USING TCL AND SWIG TO CREATE A RAPID APPLICATION DEVELOPMENT ENVIRONMENT

THESIS

Presented to the Graduate Council of
Southwest Texas State University
in Partial Fulfillment of
the Requirements

For the Degree

Master of Science

By

Beom Seog Hwang

San Marcos, Texas
May 2001

Table of Contents

List of Figures

CHAPTER 1

INTRODUCTION

Rapid Application Development (RAD) is a software development methodology that was developed to respond to the need to deliver systems very fast. RAD utilizes multiple languages, reusable components through an Application Programming Interface (API), and project management tactics to speed up the development process. Since RAD uses high-level languages that are not suitable to produce CPU intensive application, RAD sacrifices execution speed to achieve a faster development time than traditional development. Yet to achieve usable runtime performance, existing RAD tools, such as Visual Basic and Delphi, use components through Component Object Model (COM) and Common Object Request Broker Architecture (CORBA). The idea of using middleware like COM and CORBA may contribute to universal compatibility but it requires extensive programming experience and knowledge. The goal of this project is to create a simpler RAD environment that requires less programming knowledge and work than existing RAD tools by using a scripting language and a tool that allows reuse of code. This RAD environment may be more appropriate in problem domains where testing and developing algorithms are more important than producing commercial software.

Using scripting languages to integrate or glue applications is not a new idea. Extending scripting languages with components written in compiled languages is not new either. In this work we adapt both ideas to create a RAD model that achieves code

reusability and run time efficiency.

First, we review three scripting languages, Tcl, Perl, and Python, and several tools that extend them, including SWIG. We focus on finding a suitable scripting environment for RAD, especially where intensive CPU utilization is required, and adopt Tcl and SWIG for our development environment.

Second, we examine the role and the usability of Tcl and SWIG for RAD with case studies. Through these case studies we demonstrate that utilization of a scripting environment has several advantages, such as shortened development time, effective debugging environment and code reusability.

Last, we develop a system, SIDE (Simple Integrated Development Environment) to simplify and speed up the process of building extensions. We have also sketch a plan to improve SIDE to form a full-featured Integrated Development Environment (IDE) for RAD.

CHAPTER 2

BACKGROUND

In recent years, because of a more powerful computing environment, the sophistication of scripting languages has improved dramatically. But creating applications using just a scripting language may be not so feasible in applications where intensive computation is required, due to the run time inefficiency of scripting languages [1].

Application development may be done in a much faster and convenient way by adapting the power of scripting that is, gluing components together. By using existing code written in C/C++, extensions may be easily built and used within the scripting environment to develop applications without run time performance degradation, and without having to go through details of the compiled language programming [1, 2, 3].

Previously, David M. Beazley at the University of Utah applied this technique, using a scripting language to control components written in compiled languages, to his research on physics application development, in particular, molecular-dynamics simulations. He used Python to control components developed with C. The application achieved 10 Gflops sustained performance [4, 5].

2.1. Scripting Languages

In this paper, by scripting, we refer to programming with scripting languages.

Compiled languages refer to programming languages that employ compilers to produce machine dependent binary code, such as C and C++.

Scripting is more of a programming methodology than a technical term. Thus classifying scripting languages into any of the traditional language paradigms [6] is inappropriate. People have their own understanding of scripting languages, and here are some of those.

> Scripting languages support doing programmatically what otherwise is done directly by the user through direct commands [7].

> Speaking as a computer scientist, my answer is: These are not (yet) technical terms. Speaking as a linguist, my answer is: These words (like most words) are defined by prototype, not by boundary. A script is what you give the actors, and a program is what you give the audience [8].

> A programming language that is supported by and specific to a particular program. Note: A scripting program is normally used to automate complex or advanced features or procedures within the program [9].

Even though it is difficult to categorize scripting languages, there are some common characteristics shared among many scripting languages [7].

> Scripting languages often follow the syntax and semantics of command languages. For instance, many scripting languages do not require quoting of string literals, but rather require explicit evaluation of variables.

> Scripting languages make it easy to call system commands, prepare their arguments, and manipulate their results. They generally have some built-in primitives for manipulating file and directory names, argument lists, environment variables etc.

> Scripting languages generally are good at handling strings, and don't emphasize numerical manipulation.

> Since calling system commands is generally much more expensive than script execution itself, there is little emphasis on run-time efficiency, therefore they are often implemented using interpreters, byte-code interpreters, or macro processors.

Scripting is different from programming with other languages, and scripting is better considered as part of a development framework rather than understanding it as a stand-alone development method. Some of the reasons are as follows.

First, scripting languages assume that there are available components to handle complicated operations. Because of this reason, scripting languages often have facilities to extend themselves to be used with components written in compiled languages. Compiled languages offer better run time performance than scripting languages. But coding with compiled languages is less productive and requires more care [1].

Second, scripting languages are often syntactically simpler than most compiled languages, since the number of built in operators and data types are not comparable to those of compiled languages. Simpler syntax encourages readability and manageability. Thus adapting scripting into a development environment may add higher-level abstraction to codes written in compiled languages while achieving good manageability.

There is always a demand for a simple programming language to be used in a specific problem domain. Such a language is a bit more powerful than shell programming, but without the complicated learning process and development overhead of compiled languages, such as C and C++. Scripting languages are often developed to serve this need, and some of them evolve into full-fledged languages. Perl, Python, and Tcl are the most widely used full-fledged scripting languages.

2.1.1. Perl

Created by Larry Wall in the late 1980s to extract text from news messages, Perl stands for Practical Extraction and Report Language. Its use of powerful regular expressions makes Perl a good tool for text manipulation. That is why Perl became a model for CGI (Common Interface Gateway) language, which is used to control server side inclusion and to perform various server side operations by doing massive text manipulation. Perl serves this purpose very well as it is quite fast in such operations among scripting languages [1, 10]. Even though it is a powerful and widely used scripting language, many often criticize its syntax, which does not promote good readability. Examples and explanations about the problematic Perl syntax are presented in details at Perl website [11].

2.1.2. Python

Created by Guido van Rossum in the early 1990s, Python was designed as an object oriented language to link shell and C programs. Its easy syntax, elaborate library, portability, extensibility and embeddability make this language very popular. Python is a portable, interpreted, object-oriented programming language. The language has an elegant yet rich syntax and a small number of powerful high-level data types. Python can be extended in a systematic fashion by adding new modules implemented in a compiled language such as C or C++. Such extension modules can define new functions and variables as well as new object types [12].

2.1.3. Tcl

John Ousterhout created Tcl in the late 1980s as an embeddable command language for interactive tools. When supplemented with the Tk toolkit, it became popular as the fastest way to build graphical user interfaces on Unix [13]. Following are the design goals of Tcl from John Ousterhout [14].

> The language must be extensible: it must be very easy for each application to add its own features to the basic features of the language, and the application-specific features should appear natural, as if they had been designed into the language from the start.
>
> The language must be very simple and generic, so that it can work easily with many different applications and so that it doesn't restrict the features that applications can provide.
>
> Since most of the interesting functionality will come from the application, the primary purpose of the language is to integrate or "glue together" the extensions. Thus the language must have good facilities for integration.

2.2. Mechanisms for Extending Scripting Languages

Programming languages may offer extensibility through one or more of the following three mechanisms [15].

The first option is LISP's way of offering extensibility. New commands are implemented with the language itself and become the part of the language. This mechanism might be convenient, since there is no external Application Programming Interface (API) to take care. But considering that the purpose here is to find a way to build a RAD environment using existing C/C++ libraries, this mechanism is not applicable in our research.

