Design of an OpenGL Interface for the J Programming Language

William A. Randall*
John E. Howland
Department of Computer Science
Trinity University
715 Stadium Drive
San Antonio, Texas 78212-7200
Voice: (210) 736-7480
Fax: (210) 736-747
Internet: wrandall@cs.trinity.edu
Internet: jhowland@ariel.cs.trinity.edu

March, 1997

Abstract

Traditional 3D graphics programming using OpenGL in C or C++ often requires significant development time by expert graphics programmers. The time costs associated with learning to use OpenGL calls in the C or C++ environment and compiling, linking, and testing during application development are typically extensive. Therefore, new tools that reduce the development time associated with graphics programming or make graphics programming available to a broader spectrum of programmers paves the way towards speedier and more sophisticated graphical software development. An interpreted environment accommodates faster graphics application development by eliminating the need to frequently recompile code in order to observe the effects of OpenGL calls to the current graphics state. Utilizing the functional programming language J with appropriate OpenGL bindings, an interpreted approach provides an interactive environment suitable for less time consuming OpenGL-like graphics programming. This paper details design and implementation issues involved in creating OpenGL bindings for the J programming language. In addition, an explanation of one approach to creating such bindings and suggestions for possible improvements are also included.[1]

*Now at Southwest Research Institute, San Antonio, Texas

[1]This paper appears in the Journal of Computing in Small Colleges, Volume 12, Number 4, Pages 184-193, March 1997.

1 Introduction

The programming language J, a dialect of APL created by Kenneth Iverson and associates at Iverson Software Incorporated, is a modern functional programming language. The order in which operations are performed depends on the context of the operation's use with in a J sentence [Smi 95]. J sentences may be interpreted by simply reading them from left to right with the aid of a J dictionary to substitute English meanings for symbols from the standard ASCII character set. Once interpreted, the J sentence behaves precisely as the English translation is understood. For example, the following line of code:

```
sum =. + /
```

may be translated into the sentence sum is (`=.`) the addition operation (`+`) inserted (`/`) between elements of its list argument. Therefore, the verb sum might be used add `3` , `5` and `7` as follows: `sum 3 5 7` producing a result of `15`.

In addition to its straightforward translation, J also lends itself to graphical program development because of the functional nature of graphics programming. The J programming language also contains primitive operations for linear algebra and array manipulation. Graphics programming utilizes linear algebra for processes like object translation, scaling and rotation and uses arrays for tasks like grouping vertices. Graphics programming also consists of issuing commands, or using functions, that manipulate graphics states such as shape and color producing an image or series of images in a graphics view port. Therefore, once appropriate functions have been defined, the J programming language constitutes a tool well suited for graphical program development.

2 Graphics Programming Using OpenGL

In order to utilize the J environment to perform OpenGL-like graphics development, bindings between OpenGL functions and J verbs must be constructed. After constructing the bindings between J verbs and OpenGL functions, the J system may be updated by either loading code containing verb bindings into the appropriate locale via a suitably constructed J system.

3 A Sample Application: Rotating Cube with Color Interpolation

In order to describe how the OpenGL bindings are constructed, and gain an understanding of how the bindings can be utilized, an example that creates a rotating cube with color interpolation follows. The original C code will be

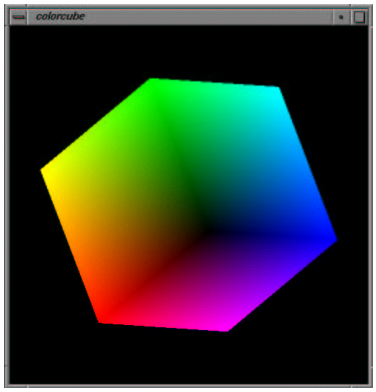

Figure 1: Rotating Cube

presented first followed by the J code. Notice the J syntax closely resembles the C syntax enabling C programmers to make use of the OpenGL bindings with only minimal effort to learn J. Figure 1 illustrates a rendering from the example.

