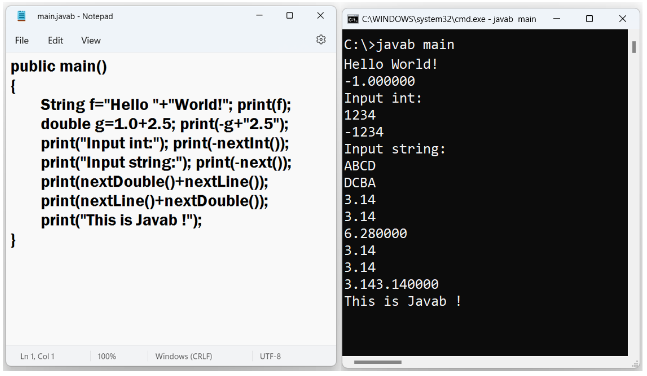
PM4VR: A Scriptable Parametric Modeling Interface for Conceptual Architecture Design in VR (Supplementary Material)

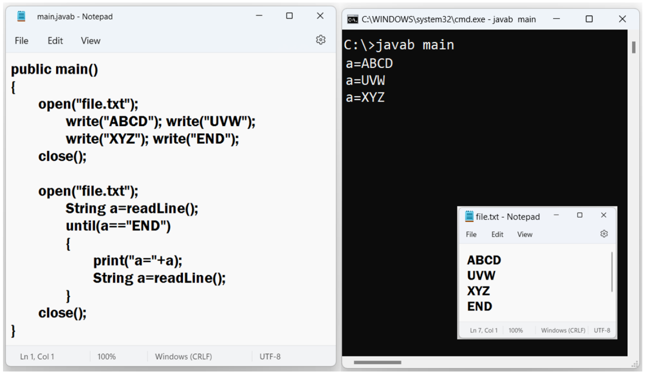
Wanwan Li
University of South Florida
Tampa, Florida, USA


main.javab - Notepad
File Edit View
Ln 1, Col 1 100% Windows (CRLF) UTF-8
C:\WINDOWS\system32\cmd.exe - javab main

```
public main()
{
      String f="Hello "+"World!"; print(f);
      double g=1.0+2.5; print(-g+"2.5");
      print("Input int:"); print(-nextInt());
      print("Input string:"); print(-next());
      print(nextDouble()+nextLine());
      print(nextLine()+nextDouble());
      print("This is Javab !");
}
```

```
C:\>javab main
Hello World!
-1.000000
Input int:
1234
-1234
Input string:
ABCD
DCBA
3.14
3.14
6.280000
3.14
3.14
3.143.140000
This is Javab !
```

Figure 1: Example of standard I/O using Java$^\flat$.


main.javab - Notepad
File Edit View
Ln 1, Col 1 100% Windows (CRLF) UTF-8
C:\WINDOWS\system32\cmd.exe - javab main
file.txt - Notepad
Ln 7, Col 1 100% Windows (CRLF) UTF-8

```
public main()
{
      open("file.txt");
            write("ABCD"); write("UVW");
            write("XYZ"); write("END");
      close();

      open("file.txt");
            String a=readLine();
            until(a=="END")
            {
                  print("a="+a);
                  String a=readLine();
            }
      close();
}
```

```
C:\>javab main
a=ABCD
a=UVW
a=XYZ
```

```
ABCD
UVW
XYZ
END
```

Figure 2: Example of file I/O using Java$^\flat$.

1 JAVA$^\flat$ LANGUAGE PROGRAMMING

1.1 Introduction to Java$^\flat$ Language

Java$^\flat$ language is a language similar to Java language, its basic syntax is similar to C language, but its style is more concise and simple than Java language. According to the definition of Java$^\flat$ language, a source code file called Java$^\flat$ script (*.javab) can only contain one class, so as to make the structure of its program very simple and straightforward. In addition, the Java$^\flat$ language satisfies the closure and completeness between operations on three data types including integers, doubles, and strings. That means the operation between every two data types is defined. At the same time, this language implements the function call of external classes and data access. In that way, one class can access multiple classes, and other classes can play an important role in data encapsulation.