The second option is to extend the language with other languages through a well-defined API. Since built in commands of programming languages are implemented with compiled languages, extensions are also implemented using compiled languages. Upon implementing extensions, extensions are compiled with the language itself to become the part of the language as built in commands.

The third option is dynamic loading of extensions. Dynamic loading allows dynamic inclusion of an implementation through defined APIs with compiled extensions during run time.

The second and the third options are more appropriate in our research, since both allow the use of C/C++ libraries. Extensions built with the second option are called static extensions, while with the third option are called dynamic extensions. Advantages and disadvantages of static and dynamic extensions are as follows;

Static Extensions.

Advantages:

- Extensions become part of language and that changes syntax as desired.
- Faster execution, since external data mappings and conversions are not required during run-time.
- Single distribution package with all extensions.

Disadvantages:

- Statically linked runtime may not be compatible with some extensions.
- Statically linked runtime can cause namespace or global symbol table collision.
- Statically linked runtime is often not compatible with integration tool kits.

Applicable situation:

- Building a proprietary scripting environment for a certain application that does not require modification or integration.

Dynamic Extensions.

Advantages:

- These extensions are much easier to manage since they are in the form of packages.
- Object oriented programming up to a certain level can be achieved by organizing packages.
- Managing symbol tables and global variables is easier since each module has its own space to store this information.

Disadvantage:

- Compiling dynamic extensions is difficult. See Chapter 5.

Applicable situation:

- Dynamic extension mechanism may apply to any situation as long as the target system supports loading dynamic extensions.

2.3. Extension Building Tools

Extensions may be built with compiled languages using API provided by scripting languages. Or extensions can be created using existing source code with interfaces, which translate between source code and scripts. Interfaces allow a scripting language to call compiled objects by specifying how a scripting language makes a call to a function within an extension, or how variables are passed between a scripting language and an extension.

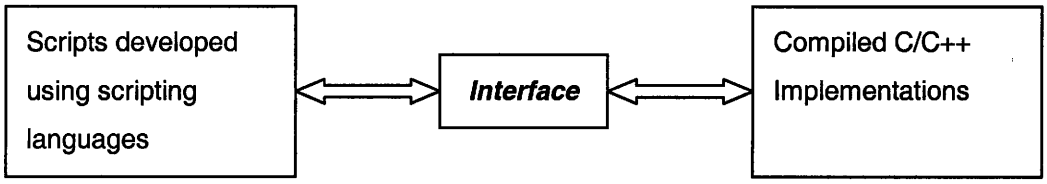

Figure. 2.3.1 Interface

Figure. 2.3.1 shows the role of the interface in scripting environment. Scripting languages that we have reviewed have their own method to implement interfaces to existing C/C++ source code or compiled binary code. An interface is also called a wrapper.

A wrapper can be made either manually or automatically, using wrapper generators. There are some advantages to using wrapper generators. First, coding a wrapper often requires understanding of details. Second, manual wrapping may require changes in source code, and that can introduce new bugs into source. Finally, manual wrapping makes big projects hard to maintain since language evolution often changes the interface protocol. If that happens, manually written parts need to be updated.

Here is an example of a wrapper for Tcl. Consider the following C function:

```
int foo(int num) {
        return 0;
}
```

In order to make the above function available to Tcl, a wrapper should collect argument information, invoke the function, and provide a return value that is recognized by Tcl. A sample wrapper for above function is as follows.

```
int wrap_foo(ClientData clientData, Tcl_Interp *interp,   int argc, char *argv[]) {
        int _result;
        int _arg0;
        if (argc != 2) {
                interp->result = "wrong # args";
                return TCL_ERROR;
        }
        _arg0 = atoi(argv[1]);
        _result = foo(_arg0);
```

```
        sprintf(interp->result,"%d", _result);
        return TCL_OK;
}
```

In addition to the above wrapper, an initialization portion of code also needs to be provided. An initialization function is a must. It tells Tcl about newly added commands. Whenever an extension is loaded into Tcl, Tcl searches for an initialization function. If the initialization function is not found, Tcl returns an error. An example initialization function for the function *foo* is as follows.

```
int Wrap_Init(Tcl_Interp *interp) {
        Tcl_CreateCommand(interp, "foo", wrap_foo, (ClientData) NULL,
                                (Tcl_CmdDeleteProc *) NULL);
        return TCL_OK;
}
```

Once all the above three functions are compiled , linked, either statically or dynamically, and properly loaded into Tcl, function *foo* is available as if it were a built in command.

Wrapper generating tools may require an input to generate a wrapper. The input is often referred to as interface definitions. Interface definitions consist of a header and a body. The header contains attributes that apply to the entire interface, and the body contains the remaining interface definitions. Interface definitions may be considered as rules to generate wrappers.

Interface Definition Language (IDL) is the most popular language to create interface definitions for COM and CORBA components. Both COM and CORBA are software architectures that allow applications to be built from binary software components and are

used in most RAD tools, such as Microsoft Visual Basic and Borland Delphi.

Using IDL and advanced programming architectures like COM and CORBA would help increasing security and portability, but there are two reasons why IDL and such RAD tools are not appropriate in certain RAD environments where testing an idea or an algorithm is more important than developing a stable commercial product. First, learning and using IDL takes too much time. Second, most existing RAD tools use IDL compilers to generate skeleton code from IDL files, but users need to fill in major portions of the generated files.

There are many wrapper generators that can be used with C/C++ libraries, such as SWIG, jWrap, Mktclapp, and SILOON. Unlike IDL compilers, these tools do not require any modification of the interface that is generated by the tools.

2.3.1. SWIG

SWIG stands for Simplified Wrapper and Interface Generator. Initially developed by David M. Beazley at University of Chicago while he was doing research on Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations at Los Alamos National Laboratory. SWIG takes an existing C/C++ library and makes an interface to many different scripting languages. By generating wrappers from the given C/C++ header files, SWIG allows scripting languages to access the underlying C/C++ code. Currently SWIG supports Unix, Windows and Macintosh environment with any ANSI compliant C/C++ compiler [16].

2.3.2. jWrap

jWrap is very similar to SWIG., but it only supports Tcl. JWrap understands more C++ advanced features than SWIG. JWrap can handle structures of structures and polymorphism. Unfortunately, jWrap is not maintained [17].

2.3.3. Mktclapp

Mktclapp was developed by D. Richard Hipp to create standalone executables from Tcl script and C/C++ code. For a C/C++ program, and an associated Tcl script, Mktclapp makes a combined binary executable. But for the interactive development or scripting with C/C++ libraries, this tool is not quite suitable. This tool becomes very handy when programmers want to generate distribution packages or give scripting capability to C/C++ code [18].

2.3.4. SILOON

SILOON stands for Scripting Interface Languages for Object-Oriented Numerics. It is a project at Advanced Computing Laboratory to make object-oriented numeric class libraries externally accessible via run-time scripting. SILOON has a very ambitious goal: making a huge set of libraries available to scripting languages without manually analyzing sources. For that reason, the SILOON project is focused on establishing a networked connection with an automatic code analyzer and library repository. In order to use SILOON users must download the Program Database Toolkit, which does C/C++ source code analysis [19]. Currently SILOON makes interfaces to Perl and Python.

