Teaching Graphics Using Ada

C. Wayne Brown
United States Air Force Academy
2354 Fairchild Dr., Suite 6G-107
USAF Academy, CO 80840-6208
Phone: 01-719-333-3131
Wayne.Brown@usafa.af.mil

ABSTRACT
This paper describes several tools related to the Ada language that were developed to support the teaching of a computer graphics course. These tools include an updated and improved OpenGL Ada specification file, a VRML-to-code conversion tool, and an Ada-to-C conversion tool. The rational for the development of these tools and some issues related to their implementation are discussed.

Categories and Subject Descriptors
D.3.3 [**Programming Languages**]: Language Contructs and Features – *Data types and structures, procedures, functions, and subroutines.*

General Terms
Documentation, Languages.

Keywords
Computer graphics, VRML, Ada, C, cross compiling, code conversion.

1. INTRODUCTION
The Computer Science department at the United States Air Force Academy (USAFA) teaches an introductory course in Computer Graphics. This course was taught by the author for the last two years – spring semester 2003 and spring semester 2004. Programming exercises assigned to the students are implemented using OpenGL. Since computer graphics programming is not language dependent and many programming languages have bindings to the OpenGL library, the author allows the students to submit assignments in the language of their choice. In 2003, all nine students choose to program in Ada because of their prior knowledge and experience with the language. In 2004, eight students developed using Ada, while three used C.

This paper presents some of the "lessons learned" and outcomes from teaching Computer Graphics using Ada. The major topics to be discussed include linking issues when combining C code written by the author with Ada projects, conversion of geometry model data into Ada and C code for manipulation, and conversion of Ada code into C code for convenience of teaching.

Copyright 2004 Association of Computing Machinery. ACM acknowledges that this contribution was authored or co-authored by a contractor or affiliate of the U.S. Government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only.
SIGAda 2004, November 14-18, 2004, Atlanta, Georgia, USA.
Copyright 2004 ACM 1-58113-906-3/04/0011...$5.00.

2. PREVIOUS WORK
When programming computer graphics applications in Ada, an OpenGL specification file, *gl.ads*, is required to provide for proper compilation of the application code and proper linking to the OpenGL run-time libraries for execution. The author is aware of two publicly available versions of the Ada specification file and its associated *glu.ads* and *glut.ads* files. One version is by W. M. Richards and is dated December 1997 [1]. It implements version 1.1 of the OpenGL specification. This specification attempts to use the strong typing features of Ada to protect programmers from passing incorrect parameter values to OpenGL subprograms. It does this by creating unique enumerated types for many of the procedure parameters. Another version of *gl.ads* is by David Holm [2]. It takes a more C approach to the specification file and implements version 1.3 of the OpenGL specification as a straight "port" of the gl.h header file. This version makes no attempt to provide semantic value checking at compile time. The advantages and disadvantages of each specification file will be discussed in the next section.

Computer graphics is a broad field that can be taught at many different application levels. The goal of the author's computer graphics course is to give students the ability to include a graphical component into future software development projects. Toward this goal, the emphasis of the course is software development. However, the creation of complex polygonal geometry for graphic models is difficult and tedious when restricted to program code. To ease this task, the author wrote a Perl script that converts Virtual Reality Modeling Language (VRML) files into Ada or C code. The students are encouraged to create polygonal mesh models using any tool of their choosing as long as the resulting data can be exported or converted to VRML format. Once the polygonal data is converted to code using the Perl script, the students modify the code to produce various graphical effects (e.g., texture mapping). The Perl script is described in Section 4. The author is not aware of any other available tools that convert polygonal geometry to programming code in this fashion.

To provide example code and demonstration programs to students during the computer graphics course, the author wrote a significant number of non-trivial programs. Due to lack of time, the demonstration and assignment programs could not be written in both Ada and C. Since a majority of the students were using Ada, all of the example programs provided to the class were written in Ada. On occasions when the author had time, the programs were converted to C code manually. The author is aware of two products that convert Ada code into C++ code. RTI's *ada2cc* program [3] is a commercial product whose large license fees make it inaccessible

for educational use. A freeware program called *Adatoccpptranslator* [4] also exists that converts Ada code to C++ code. However, it makes extensive use of objects, name spaces, and the like, which make the resulting code more complex than the graphics course requires. Therefore, the author wrote a Perl script to convert Ada programs to C (not C++) code, and this script is described in Section 4.