1.2 Java$^\flat$ Language Programming Fundamentals

The basic functions of the Java$^\flat$ language include the declaration and assignment of three data types, arithmetic expressions of integer and double-precision floating-point numbers, 3D vector operations, string operations, logical operations, array operations, function call, method call, class declaration, class initialization, instance assignment, for loop statement, while loop statement, do-while loop statement, until loop statement, break statement, function return, if-else statement, switch-case statement, this call statement, etc. This subsection will introduce some Java$^\flat$ programming examples on these fundamental topics.

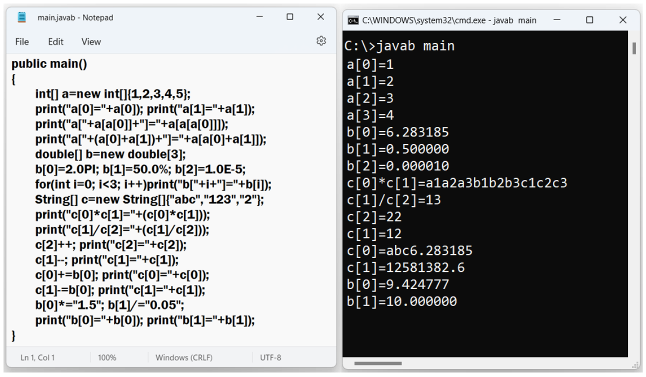
Standard I/O. The standard Input/Output (I/O) of the console is often the first program written by beginners. Here we also provides a "Hello World!" program using Java$^\flat$ script. The extension of Java$^\flat$ script is "*.javab", for example, the source code main.javab and its execution result are shown in Figure 1.

File I/O. The function of Java$^\flat$ file reading and writing is similar to the file reading and writing in C language, but the language style is the action stream style, that is, the programmer only needs to pay attention to the execution action of the instruction and does not need to care who executes the instruction. This code style is just right Avoid pointers and or some other instructions unrelated to the execution logic in the program. The code of this example and the execution result are shown in Figure 2. This action stream style is also extended to the Java$^\flat$ 3D modeling interface, so as to provide users with a convenient graphics programming platform.

Array. Java$^\flat$ supports static definition and declaration of array. User can create a static array by indicating the length of the array or by listing each element in the static array which is similar to the Java language. Figure 3 shows the execution result of a Java$^\flat$code that tests the declaration of arrays.

Method. There are five return types of methods in Java$^\flat$, one is the public type, which specifies the execution entry of the virtual machine. The other three types of methods are corresponding to three basic types (int, double, and String) and the last type is the

ACM SIGGRAPH VRCAI 2022, December 27-29, 2022, Guangzhou, China
© 2022 Association for Computing Machinery.
ACM ISBN 978-1-4503-XXXX-X/18/06…$15.00
https://doi.org/XXXXXXX.XXXXXXX