CHAPTER 3

SCRIPTING IN RAD

Initially, our goal was to design a simple programming language to be used in the computational geometry field as a prototyping tool. Our considerations on designing a language focused on following two aspects: first, limiting the number of built in operators and data types to give the user an easier environment in which to develop algorithms rather than spending time on learning a language to test his or her idea; and second, allowing the use of C/C++ libraries to provide more a flexible and programmable environment. While designing such a language, we discovered some problems associated with data type conversions during operations performed through external APIs. In order to find solutions to such problems, we studied existing extension mechanisms from various scripting languages and found many different approaches to the problems. Later, we changed our plan to use an existing scripting language and an interface generating tool to meet our design considerations to have a more stable development environment with sufficient support from both the language community and the tool community. Our initial design of a programming language helped us in choosing a scripting language and a tool for our RAD model.

3.1. Choosing a Scripting Language and a Tool

In our RAD model, extensions are considered as inputs and these may be divided into inputs, functions or applications. Input level components represent data types or data structures implemented in C/C++, such as classes, linked lists, or trees. Function level components are operations using input level components, such as copying structures, merging linked lists, and printing trees. Applications level input consists of algorithm implementations that use a collection of functions, data structures and data types.

3.1.1. Selection Criteria

Our selection criteria are based on universal usability. In other words, we have preferred simplicity of grammar and ease of use to extra functionality or better support for advanced programming concepts. Complicated data structures and procedures may be built with C/C++ as extensions. As such, the two major criteria are simplicity and extensibility.

- **Simplicity of grammar**: The number of operators and readability of syntax may measure the simplicity of a grammar. Similarity with widely used programming languages does not necessarily mean simplicity.

- **Extensibility**: Specific support for C/C++ libraries, since C/C++ are heavily used to develop components for CPU intensive operations. Supporting use of such available resources will speed up the development process as well as achieve better run-time performance.

Also there are some minor issues in choosing a language. These features are desirable but not mandatory. They are embeddability and availability of contributed

repository.

- **Embeddability**: If a language offers embeddability, developers can easily produce distribution packages converting scripts into system programming languages. Furthermore, application-specific scripting environments may be created easily by using the embeddable part, run-time to be exact, even after packaging.

- **Contributed repository**: If a well-maintained repository is available, many tasks can be easily automated by using existing scripts with minimal modification. This may cut down development time since finding necessary modules and/or scripts does not take long. Also beginners can learn how to program by looking at scripts from a repository as examples.

3.2. Why Tcl

Since Tcl was developed to control components written in compiled languages, Tcl provide a more convenient and easier way of extending the language itself as well as embedding itself into other projects done with compiled languages.

We selected Tcl for our RAD model because of the following reasons.

- **Simplicity**: Tcl is a simple language to learn and use. Even though people criticize its syntax due to unusual style with respect to other system programming languages, most of them agree that the syntax of Tcl is easier and simpler than other scripting languages [20, 21, 22]. Tcl has fewer data types. Although this often is a cause of run-time inefficiency due to massive type conversion, fewer data types make scripting a lot easier and more convenient than using compiled languages with many data types.

- **Extensibility**: Tcl supports both static and dynamic extensions [23].

- **Script Repository**: Tcl has the central repository maintained by Scriptics for the contributed scripts, which covers variety of different problem domains [24].

- **Tk**: Tk is a graphical user interface (GUI) extension originally developed for Tcl. Even though many other scripting languages ported Tk into their languages, since Tk was developed for Tcl, users can take advantage of many existing Tk widgets not available for other platforms when it is used with Tcl.

- **TclPro**: Recently Interwoven acquired Ajuba Solutions, previously Scriptics, and put TclPro into the public domain. TclPro is a commercial version of Tcl, which comes with powerful development tools, such as the Tcl compiler and a GUI debugger.

3.3. Why SWIG

After reviewing some of the possible tools that could be used to create extensions from C/C++ libraries, we decided to use SWIG. Even though SWIG does not support C++ advanced features such as template and polymorphism, there are more reasons to use SWIG over the other tools that we reviewed [25, 26].

The following summarizes key features of SWIG.

- SWIG runs on Unix, Windows and Macintosh and interfaces with five popular scripting languages. The other tools that we have reviewed mostly support one or two scripting languages.

- SWIG may be used with any ANSI compliant C/C++ compilers, and it requires minimal programming knowledge to use. SWIG only requires C/C++ header file to generate wrapper while the other tools are more tied to implementation parts.

- SWIG requires simpler input than any other existing tools with sufficient help documentation. SWIG uses ANSI standard C/C++ style input. Compared to the other tools, it is has the simplest interface requirement.

3.4. Scripting RAD Model

A RAD environment can be created with a scripting language and a tool extending the language. Such a RAD environment may become simpler, yet more programmable and flexible than popular RAD tools. Differences between the approach of traditional RAD tools and our RAD model are as follows.

First, traditional RAD tools require an interface definition with IDL, a new language that has over 140 keywords and takes significant time and effort to learn, while our RAD model takes an ANSI C style interface definition that is an input to SWIG. An example of an IDL file may look like this.

```
[
  uuid (ba209999-0c6c-11d2-97cf-00c04f8eea45),
  version(1.0),
  pointer_default(unique)
]
interface cxhndl
{
  typedef [context_handle] void *PCONTEXT_HANDLE_TYPE;
  short RemoteOpen(
      [out] PCONTEXT_HANDLE_TYPE *pphContext,
      [in, string] unsigned char *pszFile
  );
  short RemoteClose( [in, out] PCONTEXT_HANDLE_TYPE *pphContext );
  void Shutdown(void);
}
```

And the equivalent input for SWIG may look like this.

```
%module cxhndl
%typedef void *PCONTEXT_HANDLE_TYPE;
```

```
extern short RemoteOpen(PCONTEXT_HANDLE_TYPE*, unsigned char*) ;
extern short RemoteClose(PCONTEXT_HANDLE_TYPE*);
extern void Shutdown(void);
```

Second, interfaces generated from IDL require further work, since IDL compilers generate only skeleton code. On the other hand the interface generated from SWIG does not require any modification and it is ready to use. Thus we can take one step further to connect this to automatic extension generation.

Overall, traditional RAD tools require both source code preparation and interface coding along with interface definition, while our RAD model only requires small modifications on source code and interface definition.

In addition to the above differences, using a scripting language adds the following characteristics to our RAD model;

- **Programming**: A scripting language serves as a host in our RAD environment, which simplifies programming effort and cuts development time. A scripting language is used to control and layout components of all levels.

- **Abstraction**: A scripting language can add higher-level abstraction to an application while hiding details of code within extensions.

- **Tools Integration**: Scripting languages often come with facilities to perform tools integration, since this was the main purpose to begin with. When application level components are used in our RAD model, users may easily integrate those components with simple scripting.

- **Encapsulation**: Extensions are in form of compiled modules. All data member and operations related to a module are encapsulated within a compiled component.

- **Deployment**: Developed applications may be distributed in the form of scripts or

compiled byte-code, along with compiled extensions. A programmer may need to provide different extensions for different operating systems.

CHAPTER 4

CASE STUDY

We have implemented applications to explore the use of tools that we chose. The possibility of using these tools to solve real world problems was examined through the application development process.

The first application is to examine the practical usability of Tcl and SWIG as an application development environment, and the second application is to see if SWIG handles C implementations without requiring users to modify source code in detail.

4.1. Platforms

To ensure the usability of the scripting environment that we have selected, application development is done on the two systems described below. Applications have been thoroughly tested and run without error.