3. LINKING C LIBRARIES TO ADA CODE

3.1 The Ada OpenGL Specification

The Richards [1] and Holm [2] versions of *gl.ads* are significantly different. The major differences relate to how data types are defined and how parameters for subprograms are declared. The author has created a new version of *gl.ads* that incorporates the best features of each. These issues are described next.

OpenGL defines its own basic datatypes to simplify porting of code across various hardware and OS platforms. For example, OpenGL defines a GLboolean, GLbyte, and GLfloat data type, among others. Richard's specification uses the *new* keyword to define these data types, as in:

```
type GLfloat  is new C.Interfaces.C_float;
```

This choice of declaration requires the use of a cast when Ada floating point values are used in combination with the graphics code. For example:

```
glColor3f(   GLfloat(random(gen)),
             GLfloat(random(gen)),
             GLfloat(random(gen)) );
vertex(3):=GLfloat(arctan(float(vertex(1)),float(vertex(2)));
Put(float(vertex(1))); New_line;
```

This extraneous casting slows down code development. Holm's specification uses *subtypes* to define the OpenGL data types, as in:

```
subtype GLfloat  is C.Interfaces.C_float;
```

This eliminates the need for such casting. The previous examples now become:

```
glColor3f( random(gen), random(gen), random(gen) );
vertex(3) := arctan( vertex(1), vertex(2) );
Put(vertex(1)); New_line;
```

This code is quicker to type and easier to read. In addition, since there is no actually range restrictions on the subtype, there is no loss of type checking . The *subtype* declarations are used in the author's new *gl.ads* version.

OpenGL defines a significant number of integer constants to specify library attributes. In C, these are declared as *GLenum* constants which are simply *unsigned int*. In the C header file and function prototypes, there is no distinction made between various associated groups of attributes. This can lead to errors that are sometimes difficult to diagnose. For example, the author has written

```
glEnable(GL_DEPTH);
```

to enable hidden surface removable, only to have hidden surface removal not work. The correct statement is

```
glEnable(GL_DEPTH_TEST);
```

Ada has the ability to check for these types of errors at compile time and this is helpful for student programmers. Using Holm's version of *gl.ads*, these types of errors can be easily created. Richard's version uses enumerated types to prevent these types of errors, as in the following example:

```
type RenderModeEnm is
(
  GL_RENDER,
  GL_FEEDBACK,
  GL_SELECT
);
for RenderModeEnm use
(
  GL_RENDER          => 16#1C00#,
  GL_FEEDBACK        => 16#1C01#,
  GL_SELECT          => 16#1C02#
);
for RenderModeEnm'Size use GLenum'size;

function glRenderMode(mode: RenderModeEnm)
return GLint;
```

These enumerated type declarations are used in the author's new *gl.ads* version and they have been updated to include attributes and procedures from versions 1.2, 1.3, and 1.4 of the OpenGL specification. It should be noted that some of the OpenGL attributes are used for multiple subprogram calls, and the meaning of the attributes change for distinct calls. For example, the attribute GL_REPLACE can be sent to the glStencilOp and glTexEnv subprograms. However, the set of legal attributes for glStencilOp are GL_KEEP, GL_ZERO, GL_REPLACE, GL_INCR, GL_DECR, and GL_INVERT, while the legal attributes for glTextEnv are GL_MODULATE, GL_DECAL, GL_BLEND, and GL_REPLACE. Fortunately Ada allows names to be reused in individual enumerated types and no identifier conflicts occur. There is a maintenance overhead associated with this programming style because it must be guaranteed that the multiple versions of each attribute have the correct and identical constant value. The author created a Perl verification script to validate the correctness of the new OpenGL specification file.

Neither Richard's or Holm's version attempts to verify that correct array parameters are passed to subprograms. If the C version of a subprogram requires an array parameter, the Ada parameter is declared as an *access* variable of the appropriate type. The compiler can verify that the formal parameter is of the correct type and that an address is being passed, but not that a correct number of values are in the array. For example, the *glLightfv* subprogram requires a four component vector to specify the position of a light source. A common error is to pass a three component vector. The author's new version of *gl.ads* adds declarations such as:

```
type GLfloatPoint4 is array(0..3) of GLfloat;
type GLfloatPoint4Ptr is access all GLfloatpoint4;
```

and uses these types to specify appropriate array parameters. This addition type checking is helpful for student programmers.

A problem was discovered with the *glut.ads* specification when students attempted to use the bitmap font routines included in the glut library. Richard's *glut.ads* specification was written for Unix (X11) systems and incorrectly specifies the font constants for Microsoft Windows systems. The C header file handles these inconsistencies with conditional compilation syntax which Ada lacks. The author's major complaint with Ada is its lack of conditional compilation features.