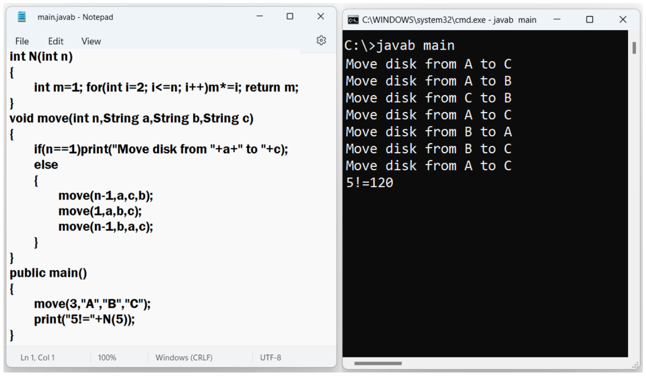


```
public main()
{
      int[] a=new int[]{1,2,3,4,5};
      print("a[0]="+a[0]); print("a[1]="+a[1]);
      print("a["+a[a[0]]+"]="+a[a[a[0]]]);
      print("a["+(a[0]+a[1])+"]="+a[a[0]+a[1]]);
      double[] b=new double[3];
      b[0]=2.0PI; b[1]=50.0%; b[2]=1.0E-5;
      for(int i=0; i<3; i++)print("b["+i+"]="+b[i]);
      String[] c=new String[]{"abc","123","2"};
      print("c[0]*c[1]="+(c[0]*c[1]));
      print("c[1]/c[2]="+(c[1]/c[2]));
      c[2]++; print("c[2]="+c[2]);
      c[1]--; print("c[1]="+c[1]);
      c[0]+=b[0]; print("c[0]="+c[0]);
      c[1]-=b[0]; print("c[1]="+c[1]);
      b[0]*="1.5"; b[1]/="0.05";
      print("b[0]="+b[0]); print("b[1]="+b[1]);
}
```

```
C:\>javab main
a[0]=1
a[1]=2
a[2]=3
a[3]=4
b[0]=6.283185
b[1]=0.500000
b[2]=0.000010
c[0]*c[1]=a1a2a3b1b2b3c1c2c3
c[1]/c[2]=13
c[2]=22
c[1]=12
c[0]=abc6.283185
c[1]=12581382.6
b[0]=9.424777
b[1]=10.000000
```

Figure 3: Example of array declaration using Java$^\flat$.

void. Methods can be called not only internally (call), but also be called externally (invoke). It can be seen that in java$^\flat$, there is no distinction between public methods and private methods. It is important to point out that the Java$^\flat$ method supports recursion, as can be seen from the running example shown in Figure 4, user can implement a recursive algorithm for Tower of Hanoi in Java$^\flat$.



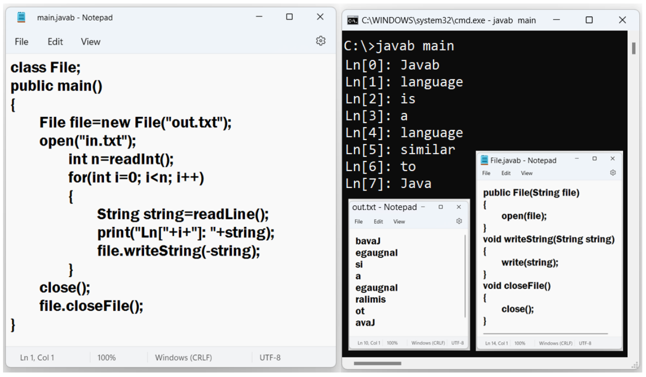
```
int N(int n)
{
      int m=1; for(int i=2; i<=n; i++)m*=i; return m;
}
void move(int n,String a,String b,String c)
{
      if(n==1)print("Move disk from "+a+" to "+c);
      else
      {
            move(n-1,a,c,b);
            move(1,a,b,c);
            move(n-1,b,a,c);
      }
}
public main()
{
      move(3,"A","B","C");
      print("5!="+N(5));
}
```

```
C:\>javab main
Move disk from A to C
Move disk from A to B
Move disk from C to B
Move disk from A to C
Move disk from B to A
Move disk from B to C
Move disk from A to C
5!=120
```

Figure 4: Example of method call using Java$^\flat$.

Class and Object. Java$^\flat$ is an object-oriented language. When compiling a Java$^\flat$ script, JVM$^\flat$ first constructs a class table according to the inclusions of the class in the first line of the Java$^\flat$ script, compiles and imports according to each class name in the list one by one in a recursive manner (that is, multiple classes can be called from one main class, and external classes can call more other classes). According to this idea, when Java$^\flat$ compiles the included class, it only needs to load the attribute fields of the method table in the class file, such as method name, method pointer, and method return type. In this way, it can compile the external class in the current syntax translator, so as to implement the object-oriented programming mechanism correctly.