4.1.1. First Platform

- Computer: Intel Pentium III 600MHz
- Memory: 256MB
- OS: Red Hat Linux Release 6.2 <Zoot> Kernel Version 2.2.14-5.0
- Compiler: GCC version egcs-2.91.66 19990314/Linux <egcs-1.1.2 release>
- SWIG: version 1.3u-20001025-2235 <Alpha 1> compiled with GCC
- Tcl: 8.0

4.1.2. Second Platform

- Computer: Intel Pentium III 500MHz
- Memory: 128MB
- OS: Microsoft Windows 2000 Professional
- Compiler: GCC version 2.95.2 for Mingw32
- SWIG: version 1.1-883 compiled with Microsoft Visual C++ 6.0
- Tcl: 8.3 with Mingw32 patch

4.2. Ray Tracer

A ray tracer is an application to draw photo-realistic 3 dimensional images by testing the relationship among rays, objects and lights. For this testing, a huge number of computations are required on points, vectors, and objects. That is the reason that we chose this application to check the possibility of using a selected scripting environment as an application development tool. For our experiment, we used a six hundred by six hundred pixel scene and one object with neither visual effects nor speed up algorithms, such as specular reflection or scan line ray tracing, respectively. The algorithm implemented for the experiment is as follows [28].

```
Set up a sphere == Sph;
Set up a position of viewer == V;
Set up a location of light == L;
Set up a screen == S;
For I=0 to I==Width of screen in pixel -1
    For J=0 to J==Height of screen in pixel –1
        If a ray from V thru S(I,J) hits Sph
                Set brightness on S(I,J);
        Else Set brightness to background color;
```

4.2.1. Development of Ray Tracer

We chose five C++ libraries written by Wilbon Davis to build data structures and essential operations as shared objects to implement the Ray tracer algorithm [29]. To work with SWIG, we modified these libraries because SWIG does not support private attributes, friend functions, and polymorphism. The libraries are point, vector, ray, sphere, and psgray. We used a simple gray scale Postscript library for visualization of output.

Here is an example header file that we have changed.

```
Before changes:
class point3 {
  float x, y, z;
public:
  point3(float, float, float);
  point3(void);
  float abs();
  float xx();
  float yy();
  float zz();
  friend vector3 operator-(const point3 &, const point3 &);
  friend point3 operator+(const point3 &, const vector3 &);
  friend point3 operator+(const vector3 &, const point3 &);
};

After changes:
class point{
public:
  float x, y, z;
  point(float, float, float);
  point(){};
  void setpoint(point);
```

```
    float abs();
};
vector point_sub(point, point);
point vector_add(point, vector);
```

We manually changed all data members and methods to public components, for two reasons. One is to prevent access violations due to the limitations of SWIG. SWIG bypasses all private members and methods when it wraps a class, and SWIG does not work with friend functions [25]. The other is to retain some level of data encapsulation by keeping methods within a class, which provides an object like interface to a scripting language.

In case of constructors, SWIG only wraps the first to be accessed from a script. We include a dummy default constructor in the header file to cheat the C++ compiler. C++ compilers do not compile sources if an object has overloaded constructors but no explicitly defined default constructor.

We manually renamed overloaded operators and functions. SWIG offers many different ways to perform these tasks [25], such as renaming overloaded function names during wrapping and adding a function to translate overloaded function names during run-time. We renamed *friend vetor3 operator-(const point3&, const point3&)* to *point_sub*, *friend point3 operator+(const point3&, const vector3&)* to *vector_add*, and removed *friend point3 operator+(const vector3 &, const point3 &)*. Another example shows what changes we have applied to the implementation files according to changes on header files.

```
Before changes:
point3::point3(float a, float b, float c){
  x = a; y = b; z = c;}
```

```
point3::point3(void){
  x = y = z = 0;}
vector3 operator-(const point3 & p, const point3 & q){
  return vector3(p.x-q.x, p.y-q.y, p.z-q.z);}
float point3::abs(void){
return sqrt(x*x + y*y + z*z);}

After Changes:
point::point(float a, float b, float c){
  x = a; y = b; z = c;};
void point::setpoint(point p){
  x = p.x; y = p.y; z = p.z;};
float point::abs(){
  return sqrt(x*x + y*y + z*z);};
vector point_sub(point p, point q){
  return vector(p.x-q.x, p.y-q.y, p.z-q.z);};
point vector_add(point p, vector d){
  return point(p.x+d.x, p.y+d.y, p.z+d.z);};
```

Since SWIG does not support function overloading, we chose just one of the constructors, which is *point(float, float, float),* because this is the only constructor invoked by external clients to initialize point class. We have added a function *void setpoint(point)* to replace the default constructor, and *point vector_add(point, vector)* was moved from the vector library to the point library.

Point library has a data structure to store x, y, and z values and functions which perform arithmetic operations with its data structure. Once we wrap the point library with SWIG, the following functions are created either from the library or by SWIG.

```
Module Point:
point_x_set self x
point_x_get self                          [ Member data: returns float  ]
point_y_set self y
```

```
point_y_get self                          [ Member data: returns float   ]
point_z_set self z
point_z_get self                          [ Member data: returns float   ]
new_point { float   } { float   } { float   }   [ Constructor: returns point * ]
point_setpoint self { point * }           [ Member : returns void   ]
point_abs self                            [ Member : returns float   ]
point_sub { point * } { point * }         [ returns vector   ]
vector_add { point * } { vector * }       [ returns point   ]
```

Except *new_point* and *vector_add*, all the other function names start with point. This naming convention will give the users a better idea about which call belongs to which library. Since all the functions listed above were packed as a shared object module, utilizing SWIG's naming convention along with careful function naming can create object oriented programming environment, at least from the interface point of view.

After compiling all the libraries into shared object modules, we developed a ray tracer with Tcl. We loaded those modules into Tcl and implemented the algorithm presented earlier. While implementing the algorithm, we could take great advantage of the Tcl interpreter. Tcl run-time gave us prompt responses upon each statement, significantly speeding up development. The Tcl interpreter contributed in two ways. First, we were able to fix syntactic errors within Tcl run time without going through source code to find mistakes or arguments incompatible to a function. Second, in case we wanted to see a function return, we were able to examine the function without writing a driver part as we do in C/C++.

Using extensions from Tcl environment may start with loading extensions to Tcl environment like this.

```
% load ./vector.so
% load ./point.so
```

Upon successful loading, we can check newly added commands by invoking the info function from Tcl environment. The return of the command serves as a simple reference of newly added commands.

```
% info command point*
point_x_get point_z_get point_x_set point_z_set point_setpoint point_y_get point_y_set
point_abs point_sub
% info command vector*
vector_x_get vector_z_get vector_x_set vector_z_set vector_mul vector_diff vector_abs
vector_y_get vector_add vector_norm vector_y_set vector_setvector vector_sum
```

When a command is used with missing arguments or incompatible data type arguments, the Tcl environment returns error messages as follows.

```
% point_setpoint
Wrong # args. :point_setpoint self { point * }   argument 0
% set A_vector [new_vector 1.5 1.5 2.5]
_8062528_vector_p
% point_x_get $A_vector
Type error. Expected _point_p:point_x_get self   argument 0
```

Here is a script that implements the ray tracing algorithm.

```
set Obs [new_point 0 0 12]
set Center [new_point .3 .4 -6]
set Light [vector_norm [new_vector 2 10 5]]
set sphptr [new_sphere $Center 2.5]
set screen [new_psgray 600 600 6 6]

for {set i 0} {$i < 600} {incr i 1} {
  for {set j 0} {$j < 600} {incr j 1} {
    set a [new_point [expr $i*.01-3] [expr $j*.01-3] 0]
```

```
        set a [point_sub $a $Obs]
        set rayptr [new_ray $Obs [vector_norm $a]]
        set t [intersect $sphptr $rayptr]
        if {$t==-1.0} {
          set B .3
        } else {
          set a [ray_at $rayptr $t]
          set a [point_sub $a $Center]
          set a [dot_product [vector_norm $a] $Light]
          set a [max 0 $a]
          set B [expr .64*$a+.16]
        }
        psgray_setpixel $screen $i $j $B
      }
    }
```

To investigate run time efficiency we also developed the same ray tracer using C++. The C++ version of the ray tracer outperforms the Tcl version as we expected. But once we rebuilt an extension with the ray tracer algorithm implemented in C++, the difference between execution time of the compiled executable and the extension invoked within Tcl almost disappeared. Execution times measured under the Windows 2000 system is as follows.

