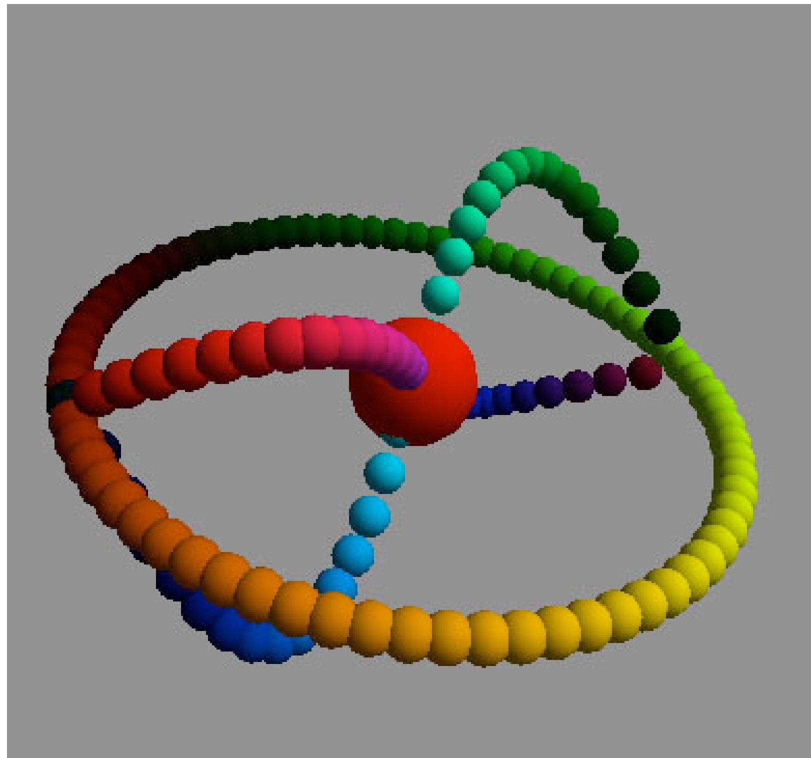
Project Final Report

A Graphic Transformation Language



May 8, 2003
COMS W4115, Spring 2003
Columbia University

Contents

Chapter 1: Language Overview

1.1 Introduction
1.2 Background
1.3 Related Work
1.4 Major Features
1.5 Primary Goal
1.6 Summary

Chapter 2: Language Tutorial

2.1 Integrated Development Environment
2.2 Basic Non-Graphic Programming
2.3 Graphic Programming

Chapter 3: Language Manual

3.1 Lexicon
3.2 Data Types and Classes
3.3 Language Structure
3.4 Comments
3.5 Data Flow Control: if-then-else
3.6 Iterations: for loop
3.7 Statements
3.7.1 Assignments
3.7.2 Class Functions
3.7.3 System Commands

Chapter 4: Project Plan

4.1 Team Responsibilities
4.2 Timeline
4.3 Development Environment
4.4 Programming Style Guide
4.5 Project Log

Chapter 5: Architectural Design

5.1 Language Components
5.2 Language Structure and interface
5.3 Design Feature

Chapter 6: Test Plan

6.1 Goal
6.2 Methods

Chapter 7: Lessons Learned

7.1 Team Member's Lessons
7.2 Advice for the future

Appendix

A. Homogeneous Coordinates and Transformation Matrices
B. jGTL Grammar
C. Code Lists
D. Demo

Reference

Chapter 1

Language Overview

1.1 Introduction

jGTL is a high-level interpreter language, designed to help the programmer generate 3D objects and perform transformation on them. jGTL language simplifies the, usually, difficult task of modeling geometric objects, through integrated routines and methods. jGTL allows the user access to complex operations, such as object scaling, translation, rotation, and projection, on 3-D graphical objects, through the use of simple commands and functions.

1.2 Background

In the context of graphic design, the programmer can use the modeling transformation to position and orient the geometric object. For instance, you can rotate, translate, or scale the 3D object and/or perform combinations of two or more of those operations.

Specifying the projection transformation is like choosing a lens for a camera. You can think of this transformation as determining what the field of view or viewing volume is and therefore what objects are inside it, and to some extent, the description of their physical representation. In addition to the field-of-view considerations, the projection transformation determines how 3D objects are projected onto the 2D screen. There are two types of projections: perspective and orthographic. Perspective projections match how you see things in real life, while orthographic projections map objects directly onto the screen without affecting their relative size.

All those geometric operations on 3D space coordinates can be implemented by computing 4x4 transformation matrices on homogeneous coordinates in the domain of (x, y, z, scale). (There are more details in Appendix A)

1.3 Related Work

Most of the graphic related libraries included in some sophisticated popular languages such as Virtual or Borland C++, Java and Delphi are very large and complicated. There is no commercial software that provides simple operational commands on geometrical

transformation of 3D objects. Furthermore, OpenGL Utility Library (GLU) provides about 150 distinct commands to specify the objects and operations needed to produce interactive 3D applications. And there are more complicated the graphical terminology and definitions in java 3D. jGTL overcomes the complexity of such graphic concepts and commands. As a basic graphic transformation language, jGTL performs viewing, modeling, projecting transformations on 3D objects and it manipulates the 4x4 matrix operations.

1.4 Major Features

The jGTL language is designed to be simple, intuitive, portable, friendly programming and object-oriented.

Simple

The simple standard basic syntax (similar to Java and C) and transformation functions help reduce the code problems. The jGTL language implemented the complicated and tedious matrix operations into the predefined functions, which as a result produces higher efficiency and simplifies 3D graphics coding.

Intuitive

The jGTL language implements computer graphical transformations of geometric figures into some intuitive commands. User can see and play those 3D objects in the graphic interface of jGTL integrated development environment (IDE) directly.

Portable

The jGTL compiler will output code to be embedded in a java class. Therefore the programmer is guaranteed to have machine-independence.

Friendly Programming

One objective is to minimize the amount of time the programmer takes to learn the language. Instead, we want to provide a very intuitive environment that will allow the user to master the language in a matter of hours and hence increase productivity. To that extent, most of jGTL commands will be specific and intuitive. For instance, if the user wants to draw a polygon connected by lines, the obvious command will be "JdrawConfig( )". Also, the graphic integrated development environment (IDE) makes the coding in jGTL easy and convenience.

Object-Oriented

The jGTL language is an object-oriented language, in which each defined geometrical point is an objects that have such attributes as transforms and matrix operations functions.

1.5 Primary Goals

Besides the basic language elements as mathematic operation, data flow control and iteration, the jGTL language implemented the following 3D graphic functions: scaling, translation, rotation and projection. As a high level interpreter, our objective is to construct a programming language that will focus on handling the following 3D geometric operations:

Set of 3D geometric models in any orientation by transformations (3D space).
Control the location in 3D space from which the model is viewed.
Transfer the 3D space coordinates of objects to planar coordinates in order to model the geometric figure into the computer screen.
Manipulate the appropriate matrix stacks responsible to control the 3D object model transformation.

1.6 Summary

The jGTL language provides a powerful and simple choice to 3D graphics transfer programming. With this object-oriented language and its graphic development environment, user can control the geometrical transformation of 3D objects easily and quickly. It is good for the user without much computer programming experience and graphical knowledge to learn and play basic 3D graphic transformation. It is also good choice for the user who wants to do general graphic transformation calculation or graphing without learning OpenGL or java 3D.

Chapter 2

Tutorial

2.1 Integrated Development Environment

jGTL interpreter was implemented with a friendly and easy-use graphic integrated development environment (IDE) interface. All the jGTL I/O operations, source code editing, debugging and running, graphic viewing, and manual referencing can all be done in this integrated graphic interface.

Using Command **$java IDE** can active the graphic interface and start programming with jGTL language. Fig 2.1 is an example of jGTL IDE main window.

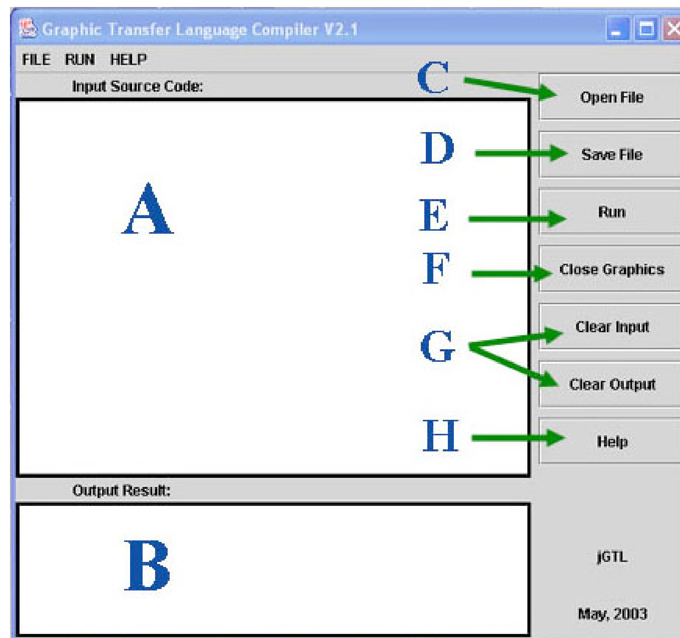


Fig 2.1 jGTL Main Window

The main window is a simple frame contains menus, buttons and text fields to help programming with jGTL interpreter.

Text field **A** is the source code edit area. User can input and edit the jGTL source code in this region. And at the running time, the program will load and parse the source code from this text area.

Text field **B** is the output area, where printing out all the running time information including I/O messages, running error reports, and program non-graphic outputs (specified by the commands like JprintLine( ) and JprintList( ), see chapter 3).

G can clear those text fields.

C and **D** is the file I/O handling to open and save source code. For example, clicking on the button "Open Files" a file open dialog could show up as in Fig 2.2.

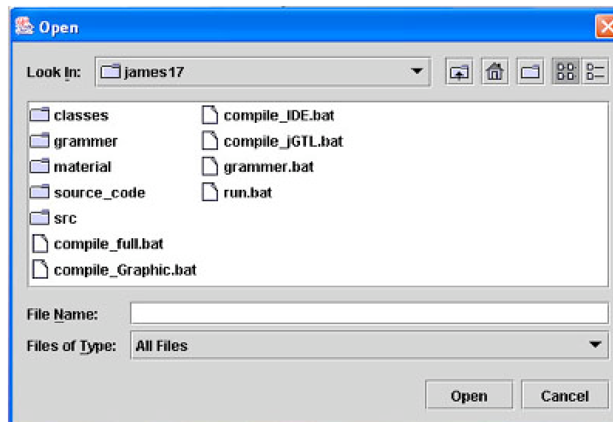


Fig 2.2 File Open Dialog in jGTL

E is the "Run" button to control running of the jGTL codes that were showed in the source code input field (**A**). Once this button is clicked, the jGTL source code in the input field will be interpreted and executed line by line. The non-graphic outputs will be printed in the output text field (**B**). If there are graphic output commands in the code, a graphic explore window will popup and show the 3D graphic results of the jGTL source code that has been executed. **F** button can turn on and off those graphic windows.

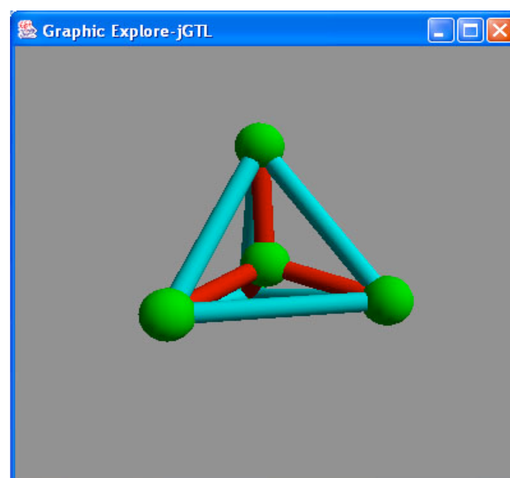


Fig 2.3 Graphic Explorer of jGTL

H is the "Help" button witch can popup a simple help documents browser. The user can use this browser to check the language reference manual, tutorial and demo at the time of coding. A snap shot of help browser is showed in Fig 2.4.

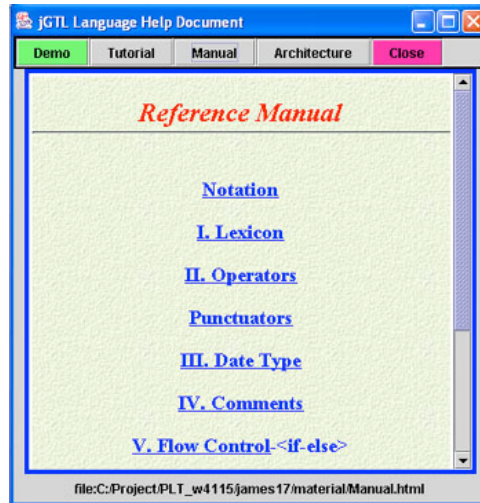


Fig 2.4 Help Browser of jGTL

The graphic IDE interface offers an easy and friendly coding environment for the jGTL user.

2.2 Basic Non-Graphic programming

jGTL language try to use a standard grammar as Java or C to make it friendly to user. The syntax and semantic of value assignments, mathematic, comparing and logic operations, comments, if-then-else data flow control, for iteration are almost the same as the standard of Java language. User with some programming experience in modern language can pick it up easily.

Here is a simple example of jGTL code to calculate the sum of numbers from 1 to 100.

```
/*******A Example to Calculate 1+..+100 ******/
n_max=100; //Initial the max number
n_result=0;  //Initial the result identify
for(ni=1; [ni<=n_max]; ni=ni+1)
{
   n_result=n_result+ni;
} //End of for loop

$JprintLine("The sum of 1 to 100 is: ", n_result); //Print out the result
/***********End ************************/
```

The output of this example code in the output field is as following:

```
Run Successfully!
The sum of 1 to 100 is: 5050.0
```

As an object-oriented language, there are six data types or classes are implemented in jGTL. They are boolean, string, number, point, group and matrix, the last three are especially for the graphic programming. In jGTL, there are some special rules for the specific class identify definitions: the identifiers for string should start with ‘s’ or ‘S’; the identifiers for number should start with ‘n’ or ‘N’; the identifiers for Boolean should start with ‘b’ or ‘B’; the identifiers for point should start with ‘p’ or ‘P’; the identifiers for group should start with ‘g’ or ‘G’ and the identifiers for matrix should start with ‘m’ or ‘M’. By this simple way, user easily declares a class by using the first letter of the identifiers. So in Java, to declare an object of Double contained number 1.2 should use: Double d = new Double (1.2). While in jGTL, just use n_d =1.2, n_d is start with n, so it is a number object.