The execution of each class is allocated with a separate JVM$^\flat$, and each virtual machine and its corresponding object have different data segments but share the same code segment so that different objects for the same class share the same instructions but present completely different execution results. The reason is that the virtual machines corresponding to the objects for each class have completely different data segments, which means that they



```
class File;
public main()
{
      File file=new File("out.txt");
      open("in.txt");
            int n=readInt();
            for(int i=0; i<n; i++)
            {
                  String string=readLine();
                  print("Ln["+i+"]: "+string);
                  file.writeString(-string);
            }
      close();
      file.closeFile();
}
```

```
C:\>javab main
Ln[0]: Javab
Ln[1]: language
Ln[2]: is
Ln[3]: a
Ln[4]: language
Ln[5]: similar
Ln[6]: to
Ln[7]: Java
```

```
bavaJ
egaugnal
si
a
egaugnal
ralimis
ot
avaJ
```

```
public File(String file)
{
      open(file);
}
void writeString(String string)
{
      write(string);
}
void closeFile()
{
      close();
}
```

Figure 5: Example of class and object using Java$^\flat$.

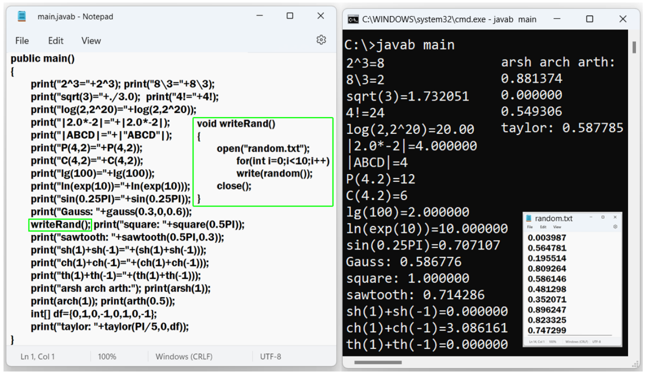
will have different running results when loaded, which just reflects the idea of object-oriented encapsulation. More specifically, when the JVM$^\flat$ is running and if there is a method call to another class in the program, the virtual machine instance of the current class will search the machine table and find that method's entry address specified in another class's corresponding virtual machine instance. Then, it will set the instruction pointer to the method entry address and execute another virtual machine until the RET command is encountered and returned to the current host virtual machine, thereby JVM$^\flat$ implements the invoke to object methods.

On Java$^\flat$ platform, its object-oriented technology supports the module encapsulation of 3D modeling, and it is of great significance in building multi-branch 3D scenes. For example, users can use object-oriented technology to write a scene in a class and import that scene by calling the method of that class. This can help user to include the parametric designs from other developers without seeing others' code, which is also the way this platform implements the idea of encapsulation. As an object-oriented language, Java$^\flat$ provides a very flexible encapsulation of classes. This encapsulation satisfies security and simplicity. A static class can be activated as a dynamic class through the new method, and multiple classes can be derived from a class through the declared method. Through "." operator, internal method of external class can be called.

Figure 5 shows an example that is using class and objects in Java$^\flat$. In this case, the main.javab achieves the function of reading a file (in.txt), printing line numbers, and writing a new file (out.txt) with each line of sentence reversed (e.g., "Javab" becomes "bavaJ"). However, as specified earlier that file I/O in Java$^\flat$ is implemented based on action stream, therefore, only one file pointer is allocated to each object. To solve this case, another class implemented in File.javab writes the out.txt while the main.javab is reading in.txt.

