue – #6495ED
LightSlateGray – #778899
DodgerBlue – #1E90FF
SteelBlue – #4682B4
SkyBlue – #87CEEB
LightBlue – #ADD8E6
CadetBlue – #5F9EA0
LightCyan – #E0FFFF
Cyan – #00FFFF
DarkTurquoise – #00CED1
DarkCyan – #008B8B
MediumTurquoise – #48D1CC
Turquoise – #40E0D0
MediumAquamarine – #66CDAA
MintCream – #F5FFFA
MediumSeaGreen – #3CB371
Honeydew – #F0FFF0
PaleGreen – #98FB98
LimeGreen – #32CD32
ForestGreen – #228B22
DarkGreen – #006400
LawnGreen – #7CFC00
DarkOliveGreen – #556B2F
OliveDrab – #6B8E23
LightGoldenrodYellow – #FAFAD2
LightYellow – #FFFFE0
Olive – #808000
LemonChiffon – #FFFACD
Khaki – #F0E68C

Gold – #FFD700
Goldenrod – #DAA520
FloralWhite – #FFFAF0
Wheat – #F5DEB3
Orange – #FFA500
BlanchedAlmond – #FFEBCD
AntiqueWhite – #FAEBD7
BurlyWood – #DEB887
DarkOrange – #FF8C00
Peru – #CD853F
SandyBrown – #F4A460
SaddleBrown – #8B4513
Sienna – #A0522D
Coral – #FF7F50
DarkSalmon – #E9967A
MistyRose – #FFE4E1
Snow – #FFFAFA
RosyBrown – #BC8F8F
Red – #FF0000
FireBrick – #B22222
Maroon – #800000
WhiteSmoke – #F5F5F5
LightGrey – #D3D3D3
DarkGray – #A9A9A9
DimGray – #696969
Cornsilk – #FFF8DC
DarkGoldenrod – #B8860B
OldLace – #FDF5E6
Moccasin – #FFE4B5
PapayaWhip – #FFEFD5
NavajoWhite – #FFDEAD
Tan – #D2B48C
Bisque – #FFE4C4
Linen – #FAF0E6
PeachPuff – #FFDAB9
Chocolate – #D2691E
Seashell – #FFF5EE
LightSalmon – #FFA07A
OrangeRed – #FF4500
Tomato – #FF6347
Salmon – #FA8072
LightCoral – #F08080
IndianRed – #CD5C5C
Brown – #A52A2A
DarkRed – #8B0000
White – #FFFFFF
Gainsboro – #DCDCDC
Silver – #C0C0C0
Gray – #808080
Black – #000000

C.4 Color Selection Chart

On the following page is a color chart showing the colors available in the X11 pallet.[7]

[7] Adapted form Wikipedia X11 color names. ⟨ http://en.wikipedia.org/wiki/Web_colors ⟩


Red colors
IndianRed
LightCoral
Salmon
DarkSalmon
LightSalmon
Crimson
Red
FireBrick
DarkRed
Pink colors
Pink
LightPink
HotPink
DeepPink
MediumVioletRed
PaleVioletRed
Orange colors
LightSalmon
Coral
Tomato
OrangeRed
DarkOrange
Orange
Yellow colors
Gold
Yellow
LightYellow
LemonChiffon
LightGoldenrodYellow
PapayaWhip
Moccasin
PeachPuff
PaleGoldenrod
Khaki
Purple colors
Lavender
Violet
Orchid
Fuchsia
Magenta
MediumOrchid
MediumPurple
BlueViolet
DarkViolet
DarkOrchid
DarkMagenta
Purple
Indigo
SlateBlue
DarkSlateBlue
Green colors
GreenYellow
Chartreuse
LawnGreen
Lime
LimeGreen
PaleGreen
LightGreen
SpringGreen
MediumSeaGreen
SeaGreen
ForestGreen
Green
DarkGreen
YellowGreen
OliveDrab
Olive
DarkOliveGreen
MediumAquamarine
DarkSeaGreen
LightSeaGreen
DarkCyan
Teal
Blue colors
Aqua
Cyan
LightCyan
PaleTurquoise
Aquamarine
Turquoise
DarkTurquoise
CadetBlue
SteelBlue
LightSteelBlue
PowderBlue
LightBlue
SkyBlue
LightSkyBlue
DeepSkyBlue
DodgerBlue
CornflowerBlue
MediumSlateBlue
RoyalBlue
Blue
MediumBlue
DarkBlue
Navy
MidnightBlue
Brown colors
Cornsilk
BlanchedAlmond
Bisque
NavajoWhite
Wheat
BurlyWood
Tan
RosyBrown
SandyBrown
Goldenrod
DarkGoldenrod
Peru
Chocolate
SaddleBrown
Sienna
Brown
Maroon
White colors
White
Snow
Honeydew
MintCream
Azure
AliceBlue
GhostWhite
WhiteSmoke
Seashell
Beige
OldLace
FloralWhite
Ivory
AntiqueWhite
Linen
LavenderBlush
MistyRose
Grey colors
Gainsboro
LightGrey
Silver
DarkGray
Gray
DimGray
LightSlateGray
SlateGray
DarkSlateGray
Black