Here is another example to show the usage of the first letter of the ID to declare an object.

```
n1=n2=3;
n3=n1-n2+7;
S1= "Welcome";
B1= true;
B2= false;
B3= B1||B2;
m1=[[1,1],[0,0]];
p1={1, 1, 1, 1};
p2={2, 2, 1, 1};
g1=<p1, p2>;
$JprintLine(S1,"  ", B3,"  ", n3, "   ", "well");
$JprintLine("We are here.");
$JprintList(p1, g1, m1);
```

And the output would be:

```
Run Successfully!
Welcome  true  7.0   well
We are here.
    1.00      1.00      1.00      1.00    #
______________________
    1.00      1.00      1.00      1.00    #
    2.00      2.00      1.00      1.00    #
______________________
```

```
1.00    1.00   #
0.00    0.00   #
```

2.3 Graphic programming

Point, group and matrix classes are especially for the graphic programming in jGTL. Once an object is declared as one of them, the object could have all the functions of the class that it is. There are 10 functions in Point class, 12 functions in Group class and 18 functions in matrix class (see chapter 3). By use those functions user can almost handle any kinds of space transformation as scaling, translating, rotating and projecting easily and in multiple ways.

There are some commands statements are offered in jGTL to draw the 3D points and lines in the graphic interface to visualize the cool 3D geometrical objects. They are as following: $JdrawPoint( ) could draw a point or group of points into the canvas, $JdrawLine( ) could draw a line to link the two parameter points , $JdrawConfig( ) could draw the lines to link the group of points in order.

The following is jGTL exampling code to draw a S2 symmetrical space group and link the three points with lines. The graphical output is as the Fig 2.5

```
/*******Draw a S2 space group ***********/
//Initial Points
p0={0.0, 0.0, 0.0, 2.0};
p1={-1.0, 0.0, 0.0, 2.0};
p2={ 1.0, 0.0, 0.0, 2.0};
//Rotate around Z-axis to get another two points.
p2:JRotate(60, 0.0, 0.0, 1.0)->p21;
p2:JRotate(-60, 0.0, 0.0, 1.0)->p22;

//Define a group including all points
g1=<p0, p1, p21, p22>;

//Draw the group of points
$JdrawPoint(g1, 0.0, 1.0, 0.0, 0.12);

/**************Draw Lines **************/
$JdrawLine(p0, p1, 1.0, 0.0, 1.0, 0.06);
$JdrawLine(p0, p21, 0.0, 1.0, 1.0, 0.06);
$JdrawLine(p0, p22, 0.0, 1.0, 1.0, 0.06);
/**************END*********************/
```

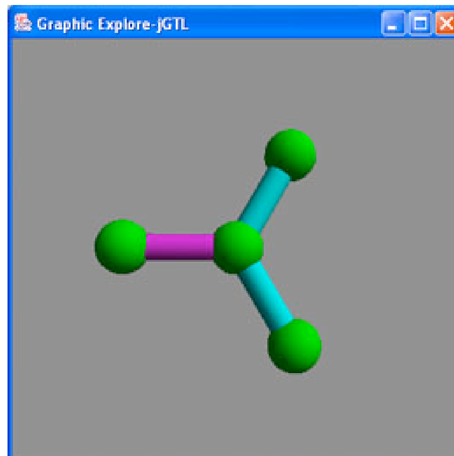


Fig 2.5 SP2 Symmetrical Group

Here is another jGTL exampling code to draw a S3 symmetrical space group and link the points. The graphical output is as the Fig 2.3

```
/************START**************/
//Initial Points
p0={0.0, 0.0, 0.0, 2.0};
p1={ 0.0, 1.0, 0.0, 2.0};
p1:JRotate(-109.5, 0.0, 0.0, 1.0)->p2;
p2:JRotate(120, 0.0, 1.0, 0.0)->p3;
p3:JRotate(120, 0.0, 1.0, 0.0)->p4;

//Define a group include all points
g1=<p0, p1, p2, p3, p4>;
$JdrawPoint(g1, 0.0, 1.0, 0.0, 0.1);

/*************Draw Lines *************/
g1:JgetSize( )->n0;
for(n1=1; [n1<n0] ; n1=n1+1)
{   g1:JgetPoint(n1)->ptemp;
    $JdrawLine(p0, ptemp, 1.0, 0.0, 0.0, 0.04);
}
for(n1=2; [n1<n0] ; n1=n1+1)
{   g1:JgetPoint(n1)->ptemp;
    $JdrawLine(p1, ptemp, 0.0, 1.0, 1.0, 0.04);
}
g3=<p2,p3,p4,p2>;
$JdrawConfig(g3, 0.0, 1.0, 1.0, 0.04);
```

More examples of jGTL source codes could be found in the Appendix D Demo part.

Chapter 3

Reference Manual

The Graphic Transformation Language, jGTL, like many other languages, tries to use a standard grammar as Java or C to make it easy and friendly to user. The specific language grammar definitions and usages are described as following.

Notation

The descriptions of lexical analysis and syntax use a BNF grammar notation.

Each rule begins with a name (which is the name defined by the rule) and a colon. A vertical bar (`|`) is used to separate alternatives; it is the least binding operator in this notation. A star (`*`) means zero or more repetitions of the preceding item; likewise, a plus (`+`) means one or more repetitions, and a question mark (?) means zero or one occurrences (in other words, this phrase is optional). The `*` and `+` operators bind as tightly as possible; parentheses are used for grouping. In lexical definitions, two more conventions are used: Two literal characters separated by two dots mean a choice of any single character in the given (inclusive) range of ASCII characters.

3.1 Lexicon

Characters. Character literals are specified just like in C. They may contain octal-escape characters (e.g., `'\377'`), and the usual special character escapes (`'\b', '\r', '\t', '\n'`).

charVocabulary ='\3'..'\377'

Whitespace. Spaces (‘ ‘), tabs (‘\t’), and newlines (‘\n’) are separators in that they can separate vocabulary symbols such as identifiers, but are ignored beyond that. For example, "`name1 name2`" appears as a sequence of two token references. White space is only meaningful to separate tokens.

End of line. The end of a logical line is represented by the token NEWLINE, which is ‘\n’, ‘\r’ or ‘\r’‘\n’. Statement can’t cross the logical line boundary.

End of file. The EOF token is automatically generated at the end of the source code.

Identifiers. An identifier is a sequence of letters or digits, the first of which must be a letter. There is no limit on the length of an identifier. Two identifiers are the same if they have the same character for every letter and digit.

ID options {testLiterals=true;} : ('a'.. 'z'|'A'..'Z') ('a'..'z'|'A'..'Z'|'_'|'0'..'9')+ ;

In jGTL, there are some special rules for the specific class definition: the identifiers for string should start with ‘s’ or ‘S’; the identifiers for number should start with ‘n’ or ‘N’; the identifiers for Boolean should start with ‘b’ or ‘B’; the identifiers for point should start with ‘p’ or ‘P’; the identifiers for group should start with ‘g’ or ‘G’ and the identifiers for matrix should start with ‘m’ or ‘M’. By this simple way, user declares a class by using the first letter of the identifiers.

Keywords. The following identifiers are used as reserved words, or keywords of the language, and cannot be used as ordinary identifiers. They must be spelled exactly as written here:

if	then	else	for	
false	true			
sqrt	bas	sin	cos	tan
JTranslate	JRotate	JScale	JOrtho	JFrustum
JMulti	JUnit	JAppend	JsetScale	
JInverse	JNegate	JsetIdentity	JsetElement	JsetRow
JsetColumn	JsetValue	JgetElement	JgetRow	JgetColumn
JgetPoint	JgetSize	JaddScale	JaddMatrix	
JsubScale	JsubMatrix	JmulMatrix	JmulScale	
JprintLine	JprintList	JdrawPoint	JdrawLine	JdrawConfig

Table 3.1 jGTL keywords

Operator. An operator is a token that specifies an operation on at least one operand, and yields some result (a value, side effect, or some combination).

ALL **Mathematic operations** in jGTL are listed in Table 3.2, their operands are expressions or constants (a form of expression). The first order operators sqrt(), abs(), sin(), cos() and tan() can have only one operand. For example: N1=sqrt(2); N2=abs(-4). +(plus) and –(minus) would have one or two operand. For example: N3= -6; N4=+2.3; N3 = N1+ 2; other operators must have two operands. For example: N5=2*4; N6=7^2; N7=5/2; N8=3%2;.

All operators are ranked by precedence, a ranking system determining which operators are evaluated before others in a statement. In jGTL, operators sqrt(), abs(), sin(), cos()

and tan() are the first order operations, ^(power) is in the second order, *(multiple), /(divide) and %(mod) are the third order operations, +(plus), -(minus) are the last order operations. As in C and Java, ( ) are used to assign the highest order of all operators, and all the expressions are left-associated. For example: (2+3)/2.

```
   expr : (mexpr (( ‘+’ | ‘-‘ ) mexpr)*)| (( ‘+’ | ‘-‘ ) mexpr)+;
mexpr: pexpr (( ‘*’ | ‘/’ | ‘%’ ) pexpr)*;
pexpr: mole  ( ‘^’  mole)?;
mole : (SQRT^|ABS^|SIN^|COS^|TAN^)atom|atom;
atom : NUMBER|(‘(‘! expr ‘)’!)
          | ID;
```

Operator	Name	Order	Example
sqrt( )	Squre root	First	sqrt(4)=2
abs( )	Absolute val	First	abs(-5)=5
sin( )	sin function	First	sin(1.57)=1
cos( )	cos function	First	cos(3.14)=1
tan( )	tan function	First	tan(0.785)=1
^	Power	Second	2^3=4
*	Multiple	Third	2*3=6
/	Divid	Third	2/3=0.666
%	Mod	Third	2%3=2
+	Plus	Fourth	2+3=5
-	Minus	Fourth	2-3=-1

Table 3.2 jGTL Mathematic Operators

The **comparison operations** in jGTL are “==”, “!=”, “>=”, “<=”, “>” and “<”, them have the same priority, which is lower than that of any arithmetic. Each of them must have two operands at both sides. In jGTL, the operands of a comparison operation is two arithmetic expressions, and the comparison operation must be encompassed in a pair of [ ]. The jGTL compile will evaluate the comparison by return a Boolean “true” or “false”. Notice that the jGTL comparison operations are very strict about parentheses: every expression must be enclosed in parentheses.

```
condi_comp : ‘[‘ ! (expr)
   ( “==”| “!=”| “>=”| “>”| “<=”| “<”)
        (expr) ‘]’ !
        ;
```

Operator	Name	Example

==	Equal	[n1==1]
!=	Not Equal	[n1!=2]
>=	Great than or Equal	[n1>=2]
<=	Less than or Equal	[n2<=3]
>	Great than	[n1>0]
<	Less than	[n2<0]

Table 3.3 jGTL Comparison Operators

Relation operations "&&", "||", "!" are used in jGTL by control flow statements if-else and for-loop conditions. Relation operations have the lowest priority of all jGTL operations, and ( ) could be also used to assign the evaluation order of Relation operation. The operands of relation operations must be comparison operations. The NOT has only one operand, AND and OR both has two operands. As in arithmetic operation, the relation operation is also left-most associated. For example: [n1>2]&&[n2<0] || [n3!=2] is as ([n1>2]&&[n2<0]) || [n3!=2].

```
condition : (condi_comp|( '('!condition ')'!) |ID | TRUE | FASLE)
              ("&&"| "||" | "~" )
            (condi_comp|( '(' !condition ')'!)| ID | TRUE | FASLE))*
          ;
```

<table>
<tr><th>Operator</th><th>Name</th><th>Example</th></tr>
<tr><td>&&</td><td>AND</td><td>[n1==1]&&[n2==0]</td></tr>
<tr><td>| |</td><td>OR</td><td>[n1!=2] | | [n2>1]</td></tr>
<tr><td>!</td><td>NOT</td><td>![n1>=2]</td></tr>
</table>

Table 3.4 jGTL Relation Operators

There is an example for relation operation:

```
        n1= 3;
        n2= 5;
    [(n1+n2)>5]&&(!([n1>1]||[n2>6])) will return false
    [(n1+n2)>5]&&(([n1>1]||[n2>6]))   will return true
    [(n1+n2)>9]&&(([n1>1]||[n2>6])) will return false
```

Punctuators. Some characters in jGTL are used as punctuators, which have their own syntactic and semantic significance. Punctuators are not operators or identifiers. Table 2 lists the jGTL punctuators.

Punctuators	Use	Example
{ }	compound statement delimiter for for-loop and	for(n1=0; [n1<4]; n1=n1+1) { n2=n2+1;

<table>
<tr><td></td><td>if-else statement; Point expressions</td><td>n3=n2+1;}

{1.1, 1.2, 1.3, 1.0}</td></tr>
<tr><td>( )</td><td>Function and Command parameter list delimiter; also used in operate expression</td><td>P1:JRotate(180, 1,1,0)->P2;
$draw(P1);

2*(1+3);</td></tr>
<tr><td>< ></td><td>Group expression</td><td><p1, p2, p3></td></tr>
<tr><td>[ ]</td><td>Matrix expression; comparison operation</td><td>[[1 2 3], [2 3 1], [3 21]]
[n1>2]</td></tr>
<tr><td>;</td><td>Statement ends</td><td>N1=1;</td></tr>
<tr><td>=</td><td>Identify assignment</td><td>N1=2; p1={1, 2, 3, 0};</td></tr>
<tr><td>#</td><td>Multiple Identifiers assignment</td><td>N1=N2=2.3;</td></tr>
<tr><td>: and -></td><td>Function Assignment</td><td>P1:JTranslate(1, 2, 3)->P2;</td></tr>
<tr><td>$</td><td>System Command</td><td>$draw(P1);</td></tr>
<tr><td>“ ”</td><td>String</td><td>S1= “Hello.”</td></tr>
<tr><td>// and /* , */</td><td>Comment</td><td>//This is a comment.</td></tr>
<tr><td>*, /, +, -, ^, %</td><td>Arithmetic Operation</td><td>N1+N2</td></tr>
<tr><td><, <=, >, >=, != and ==</td><td>Comparison Operation</td><td>[N1<=2]</td></tr>
<tr><td>&&, | |, !</td><td>Relation Operation</td><td>[N1==2]&&[N3!=2]</td></tr>
</table>

Table 3.5 jGTL punctuators

3.2 Data Type and Classes

There are six data types or classes are implemented in jGTL (Table 3.6), point, group and matrix data types are special for graphic programming.