- Tcl Version with the algorithm implemented in Tcl 102 seconds
- Tcl Version with the algorithm implemented in C++ 1.5 seconds
- C++ Version 1.3 seconds

Also we measured execution times under Linux system.

- Tcl Version with the algorithm implemented in Tcl 53 seconds
- Tcl Version with the algorithm implemented in C++ 1.2 seconds
- C++ Version 0.8 seconds

The first Tcl version uses input level components and implements the algorithm within the Tcl environment. The algorithm uses a doubly nested for loop with many calls to external components. This demonstrates that the level of abstraction of the source classes and methods determines the runtime performance of applications written with a generated interface. High frequency calls to extensions can easily lead to unsatisfactory runtime performance. Thus it will be necessary to provide higher-level inputs that combine the necessary components so that high frequency calls are not needed by the scripts.

4.3. Voronoi diagrams

In the first application we modified the source codes written in C++ because SWIG does not support advanced features of C++. On the other hand, SWIG claims to be compatible with C [25]. So in the second application we examined whether C sources can work with SWIG without requiring users to understand source code.

The code used in our second experiment was written by Steven Fortune to compute Voronoi diagrams [30, 31]. This is a widely used 2D code for Voronoi diagrams and Delauney triangulations, based on Steven Fortune's sweepline algorithm. Voronoi diagrams are useful for various problem domains, such as nearest neighbor search, facility location, largest empty circle, and path planning. A Voronoi diagram is a geometric structure that represents proximity information about a set of points. The program computes Voronoi diagram with a set *S* of points *p_1,...,p_n*, and program returns the answer that is a decomposition of space into regions around each point, such that all the points in the region around *p_i* are closer to *p_i* than any other point in *S* [32,

33].

4.3.1. Working with the Voronoi code

We realized that the original Kernighan & Ritchie coding style function headers [34] used throughout the Voronoi code were not recognized by SWIG. We solved this problem by replacing those function headers with ANSI standard style headers.

```
Before change:
voronoi(triangulate, nextsite)
int triangulate;
struct Site *(*nextsite)();{...}

After change:
voronoi(int triangulate, struct Site* (*nextsite)()){...}
```

Even though SWIG was able to understand sources there were still two more problems. One was due to a complex pointer, such as function pointer, and the other was caused by arrays. We solved the first problem by adding a *typedef* to replace the function pointer with a void pointer, and the second problem by adding type-mapping codes to access array contents [25]. SWIG works smoothly after above modifications and we generated a dynamic linked library on our second attempt.

```
Code added to Voronoi interface file voronoi.i:
%include typemaps.i
%typemap(memberin)struct Site * [2] {
        $target=$source;
}
%typedef struct Site* (*NS_FUNC)();
```

```
Code added to Voronoi.c:
#ifdef SWIG
void voronoi(int triangulate, NS_FUNC nextsite)
#endif
#ifndef SWIG
void voronoi(int triangulate, struct Site* (*nextsite)())
#endif
```

When we tried to load the Voronoi extension into Tcl shell, we found that Tcl did not load a module with function declarations that lacked definitions. After we commented out non-defined function declarations, extension loaded properly and we were able to use functions from the voronoi source code.

4.3.2. Tcl Voronoi Script and Helper Functions

Upon loading the Voronoi extension to Tcl shell, we discovered small problems when we were trying to call the *main()* function from the Tcl environment. One was related with top level C function *main()* and the other was with the arguments of the *main()*.

The *main()* problem was solved during compilation of the extension by turning on the *–Dmain* option, which most of compilers have, and replaced the name *main()* with another name [35].

In order to pass arguments of the *main()* from Tcl we developed helper functions and a script. The helper functions were defined using SWIG's inline function defining facility to create a C style array, and the script was written with Tcl to change a Tcl list into a C style array. By doing this we could pass options to the C pre-compiled module as if they are *int argc* and *char **argv* [25].

Voronoi.tcl:

```
load ./voronoi.so
load ./carray.so
proc voronoi {option channel} {
        set argptr "\-$option $channel"
        cvoronoi 1 $[ChListToArray argptr]
}
proc ChListToArray {l} {
        set length [llength $l]
        set a [ch_array $length]
        set i 0
        foreach item $l {
                ch_set $a $i $item
                incr i 1
        }
        return a
}
```

Carray.i:

```
%inline %{
char *ch_array(int size) {
    return (char *) malloc(size*sizeof(char));}
char ch_get(char *a, int index) {
    return a[index];}
char ch_set(char *a, int index, char value) {
    return (a[index] = value);}
%}
%name(ch_destroy) void free(void *);
```

Voronoi.tcl takes an option and passes it to *cvoronoi*, which is equivalent to the top level C function *main()*. When it passes arguments to *cvoronoi*, it calls *ChListToArray*, which is actually a C implementation, to convert a list to a character array.

Upon experimenting the Voronoi code, we have tested a few C/C++ implementations from Weisses Data Structures [36]. Those are linked list, stack, and queue. We were able to generate DLLs from this source code. Generating DLLs from this code did not require any modification or helper function. This shows the possibility of using the system without any modification on source code.

CHAPTER 5

SIDE

5.1. Simple Integrated Development Environment

SIDE is an add-on utility to existing scripting languages and SWIG to provide more convenient and accessible development environment that we have explored. SIDE uses SWIG, scripting languages supported by SWIG, and an ANSI C compiler to form an IDE for RAD.

We initially developed SIDE for two purposes. First, it was developed as a stub generator to create interface templates that users can fill in easily. Second, under the Windows environment, SIDE works as a Dynamic-Link Library (DLL) generator that requires as little as two mouse button clicks. The current version of SIDE supports not only these but also following features.

- SIDE was developed using Tcl/Tk, which allows porting it to other operating systems.
- It has a clean and easy to understand GUI.
- It can load and save interface files from/to a file.
- It can select a preferred documentation form.
- It has built-in html style help file system.
- Navigation-help system is built into main window.