Throughout the discussion that follows the following preliminary J bindings will be employed.

```
monad =: 3
define =: :
script =: 0
from =: {
```

Important to all graphics applications is the display function. The display function is executed as events occur requiring a refresh of the screen. In C, the code for the display function in the rotating cube example appears below.

```
void display(void)
 {
  /* display callback, clear frame buffer and z buffer,
     rotate cube and draw, swap buffers */
     glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
     glLoadIdentity();
     glRotatef(theta[0], 1.0, 0.0, 0.0);
     glRotatef(theta[1], 0.0, 1.0, 0.0);
     glRotatef(theta[2], 0.0, 0.0, 1.0);
     colorcube();
     glFlush();
     glutSwapBuffers();
 }
```

The display function begins by clearing the color and depth buffers and loading the identity transformation matrix. Then the display function rotates

the transformation matrix for the new position of the cube, and calls a function containing the cube geometry to render and color the cube. Finally, the display function calls glFlush to force complete execution of the preceding OpenGL calls and then swaps the buffers for smoother looking motion.

In J, the display function would look like the following.

```
display =. monad define script
NB. display callback, clear frame buffer and z buffer,
NB. rotate cube and draw, swap buffers
gl 'Clear' ; 'GL_COLOR_BUFFER_BIT' ; 'GL_DEPTH_BUFFER_BIT'
gl 'LoadIdentity'
gl 'Rotatef' ; (0 from theta) ; 1.0 ; 0.0 ; 0.0
gl 'Rotatef' ; (1 from theta) ; 0.0 ; 1.0 ; 0.0
gl 'Rotatef' ; (2 from theta) ; 0.0 ; 0.0 ; 1.0
colorcube ''
gl 'Flush'
glut 'SwapBuffers'
)
```

The structure of the OpenGL bindings for J centers around external conjunctions. An external conjunction, appropriately named gl following the naming conventions of OpenGL, is used to construct a J sentence that performs an OpenGL function. The gl external conjunction is followed by the rest of the OpenGL function name, like Translatef or Rotatef, and the argument list corresponding to the C version of the OpenGL function. The J syntax to construct the external conjunction looks similar to the folowing:

```
gl =: 1024 !: 0
```

In J syntax, the description of the OpenGL function and its arguments are strung together through links forming a boxed list. A sample OpenGL sentence in J looks like the following:

```
gl 'translate3f' ; 0.0 ; _1.0 ; 5.0
```

The first element of the boxed list corresponds to one of approximately 120 OpenGL functions and serves as a hash key used by the J system to identify the correct function [Nei 93]. Once the function has been identified, the J system has an idea of what proper arguments to the function should look like. The remaining elements in the boxed list constitute arguments to the OpenGL function. If the remainder of the boxed list contains an argument list inconsistent with that of the original C prototype, J will alert the user to the presence of an error. If the boxed list elements are consistent with the original C prototype, the J system proceeds to process the sentence and perform the graphical side effects associated with the OpenGL function.

Next, consider the code for the main function of the graphics application. The C code follows.

```
void main(int argc, char **argv)
{
   glutInit(&argc, argv);
/* need both double buffering and z buffer */
   glutInitDisplayMode(GLUT_DOUBLE | GLUT_RGB | GLUT_DEPTH);
   glutInitWindowSize(500, 500);
   glutCreateWindow("colorcube");
   glutReshapeFunc(myReshape);
   glutDisplayFunc(display);
   glutIdleFunc(spinCube);
   glutMouseFunc(mouse);
   glEnable(GL_DEPTH_TEST); /* Enable hidden--surface--removal */
   glutMainLoop();
}
```

Notice that the main function in the rotating cube example contains calls to the GLUT libraries. GLUT library calls are not a problem for the J system. The J system simply relies on another external conjunction to include GLUT functionality. The GLUT external conjunction is constructed in a manner similar to the gl external conjunction:

```
glut =: 1024 !: 1
```

Also, notice the main C function uses callbacks. Callbacks constitute another case that the J system must handle, and therefore establishes a need for another external conjunction.