Data Type	Examples	Comments
Boolean	true, false, [n>2]	The expression must in [ ]
String	“This is a “”Test”” string.”	Double quote;
Number	12, 12.3, 12e-2, 12.2e1	Include C’s INT and Float
Point	{1, n1*2, -3}, {1, n1, 3, 4}	1<=(element number)<= 4
Group	<p1, p2, p3>	A list of Point
Matrix	[[1, 2, 3], [3, 2, 1], [2, 1, 3]]	Row number = Column number

Table 3.6 jGTL Data Types and Classes

Boolean. Boolean class is used for the logic operations. It have two values "true" or "false", it also could be the logic results of comparison and relation operations. For example: B1=true; B1=false; B3=[n1>=1]; B4=!(B1||(B2&&B3)).

String. String constants are enclosed in quotes. String constant must be contained on a single line and may contain double quotes. For example: "This is a constant with ""double quotes"".", jGTL compiler will recognize the string as: This is a constant with "double quotes".

```
String : '"'! (~('"' | '\n') | ('"'!'"'))*  '"'!  ;
```

Number. Simply, the number includes the INT and Float that defined in C language. E.g. the following are the legal number: 12(Integer),   12. , .3 , 12E2 , 12.3 , 12.E2 , .3E2 , 12.3E2 . And others are illegal number expression.

```
DIGIT : '0'..'9';
INT   : (DIGIT)+;
EXPONENT : ('e'|'E')('+'|'-')?(DIGIT)+;

NUMBER : ( (INT) ((('.')( ((INT)(EXPONENT)?) | (EXPONENT)? ))|EXPONENT)? )
       | ('.'(INT)(EXPONENT)?) ;
```

Negative parts of all the above number definitions are allowed in jGTL. Although it is defined as an operate expression, For example, -23 will be recognized as 0-23, there is no difference to user.

Point. Point class is a list of numbers with the format {x, y, z, scale}. For example: a two dimensional point {1.1, 2.1, 0, 1.0}, and a four-dimensional point {1.1, 1.2, 1.3, 1.0}. In jGTL, the number of point elements is limited to 4. You could define a point with less than four elements, and the undefined parts will be set to the default value {0, 0, 0, 1.0}. For example: input { }, the program will automatically recognize it as {0, 0, 0, 1.0}. The element of point would be any number, predefined number ID and operate expression. For example: n1=1.2; n2=2.3; p1={n1, -n2, 3.3, n1*n2}.

Group. Group class is a list of pre-defined point identifiers or point expressions with the format <p1, p2 …>. For example: p1={1, 2, 3, 0}; g1=<p1, {2, 3, 4, 1}>;. . User can construct an object by specifying a group of points in that object. In jGTL, the number of group element—point is unlimited. The empty group can be declared as g1=< >; , while all functions of Group data type except JAppend( ) require the current group be nonempty.

Matrix. The matrix class is a two dimensional list of number with the format [row1, row2, row3 …] which has the same row number and column number. For example: [[1.1, 1.2, 1.3], [2.1, 2.2, 2.3], [3.1, 3.2, 3.3]]. In jGTL, the number of elements in each row is unlimited and must be large than one. A 4x4 matrix is always used for the three-

dimensional coordinates transformation operations. The element of matrix would also be any number, predefined number ID and operate expressions. For example: n1=1.2; n2=2.3; m1=[[n1, n2], [3.3, n1-n2]].

3.3 Language Structure

The jGTL is a kind of line-interpreted languages. It executes the source code line by line. There are four basic logic line formats that are listed in Table 3.7, and their format, grammar and usage will be described in following several parts.

Logic-line					
Comment	If-Else	For-Loop	**Statement;**		
// one line or /* BLOCK*/	**IF** condition THEN {Logic-line} ELSE {Logic-line}	**For** (initial-condition) {Logic-line}	Assignment	Function	System
			ID = Element	ID **:** Function-Name (args) -> ID	$Command (args)

Table 3.7 jGTL Language Structure

3.4 Comments

jGTL accepts line and block comments similar to Java-style. For example,

Line comment: A single line start with at least two slashes--'//'.
// This a line comment start with two slashes. Or
/// This is a line comment start with three slashes.

Block comment: Several continue lines start with '/*' and end with '*/', no extra slash '/' is allowed in block.
/******This is a block comment ********
@Author
@Tile
**********************************/

jGTL use Java-style flow control if-else and iteration for-loop.

3.5 Data Flow Control: If-then-else

The `if`-else logic line has the following syntax:

```
if_else  : IF  condition
              THEN { (logic-line)*}
              (  ELSE {(logic-line)*} )?
          ;
```

```
condition : (condi_comp|( '('!condition ')'!) |ID | TRUE | FASLE)
                ("&&"| "||" | '!' )
              (condi_comp|( '(' !condition ')'!)| ID | TRUE | FASLE))*
            ;                              (see Table 3.3)
```

```
condi_comp : '[' ! (expr)
      ( "=="| "!="| ">="| ">"| "<="| "<")
          (expr) ']' !
          ;                              (see Table 3.4)
```

logic-line : Comment| If-else | For-loop| Statement ; (see Table 3.7)
expression: is defined as all kinds of arithmetic operation. (see Table 3.2)

```
For example:   if  [n_scale >0] then {n_scale= - n_sacle;}
               If [n1<0 ] then {n2=n2+n1; n3=n2+2;} else {n3=0;}
```

3.6 Iteration: for loop

The `for-loop logic line` has the following syntax:

```
for_loop : FOR  for_init
            { (logic-line)*};
```

```
for_init :  '(' assignment ';'condition ';' assignment ')';
assignment : ID = Element ;  (Element is one of data type. See Table 3.6)
```

```
For example:  for(n1=1; [n1<10]; n1=n1+1) { n_f=n_f*n1; }
```

The nested If-else and For-loop and the intersection of If-else and for-loop are accepted in jGTL. For example:

```
          n1= -1;
          g1=< >;
  if [n1<0] then
      { for ( n2=0;[n2<3]; n2=n2+1)
        { p1={n2, n2, -n2,-n2};
          g1:JAppend(p1)->g1; } //End of for
  } //End of if
```

3.7 Statements

A statement is a logic line ended with a semicolon (;), It could be an assignment as n1=1+2*(3+2);, or a function as p1:Translate(1.1, 1.2, 1.3); or a system command as $write("Hello, world.");.

3.7.1 Assignments

In jGTL, assignment statement is defined as ID = Element; where ID is one following case: the identifiers for boolean start with 'b' or 'B'; the identifiers for string start with 's' or 'S'; the identifiers for number start with 'n' or 'N'; the identifiers for point start with 'p' or 'P'; the identifiers for group start with 'g' or 'G' and the identifiers for matrix start with 'm' or 'M'. The elements are the corresponding data structure: number, point, group and matrix as defined in (2) Data Types. Here are some examples.
B1= true; S1= "Hello"; N1=2.3+34/2; P1= {1.1, 1.2, 1.3, 0};
G1=<{1.1, 1.2, 1.3 0}, P1>; M1=[[1, 2, 3], [2, 1, 3], [3, 2, 1]];

The element can be assigned to several identifiers at the same line in jGTL:
N1=N2=3=2.3; P1=P2=P3= {1.1, 1.2, 1.3, 0};.

The right side of the assignment could be the original corresponding data type or the corresponding operation results. For example:
N1= 2+3/2; N1= N1-2; B1=true; B2= [N1>=2]&&B1; P1={N1, N2, N1-N2, 1};
G1= <p1, {N1, N2, -N1, -N2}>; M1=[[n1, n2], [3.3, n1-n2]].

Empty assignments for point and group are allowed. If point is assigned as p1={ }; the compile will automatically assign the default values to p1 as {0, 0, 0, 1}. If a group ID is assigned as g1=< >;, this group will contain nothing, and it can only use JAppend( ) function as a group data type.

3.7.2 Class Functions

In jGTL, function statement of class point, group and matrix is defined as ID:function-name(args)->ID. The First ID before ":" is the current class object to be operated, and the second ID after "->" is the return object ID, the result will be stored in the return object, and it does not need to be predefined. The number and format of arguments of each function is fixed and will be checked while compiling.

The jGTL function-names and their arguments for Point, Group and Matrix class are summarized as following.

(1) Functions for **point** class:

Point: Summary	
Return data type	**Function Fields**
Point	**JTranslate(number x, number y, number z)** Translate a point by the given ***x, y, z*** values
Point	**JRotate(number angle, number x, number y, number z)** Rotate a point in a counterclockwise direction about the vector from the original to the point ***(x, y, z)***. The ***angle*** parameter specifies the angles of rotation in degrees.
Point	JFrustum(number left, number right, number bottom, number top, number near, number far) Do a perspective-view frustum projection to a point. The frustum's view volume is defined by the parameters: ***(left, bottom, -near)*** and ***(right, top, -near)*** specify the (x, y, z) coordinates of the lower-left and upper-right corners of the near clipping plane; ***near*** and ***far*** give the distances from the viewpoint to the near and far clipping planes. They should always be positive.
Point	JOrtho(number left, number right, number bottom, number top, number near, number far) Do an orthographic parallel view volume projection to a point. **(*left, bottom, -near*)** and ***(right, top, -near)*** are points on the near clipping plane that are mapped to the lower-left and upper-right corners of the viewpoint windows, respectively. **(*left, bottom, -far*)** and ***(right, top, -far)*** are points on the far clipping plane that are mapped to the same respective corners of the viewpoint. Both ***near*** and ***far*** can be positive and negative.
Point	**JScale(number x, number y, number z)** Stretch (>1), shrink (<1), or reflect (<0) a point along the axes. The x, y, z coordinates of point is multiplied by the corresponding arguments ***x, y,*** or ***z***. Scaling with values greater than 1.0 stretches an object and using values less than 1.0 shrinks it. Scaling with a –1.0 value reflects an object across an axis.
Point	JUnit(number radius) Move the point towards or backwards the original point (0, 0, 0). The distance between the current point and the original point is specified by parameter ***radius***.
Point	JMulti(matrix m) Multiple the current point with the parameter 4x4 matrix ***m***, return a new point value.

Point	JsetScale(number scale) Re-organize the homogeneous coordinates of the current point. Return a same point with a different scale (the fourth coordinate) as the parameter ***scale***.
Group	JAppend(point p) This function will append the parameter point ***p*** at end of the current point and combine the two points into group and return the group.
Group	JAppend(group g) This function will append the parameter point group ***g*** at end of the current point and combine those points into a group and return the group.

(2) Functions for **group** class:

Group: Summary	
Return data type	**Function Fields**
Group	**JTranslate(number x, number y, number z)** Translate a group of points by the same given ***x, y, z*** values
Group	**JRotate(number angle, number x, number y, number z)** Rotate a group of points in a counterclockwise direction about the vector from the original to the point ***(x, y, z)***. The ***angle*** parameter specifies the angles of rotation in degrees.
Group	JFrustum(number left, number right, number bottom, number top, number near, number far) Do a perspective-view frustum projection to a group of points. The frustum's view volume is defined by the parameters: ***(left, bottom, -near)*** and ***(right, top, -near)*** specify the (x, y, z) coordinates of the lower-left and upper-right corners of the near clipping plane; ***near*** and ***far*** give the distances from the viewpoint to the near and far clipping planes. They should always be positive.
Group	JOrtho(number left, number right, number bottom, number top, number near, number far) Do an orthographic parallel view volume projection to a group of points. (***left, bottom, -near***) and ***(right, top, -near)*** are points on the near clipping plane that are mapped to the lower-left and upper-right corners of the viewpoint windows, respectively. (***left, bottom, -far***) and ***(right, top, -far)*** are points on the far clipping plane that are mapped to the same respective corners of the viewpoint. Both ***near*** and ***far*** can be positive and negative.
Group	**JScale(number x, number y, number z)** Stretch (>1), shrink (<1), or reflect (<0) a group of points along the axes. Each x, y, z coordinates of every point in the group is multiplied by the

	corresponding arguments ***x, y,*** or ***z***. Scaling with values greater than 1.0 stretches an object and using values less than 1.0 shrinks it. Scaling with a –1.0 value reflects an object across an axis.
Group	JUnit(number radius) Move the point towards or backwards the original point (0, 0, 0). The distance between the current point and the original point is specified by parameter ***radius***. By using this function, a group of points can be constrained on a sphere surface that centered on the original point.
Group	Jmulti(matrix m) Multiple the each of point in the current point with the parameter 4x4 matrix ***m***, return a group of new point values.
Group	JsetScale(number scale) Re-organize the homogeneous coordinates of the current group. Return the same points with a same scale (the fourth coordinate) as the parameter ***scale***.
Group	JAppend(group g) This function will append the parameter point group ***g*** at end of the current group and combine those two groups of points into a group and return the group.
Group	JAppend(point p) This function will append the parameter point ***p*** at end of the current group and return the new group.
Point	JgetPoint(number i) This function will return the point at **i**th of the current group. If the index I is out of the array, an empty point {} will be returned.
Number	JgetSize( ) Return the size of the current group (the number of points in the current group).

(3) Functions for **4x4 Matrix** class: (all the following functions are only for 4x4 matrix, The 4x4 matrix operations are commonly used in graphic 4D coordinates transfer operations.)

.