3.2 Linking to Non-Ada Code

The author has developed various support software for the student's use over the years. Two libraries of C code are particularly important – a movie library and an image processing library. The movie software uses Apple's QuickTime Software Development Kit (SDK) to save the contents of a window into a QuickTime movie file to create animations. The imaging code reads BMP files into memory and crops images to dimensions that are a power of 2 for use as texture maps. It was very difficult to make these libraries work with Ada, even though the process was fairly straightforward. The Ada programs would compile and link correctly, but would not execute correctly. Eventually the problem was identified as a conflict with the Microsoft I/O routines. After converting the code from static libraries to dynamic libraries, the code worked correctly. The author learned a valuable lesson: "A resident Ada expert is a good thing."

4. CODE GENERATION

4.1 Converting Geometric Models to Code

The students were encouraged to use graphical modeling tools to create polygonal mesh models. These models could have been read into their programs as data, but the author desired that the students manipulate the models through programming. Therefore, the author wrote a Perl script that converts Virtual Reality Modeling Language (VRML) files into Ada or C programs. The conversion is straightforward, with only a few exceptions. VRML files define hierarchical data. Since Ada allows embedded procedures, a straightforward mapping is possible between VRML nodes and Ada procedures. However, the author also desired to produce C code from VRML data, which does not support embedded functions. Therefore, the Perl script is implemented recursively but during the recursion VRML nodes are converted to functions in a linear order. This creates potential identifier conflicts between common names in different branches of the hierarchy. Therefore, the script modifies any identifier names that create a conflict. Another issue relates to variable initialization. In C, a local variable inside a function that is declared as *static* is initialized once (and only once) and retains its value between procedure invocations. This feature of the language works well for large arrays of vertices that must be declared and initialized only once, and which have local scope, i.e., the vertices are used by only one model and therefore in only one function. However, if such arrays are declared as local variables in Ada, then they are re-initialized on each entry to the subprogram. This is very inefficient and can potentially slow graphics rendering. Therefore, when the script converts data into Ada code it creates all Ada arrays with global file scope. This also requires some identifier name de-confliction.

Complex polygonal models that contain many vertices are easily created with modern modeling tools and this translates into large Ada and C code files. The AdaGIDE development environment used by the Ada students was modified to handle larger file sizes. Currently files larger than approximately 2MB and 32,000 lines of code cannot be efficiently edited with AdaGIDE. Similarly, the Microsoft Visual C/C++ IDE cannot handle code files with more than 65,536 lines of code. The Perl script was modified to reduce the line count for large models so that more of the student's files could be edited and compiled in the normal way. For several large files the students had to use Microsoft Word for editing. The GNAT Ada compiler had no problems compiling the large generated Ada files. The Microsoft compiler would compile the large C files, but it took a long time to do so (e.g., up to 5 minutes for a single file).

4.2 Converting Ada Code to C Code

For convenience of teaching, the author desired a tool that would convert straightforward Ada code into C code. A Perl script was developed that does this conversion. The script prompts for a file name and, for example, if the user enters *foo*, the script attempts to convert *foo.ads* to *foo.h* and *foo.adb* to *foo.c*. Translation of most statement types is straightforward and is not discussed further here. The major difficulties arise from parameter passing and these issues are discussed in the following paragraphs. The translation script is not intended to be robust or comprehensive. It does not deal with Ada objects or the unique specifics of most standard Ada libraries. It does convert I/O statements and random number generation statements correctly because these are commonly used in the author's code. The script produces ANSI C code. Ada code that the script does not understand is added to the C file as comments with the syntax "`/* FIX: ... */`." After the automatic conversion, these comments can typically be manually fixed with minimal effort.

Parameter passing and function return values are difficult to convert during the language conversion due to distinct language features. ANSI C does not allow arrays or structures as return values from functions. However, Ada does and it is standard practice to do so. For example, a function that creates a vector from two points might be declared as:

```
function CreateVector (
    Tailpoint : in Point;
    Headpoint : in Point  ) return Vector;
```

This allows the function call to be embedded inside other vector processing statements, as in:

```
v3 := CrossProduct(    CreateVector(p1, p2),
                       CreateVector(p2, p3) );
```

Since C code does not support the return of entire arrays, the C code could be translated as:

```
Point *CreateVector (
    Point *Tailpoint,
    Point *Headpoint, Point *result  );
```

where the memory for the *result* is allocated by the calling subprogram, but the function returns the address of the result so that the function can be used as above. For the example function call above, the C calling statement might now be:

```
CrossProduct(  CreateVector(p1, p2, v1),
               CreateVector(p2, p3, v2), v3 );
```

As shown from this example, the change in parameter passing requires not only a change in the function prototype, but also in the internal function code, the return statement, how the function is called, and the required variables in the calling program. The conversion script makes these changes where appropriate when it creates the C code.