1.3 Advanced Java$^\flat$ Language Programming

In order to facilitate the 3D programming in Java$^\flat$, we implement advanced math calculation and data structures in Java$^\flat$ that support special calculations related to parametric modeling. There are three new data types defined for this purpose, they are 3D vector, parametric function, and transform group. As advanced programming topics, these data structures are extremely useful in parametric modeling with Java$^\flat$ and PM4VR. This subsection will introduce some Java$^\flat$ programming examples on these advanced topics.



```
public main()
{
      print("2^3="+2^3); print("8\3="+8\3);
      print("sqrt(3)="+./3.0);  print("4!="+4!);
      print("log(2,2^20)="+log(2,2^20));
      print("|2.0*-2|="+|2.0*-2|);
      print("|ABCD|="+|"ABCD"|);
      print("P(4,2)="+P(4,2));
      print("C(4,2)="+C(4,2));
      print("lg(100)="+lg(100));
      print("ln(exp(10))="+ln(exp(10)));
      print("sin(0.25PI)="+sin(0.25PI));
      print("Gauss: "+gauss(0.3,0,0.6));
      writeRand(); print("square: "+square(0.5PI));
      print("sawtooth: "+sawtooth(0.5PI,0.3));
      print("sh(1)+sh(-1)="+(sh(1)+sh(-1)));
      print("ch(1)+ch(-1)="+(ch(1)+ch(-1)));
      print("th(1)+th(-1)="+(th(1)+th(-1)));
      print("arsh arch arth:"); print(arsh(1));
      print(arch(1)); print(arth(0.5));
      int[] df={0,1,0,-1,0,1,0,-1};
      print("taylor: "+taylor(PI/5,0,df));
}
```

```
void writeRand()
{
      open("random.txt");
            for(int i=0;i<10;i++)
            write(random());
      close();
}
```

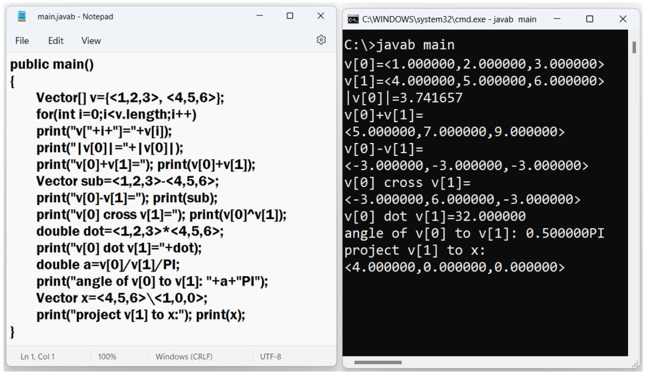
```
C:\>javab main
2^3=8                 arsh arch arth:
8\3=2                 0.881374
sqrt(3)=1.732051      0.000000
4!=24                 0.549306
log(2,2^20)=20.00     taylor: 0.587785
|2.0*-2|=4.000000
|ABCD|=4
P(4.2)=12
C(4.2)=6
lg(100)=2.000000
ln(exp(10))=10.000000
sin(0.25PI)=0.707107
Gauss: 0.586776
square: 1.000000
sawtooth: 0.714286
sh(1)+sh(-1)=0.000000
ch(1)+ch(-1)=3.086161
th(1)+th(-1)=0.000000
```

```
0.003987
0.564781
0.195514
0.809264
0.586146
0.481298
0.352071
0.896247
0.823325
0.747299
```

Figure 6: Example of math calculations using Java$^\flat$.

Math. Java$^\flat$ has powerful math calculation support for commonly used math functions such as trigonometric functions, it also has some specially designed simplified math representations such as nth power: x^n, nth root: $\sqrt[n]{x}$, square root $\sqrt{x}$, factorial n!, absolute value |x|, permutation $P(m,n)$, combinations $C(m,n)$, sawtooth function, hyperbolic functions, Taylor series expansion, etc. Some example code for testing math calculations in Java$^\flat$ are written in main.javab and its execution result is shown in Figure 6.

Vector. As a special data structure supported by Java$^\flat$, a 3D vector consists of three doubles as its components of x, y, and z. User can declare a vector directly using **Vector v**=<x, y, z>; There are multiple vector operations directly implemented in Java$^\flat$ compiler inclduing add, subtract, magnitude or length, cross product, dot product, angle between to vectors, project one vector to another, etc. Some example code for testing vector calculations in Java$^\flat$ are written in main.javab and its execution result is shown in Figure 7.