Matrix (4x4): Summary	
Return data type	Function Fields
Matrix(4x4)	JInverse( ) Get the inverse matrix of current matrix. If current matrix is singular, a empty 4x4 matrix will be returned. (each element is 0 in an empty matrix)
Matrix(4x4)	JsetIdentity( )

	Set an identity matrix. It will return a 4x4 identity matrix I. (In an identity matrix, each diagonal element is 1, and all other elements are 0.)
Matrix(4x4)	JsetValue(point p1, point p2, point p3, point p4) Set values to a 4x4 matrix. The rows of the matrix are assigned by the values of the parameter point ***p1, p2, p3, p4*** in order.
Matrix(4x4)	JsetValue(group g) Set values to a 4x4 matrix. The rows of the matrix are assigned by the values each point of parameter group ***g*** in order. Group ***g*** must have four points.
Matrix(4x4)	JsetElement(number row, number column, number value) Set the specific element of matrix as value ***value***, at the position (***row, column***).
Number	JgetElement(number row, number column) Get the specific element value of matrix at the position (***row, column***).
Matrix(4x4)	JsetRow(number row, point p) Set the specific row of a 4x4 matrix with the values of parameter point ***p***, the reset row number is given by parameter ***row***.
Point	JgetRow(number row) Get the specific row of current 4x4 matrix. The elements in the row which is given by parameter ***row*** will be returned as a point data type.
Matrix(4x4)	JsetColumn(number column, point p) Set the specific column of a 4x4 matrix with the values of parameter point ***p***, the reset column number is given by parameter ***row***.
Point	JgetColumn(number column) Get the specific column of current 4x4 matrix. The elements in the column which is given by parameter ***column*** will be returned as a point data type.
Matrix(4x4)	JTranspose( ) Return a matrix that is the transposed matrix of current 4x4 matrix. (Transposition means exchanging the element at (i, j) with the element at (j,i).)
Matrix(4x4)	JNegate() Return a matrix in which each element is the negative value of corresponding element of current matrix.
Matrix(4x4)	JaddScale(number n) Add each element of current 4x4 matrix with number ***n***.
Matrix(4x4)	JsubScale(number n) Subtract each element of current 4x4 matrix with number ***n***.
Matrix(4x4)	JmulScale(number n) Multiply each element of current 4x4 matrix with number ***n***.
Matrix(4x4)	JaddMatrix(matrix m) Add the current matrix with the parameter matrix ***m***.
Matrix(4x4)	JsubMatrix(matrix m)

	Subtract the current matrix with the parameter matrix ***m***.
Matrix(4x4)	**JmulMatrix(matrix m**) Multiply the current matrix with the parameter matrix ***m***.

3.7.3 System Commands

In jGTL language, system command is defined as $command (args);

$JprintLine((string s | number n | Boolean b)*) Print the corresponding values of the arguments in the same line of the console. There is no limit on the number of the arguments, and they could be any data type of string, number or Boolean. If the number of arguments is zero, an empty line will be printed.
$JprintList((point p| group g | matrix m)*) Print the corresponding values of the argument point ***p*** in a row, group ***g*** in several rows with each point in a row or matrix ***m*** in the matrix format on the console. If the number of arguments is zero, an empty line will be printed.
$JdrawPoint(point p, number red, number green, number blue, number radius) Draw a point as a sphere with color (**red, green, blue**) and **radius** on the graphic canvas. (Parameters **red, green and blue** are limited in the range of [0.0, 1.0])
$JdrawPoint(group g, number red, number green, number blue, number radius) Draw a group of point as spheres with color (**red, green, blue**) and **radius** on the graphic canvas. (Parameters **red, green and blue** are limited in the range of [0.0, 1.0])
$JdrawLine(point p1, point p2, number red, number green, number blue, number radius, number length) Draw a line between point **p1** and **p2** with color (**red, green, blue**) and **radius, length** on the graphic canvas. (Parameters **red, green and blue** are limited in the range of [0.0, 1.0])
$JdrawConfig(group g, number red, number green, number blue, number radius) Draw lines to link a group of points in sequence with color (**red, green, blue**) and **radius** on the graphic canvas. (Parameters **red, green and blue** are limited in the range of [0.0, 1.0])

Chapter 4

Project Plan

1.1 Team Responsibilities

As a media software project, each team member was given primary responsibility for the certain project goals as the table of 4.1. And each member should collaborate with each other in each software development phase. Every key Feature of the jGTL software was discussed among the members and decided by the major.

Member	Responsibilities
Zhenyu Zhu	Project Architecture Design, Tree walk, Graphic interface programming, 3D graphic programming, Documentation, Error Handling.
Santiago Ordonez	Project Organization, Documentation, Regression Testing, Demo.
Corey Kasten	Lexer and Parser, Compiler back-end, Regression Testing.

Table 4.1 Team Responsibilities

1.2 Project Timeline

The following timeline for the goals of the jGTL development were set for control the project process.

Software Process	Date	Software Development Goals
Requirement	01-30-03	Language Design, Topic Specification
Specification	02-18-03	White Paper complete.
	02-20-03	Development environment, tool, and programming style guide specification.
	02-20-03	Language grammar specification.
	03-27-03	Language reference manual complete.
Implementation	03-13-03	Lexer complete.
	03-15-03	Parser complete.
	04-08-03	Tree Walker complete.

	04-17-03	Graphic Interface complete.
Integration	04-22-03	Language Integration.
Maintenance	04-24-03	Error Handling.
	05-01-03	Program Optimization.
	05-08-03	Demo and Test complete.
Documentation	05-12-03	Final Paper complete.
Complete	05-14-03	Final Presentation and Review

Table 4.2 Project Timeline

4.3 Development Environment

The overall project was developed by Java SDK 1.4.0 in Windows XP operation system. Antlr 2.7.2 was used to handle source code scan, parse, and tree walk. The graphic interface was developed by java swing, and 3D graphic explorer was implemented by java 3D 1.3.

4.4 Programming Style Guide

The purpose of this document is to provide basic standards for collaborative code development and improve the readability of the codes. The guidelines contained herein reflect core best practices in pursuit of making one's code perspicuous and maintainable. Team member should try to follow these guidelines as much as possible in their work.

General Principle

Code files should be easy to read and sensibly laid out. Blocks should be intended a consistent width. Variables should have the names that clearly indicate their purpose. Non-trivial blocks of code should be accompanied by explanatory comments.

Declaration

One declaration per line is recommended since it encourages commenting. Do not put different types on the same line. Try to initialize local variables where they're declared. The only reason not to initialize a variable where it's declared is if the initial value depends on some computation occurring first.

Tabs, Indenting and Wrapping

Always use tab characters for indenting rather than spaces, and always use one tab per level of nested brackets. Keep line lengths no more than 80 columns (with tabs set to 4 columns). When wrapping a line, use a single additional tab for indenting the wrapped text.

Comments and Documentation

Use inline comments ("//" lines) to narrate the work being done by the code for any code blocks longer than a few lines. Indent the inline comment to the same level as the code, and use a blank line before the comment to emphasize the blocking.

Use JavaDoc comments ("/** ... */" blocks) for all classes, methods, and non-local variables. The only exception is when a number of static public final definitions are used to define constants, and the names of the constants describe their purpose.

Always make sure the comments are kept current-- out of date comments are worse than no comments at all, and code without comments is close to worthless.

4.5 Project Log

Date	Notes
01-27-03	Group birthday! First meeting.
01-28-03	Project initiated. Topic decided.
01-30-03	Language Designed. Main functions focus on graphic transformation.
02-18-03	White Paper complete.
02-20-03	Development environment, tool, and programming style guide specification.
02-20-03	Language grammar first draft.
02-23-03	Language manual first draft.
02-25-03	Initial lexer and paser with antlr
02-27-03	Test Phase I at the jGTL lexer, parse
03-27-03	Language reference manual complete.
03-13-03	Lexer complete.
03-15-03	Pares complete.
03-20-03	Start Tree walker

03-20-03	Solve list tree walk problem
03-25-03	Solve symbol table problem
03-27-03	Start java interface programming.
04-01-03	Test Phase II at the jGTL tree walker.
04-08-03	Tree Walker complete.
04-10-03	A simple Compiler ready without graphic interface.
04-14-03	The main window of interpreter complete.
04-16-03	The graphic explorer complete.
04-17-03	Graphic Interface complete.
04-20-03	Test Phase III at the jGTL graphic tree walker.
04-22-03	Language Integration.
04-28-03	Test Phase IV graphic interface&&overall test.
04-30-03	Error Handling.
05-01-03	Program Optimization.
05-03-03	Grammar Optimization.
05-06-03	Test Phase V program complexity test.
05-08-03	Demo and Test complete.
05-12-03	Final Paper complete.
05-14-03	Final Presentation and Review

Table 4.3 jGTL Project Log

Chapter 5

Architectural Design

5.1 Interpreter Components and interfaces

The major components of jGTL interpreter are lexer (jGTL.g), parser (jGTL.g), error checking (jGTL.g & jGTL.java), tree walker (jGTL.g & jGTL.java) and outputs control (IDE.java).

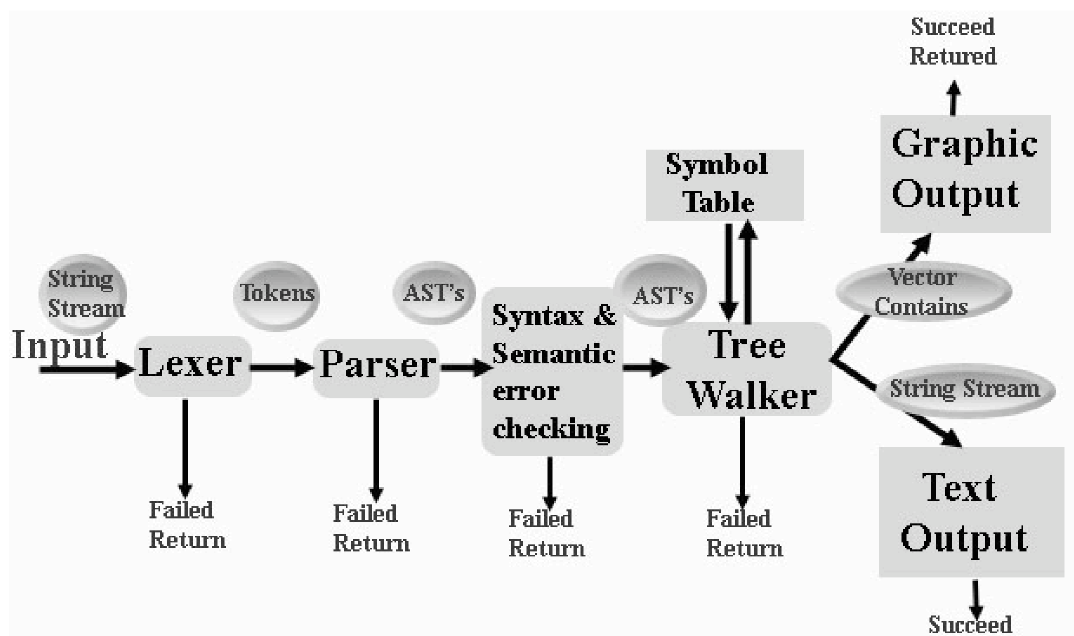


Fig 5.1 Major Components and interfaces of jGTL

In the format of string stream, the source code would pass through lexer (lexical analysis) of jGTL to become tokens. The tokens then pass through parser (syntax analysis) of jGTL to become Abstract Syntax Tree (AST) structures. Those AST's could go through the tree walker of jGTL to be executed one by one. At the running time Symbol Table of current program is created and updated. The error checking is combined with tree walker execution. Syntax and semantic checking were carried before the execution of each AST.

After tree walkers, there are two kinds of output. One is the non-graphic text output. It will be appended to the output text fields as buffered string stream. Another one is the graphic outputs. The jGTL will pass the information of points and lines to the graphic

explorer by vector contains. The explorer will read in those vectors and pass the parameters in them and draw the points and lines on the screen.

Any error during the scanning, parsing and tree walking process will terminate the running process to report the error messages.

5.2 Language Structure

The jGTL is a kind of line-interpreted languages. It will execute the source code line by line. There are four basic logic line formats that are listed in Table 5.1. The language will skip the comment lines. In If-Else case, the language would check the condition first, if it is true, then block of lines will be examined next; if it is false, else block of lines will be examined. In For-loop case, the language will iteratively execute the loop blocks until the condition is failed. In statements case (ended with semiclone ;), it could be assignment (characterized by = sign), function (characterized by : sign) and system commands (characterized by $ sign). So the language will execute each logic line in order. The symbol table will be kept and updated with line execution. Once there is error in the line execution, the running procedure will be terminated.

<table>
<tr><th colspan="6">Logic-line</th></tr>
<tr><td>Comment</td><td>If-Else</td><td>For-Loop</td><td colspan="3">Statement;</td></tr>
<tr><td rowspan="2">// one line
or
/*
BLOCK*/</td><td rowspan="2">IF condition
THEN
{Logic-line}
ELSE
{Logic-line}</td><td rowspan="2">For
(conditions)
{Logic-line}</td><td>Assignment</td><td>Function</td><td>System</td></tr>
<tr><td>ID =
Element</td><td>ID :
Function-
Name (args)
-> ID</td><td>$Command
(args)</td></tr>
</table>

Table 5.1 jGTL Language Structure

The main modules of tree walker are listed in Table 5.2

Parser	Tree walker function
Comment	skip
If-Else	Line_Walk( )
For-Loop	Line_Walk( )
Statements	Line_Walk( )
Assignment	Assignment_Walk( )
Function	function( )
System	command( )

Table 5.2 Main tree walker functions

5.3 Design Feature

As a project in Programming and Language Transfer course, jGTL is characterized as a graphic programming interpreter. There are some special features that have been implemented in jGTL V2.1 language. Those great features integrated into a friendly, robust, intuitive, and easy-use object-oriented graphic language – jGTL.

5.3.1 Graphic interface IDE

There is combined graphic development environment with the jGTL compiler. The user can use this language in a graphic interface. User can open and save source files by just clicking buttons, the source code could be edited on the text area. All the user should do to run piece of code is clicking "Run" button, them do not need go to console and handle javac and java commands. With this feature, the user can easily focus on the language grammar and graphic programming.

5.3.2 3D Graphic output and text output

In this project, java 3D API was used to produce the wonderful 3D graph on a simple Applet. And the text output of the jGTL language will be printed in the output fields with the error message and system information. The user can even turn off the graphic output. In jGTL v2.1, there are two kinds of outputs specified by the system commands. By using "JprintLine" or "JprintList", the program will print the values of the input ID into the output text fields. And by using one of "JdrawPoint", "JdrawLine" and "JdrawConfig", the program could popup a new window to show the 3D graph of the current configuration of points and lines. In this window, the user can rotate, zoom and translate the objects by mouse actions to change the perspective of viewing.