Bibliography

[Aho et al., 1986] Aho, A. V., Sethi, R., and Ullman, J. D. (1986). *Compilers: Priciples, Techniques, and Tools*. Addison-Wesley, Reading, MA.

[Bentley, 1986] Bentley, J. (1986). Programming pearls: little languages. *Commun. ACM*, 29(8):711–721.

[Brown et al., 1995] Brown, D., Mason, T., and Levine, J. R. (1995). *lex & yacc*. O'Reilly and Associates, Inc., Sebastopol, CA.

[Brown, 1983] Brown, P. J. (1983). Error messages: the neglected area of the man/machine interface. *Commun. ACM*, 26(4):246–249.

[Freeman and Pryce, 2006] Freeman, S. and Pryce, N. (2006). Evolving an embedded domain-specific language in java. In *OOPSLA '06: Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 855–865, New York, NY, USA. ACM.

[Friedl, 1997] Friedl, J. E. (1997). *Mastering Regular Expressions*. O'Reilly and Associates, Inc., Sebastopol, CA.

[Golledge and Stimson, 1997] Golledge, R. G. and Stimson, R. J. (1997). *Spatial behavior : a geographic perspective*. Guilford Press, New York, NY.

[Hudak, 1996] Hudak, P. (1996). Building domain-specific embedded languages. *ACM Comput. Surv.*, 28(4es):196.

[Hunt and Thomas, 1999] Hunt, A. and Thomas, D. (1999). *The Pragmatic Programmer*. Addison-Wesley, Reading, MA.

[Louden, 2003] Louden, K. C. (2003). *Programming Languages – Principles and Practices*. Thomson, Boston, MA.

[Mernik et al., 2005] Mernik, M., Heering, J., and Sloane, A. M. (2005). When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344.

[Meyer, 2004] Meyer, E. A. (2004). *Cascading style sheets : the definitive guide, 2nd Edition.* O'Reilly and Associates, Inc., Sebastopol, CA.

[Pagan, 1981] Pagan, F. G. (1981). *Formal Specification of Programming Languages: A Panoramic Primer.* Prentice-Hall, Inc., Englewood Cliffs, NJ.

[Provins, 1998] Provins, D. A. (1998). Take command - lex and yacc: Tools worth knowing. *Linux J.*, 1998(51es):18.

[Steele, 1998] Steele, G. L. (1998). Growing a language. *1998 Proceedings of the Conference on Object-Oriented Programming, Systems, Languages, and Applications.*

[van Deursen et al., 2000] van Deursen, A., Klint, P., and Visser, J. (2000). Domain-specific languages: an annotated bibliography. *SIGPLAN Not.*, 35(6):26–36.

[Walsh and Muellner, 1999] Walsh, N. and Muellner, L. (1999). *DocBook: The Definitive Guide.* O'Reilly Media, Inc., Sebastopol, CA.