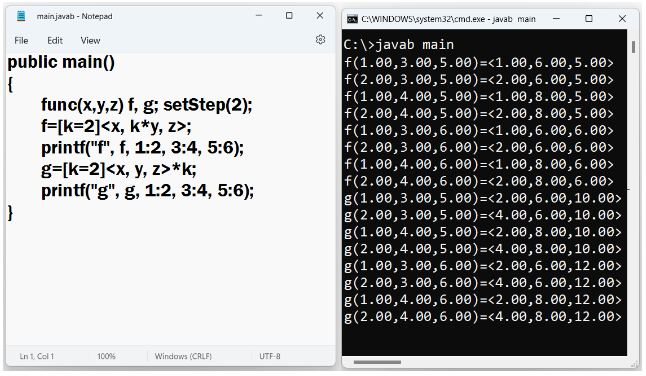


```
public main()
{
      Vector[] v={<1,2,3>, <4,5,6>};
      for(int i=0;i<v.length;i++)
      print("v["+i+"]="+v[i]);
      print("|v[0]|="+|v[0]|);
      print("v[0]+v[1]="); print(v[0]+v[1]);
      Vector sub=<1,2,3>-<4,5,6>;
      print("v[0]-v[1]="); print(sub);
      print("v[0] cross v[1]="); print(v[0]^v[1]);
      double dot=<1,2,3>*<4,5,6>;
      print("v[0] dot v[1]="+dot);
      double a=v[0]/v[1]/PI;
      print("angle of v[0] to v[1]: "+a+"PI");
      Vector x=<4,5,6>\<1,0,0>;
      print("project v[1] to x:"); print(x);
}
```

```
C:\>javab main
v[0]=<1.000000,2.000000,3.000000>
v[1]=<4.000000,5.000000,6.000000>
|v[0]|=3.741657
v[0]+v[1]=
<5.000000,7.000000,9.000000>
v[0]-v[1]=
<-3.000000,-3.000000,-3.000000>
v[0] cross v[1]=
<-3.000000,6.000000,-3.000000>
v[0] dot v[1]=32.000000
angle of v[0] to v[1]: 0.500000PI
project v[1] to x:
<4.000000,0.000000,0.000000>
```

Figure 7: Example of vector calculations using Java$^\flat$.

Function. Another important feature of Java$^\flat$ is the introduction of the definition of parametric function. The concept of function in Java$^\flat$ is different from the function call in C. Rather, in Java$^\flat$, a function is a mathematical representation of a 3D vector or a 1D scalar that is changing along with parameters. Users can define any mathematical function using this type of coding structure: **func**(param1, param2, ...) **f**=[arg1=..., arg2=..., ...] <x, y, z>; After setting up the steps for each argument in **f**, users can print the values of function **f** by calling Java$^\flat$method **printf**(**f**, range1, range2, ...);