5.3.3 Object-oriented language

jGTL is a kind of object-oriented languages. Once the user initial an object as one of point, group and matrix classes, all the predefined methods of those classes can be used directly. This language also simplified the declaration of objects by the figuring the class name from the first letter of object identities.

5.3.4 Compile error report and handle

The developers of jGTL language spent a lot of time on error handing. In jGTL V2.1, first of all, if any error occurred, the running of line execution would be terminal to avoid further serious problems. Second, the compiler could report where is the error by print the logic line number of the source code. Third, the compiler could report to the user the reason of the error in most cases. If there is no error at all, a message will be printed to inform user that the running is succeed. All the error messages will be printed in the output text field.

5.3.5 Simultaneous help document checking

There is a “Help” button in the left side of the main window. Clicking it, a simple browser could popup to show the documents of jGTL language. The user could check out demo, tutorial, reference manual and language architecture while programming. This browser could read html format files, so it is easy to add new documents by future development.

Chapter 6

Test Plan

6.1 Goal

Software testing is a process of analyzing or operating software for the purpose of finding bugs. No test plan can aspire to catch every bug in a program. It is not realize to test every possible input to jGTL, here we just try to discover bugs as soon as possible in the process of software development. With carefully choice of unit tests, regression tests, white box and black box tests, the development process can evolve smoothly. The goal of test on jGTL project is to test systematically, to write enough tests to test each module of jGTL project separately and thoroughly.

6.2 Method

In jGTL project, rapid prototyping software development model was used, because there is a time limit and we did not know how far we could walk before end of the project. So there is totally five test phases were set up during the development process to guarantee the stabilization of out project. In each testing phase, the test will focus on the new implementations.

Corey Kasten	Test Phase I	lexer, parser
Corey Kasten	Test Phase II	tree walker.
Zhenyu Zhu	Test Phase III	tree walker on graphics.
Zhenyu Zhu	Test Phase IV	Graphic interface.
Santiago && Corey	Test Phase V	Overall maintenance.

Table 6.1 Project Test

There are two aspects to testing changes to a product. The first is checking that the required changes have been implemented correctly. The second aspect is ensuring that, in the course of making the required changes to the product, no other inadvertent changes were made. Therefore, once the programmer has determined that the desired changes have been implemented, the product must be tested against the previous test cases to make certain that the functionality of the rest of the product has not been compromised. So in last maintenance test phase, the regression testing method was used. We set up sixteen test cases, each case focus on some of the language important features. Those test cases are listed in Table 6.2. Regression testing was carried at each time we modify the source codes. A batch file was created in Windows to run all of the test cases one by one automatically and report the output and error message.

Test1.txt	Boolean, string, and number assignment and print
Test2.txt	Point, group, and matrix assignment and print
Test3.txt	Mathematic operation on number
Test4.txt	Logic operations on boolean
Test5.txt	comment
Test6.txt	If-else, for-loop
Test7.txt	Nested if-else and for-loop
Test8.txt	Point and Group functions
Test9.txt	Matrix functions
Test10.txt	Print command
Test11txt	Point and group drawing
Test12.txt	Line and configuration drawing
Test14.txt	Point scaling, translation, rotation and projection
Test15.txt	Lexer and parse Erroe handle
Test16.txt	Tree walker Error handle

Table 6.2 Test cases in Overall Testing

Here is an example of Text4.txt to test the logic operation of Boolean data type in jGTL.

```
n1= (sqrt(4)+2*3)/sqrt(2*2);
n2= abs(-1.5)*sin(3.14159/2);
B1= [n1>2];
B2= [n2==3]&&B1;
B3= B1||B2;
B4= [(n1+n2)>5]&&(!([n1>9]||[n2>1]));
B5= true;
B6= false;
```

```
if B6 then {n3=3;n4=7;} else {n3=4;}
$JprintLine("n1:   ", n1, "   n2:    ", n2);
$JprintLine("B1:   ", B1, "   B2:    ", B2);
$JprintLine("B3:   ", B3, "   B4:    ", B4);
$JprintLine("B5:   ", B5, "   B6:    ", B6);
```

If the test past, the out put would be as following:

```
Run Successfully!
n1:   4.0   n2:    1.4999999999986797
B1:   true   B2:    false
B3:   true   B4:    false
B5:   true   B6:    false
```

Chapter 7

Lessons Learned

7.1 Team Member's Lessons

<Zhenyu Zhu>

From this "huge" project, I learned a lot new staffs about compiler implementation, computer graphic and software engineering. I spend a lot spare time on this graphic transformation language project to make it robust, flexible, standard and friendly to user.

As a project of <Programming and Language Transformation> course, I get great training on implementing a computer language, especially the scanning, parsing, tree walking parts of the compiler implementation. In this project, we found that tree walker is more difficult than scanner and parser. One the language was specify, scanner and parser can be done quickly. We met two major difficulties in this project. They are list tree walker, and symbol table setting. Antlr is used as tool in this project. I found that sometimes using the class of Antlr directly is more easy and efficient than specifying a .g grammar file, because I am not family with the Antlr grammar. Antlr simplify the work of scanner, parser and tree walker.

I start to learn computer graphic since this project. I studied the transformation part of the graphic fields. I found it really interested me. The projection part was the most difficult of all the transformation to me. I spend a whole evening on figuring out the parameters that were used in perspective projection. Right those staff could be handled by java 3D API.

While the implementation of this project, a lot time was spent on the error message output and software maintenance. I found it is most challenge for a compiler to catch any syntax and semantic error as early as possible. Sometimes, it an error was not figured out early and handled could crash the program in the next steps. It is also difficult for a compiler to tell the user where the error came from.

Another difficult thing in compiler implementation that I learned from the jGTL project is making the grammar standard as most popular language. We tried to let jGTL language grammar similar as C/C++ and Java. But there are still some differences between jGTL language and standard language grammar due to the non-determination LR parsing.

As a “large” project, the jGTL required team working with other members. Some software engineering knowledge is very useful here, like uniform programming style, interface testing and team organization. Usually, our group would meet right after the class once a week. We worked together and learned from each other.

< *Santiago Ordonez*>

The project demanded meticulous planning and devoted collaboration of all the team members. We started this project with one idea in mind, and that was to develop a programming language that would facilitated the drawing of points and lines in 3 dimensional space. The fact that we had a clear and distinct idea about our project, gave us an edge to complete our project. From it’s initial stages we knew what was need and we starting distributing the project into tasks of manageable size. Through this process our project at each stage of the developing cycle, started to take shape and form. At each step we knew what could have been implemented and what has to be taken off the project. Since we were limited by time constrains we had to be very practical and implement tools and techniques that would allow us to complete the project. Improvisation such as replacing OpenGL with Java3D is one example. While the former is very strong and robust for intensive graphical applications, the latest is almost as strong and gave us the flexibility to integrate it quickly into the project and thus guarantying the successful completion of our programming language.

As a program manager I was mostly involved with the team integration and collaboration. Proper planning was indispensable for the successful completion of the jGTL project. I learned early in the developing cycle that having the team talking to each other as often as possible was key for the integration process. It reduced regression time made us more efficient. Each team member worked in the area in which they felt strongest thus increasing productivity and efficiency. Managing a project while taking other intense courses during a semester can be extremely stressful. But there is one thing I learned for sure out of this assignment; I learned to trust my team members and their work. Without the participation of even one of them, this project could not have been realized. Zhenyu Zhu with his keen programming and mathematical skills, Corey with his abstract visualization for the lexical analyzes and the parsing techniques and myself with my managerial skills and testing abilities came out to be one solid team. The key to our success was TEAM WORK.

< *Corey Kasten*>

Out of the many things I learned from this project, the most helpful are the tricks of working efficiently in a team. Since this was my first team project in my career, it has been a great learning experience. Of the many issues of team work, some of the lessons learned were the stress on modularization, documentation, and version control. I learned both from the successful use of some of these techniques as well as the failure to use others.

The first lesson is about modularization. The biggest key to succeeding to get a big project to work, like a compiler, is probably modularization. The whole ideology of the compiler is split into two phases: the front end, and the back end. The front end consists of the scanning, translating, and organization of source code into a useful form such as an abstract syntax tree. Essentially the whole front end was simplified by ANTLR, providing a notation to capture most functionality of a parser, lexer, and tree walker, with simple code. The only difficulty we ran into during the front end phase, was a modularization issue. Instead of learning how to usefully use ANTLR notation, we went in and hard coded much of the tree walking code manually in our jGTL.java file. Here we use the ANTLR library functions to explicitly walk the tree during static semantic analysis. We broke up the static semantic analysis into two parts: one defined in our jGTL.g file and the other in the jGTL.java file. This was probably a big mistake, since is brakes the modularization model and would make it difficult to expand the language. On a good note, we successfully managed to modularize our utility classes, handling the representation of the objects, defined in the source code, as java objects.

Another big lesson I learned is the importance of documentation. This goes hand in hand with version control. In order to keep on track of the progress of the project, we needed a good understanding of the additions and modifications made by any of the members of the project. This did not come so easily to us, as we did not use any standard version control tool, such as RCS. Instead we shipped around zipped files containing all the project work. The progress was not documented, rather orally transmitted among the members. Luckily there were only four members in our project and the project was not so massive, and we still were able to get the project together. In future projects, I hope to take advantage of the tools out there to strengthen team work efficiency.

The last lesson I learned from this project is to make sure you have your goals set. At the beginning of the semester we were not sure exactly how big we wanted our language to be and what format would we compile the source code into. I think we made a good choice to have a language with limited but powerful functionality. Instead of making a huge language, we have a simple one which extremely easy to learn and to quickly generate graphics. In the end we decided to compile our code inside an IDE. This makes it easy for the user to write programs compile them and see the results all in one window. It also took another step out of the process, namely, the intermediate representation, so that jGTL runs more like a scripting language.

7.2 Advice for the future

In the future, we should make the jGTL language more interesting, robust, flexible, standard and friendly to user.

We should make jGTL language consist with the most popular language like C/C++ and Java. The more standard the language is, the more friendly to user and more easy to learn.

Right now, jGTL is only an interpreter depend on Java. Later. If have more time, we should do the backend of compile implementation to free it from java. Otherwise, we could let the jGTL output the java source code to be a translator. This could let jGTL be a graphic tool in Java language. It is more useful, since user can embed jGTL source code to Java source code and use it with Java.

The jGTL could be extended later, now only sphere (point) and cylinder (line) are implemented, more other configuration could be created in jGTL. It could be more fun to play jGTL. And we also considered that let jGTL handle animation. It will be interesting and complete as a basic graphic language.

More graphic issue could be added to the jGTL language. Right now, jGTL only could handle graphic transformation. There are other interesting computer graphic topic could added to it, such as lighting, blending, text mapping.

Now, we have not included error recover. If the interpreter finds an error, it just stops running. Later, error recovering could be implemented to bypass some trivia errors. More error message could be implemented in the future to improve the robustness of the language.

Appendix A

Homogeneous Coordinates and Transformation Matrices

Homogeneous Coordinates

jGTL language usually deal with two- and three-dimensional vertices, but in fact all are treated internally as three-dimensional homogeneous vertices comprising four coordinates. Every column vector (x, y, z, w) represents a homogeneous vertex if at least one of its elements is nonzero. If the real number a is nonzero, then (x, y, z, w) and (ax,a y, az, aw) represent the same homogeneous vertex. A three-dimensional Euclidean space point (x, y, z) becomes the homogenous vertex with coordinates (x, y, z, 1.0), and the two-dimensional Euclidean point (x, y) becomes (x, y, 0.0, 1.0).

As long as w is nonzero, the homogeneous vertex (x, y, z, w) corresponds to three-dimensional point (x/w, y/w, z/w). if w=0.0, it corresponds to no Euclidean point, but rather to some idealized “point at infinite”.

Transforming Vertices

Vertex transformations (such as rotations, translations, scaling and shearing) and projections (such as perspective and orthographic projections) can all be represented by applying an appropriate 4×4 matrix to the coordinates representing the vertex. If **V** represents a homogeneous vertex and **M** is a 4×4 transformation matrix, than **MV** is the image of **V** under the transformation by **M**. (In computer-graphics applications, the transformations used are usually nonsingular –in other words, the matrix **M** can be inverted. This isn’t required, but some problems arise with nonsingular transformations.)

Transformation Matrices

Although any nonsingular matrix **M** represents a valid projective transformation, a few special matrices are particularly useful. Those matrices are listed in the following subsections.

Translation

Translate by a vector (x, y, z).

$$T = \begin{bmatrix} 1 & 0 & 0 & x \\ 0 & 1 & 0 & y \\ 0 & 0 & 1 & z \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } T^{-1} = \begin{bmatrix} 1 & 0 & 0 & -x \\ 0 & 1 & 0 & -y \\ 0 & 0 & 1 & -z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Scaling

Scale a vertex by (x, y, z).

$$S = \begin{bmatrix} x & 0 & 0 & 0 \\ 0 & y & 0 & 0 \\ 0 & 0 & z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and } S^{-1} = \begin{bmatrix} 1/x & 0 & 0 & 0 \\ 0 & 1/y & 0 & 0 \\ 0 & 0 & 1/z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Notice that $\mathbf{S}^{-1}$ is defined only if x, y, z are all nonzero.

Rotation

Rotate α clockwise around a vector $v = (x, y, z)^T$, and $u = v / \|v\| = (x', y', z')^T$.
Also let

$$S = \begin{bmatrix} 0 & -z' & y' \\ z' & 0 & -x' \\ -y' & x' & 0 \end{bmatrix} \text{ and } M = uu^T + (\cos\alpha)(1 - uu^T) + (\sin a)S$$

Then

$$R = \begin{bmatrix} m & m & m & 0 \\ m & m & m & 0 \\ m & m & m & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ where m represent elements from } \mathbf{M}\text{, which is a 3X3 matrix}$$

The **R** matrix is always defined. If x=y=z=0, then **R** is the identity matrix. You can obtain the inverse of R, by substituting -α for α.