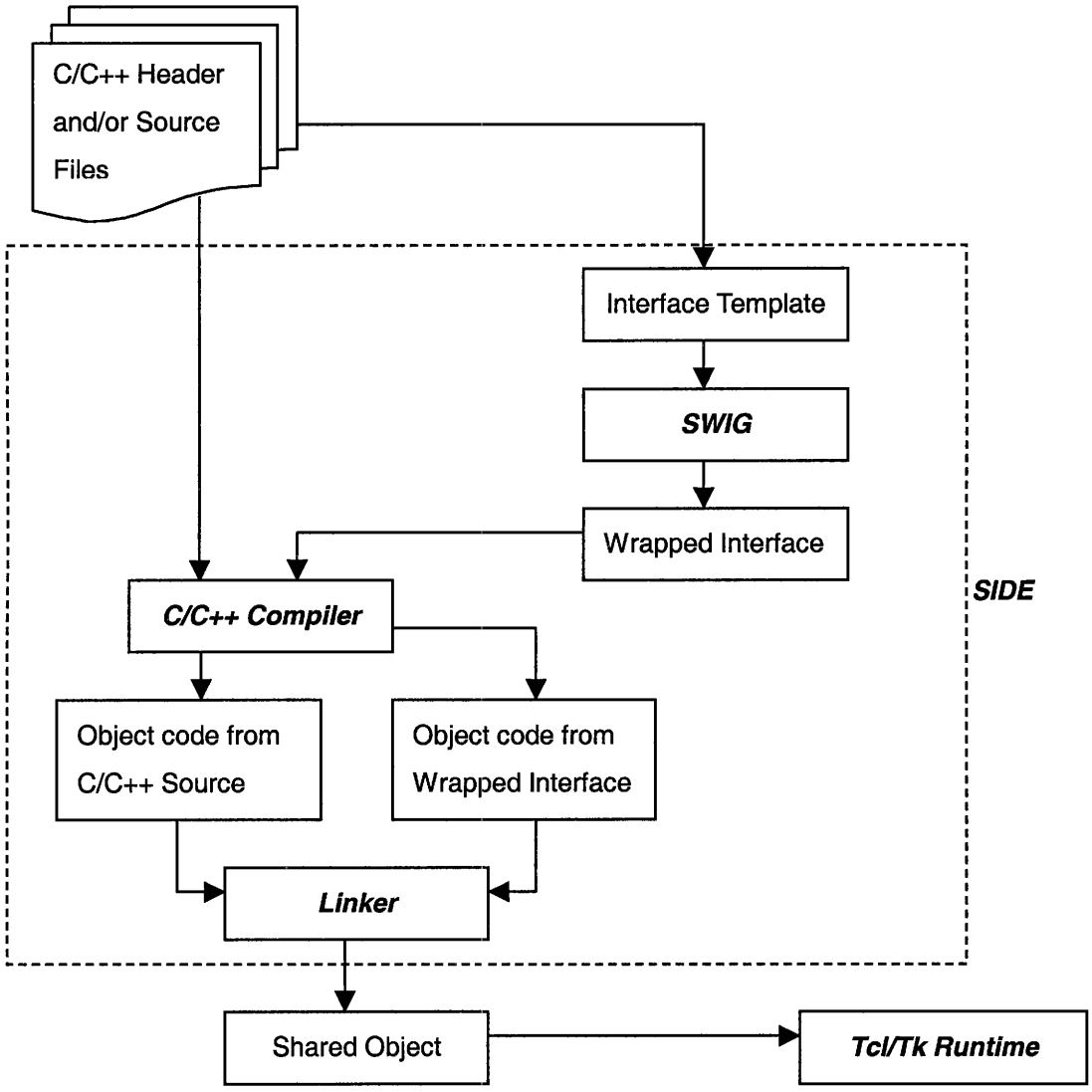

Figure. 5.1.1 SIDE

Figure 5.1.1 shows the layout of SIDE. SIDE includes SWIG, a C/C++ compiler, and a Linker to provide one simple development environment to users. SIDE takes existing C/C++ headers and/or implementations as inputs and generates a shared object as an output. Internally when input is given to SIDE, SIDE generates an interface template and provides it to SWIG as an input that is required by SWIG in order to

generate a wrapper. When the build command is issued by a user, SIDE generates a shared object by invoking SWIG, the C/C++ compiler and linker calls in appropriate order.

SIDE may be used in two different ways, as shown below.

The first scenario is to start from an existing interface file. In this case, load the existing interface file from menu option Load under File section, as shown in figure 5.1.2, set path and library include information, as shown in figure 5.1.3, set compile options as shown in figure 5.1.4, and build a shared object by choosing menu option Build under File section, as shown in figure 5.1.5.