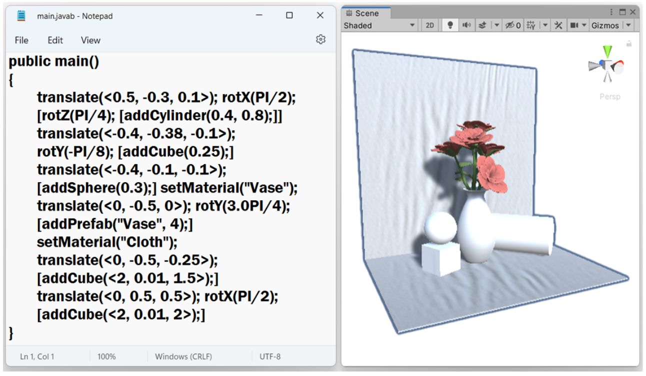
```
public main()
{
      func(x,y,z) f, g; setStep(2);
      f=[k=2]<x, k*y, z>;
      printf("f", f, 1:2, 3:4, 5:6);
      g=[k=2]<x, y, z>*k;
      printf("g", g, 1:2, 3:4, 5:6);
}
```

```
C:\>javab main
f(1.00,3.00,5.00)=<1.00,6.00,5.00>
f(2.00,3.00,5.00)=<2.00,6.00,5.00>
f(1.00,4.00,5.00)=<1.00,8.00,5.00>
f(2.00,4.00,5.00)=<2.00,8.00,5.00>
f(1.00,3.00,6.00)=<1.00,6.00,6.00>
f(2.00,3.00,6.00)=<2.00,6.00,6.00>
f(1.00,4.00,6.00)=<1.00,8.00,6.00>
f(2.00,4.00,6.00)=<2.00,8.00,6.00>
g(1.00,3.00,5.00)=<2.00,6.00,10.00>
g(2.00,3.00,5.00)=<4.00,6.00,10.00>
g(1.00,4.00,5.00)=<2.00,8.00,10.00>
g(2.00,4.00,5.00)=<4.00,8.00,10.00>
g(1.00,3.00,6.00)=<2.00,6.00,12.00>
g(2.00,3.00,6.00)=<4.00,6.00,12.00>
g(1.00,4.00,6.00)=<2.00,8.00,12.00>
g(2.00,4.00,6.00)=<4.00,8.00,12.00>
```

Figure 8: Example of function programming using Java$^\flat$.

Figure 8 shows an example of Java$^\flat$programming using function. In this example, the user defines two 3D vector functions which are **f** and **g** respectively. Both have three parameters of x, y, and z along with one constant argument of $k = 2$. In this case, the user setup steps as 2 and setup range of $x \in [1,2]$, $y \in [3,4]$, and $z \in [5,6]$, therefore there will be $2 \times 2 \times 2 = 8$ steps print out for each function. This feature is the basis for parametric modeling in Java$^\flat$, as most parametric designs use this kind of mathematical representation.

Transform Group. Transform group is a specially designed grammar in Java$^\flat$ so as to transform a group of 3D objects in the scene. The transform group is following with a tree structure. That means in the transform group, a child's transformation is based on the father's transformation. Due to this feature, we use a specific type of code block to represent the tree structure of the transform group. This code block is denoted as "[" and "]".



```
public main()
{
      translate(<0.5, -0.3, 0.1>); rotX(PI/2);
      [rotZ(PI/4); [addCylinder(0.4, 0.8);]]
      translate(<-0.4, -0.38, -0.1>);
      rotY(-PI/8); [addCube(0.25);]
      translate(<-0.4, -0.1, -0.1>);
      [addSphere(0.3);] setMaterial("Vase");
      translate(<0, -0.5, 0>); rotY(3.0PI/4);
      [addPrefab("Vase", 4);]
      setMaterial("Cloth");
      translate(<0, -0.5, -0.25>);
      [addCube(<2, 0.01, 1.5>);]
      translate(<0, 0.5, 0.5>); rotX(PI/2);
      [addCube(<2, 0.01, 2>);]
}
```

Figure 9: Example of transform group using Java$^\flat$.

More specifically, when the interpreter implemented in the **JavabCompiler.cs** reads "[", it will create a child for the current gameobject (called root), and set this child as a new root. And every time when **JavabCompiler.cs** reads "]", it will set the current root's father as the new root. In this way, every gameobject added between "[" and "]" will belongs to the child level of the root. Figure 9 shows an example of Java$^\flat$programming using transformation and transform group code blocks. In this example, the user uses Java$^\flat$ script to generate a scene with a vase (a prefab in Unity), a cube, a sphere, and a cylinder. Two orthogonal cloth boards (flat cubes) are loaded as the background of the scene.