Ada also allows for the passing of arrays "by value" using the keyword *in*. This is advantageous in situations where the input and output variables are identical, and the procedure modifies the input values during its processing. For example, the Ada procedure below multiplies a three component vector by a 4x4 matrix, where the matrix is stored in a single dimensional, column-major order, array:

```
Function Transform( m        : in Matrix;
                    original : in Vector)
  return Vector is
  newVector  : Vector;
begin
  newVector(1) :=  m(1) * original(1) +
                   m( 5) * original(2) +
                   m( 9) * original(3);
  newVector(2) :=  m(2) * original(1) +
                   m( 6) * original(2) +
                   m(10) * original(3);
  newVector(3) :=  m(3) * original(1) +
                   m( 7) * original(2) +
                   m(11) * original(3);
  return newVector;
end Transform;
```

Making a function call such as:

```
p1 := Transform(m, p1);
```

works fine because the function is working on a copy of p1 and returns new values after the entire computation is finished. If this function is modified as described in the previous paragraph, the function prototype might become:

```
Vector *Transform(  GLfloat  *m,
                    Vector   *original,
                    Vector   *result);
```

with the function call becoming:

```
Transform(m, p1, p1);
```

This would calculate incorrect results because the values in the array would change during the processing of the final result. This can be corrected by making a copy of the input array before the processing begins, as in the following transformed C code:

```
Vector *Transform( GLfloat   *m,
                   Vector    *original,
                   Vector    *result)
{
  Vector  originalCopy;
  memcpy(originalCopy, original, sizeof(Vector));

  result[0] =  m[0] * original[0] +
               m[4] * original[1] +
               m[8] * original[2];
  result[1] =  m[1] * original[0] +
               m[5] * original[1] +
               m[9] * original[2];
  result[2] =  m[2] * original[0] +
               m[6] * original[1] +
               m[10] * original[2];
  return result;
}
```

The conversion script creates C code that allocates and manipulates local copies of arrays if the Ada parameter type is an "*in*" array.

The conversion tool is not perfect, but it automates a major portion of the code creation and eases the burden of teaching a multi-language programming course.

5. RESULTS

The tools and issues discussed in this paper were developed and learned while teaching two semesters of a computer graphics course. The students had access to some of these tools and not to others. The computer graphics class of spring 2003 worked in three teams of three students each. Each team developed a computer graphics animation over the course of the semester. They had use of the movie and image processing libraries discussed, and a version of the Perl script that converted VRML 2.0 into Ada code. The quality of the resulting student animations was not outstanding.

The class of spring 2004 worked in teams of three students each to also produce computer graphic animations over the course of the semester. In contrast, they had access to an improved version of the Perl script that converted VRML 1.0 files into Ada and C code. Overall they produced higher quality animations. The author attributes the higher quality animations to better tools and higher motivated students.

The new version of *gl.ads* was developed out of lessons learned from teaching these courses and was not used by either class.

6. CONCLUSIONS

Ada's strong type checking prevents students from making some types of common errors and thus has the potential to increase the speed of program development by lowering the amount of time required for debugging. The author sees no major quality difference in the final resulting computer graphics animations produced from the Ada and C programming groups.

It is hoped that by making the new version of *gl.ads*, *glu.ads*, and *glut.ads* freely available, along with the other tools mentioned in this paper, that others can benefit from their development. The files and tools are available at [5].

7. ACKNOWLEDGMENTS

The author is grateful to Lt Col Ricky Sward for his encouragement and support in documenting this work. In addition, I am grateful to Dr. Martin Carlisle for his support and his knowledge of Ada. Dr. Carlisle assisted with the resolution of linking errors and modifications to AdaGIDE.

8. REFERENCES

[1] W. M. Richards, (one source of his sepecification files is http://www.niestu.com/oglada/).

[2] David Holmes, http://adaopengl.sourceforge.net/.

[3] Rehosting Technologies, Inc., 11505 Henegan Pl, Spotsylvania, VA, 22553, http://www.ada2cc.com/.

[4] http://adatoccpptranslator.free.fr/eg/.

[5] http://usafa.af.mil/dfcs/papers/Brown/ResearchWebPage.