Often, you are rotating about one of the coordinate axes, the corresponding matrices are as following:

$$R_x = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos a & -\sin a & 0 \\ 0 & \sin a & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_y = \begin{bmatrix} \cos a & 0 & \sin a & 0 \\ 0 & 1 & 0 & 0 \\ -\sin a & 0 & \cos a & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$R_z = \begin{bmatrix} \cos a & -\sin a & 0 & 0 \\ \sin a & \cos a & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Perspective Projection

Do a perspective-view frustum projection to a vertex. The frustum's view volume is defined by the parameters: ***(l, b, -n)*** and ***(r, t, -n)*** specify the (x, y, z) coordinates of the lower-left and upper-right corners of the near clipping plane; ***n*** and ***f*** give the distances from the viewpoint to the near and far clipping planes. They should always be positive.

$$R = \begin{bmatrix} 2n/_{r-l} & 0 & r+l/_{r-l} & 0 \\ 0 & 2n/_{t-b} & t+b/_{t-b} & 0 \\ 0 & 0 & -(f+n)/_{f-n} & -2fn/_{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix} \text{and}$$

$$R^{-1} = \begin{bmatrix} r-l/_{2n} & 0 & 0 & r+l/_{2n} \\ 0 & t-b/_{2n} & 0 & t+b/_{2n} \\ 0 & 0 & 0 & -1 \\ 0 & 0 & -(f-n)/_{2fn} & f+n/_{2fn} \end{bmatrix}$$

R is defined as long as $l \neq r$, $t \neq b$ and $n \neq f$.

Orthographic Projection

Do an orthographic parallel view volume projection to a vertex. ***(l, b, -n)*** and ***(r, t, -n)*** are points on the near clipping plane that are mapped to the lower-left and upper-right corners of the viewpoint windows, respectively. ***(l, b, -f)*** and ***(r, t, -f)*** are points on the far clipping plane that are mapped to the same respective corners of the viewpoint. Both ***n*** and ***f*** can be positive and negative.

$$R = \begin{bmatrix} \frac{2}{r-l} & 0 & 0 & \frac{r+l}{r-l} \\ 0 & \frac{2}{t-b} & 0 & \frac{t+b}{t-b} \\ 0 & 0 & \frac{-2}{f-n} & \frac{f+n}{f-n} \\ 0 & 0 & 0 & 1 \end{bmatrix} \text{ and}$$

$$R^{-1} = \begin{bmatrix} \frac{r-l}{2} & 0 & 0 & \frac{r+l}{2} \\ 0 & \frac{t-b}{2} & 0 & \frac{t+b}{2} \\ 0 & 0 & \frac{f-n}{-2} & \frac{n+f}{2} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Appendix B

jGTL Grammar (BNF Notation)

```
//Main Structure
exprlist    :(sentence)*EOF^
             ;

sentence    :  Comment
               | if_else
               | for_loop
               | statement
                ;
```

/**********Part I   Comment *****************************/**

```
// Comment for a single line

Comment_line
      : DSLASH   (~('\n'|'\r'))+  ('\n'|'\r'|('\r''\n'))
     ;
// Comment for a block
Comment_block
     : SSLASH  (~'/')+ FSLASH
      ;
//Include the Comment for line and block
Comment
      :Comment_line  |Comment_block
      ;
```

/***********Part II if-else && Part III for loop **************/**

```
block    : LCUR^ (sentence)*RCUR!
           ;
// If-else and For-loop condition.

condition : (condi_sing|(LPAREN!condition RPAREN!)|ID_B|TRUE|FALSE)?
           ((AND^|OR^|NOT^)
           (condi_sing|(LPAREN!condition RPAREN!)|ID_B|TRUE|FALSE))*
        ;


condi_sing :LBRA! (expr)
```

```
        (EQUAL^|NOT_EQUAL^|LTE^|LT^|GTE^|GT^)
        (expr) RBRA!
        ;

if_else  : IF^  condition  THEN! block
        (ELSE! block)? {System.out.print("****If-Else****");}
       ;

for_init : LPAREN!assign_n SEMI!condition SEMI! assign_n RPAREN!
       ;

for_loop : FOR^ for_init block {System.out.print("****For-Loop****");}
       ;
```

/***************Part IV.   Statement ********************/**

```
statement :   (assi_function | system_command)SEMI^
            ;

assi_function:   ((ID_P|ID_G|ID_M)FUNC) =>
            (ID_P|ID_G|ID_M)FUNC^ FUNC_NAME
               LPAREN! (
               (expr|ID_M|ID_P|ID_G|matrix|point|group)
          (PERI!(expr|ID_M|ID_P|ID_G|matrix|point|group))*
               )?   RPAREN!
               RETURN (ID_P|ID_G|ID_M|ID_N)
        | (assign_n| assign_s|assign_p|assign_g|assign_m|assign_b)
      ;

system_command: SYST^ SYST_NAME LPAREN! (
            (  ID_S|ID_B|ID_P|ID_G|ID_M
             |StringConstant|point|group|matrix|expr)
             (PERI!(expr|ID_P|ID_B|ID_M|ID_S|ID_G
             |StringConstant|point|group|matrix))*)?
            RPAREN!      {System.out.print("**System_Command**");}
            ;



//assign  : assign_n| assign_s|assign_p|assign_g|assign_m|assign_l
//       ;

assign_n : ID_N ASSI^ (ID_N ASSI!)* expr
      ;
```

```
assign_s : ID_S ASSI^ (ID_S ASSI!)* StringConstant
       ;

assign_p : ID_P ASSI^ (ID_P ASSI!)*  point
       ;

assign_g : ID_G ASSI^ (ID_G ASSI!)* group
       ;

assign_m : ID_M ASSI^ (ID_M ASSI!)* matrix
       ;

assign_b : ID_B ASSI^ (ID_B ASSI!)* condition
         ;

// calculation
expr : (mexpr ((PLUS^|MINUS^) mexpr)*)| ((PLUS^|MINUS^) mexpr)+;
mexpr: pexpr ((STAR^|DIV^|MOD^) pexpr)*;
pexpr: mole  (POW^ mole)?;
mole : (SQRT^|ABS^|SIN^|COS^|TAN^)atom|atom;
atom : NUMBER|(LPAREN! expr RPAREN!)
      | ID_N
      ;

//list : LCUR^ (list | NUMBER)* RCUR! {System.out.print("**list**");}
//     ;

point: LCUR^ ((expr)(PERI! expr)*)? RCUR!{System.out.print("**point**");}
     ;

group: LT^ ((ID_P|point)(PERI! (ID_P|point))*)? GT!{System.out.print("**group**");}
     ;

matrix : LBRA^((matrix_p)(PERI! matrix_p)+)?RBRA!
{System.out.print("**matrix**");}
     ;

protected
matrix_p: LBRA^ (expr)(PERI! expr)* RBRA!
      ;

//******************Lexer ******************************

WS:     ( ' ' | '\t' | '\n'{newline();} | '\r''\n' {newline();})
           ;
```

```
LPAREN: '(';
RPAREN: ')';
LBRA :  '[';
RBRA :  ']';
LCUR :  '{';
RCUR :          '}';
STAR  : '*';
PLUS  : '+';
MINUS : '-';
DIV   : '/';
MOD   : '%';
POW   : '^';
ASSI  : '=';
FUNC  : ':';
POUND : '#';
SEMI  : ';';
SYST  : '$';
PERI  : ',';
DSLASH : "//" ;
SSLASH : "/*" ;
FSLASH : "*/" ;

EQUAL     : "==";
NOT_EQUAL : "!=";
LTE       : "<=";
LT        : '<';
GTE       : ">=";
GT        : '>';
IF        : "if"|"IF";
THEN      : "then"|"THEN";
ELSE      : "else"|"ELSE";
FOR       : "for"|"FOR";
NOT       : '!';
AND       : "&&";
OR        : "||";
SQRT      : "sqrt";
ABS       : "abs";
SIN       : "sin";
COS       : "cos";
TAN       : "tan";
```

```
ID_N  options {testLiterals=true;}
    : ('n'|'N')
      ('a'..'z'|'A'..'Z'|'_'|'0'..'9')+
    ;

ID_S  options {testLiterals=true;}
    : ('s'|'S')('a'..'h'|'j'..'p'|'r'..'z'|'A'..'Z'|'_'|'0'..'9')
      ('a'..'z'|'A'..'Z'|'_'|'0'..'9')*
    ;

ID_P  options {testLiterals=true;}
    : ('p'|'P')
      ('a'..'z'|'A'..'Z'|'_'|'0'..'9')+
    ;

ID_G  options {testLiterals=true;}
    : ('g'|'G')
      ('a'..'z'|'A'..'Z'|'_'|'0'..'9')+
    ;

ID_M  options {testLiterals=true;}
    : ('m'|'M')
      ('a'..'z'|'A'..'Z'|'_'|'0'..'9')+
    ;

ID_B  options {testLiterals=true;}
     : ('b'|'B')
        ('a'..'z'|'A'..'Z'|'_'|'0'..'9')+
     ;

protected
DIGIT : '0'..'9';

protected
INT   : (DIGIT)+;

protected
EXPONENT : ('e'|'E')('+'|'-')?(DIGIT)+;

//C like float and integer
NUMBER :
        (  (INT) ((('.')( ((INT)(EXPONENT)?) | (EXPONENT)? ))|EXPONENT)? )
       | ('.'(INT)(EXPONENT)?)
       ;
```

```
StringConstant
   : '"'!
     (~('"' | '\n') | ('"'!'"'))*
     '"'!
    ;

RETURN    : "->";

FUNC_NAME :  "JRotate"|"JScale"|"JFrustum"|"JOrtho"|"JTrans"("late"|"pose")
            |"JMulti"|"JUnit"|"JAppend"|"JInverse"|"JNegate"
            |"Jset"("Identity"|"Element"|"Row"|"Column"|"Value"|"Scale")
            |"Jget"("Element"|"Column"|"Row"|"Point"|"Size")
            |("Jadd"|"Jsub"|"Jmul")("Scale"|"Matrix")
          ;

SYST_NAME :  "JprintL"("ine"|"ist")|"Jdraw"("Point"|"Line"|"Config")
          ;
TRUE    :  "true"|"TRUE"
        ;
FALSE   :  "false"|"FALSE"
        ;
```

Appendix C

Code List

There are totally one antlr grammar file jGTL.g (lexer, parser and tree walker) and nine jave files jGTL project V2.1(May, 2003). IDE.java is the main code to control the whole integrated development environment of jGTL language. Browser_Help.java is a help documents reading frame for IDE and the MyExplore.java offer an 3D explore window for the graphic output of jGTL language. IDE use jGTL.java to do line execution of the input source code. There are four auxiliary classes jList, jPoint, jMatrix4D and jDictionary for the tree walker and execution.

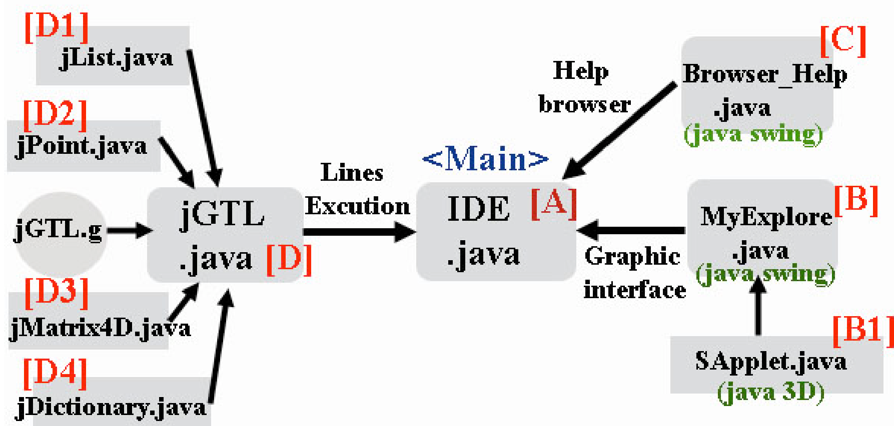


Fig C.1 Source Codes of jGTL Project

[A. IDE.java]

```
/* ***********************************
*** IDE.java
*** Author: Zhenyu Zhu
*** Date:  04/12/03
*** Main method of jGTL Interpreter
*** IDE control
*** Extend javax.swing.JFrame
*** Use MyExplore.java
*** Use Browser_Help.java
*** Use Antlr 2.7.2
*************************************/
//Initial Frame
initComponents()

//Open a source file
jButton1ActionPerformed(java.awt.event.ActionEvent evt)
```

```
//Save a source file
jButton2ActionPerformed(java.awt.event.ActionEvent evt)

//Line Execute from the source txt of input field
jButton3ActionPerformed(java.awt.event.ActionEvent evt)

//Control the Graphic Explorer
jButton4ActionPerformed(java.awt.event.ActionEvent evt)
//Clear input text field
jButton5ActionPerformed(java.awt.event.ActionEvent evt)

//Clear text output field
jButton6ActionPerformed(java.awt.event.ActionEvent evt)

//Open the Help browser
jButton7ActionPerformed(java.awt.event.ActionEvent evt)
```

[B MyExplore.java]

```
/* ************************************
*** MyExplore.java
*** Author: Zhenyu Zhu
*** Date:  04/16/03
*** Explorer of graphic output
*** Extend javax.swing.JFrame
*** Use Sapplet.java
************************************/
//Initial Frame
init()

//Open the applet to show the graphic output
open()
```

[B1 Sapplet.java]

```
/* ************************************
*** IDE.java
*** Author: Zhenyu Zhu
*** Date:  04/16/03
*** A Applet to show the language's graphic output
*** Extend java.awt.Applet
*** Use java 3D 1.3 API
************************************/
//Initial Applet and draw points and lines with the parameters
// java 3D is used to draw the 3D objects
init()
```

```
//Draw the open graphic interface of jGTL compiler.
//java 3D is used to generate the animation.
open_interface()
```

[C Browser_Help.java]

```
/* ************************************
*** Browser_Help.java
*** Author: Zhenyu Zhu
*** Date:  04/27/03
*** Help Files Reader – A simple Browser
*** Extend javax.swing.JFrame
*************************************/
//Initial the Frame
initComponents()

//Show Demo page
jButton1ActionPerformed(java.awt.event.ActionEvent evt)

//Show Tutorial page
jButton1ActionPerformed(java.awt.event.ActionEvent evt)

//Show Manual page
jButton1ActionPerformed(java.awt.event.ActionEvent evt)

//Show Architecture page
jButton1ActionPerformed(java.awt.event.ActionEvent evt)

//Close this Frame
jButton1ActionPerformed(java.awt.event.ActionEvent evt)
```