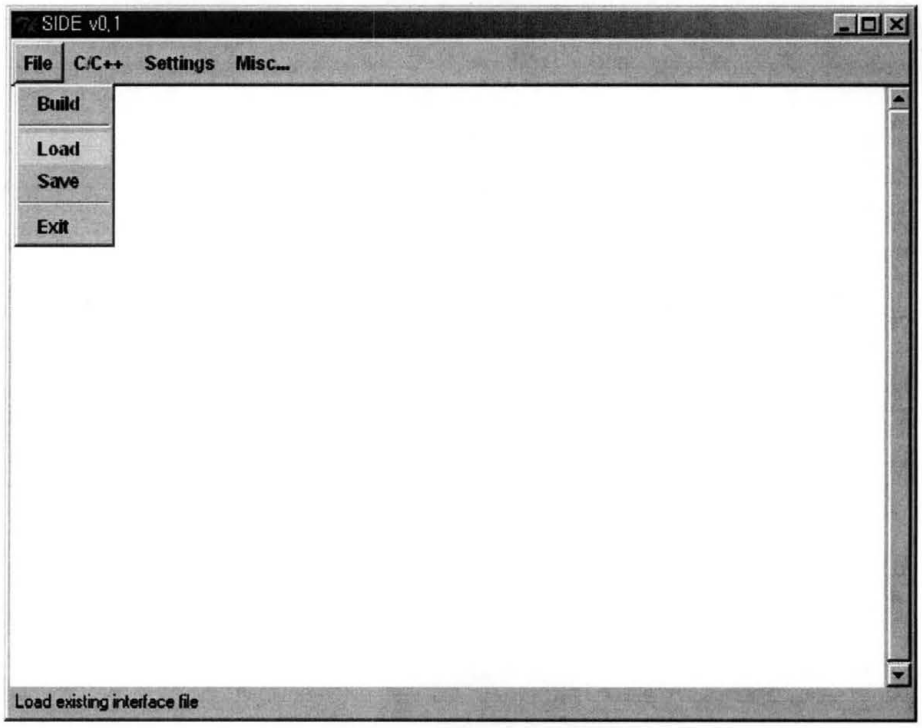

Figure. 5.1.2 Initial Program Screen

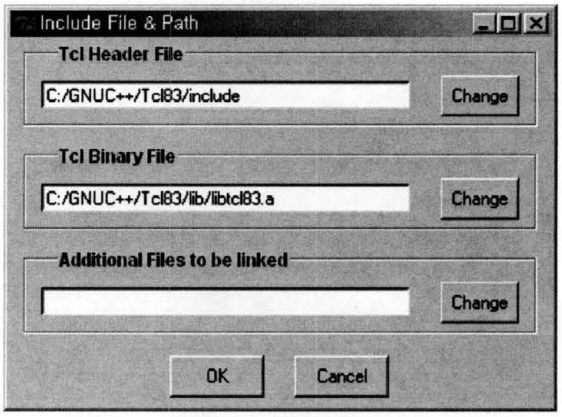

Figure. 5.1.3 Setting the Include File & Path

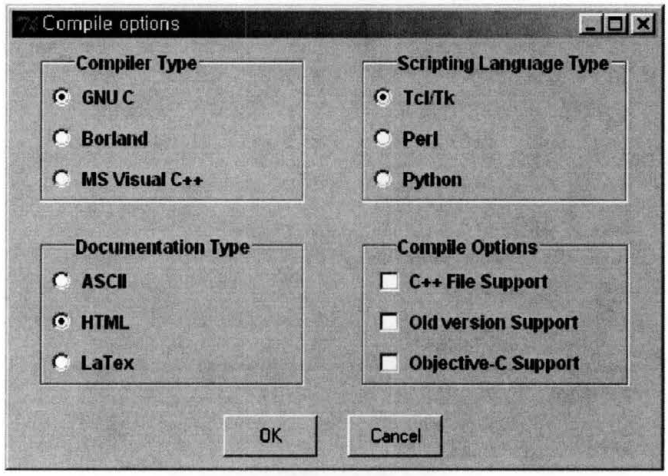

Figure. 5.1.4 Setting the Compile options

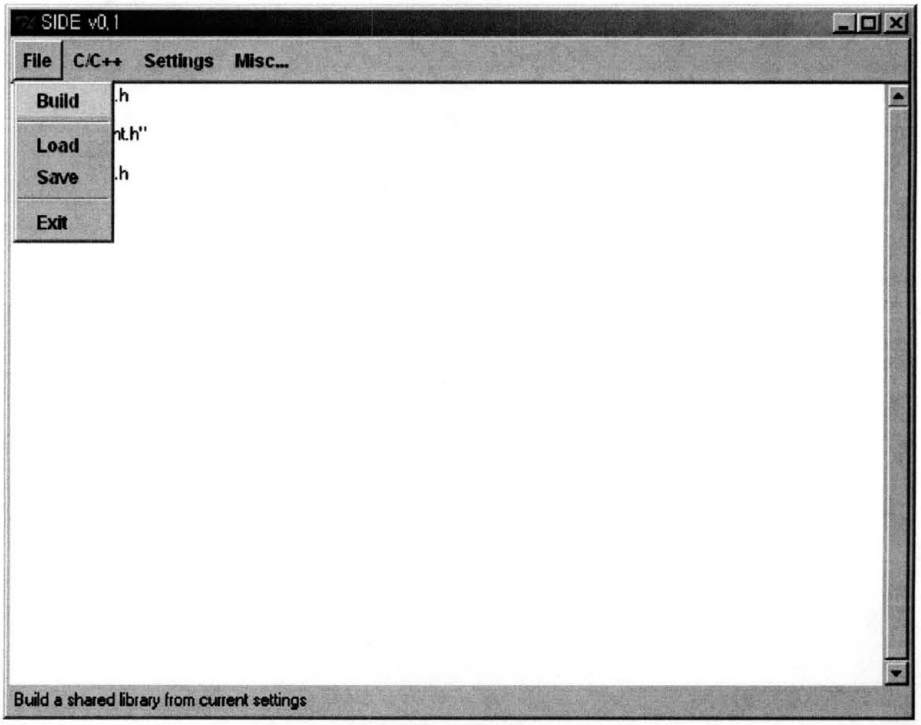

Figure. 5.1.5 Building a shared object

The second possibility is to start from ground zero and build a shared object. In this case, open SIDE and select Add Library or Add Functions from menu option C/C++. Upon reading a C/C++ source code, SIDE will display an interface file template on the main screen. In many cases, the user may build shared objects without editing this template interface file by choosing Build under menu option File. The user can edit contents in the main window, but the changes will not be saved unless the user selects Save or Build.

5.2. Generating Dynamic-Link Libraries

A Dynamic-Link Library (DLL) is a collection of modules that contain functions and data. It is the form of the scripting language extension under the Windows environment in our project. A DLL is loaded at run time by the scripting language. Under

UNIX systems, a shared object is the counter part of a DLL. It is fairly simple to generate shared objects under UNIX systems, but in the Windows environment there is no simple way to generate DLLs. One of the main functions of SIDE is automatic DLL generation.

DLL generation requires resolving dependencies, including appropriate precompiled objects, and invoking many compiler and linker commands. The number of commands need to be invoked depends on the compiler. In our project we use the GNU ported Ming32 C/C++ compiler. In order to generate a DLL for Tcl from a simple hello world program written in C++, we need to invoke at least seven compiler and linker commands as follows [35].

```
swig -tcl -c++ helloworld.i

c++ -DBUILDING_DLL=1 -g -c -I<Tcl_Include> helloworld.cpp

c++ -Wl,--base-file,<base_file> -mdll -Wl,-e,_DllMainCRTStartup@12 \
    -o helloworld.dll helloworld.o helloworld_wrap.o <Tcl_Lib>

dlltool --base-file <base_file> --output-exp <exp_file> \
        --def helloworld.def

c++ -Wl,--base-file,<base_file> <exp_file> -mdll \
    -Wl,-e,_DllMainCRTStartup@12 \
    -o helloworld.dll helloworld.o helloworld_wrap.o <Tcl_Lib>

dlltool --base-file <base_file> --output-exp <exp_file>   \
        --def helloworld.def

c++ <exp_file> -mdll -Wl,-e,_DllMainCRTStartup@12 \
    -o helloworld.dll helloworld.o helloworld_wrap.o <Tcl_Lib>
```

SIDE replaces these calls with as little as one mouse button click by resolving file

dependencies and library inclusions upon adding a C/C++ file into the system. Detailed information about DLL specifications can be found in the Microsoft Developer Network Library [37, 38].

5.3. Setting up the Development Environment

We developed SIDE to be used under the Windows platform. The following steps show how to set up the development environment for Windows.

Download and install TclPro. Since SIDE uses many extensions in TclPro, installing TclPro would prevent most errors due to missing packages. Alternatively, install Tcl and add *[incr Tcl]*, *TclX*, and *[incr Widgets]*. TclPro and extensions are available at Tcl homepage [13].

Download a stable version of SWIG, not the latest version or beta version. Experimental versions often cause problems with compilers and operating systems. Compiling SWIG under the Windows system requires a makefile modification unless compiler is Microsoft Visual C++ or Borland C, since SWIG provides makefile templates for those two compilers.

Since SIDE invokes the compiler from a command line interface, the environment variables, such as bin path, include path, and lib path, must set correctly. SIDE gets default lib path and include paths from system variables. Default path variables may be changed within SIDE but SIDE will not change the system variables permanently. If the bin path for the scripting language and compiler are not found from environment variables than SIDE will fail to build shared objects.

5.4. SIDE integration plan

While developing SIDE, we identified some features which may be added to the system later. These possible improvements in next version of SIDE are as follows.

- Platform independence by implementing OS checking and extending initialization to cover various environments.

- Semi-automatic interface template generation that overcomes the limitations of SWIG by validating the source code. For instance, if a source contains a complex function definition, then SIDE will generate a stub to replace the complex function definition with a pointer.

- Database facility to keep track of shared objects created by SIDE.

- Project template to expedite application development by providing shared objects from the database.

CHAPTER 6

CONCLUSION

6.1. Summary

We have demonstrated that the scripting RAD model using SWIG and Tcl could be used to build extensions from existing C++ libraries and to develop a CPU intensive application like RayTracer.

Total development time of the Tcl version of RayTracer including time taken to prepare the modules is roughly half of the time taken to develop the C++ version of the same application. Although total development time may depend on developer's experience and expertise with the given development environment and problem domain, our result is consistent with a recent article comparing development times of scripting languages and compiled languages [1].

Even though the Voronoi code that we have used here is a highly recommended program in computational geometry field, since it was developed quite a while ago, the code itself was not easy to understand due to its use of different style of programming. But without understanding the details of the Voronoi diagram algorithm, by using mechanisms provided by SWIG, we were able to build a shared object to be used as an extension within Tcl. SWIG could handle C very well, which confirms usability of this tool in a RAD environment.