The callback external conjunction follows the same construction pattern as the previous OpenGL external conjunctions, but is processed a little differently. The callback external conjunction appears as follows:

```
callback =: 1024 !: 4
```

The callback external conjunction produces a C function pointer to the function specified in the second element of the boxed list, for example display. When called with appropriate arguments, if any, the J function (in this example display) is executed. As a result, the callback function is properly executed for its side effect.

The J code for the main function of the rotating cube example therefore appears as follows.

```
main =. monad define script
('argc' ; 'argv') =. y.
glut 'Init' ; 'argc' ; 'argv'
NB. need both double buffering and z buffer
glut 'InitDisplayMode' ; 'GLUT_DOUBLE' ; 'GLUT_RGB' ; 'GLUT_DEPTH'
glut 'InitWindowSize' ; 500 ; 500
glut 'CreateWindow' ; callback 'colorcube'
```

```
glut 'ReshapeFunc' ; callback 'myReshape'
glut 'DisplayFunc' ; callback 'display'
glut 'IdleFunc' ; callback 'spinCube'
glut 'MouseFunc' ; callback 'mouse'
NB. Enable hidden--surface--removal
gl 'Enable' ; 'GL_DEPTH_TEST'
glut 'MainLoop'
)
```

The remainder of the rotating cube example code appears in the Appendix in both C and J. All that remains are the definitions of the callback functions, some global variables, and the cube geometry and color functions.

4 Benefits to OpenGL Programming in J

The environment presented by the OpenGL bindings allows for faster graphics programming and development by increasing interactivity. When graphics commands are issued to the J system they are immediately interpreted and executed. The programmer is therefore allowed to experiment with translation values or red-green-blue-alpha values (or any other graphics parameter) and perceive the results of his or her changes relatively quickly. In other words, the graphics state updates as the programmer issues commands inside the J environment. As a result, the programmer immediately sees the effect of his or her actions without having to compile, link, and test the application changes as required when working in the C or C++ development environments. Even with the convenience of make scripts that automate the build process, building the C or C++ executable to evaluate the effects of added OpenGL commands is time consuming and still requires the programmer to run the freshly built executable.

As for the selection of the OpenGL graphics programming libraries, OpenGL offers a robust set of utilities and commands suitable to a wide range of two and three dimensional graphical programming endeavors. OpenGL has also evolved as a standard amongst 3D graphics programming languages and has been ported to and extensive variety of platforms and operating systems. In any case, the creation of J bindings for OpenGL certainly does not exclude or prevent the creation of J bindings for other graphics programming libraries in the future.

5 Conclusions

The ability to interactively experiment with OpenGL commands in the J environment may increase the learning speed of new comers to graphics programming and accelerate their progression along the learning curve associated with OpenGL and 3D graphics programming. J bindings for OpenGL decrease the feedback latency between trial and result (hopefully not error) which grants programmers the freedom to think and work faster. The interactive feedback

offered by J also has a better chance of captivating and maintaining the developers interest when compared with the traditional C or C++ development environments which follow more of a trial ... wait ... wait ... wait ... result model.

The creation of J bindings to OpenGL also offers potential spin-off and enhancement projects. For example, while it may be faster to develop graphics applications in an interpreted environment, applications with real-time needs may still require a compiled executable to meet performance demands. A reasonable project would therefore be to write a translator that would take a J script file and generate a C or C++ file which would perform the same operations but would be compiled for optimal runtime performance.

Another spin-off project might be to create a GUI environment for object modeling in OpenGL. Such a development tool would certainly save time over the trial ... wait ... result sequences involved in using C and C++ to create models, and would also abstract away the users need for knowledge of the J syntax to utilize the bindings to OpenGL for modeling.