[D. jGTL.java]

```
/* ************************************
*** jGTL.java
*** Author: Zhenyu Zhu
*** Date:  04/01/03
*** Main method of jGTL Line Interpreter
*** Tree Walking control and Error Checking
*** Use jList.java
*** Use jPoint.java
*** Use jMatrix4D.java
*** Use jDictionary.java
*** Use Antlr 2.7.2
*************************************/
```

//Walking on each line of input source code
Line_Walk(AST parse_s)

//Assignment tree walker
Assignment_Walk

//Functions of Class Point, group and matrix tree walker
Function_Execute(jList li_e)

//System command tree walker
System_Command(jList li_e)

//Print a double number with specified digital number
df_print(Object o)

[D1 jList.java]

```
/* ************************************
*** jList.java
*** Author: Corey Kasten
*** Date:  02/27/03
*** jList interface
*** Extend java.util.Vector
************************************/
//Add a string, double, float and jList
add()

//Return the element number of this list
dim()

//Check whether the list is a matrix format
isMatrix()

//Print all the elements
print()
```

[D2 jPoint.java]

```
/* ************************************
*** jPoint.java
*** Author: zhenyu Zhu
*** Date:  03/10/03
*** jPoint interface
************************************/
//Assign A point by a jList
list_assign(jList pi)
```

//Return the point as format of jList
jList get_list()

//All kinds of point transformation methods for graphic operations
unit(jList param)
set_scale(jList param)
translate(jList param)
scale(jList param)
rotate(jList param)
frustum(jList param)
ortho(jList param)
multi(jList param)

[D3 jMatrix4D.java]

/* ************************************
*** jMatrix4D.java
*** Author: zhenyu Zhu
*** Date: 03/12/03
*** jMatrix4D interface –Only 4x4 matrix
*** Modify from javax.vetmath
************************************/
//Assign A Matrix by a jList
list_assign(jList pi)

//Return the Matrix as a jList Type
jList get_list()

//All kinds of Matrix4D methods for graphic operations
setIdentity()
setElement(jList param)
getElement(jList param)
getRow(jList param)
getColumn(jList param)
setRow(jList param)
setColumn(jList param)
add_scale(jList param)
sub_scale(jList param)
mul_scale(jList param)
mul_matrix(jList param)
add_matrix(jList param)
sub_matrix(jList param)
transpose()
invert()
equals(jList param)
negate()

[D4 jDictionary.java]

```
/* ************************************
*** jDictionary.java
*** Author: zhenyu Zhu
*** Date:  03/11/03
*** jDictionary interface
*** For Symbol Table Record
*** Extend java.util.Hashtable
*************************************/
//Assign the dictionary by a jList
jlist_assign(jList assi_list)

//Get element from the dictionary by the key
getElement(Object key)

//Get elements as jList format
getList(Object key)

//Return the size of the dictionary
getSize()

//Add new element
putElement(Object key, Object item)

//print the dictionary
print()

/************************
***** jGTL.g
***** Author: Corey Kasten
***** Author: zhenyu Zhu
***** Language Lexer, Parser,
***** and Tree Walker
*************************/
```

Appendix D

jGTL Demo

jGTL is a high-level Interpreter graphic language. It is a powerful 3D graphic tool for implementing high quality 3D graph by simple coding.

1. Drawing Beautiful 3D Graph

1.1 Sphere Ring

jGTL Code:

```
//Initilization
n_max_x = 0.8; //x direction dimension
n_max_z = 0.4; // z direction dimension
n_max_y = 0.4; //y direction dimension
n_r = 0.05;  //Sphere readius

/*********Draw Double Courves**************/
for(ni=-n_max_x; [ni<=(n_max_x+0.05)]; ni=ni+0.05)
{
 n_para=sin(ni*3.1416/n_max_x);
  n_z = n_max_z*n_para;
  n_y= n_max_y*n_para;
  n_cr= 0.5+0.5*n_para;
  n_cg=0.5+0.5*sin( (ni/n_max_x+1/3)*3.1416);
  n_cb=0.5+0.5*sin( (ni/n_max_x+2/3)*3.1416);

  p1={ni, n_y, -n_z};
  p2={-ni, n_y,  n_z};
  $JdrawPoint(p1, 0.0, n_cg, n_cb, n_r);
  $JdrawPoint(p2, n_cr, 0.0, n_cb, n_r);
}

//Draw a color circle
for(ni=0; [ni<=360]; ni=ni+5)
{
    n_cr=0.5+0.5*sin( (2*ni/360)*3.1416);
    n_cg=0.5+0.5*sin( (2*ni/360+1/3)*3.1416);

   p3={n_max_x*cos(ni*2*3.1416/360), 0.0,
         n_max_x*sin(ni*2*3.1416/360)};
   $JdrawPoint(p3, n_cr, n_cg, 0.0, n_r);
```

```
  }
//Draw the center point
p0={0.0, 0.0, 0.0};
$JdrawPoint(p0, 1.0, 0.0, 0.0, 0.16);
```

Graphic output:

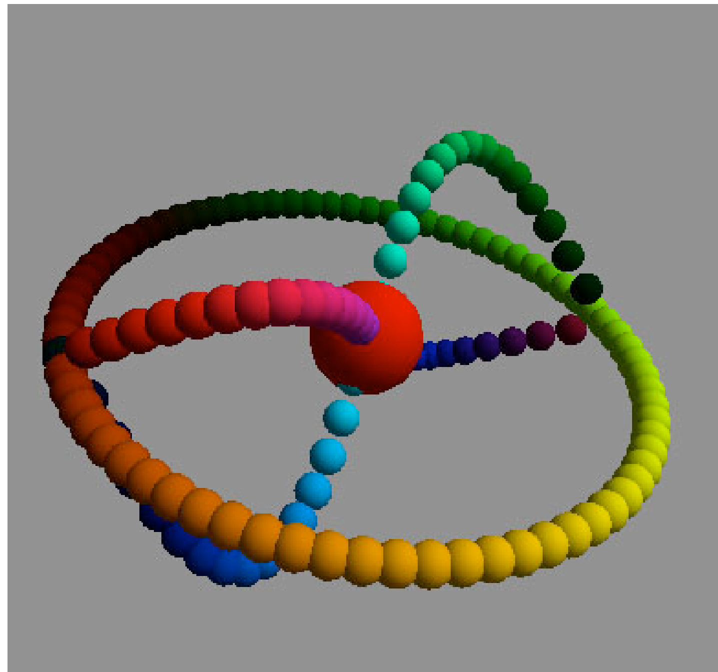


Fig D.1 Sphere Ring

1.2 Colorful Circle

jGTL Code:

```
//Initial Points
p0={0.0, 0.0, 0.0, 2.0};
p1={-1.0, 0.0, 0.0, 2.0};
p2={ 1.5, 0.0, 0.0, 2.0};
//Rotate around Z-axis
p2:JRotate(60, 0.0, 0.0, 1.0)->p21;
p2:JRotate(-60, 0.0, 0.0, 1.0)->p22;

//Define a group include all points
g1=<p0, p1>;
$JdrawPoint(g1, 1.0, 0.0, 0.0, 0.16);
```

```
/*************Draw Line*************/
$JdrawLine(p0, p1, 1.0, 0.0, 1.0, 0.06);

/******** rotation ***********/
for(ni=5; [ni<=360]; ni=ni+5) {
     p21:JRotate(ni, 1.0, 0.0, 0.0)->pt1;
     n_cr= 0.5+0.5*sin(3.1416*(ni/180+2/3));
     n_cg= 0.5+0.5*sin(3.1416*ni/180);
     n_cb= 0.5+0.5*sin(3.1416*(ni/180+1/3));
    $JdrawPoint(pt1,      n_cr, n_cg, n_cb, 0.06);
    $JdrawLine(p0, pt1, n_cr, n_cg, n_cb, 0.04);
 }
```

Graphic output:

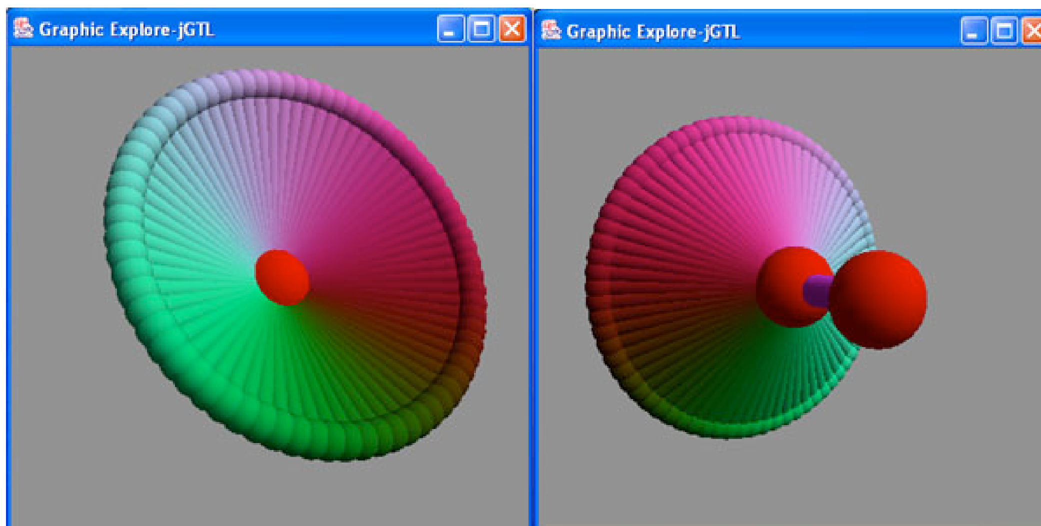


Fig D.2 Colorful Circle

2. Drawing Solid State Structure -- Applications in Physics

2.1 Body Center Structure –NaCl

jGTL Code:

```
//Initilization
n_bond=0.4; //Bond length
n_ra =0.12;   // A type Atom radius
n_rb= 0.18;    // B type Atom radius


//Initial All A type Points
```

```
g1= < > ;
for(ni=-n_bond; [ni<= n_bond]; ni=ni+n_bond){
   for(nj=-n_bond; [nj<= n_bond]; nj=nj+n_bond){
      for(nk=-n_bond; [nk<= n_bond]; nk=nk+n_bond){
      p_ijk={ni, nk, nj};
      g1:JAppend(p_ijk)->g1;
     }
   }
} //End of for

//Initial B type Points
g2= < > ;
n0=-n_bond+n_bond/2;
for(ni=n0; [ni<=n_bond]; ni=ni+n_bond){
   for(nj=n0; [nj<=n_bond]; nj=nj+n_bond){
      for(nk=n0; [nk<=n_bond]; nk=nk+n_bond){
       p_b1={ni, nk, nj};
      g2:JAppend(p_b1)->g2;
     }
   }
} //End of for

/*******Draw  All Atoms ********************/
 $JdrawPoint(g1, 0.0, 1.0, 0.0, n_ra);
 $JdrawPoint(g2, 1.0, 0.0, 0.0, n_rb);
//End
```

Graphic output:

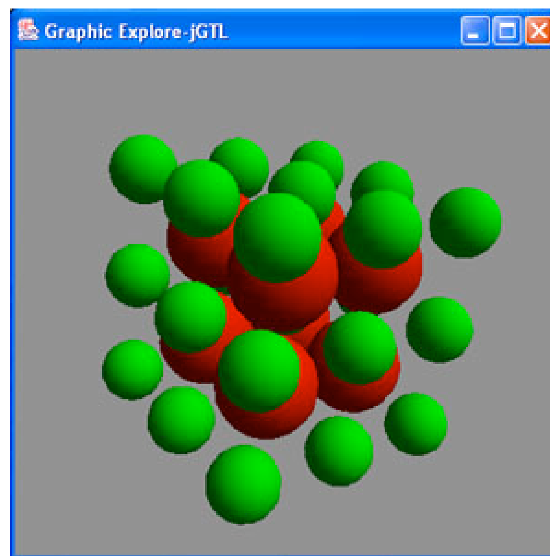


Fig D.3 BCC Structure

2.2 Face Center Structure – CsCl

jGTL Code:

```
//Initilization
n_bond=0.4; //Bond length
n_ra =0.10;   // A type Atom radius
n_rb= 0.12;    // B type Atom radius


//Initial All A type Points
g1= < > ;
for(ni=-n_bond; [ni<= n_bond]; ni=ni+n_bond){
   for(nj=-n_bond; [nj<= n_bond]; nj=nj+n_bond){
      for(nk=-n_bond; [nk<= n_bond]; nk=nk+n_bond){
     p_ijk={ni, nk, nj};
     g1:JAppend(p_ijk)->g1;
    }
  }
} //End of for

//Initial B type Points
g2= < > ;
n0=-n_bond+n_bond/2;
n1= n_bond+n_bond/2;
for(ni=n0; [ni<=n1]; ni=ni+n_bond){
   for(nj=n0; [nj<=n1]; nj=nj+n_bond){
      for(nk=n0; [nk<=n1]; nk=nk+n_bond){

         if ([ni<=n_bond]&&[nk<=n_bond])
             then   {  p_b1={ni, nk, nj-n_bond/2};  }

         if ([nk<=n_bond]&&[nj<=n_bond])
            then {p_b2={ni-n_bond/2, nk, nj};}

         if ([ni<=n_bond]&&[nj<=n_bond])
           then {p_b3={ni, nk-n_bond/2, nj};}
     g2:JAppend(p_b1)->g2;
     g2:JAppend(p_b2)->g2;
     g2:JAppend(p_b3)->g2;
    }
  }
} //End of for

/*******Draw  All Atoms ********************/
```

```
  $JdrawPoint(g1, 0.0, 1.0, 0.0, n_ra);
  $JdrawPoint(g2, 1.0, 0.0, 0.0, n_rb);
//End
```

Graphic output:

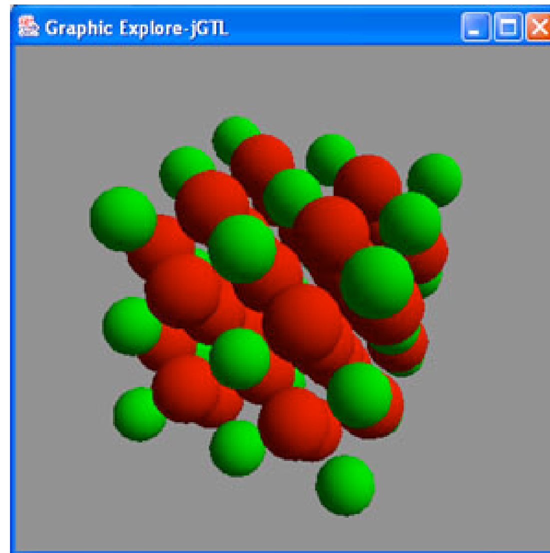


Fig D.4 FCC Structure

2.3 S3 Space Group Structure –Silicon

jGTL Code:

```
//Initilization
n_bond=0.2; //Bond length
n_r1=0.05;   //Atom radius
n_r2=0.02;  //Bond radius

//Initial Four Points for each lattice
p0={0.0, 0.0, 0.0};
p1={ 0.0, n_bond, 0.0};
p1:JRotate(-109.5, 0.0, 0.0, 1.0)->p2;
p2:JRotate(120, 0.0, 1.0, 0.0)->p3;
p3:JRotate(120, 0.0, 1.0, 0.0)->p4;

//Calculate the translation displacements
n_angle = (109.5-90.0)*3.1416/180;
n_dy   =  n_bond*(1+sin(n_angle));
n_dx1 = n_bond*cos(n_angle);
n_dx2 = n_dx1/2;
n_dz2  = n_dx1*sin(3.1516/3);
```

```
//$JprintLine("n_dx1:   ", n_dx1, "  n_dx2:  ", n_dx2);
//$JprintLine("n_dy:   ", n_dy);
//$JprintLine("n_dz2:    ", n_dz2);

//Define a group include all points in a lattice
g10=<p0, p1, p2, p3, p4>;

//Get the first layer lattices---Seven lattices
g10:JTranslate(n_dx1+n_dx2, 0.0, n_dz2)->g11;
g10:JTranslate(n_dx1+n_dx2, 0.0, -n_dz2)->g12;
g10:JTranslate(-n_dx1-n_dx2, 0.0, n_dz2)->g13;
g10:JTranslate(-n_dx1-n_dx2, 0.0, -n_dz2)->g14;
g10:JTranslate(0.0, 0.0, 2*n_dz2)->g15;
g10:JTranslate(0.0, 0.0, -2*n_dz2)->g16;

//Orginaze the atoms into a group
g11:JAppend(g12)->g11;
g13:JAppend(g14)->g13;
g15:JAppend(g16)->g15;

g10:JAppend(g11)->g1;
g13:JAppend(g15)->g13;

/***********Get First layer atoms ************/
g1:JAppend(g13)->g1;
/***********Get four layers ****************/
g_total=<>;
g_total:JAppend(g1)->g_total;
for(ni=1; [ni<=3]; ni=ni+1)
{
   g1:JTranslate(ni*n_dx1, -ni*n_dy, 0.0)->g_temp;
   g_total:JAppend(g_temp)->g_total;
}

g_total:JTranslate(-0.2, 0.4, 0.0)->g_total;

/*******Draw  All Atoms ********************/
$JdrawPoint(g_total, 0.0, 1.0, 0.0, n_r1);


/***Draw Lines in a lattice ***Double for loop***/
g_total:JgetSize( )->n_total;
$JprintLine(n_total);
for(ni=0; [ni<n_total]; ni=ni+5)
{
  g_total:JgetPoint(ni)->p_center;
```

```
    for(nj=ni+1; [nj<=(ni+4)]; nj=nj+1)
     {   g_total:JgetPoint(nj)->ptemp;
         $JdrawLine(p_center, ptemp, 1.0, 0.0, 0.0, n_r2);
       }
}
//End
```

Graphic output:

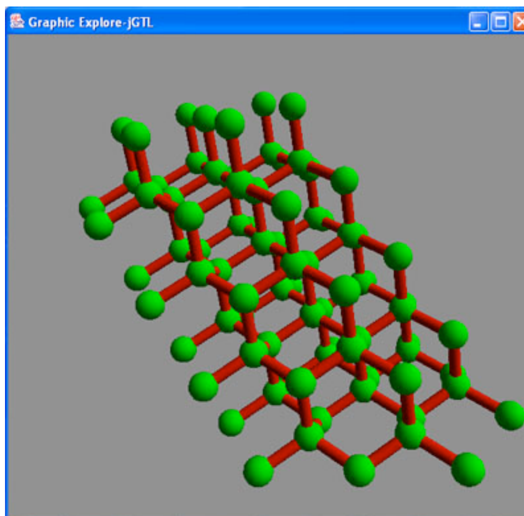


Fig D.5 Silicon Structure

3. Drawing Molecular Structure – Applications on Biology

3.1 Amino Acids –Aspartic acid

jGTL Code:

```
/Parameters initialization
n_r_c=0.08;  // C Atom radius
n_r_o=0.11; // O Atom radius
n_r_n=0.1; // N Atom radius
n_r_h=0.06; // H Atom radius
n_r_bond=0.03; // Basic Bond radius
n_r_bond_s=0.04; // Strong Bond readius
n_r_bond_w=0.02; // Week Bond readius


//Initialization All Atom Coordinates
p_c   ={0.49, 0.33, 0.13 };
```

```
p_o   ={0.54, 0.43, 0.35 };
p_n   ={0.14, 0.36, -0.21};
p_hn ={0.08, 0.23, -0.35};
p_ca ={0.20, 0.26, 0.06};
p_ha ={0.07, 0.35, 0.17};

p_cb  ={0.17, -0.04, 0.09};
p_hb1={0.24, -0.13, -0.06};
p_hb2={0.26, -0.10,  0.25};

p_cg   ={-0.12, -0.13, 0.11};
p_od1 ={-0.29, 0.03,  0.21};
p_od2 ={-0.20, -0.35, 0.04};
p_ht    ={-0.47, -0.05, 0.21};

//Group the same atom type
g_c=<p_c, p_ca, p_cb, p_cg>;
g_o=<p_o, p_od1, p_od2>;
g_n=<p_n>;
g_h=<p_hn, p_ha, p_hb1, p_hb2, p_ht>;

$JdrawPoint(g_c, 1.0, 1.0, 0.0, n_r_c);//yellow
$JdrawPoint(g_o, 1.0, 0.0, 0.0, n_r_o); //red
$JdrawPoint(g_n, 0.0, 1.0, 0.0, n_r_n);   //green
$JdrawPoint(g_h, 0.0, 0.0, 1.0, n_r_h); //blue

//Draw C-C Bond
$JdrawConfig(g_c, 1.0, 1.0, 0.0, n_r_bond); //yellow
///Draw C-N Bond-strong bound
$JdrawLine(p_ca, p_n, 0.0, 1.0, 0.0, n_r_bond_s); //green
///Draw C-O Bond-strong bound
$JdrawLine(p_c, p_o, 1.0, 0.0, 0.0, n_r_bond_s); //red
$JdrawLine(p_cg, p_od1, 1.0, 0.0, 0.0, n_r_bond_s); //red
$JdrawLine(p_cg, p_od2, 1.0, 0.0, 0.0, n_r_bond_s); //red
///Draw H Bond-week bound
$JdrawLine(p_n, p_hn, 0.0, 0.0, 1.0, n_r_bond_w); //blue
$JdrawLine(p_ca, p_ha, 0.0, 0.0, 1.0, n_r_bond_w); //blue
$JdrawLine(p_cb, p_hb1, 0.0, 0.0, 1.0, n_r_bond_w); //blue
$JdrawLine(p_cb, p_hb2, 0.0, 0.0, 1.0, n_r_bond_w); //blue
$JdrawLine(p_od1, p_ht, 0.0, 0.0, 1.0, n_r_bond_w); //blue
```

Graphic output:

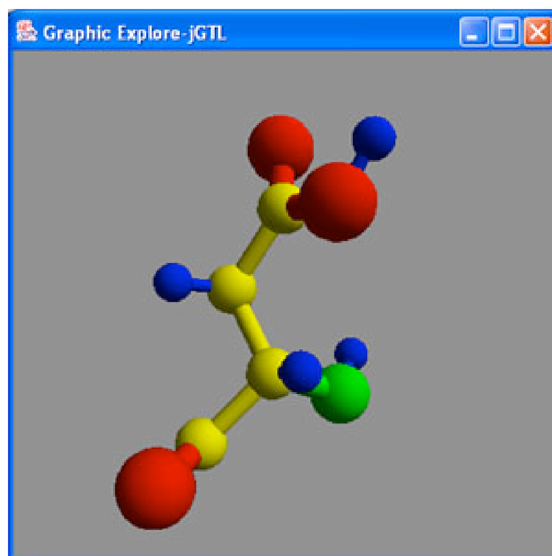


Fig D.6 Aspartic Acid Structure

3.2 Residue Conformation –Aspartic acid

jGTL Code fragments:

```
/****************Rotation Start**********/
//Define rotation group
g1=< p_od1, p_od2, p_ht>;
//Find the rotation bond
n_x=-0.12-0.17;
n_y=-0.13+0.04;
n_z= 0.11-0.09;
n_r=sqrt(n_x*n_x+n_y*n_y+n_z*n_z);
n_x=n_x/n_r;
n_y=n_y/n_r;
n_z=n_z/n_r;
//p_r={n_x, n_y, n_z};
//p_0={ 0.0, 0.0, 0.0};
//$JdrawLine(p_0, p_r, 0.0, 0.0, 0.0, 0.04);
//Rotate
g1:JTranslate(-0.17, 0.04, -0.09)->g1;
g1:JRotate(45, n_x, n_y, n_z)->g2;
g1:JRotate(90, n_x, n_y, n_z)->g3;
g1:JRotate(135, n_x, n_y, n_z)->g4;


g2:JAppend(g3)->g2;
g2:JAppend(g4)->g2;
```

```
g2:JTranslate(0.17, -0.04, 0.09)->g2;

//Group the same atom type
g_c=<p_c, p_ca, p_cb, p_cg>;
g_o=<p_o, p_od1, p_od2>;
g_n=<p_n>;
g_h=<p_hn, p_ha, p_hb1, p_hb2, p_ht>;

//Apend the new Atoms
g2:JgetSize( )->n0;
//$JprintLine(n0);
for(ni=0; [ni<n0]; ni=ni+3){
        g2:JgetPoint(ni)->p_temp_o1;
        g_o:JAppend(p_temp_o1)->g_o;
        g2:JgetPoint(ni+1)->p_temp_o2;
        g_o:JAppend(p_temp_o2)->g_o;
        g2:JgetPoint(ni+2)->p_temp_h;
        g_h:JAppend(p_temp_h)->g_h;
     //Draw New Bonds
       $JdrawLine(p_temp_o1, p_temp_h,
                 0.0, 0.0, 1.0, n_r_bond_w); //blue
       $JdrawLine(p_cg, p_temp_o1,
                 1.0, 0.0, 0.0, n_r_bond_s);//red
       $JdrawLine(p_cg, p_temp_o2,
                 1.0, 0.0, 0.0, n_r_bond_s);//red
}
/*************Rotation End *************/
```

Graphic output:

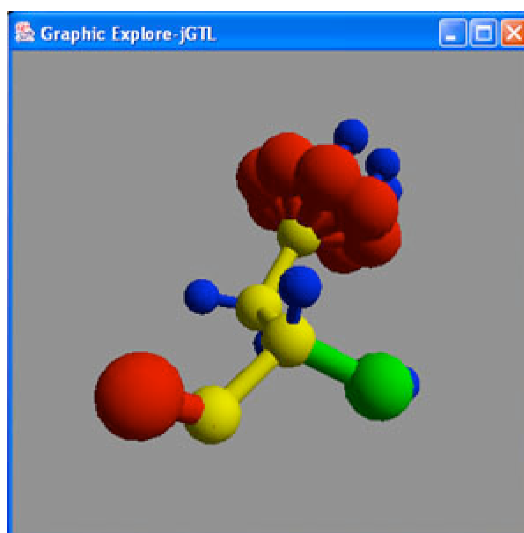


Fig D.7 Aspartic acid Conformations

3.3 Residue Rotomer –Aspartic acid

Graphic output:

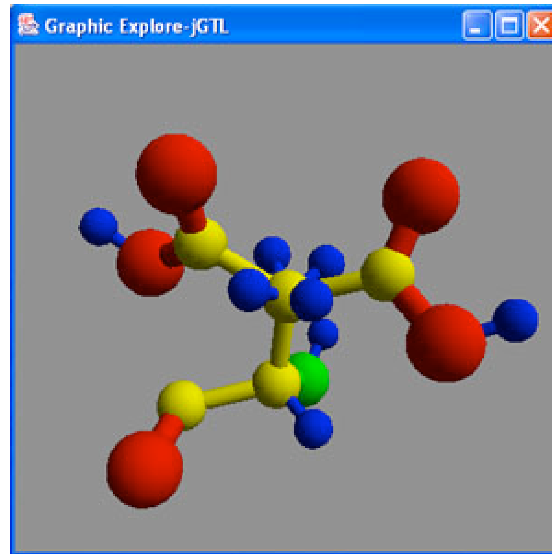


Fig D.8 Aspartic Acid Rotomer

Reference

[1] Christopher Conway, Cheng-Hong Li and Megan Pengelly, ***Pencil: A Petri Net Specification Language for Java***, December 2002.

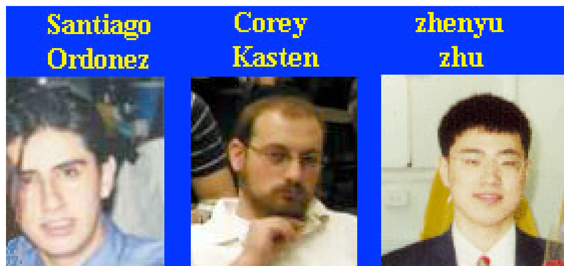
[2] Mason Woo, Jackie Neider and Tom Davis, ***Open GL programming Guide***, second edition, ISBN: 0201461382, 1997.

[3] ***The Java Language, An Overview*** , http://java.sun.com/docs/overviews/java/java-overview-1.html.

[4] Stephen R. Schach, **Object-Oriented and Classical Software Engineering,** Fifth Edition, ISBN: 0072395591, 2002.

[jGTL Group member:]

zhenyu zhu zz2103@columbia.edu
Santiago Ordonez rso2003@columbia.edu
Corey Kasten crk2009@columbia.edu



Graphical Transformation Language
May, 2003