Overall, we conclude that the use of a scripting language, Tcl in particular here, and SWIG may serve as an efficient alternative to existing RAD tools, such as Microsoft Visual Basic and Borland Delphi.

6.2. Future Work

The next version of Tcl will include *[incr Tcl]*, either as a plug-in extension or as a built-in feature [39]. With this improvement, Tcl will support object oriented programming and complex data structures.

It is possible to overcome the limitations of SWIG by applying techniques described in the SWIG manual, or by obtaining solutions from the SWIG community. The SWIG community often provides surprising solutions for the limitations of SWIG, such as polymorphism. SWIG will directly support these techniques in the future. But since SWIG is solely a voluntary work it is hard to say when those techniques will be added to the system. Meanwhile the techniques from either the SWIG manual or the SWIG community will be implemented in SIDE.

SIDE may be used to expedite the application development process by shortening the time required for building extensions, and once SIDE is improved, as we described in Chapter 5, it will become a full-fledged IDE for RAD.

Appendix

Attached disk contains following files.

- Examples
  - RayTracer
    - Executable
      - C_RayTracer.tcl
      - Tcl_RayTracer.tcl
      - RayTracer.exe
    - Extensions
      - Linux
        - point.so
        - psgray.so
        - ray.so
        - RayTracer.so
        - sphere.so
        - vector.so
      - Windows
        - point.dll
        - psgray.dll
        - ray.dll
        - RayTracer.dll
        - sphere.dll
        - vector.dll
    - Interface Files
      - point.i
      - psgray.i
      - ray.i
      - RayTracer.i

- sphere.i
- vector.i

Modified Source code

- C_RayTracer.cpp
- point.cpp
- point.h
- psgray.cpp
- psgray.h
- ray.cpp
- ray.h
- RayTracer.cpp
- RayTracer.h
- sphere.cpp
- sphere.h
- vector.cpp
- vector.h

Original Source code

- point.cpp
- point.h
- psgray.cpp
- psgray.h
- ray.cpp
- ray.h
- sphere.cpp
- sphere.h
- vector.cpp
- vector.h

Voronoi

Extensions

- carray.so
- voronoi.so

Interface Files

carray.i

voronoi.i

SIDE

SIDE0.1.tcl

SWIG

Contains Pre-compiled SWIG for Windows

Tcl83

Contains Tcl 8.3 with patch for Ming32 GNU compiler

SIDE expects to be used with Ming32 GNU compiler in current version. Ming32 GNU compiler may be obtained from http://www.xraylith.wisc.edu/~khan/software/gnu-win32/index.html.

Tcl folder and SWIG folder may be copied over to a target machine. Please make sure that the SWIG folder copied into C:\SWIG or append the path with whatever the directory SWIG is located, and update Tcl folder location within SIDE.

References

[1] Lutz Prechelt. An empirical comparison of C, C++, Java, Perl, Python, Rexx, and Tcl for a search/string-processing program Technical Report 2000-5, 34 pages, Universität Karlsruhe, Fakultät für Informatik, Germany, March 2000

[2] John K. Ousterhout. Scripting: Higher Level Programming for the 21st Century, IEEE Computer magazine, March 1998

[3] David M. Beazley: (1996) SWIG: An Easy to Use Tool for Integrating Scripting Languages with C and C++, Proceedings of the USENIX Fourth Annual Tcl/Tk Workshop, Monterey, California, July 1996

[4] David M. Beazley: (1996) Lightweight Computational Steering of Very Large Scale Molecular Dynamics Simulations, Supercomputing '96 Conference Proceeding, Pittsburgh, PA. November 17-22, 1996

[5] David M. Beazley: (1997) Feeding a Large-scale Physics Application to Python, Presented at the 6th International Python Conference, San Jose, California. October 14-17, 1997

[6] Doris Appleby: (1991), Programming Languages: Paradigm and Practice, McGraw-Hill, Inc.

[7] Stavros Macrakis. Usenet message from comp.compilers, Mar 1995

[8] Larry Wall. Usenet message from comp.compilers, Mar 1995

[9] Bahorsky, R. (ed.), Official Internet Dictionary, Government Institutes. 1998

[10] The www.perl.com home page, http://www.perl.com

[11] Perl4 Gotchas, http://www.perl.com/CPAN-local/doc/misc/ancient/Gotchas4

[12] Python language home page, http://www.python.org

[13] Tcl/Tk home page, http://www.scriptics.com

[14] John K. Ousterhout. History of Tcl, http://dev.scriptics.com/advocacy/tclHistory.html

[15] Juergen Wagner. Re: GNU Extension Language Plans, Usenet message from comp.lang.tcl, October 1994

[16] SWIG homepage, http://www.swig.org

[17] jWrap homepage, http://www.fridu.com/Html/jWrap.html

[18] Mktclapp homepage, http://www.hwaci.com/sw/mktclapp/index.html

[19] SILOON homepage, http://www.acl.lanl.gov/siloon/index.html

[20] Richard Stallman. Why you should not use Tcl, Usenet message <9409232314.AA29957@mole.gnu.ai.mit.edu>, September 1994

[21] John Ousterhout. Re: Why you should not use Tcl, Usenet message <367307$1un@engnews2.Eng.Sun.COM>, September 1994

[22] Cameron Laird and Kathryn Soraiz. Choosing a scripting language; Perl, Tcl, and Python: they're not your father's scripting languages, SunWorld Online - http://www.sunworld.com/swol-10-1997/swol-10-scripting.html, October 1997

[23] Brent Welch and Michael Thomas. Tcl Extension Architecture. Presented at the Tcl/2k conference, February, 2000

[24] Tcl Software Resources, http://dev.scriptics.com/software/

[25] SWIG online user manual, http://www.swig.org/Doc1.1/HTML/Contents.html

[26] David M. Beazley: (1998) Using SWIG to Control, Prototype, and Debug C

Programs with Python, 4th International Python Conference

[27] Geometry in Action, http://www.ics.uci.edu/~eppstein/geom.html

[28] Wilbon Davis. Ray tracer algorithm, October 1999

[29] Wilbon Davis. Point3, Vector3, Ray3, Sphere, Psgray Libraries, October 1999

[30] Steven Fortune. A sweepline Algorithm for Voronoi Diagrams, Algorithmica 2: 153~174, 1987

[31] Steven Fortune. Voronoi Code. http://netlib.bell-labs.com/netlib/voronoi

[32] Steven S. Skiena. The Algorithm Design Manual, November 1997

[33] Joseph O'Rourke. Computational Geometry in C 2nd ed, September 1998

[34] Brian W. Kernighan and Dennis M. Ritchie. The C Programming Language, Second Edition, Prentice Hall, Inc., 1988

[35] GCC user manual, http://gcc.gnu.org/onlinedocs/gcc_toc.html

[36] C Language Implementations; Weisses Data Structures, http://www.cs.sunysb.edu/~algorith/implement/c.shtml

[37] Dynamic-Link Libraries, MSDN SDK, http://msdn.microsoft.com/library/psdk/winbase/dll_512r.htm

[38] Ruediger R. Asche. Rebasing Win32 DLLs: The Whole Story. http://msdn.microsoft.com/library/techart/msdn_pagetest.htm, September 1995

[39] Tcl Improvement Proposal: (2000), http://www.cs.man.ac.uk/fellowsd-bin/TIP/