In the future, perhaps even more sophisticated graphics APIs could be utilized interactively through the creation of new J bindings. In particular, Performer and its ability for multiprocessing poses interesting questions about how J might handle multiple process graphical applications.

References

[Bur 96] Burke, Chris, *J User Manual*, Iverson Software Inorporated, Toronto, Ontario, 1996.

[Hui 92] Hui, Roger K. W., *An Implementation of J*, Iverson Software Inorporated, Toronto, Ontario, 1992.

[Ive 96] Iverson, Kenneth E., *J Intorduction and Dictionary*, Iverson Software Inorporated, Toronto, Ontario, 1996.

[Kil 96] Kilgard, Mark J., "The OpenGL Utility Toolkit (GLUT) Programming Interface: API Version 3", Silicon Graphics Incorporated, 1996.

[Nei 93] Neider, Jackie et al., *Open OpenGL Programming Guide*, Addison-Wesley Publishing Company, New York, 1993.

[Seg 93] Segal, Mark and Kurt Akeley, *The OpenGL Graphics System: A Specification (Version 1.0)*, Silicon Graphics Incorporated, 1993.

[Smi 95] Smillie, Keith, "Some Notes on Introducing J with Statistical Examples" Revised Edition, June 1995.

[Smi 93] Smith, Kevin P., "The OpenGL Graphics System Utility Library", Silicon Graphics Incorporated, 1993.

Appendix

A Remaining C code

```
GLfloat vertices[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

GLfloat normals[][3] = {{-1.0,-1.0,-1.0},{1.0,-1.0,-1.0},
{1.0,1.0,-1.0}, {-1.0,1.0,-1.0}, {-1.0,-1.0,1.0},
{1.0,-1.0,1.0}, {1.0,1.0,1.0}, {-1.0,1.0,1.0}};

GLfloat colors[][3] = {{0.0,0.0,0.0},{1.0,0.0,0.0},
{1.0,1.0,0.0}, {0.0,1.0,0.0}, {0.0,0.0,1.0},
{1.0,0.0,1.0}, {1.0,1.0,1.0}, {0.0,1.0,1.0}};

void polygon(int a, int b, int c , int d)
{
/* draw a polygon via list of vertices */
glBegin(GL_POLYGON);
glColor3fv(colors[a]);
glNormal3fv(normals[a]);
glVertex3fv(vertices[a]);
glColor3fv(colors[b]);
glNormal3fv(normals[b]);
glVertex3fv(vertices[b]);
glColor3fv(colors[c]);
glNormal3fv(normals[c]);
glVertex3fv(vertices[c]);
glColor3fv(colors[d]);
glNormal3fv(normals[d]);
glVertex3fv(vertices[d]);
glEnd();
}

void colorcube(void)
{
/* map vertices to faces */
polygon(0,3,2,1);
polygon(2,3,7,6);
polygon(0,4,7,3);
polygon(1,2,6,5);
polygon(4,5,6,7);
polygon(0,1,5,4);
}
```

```
static GLfloat theta[] = {0.0,0.0,0.0};
static GLint axis = 2;

void spinCube()
{
/* Idle callback, spin cube 2 degrees about selected axis */
   theta[axis] += 2.0;
   if( theta[axis] > 360.0 ) theta[axis] -= 360.0;
   display();
}

void mouse(int btn, int state, int x, int y)
{
/* mouse callback, selects an axis about which to rotate */
   if(btn==GLUT_LEFT_BUTTON & state == GLUT_DOWN) axis = 0;
   if(btn==GLUT_MIDDLE_BUTTON & state == GLUT_DOWN) axis = 1;
   if(btn==GLUT_RIGHT_BUTTON & state == GLUT_DOWN) axis = 2;
}

void myReshape(int w, int h)
{
   glViewport(0, 0, w, h);
   glMatrixMode(GL_PROJECTION);
   glLoadIdentity();
   if (w <= h)
     glOrtho(-2.0, 2.0, -2.0 * (GLfloat) h / (GLfloat) w,
      2.0 * (GLfloat) h / (GLfloat) w, -10.0, 10.0);
   else
     glOrtho(-2.0 * (GLfloat) w / (GLfloat) h,
      2.0 * (GLfloat) w / (GLfloat) h, -2.0, 2.0, -10.0, 10.0);
   glMatrixMode(GL_MODELVIEW);
}
```

B Remaining code in J

```
vertices =. 8 3 $ _1.0 _1.0 _1.0  1.0 _1.0 _1.0 1.0 1.0 _1.0
                  _1.0 1.0 _1.0   _1.0 _1.0 1.0 1.0 _1.0 1.0
                  1.0 1.0 1.0 _1.0 1.0 1.0

normals =. 8 3 $ _1.0 _1.0 _1.0 1.0 _1.0 _1.0 1.0 1.0 _1.0
                 _1.0 1.0 _1.0  _1.0 _1.0 1.0 1.0 _1.0 1.0
                 1.0 1.0 1.0  _1.0 1.0 1.0

colors =. 8 3 $ 0.0 0.0 0.0 1.0 0.0 0.0 1.0 1.0 0.0 0.0 1.0 0.0
```

```
                0.0 0.0 1.0 1.0 0.0 1.0 1.0 1.0 1.0 0.0 1.0 1.0

polygon monad define script
('a' ; 'b' ; 'c' ; 'd') =. y.

NB. draw a polygon via list of vertices

gl 'Begin' ; 'GL_POLYGON'
gl 'Color3fv' ; a from colors
gl 'Normal3fv' ; a from normals
gl 'Vertex3fv' ; a from vertices
gl 'Color3fv' ; b from colors
gl 'Normal3fv' ; b from normals
gl 'Vertex3fv' ; b from vertices
gl 'Color3fv' ;  c from colors
gl 'Normal3fv' ; c from normals
gl 'Vertex3fv' ; c from vertices
gl 'Color3fv' ; d from colors
gl 'Normal3fv' ; d from normals
gl 'Vertex3fv' ; d from vertices
gl 'End'
)

colorcube =. monad define script
NB. map vertices to faces
polygon 0 3 2 1
polygon 2 3 7 6
polygon 0 4 7 3
polygon 1 2 6 5
polygon 4 5 6 7
polygon 0 1 5 4
)

theta =. 0.0 0.0 0.0
axis =. 2


spinCube =. monad define script
NB. Idle callback, spin cube 2 degrees about selected axis
2+ (axis from theta)
if. 360.0 < axis from theta
  do.   360 - axis from theta
end.
display ''
}
```

```
mouse =. monad define script
('btn' ; 'state' ; 'x' ; 'y') =. y.
NB. mouse callback, selects an axis about which to rotate
  if.(btn = GLUT_LEFT_BUTTON) and (state = GLUT_DOWN)
    do.  axis =. 0
  elseif.(btn = GLUT_MIDDLE_BUTTON) and (state = GLUT_DOWN)
    do. axis =. 1
  elseif.(btn = GLUT_RIGHT_BUTTON) and (state = GLUT_DOWN)
    do. axis =. 2
  end.
)

myReshape =. monad define script
('w' ; 'h') =. y.
gl 'Viewport' ; 0 ; 0 ; w ; h
gl 'MatrixMode' ; 'GL_PROJECTION'
gl 'LoadIdentity'
  if. (w < h) or (w = h)
    do.
      gl 'Ortho' ; _2.0 ; 2.0 ; _2.0 * h % w ; 2.0 * h % w ; _10.0 ; 10.0
    else.
      gl 'Ortho' ; _2.0 * w % h ; 2.0 * w % h ; _2.0 ; 2.0 ; _10.0 ; 10.0
      gl 'MatrixMode' ; 'GL_MODELVIEW'
  end.
)
```