Addressing Methods	Examples
Direct access of constant value	E.g. "5" means: 5
Direct access of identifier table	E.g. ”%5” means: IdentifierTable[5]
Direct access of integer table	E.g. "@5" means: IntegerTable[5]
Direct access of double table	E.g. "#5" means: DoubleTable[5]
Direct access of string table	E.g. "$5" means: StringTable[5]
Direct access to the vector table	E.g. "\5" means: VectorTable[5]
Initialize vector from double table	E.g. "~5" means: new Vector(#5, #6, #7)
Indirect access of the integer table	E.g. "&5" means: IntegerTable[IdentifierTable[5]]
Indirect access of the double table	E.g. "^5" means: DoubleTable[IdentifierTable[5]]
Indirect access of the string table	E.g. "*5" means: StringTable[IdentifierTable[5]]
Indirect access of the vector table	E.g. "<5" means: VectorTable[IdentifierTable[5]]
Secondary indirect access of integer table	E.g. "&5" means: IntegerTable[IntegerTable[IntegerTable[5]]
Secondary indirect access of floating point table	E.g. "!5" means: DoubleTable[IntegerTable[IntegerTable[5]]
Secondary indirect access of string table	E.g. "?5" means: StringTable[IntegerTable[IntegerTable[5]]
Secondary indirect memory access of vector table	E.g. ">5" means: VectorTable[IntegerTable[IntegerTable[5]]
Secondary indirect memory access of vector table	E.g. ">5" means: VectorTable[IntTable[IntTable[5]]

Table 1: Addressing methods in ASM$^\flat$ instructions.

2 ASM$^\flat$ LANGUAGE INSTRUCTION SET

In order to implement a high-efficiency virtual machine, the traditional assembly instructions are difficult to meet the requirements. Therefore, an instruction set specially designed for the Java$^\flat$ Virtual Machine (JVM$^\flat$), namely, the ASM$^\flat$ instruction set, is used here. The style of the ASM$^\flat$ instruction set is inspired by the standard ARM instruction set which is a variable-length instruction set. After the instruction is read from the class file, it is loaded into the virtual machine, then a table structure (similar to the interrupt vector table) is created. Then, each instruction is loaded to JVM$^\flat$ and executes different operations according to different instructions' identifiers.

2.1 Addressing Methods

In the ASM$^\flat$ instruction set, different memory addressing methods are used to manipulate different data, thereby achieving different execution results. There are several different addressing methods as show in in Table 1. Based on different addressing methods, the virtual machine can flexibly access and change the data after loading the data segment from the class, so as to properly execute the instructions, such as the meaning of this instruction **MUL \4,<5,*>6** is to perform the dot product operation of vectors and its mathematical calculation is:

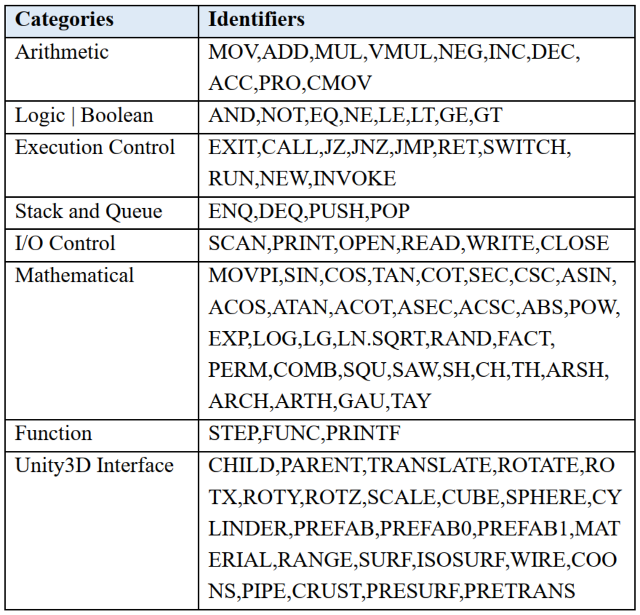
$$VectorTable[4] = VectorTable[IdentifierTable[5]] \\ \cdot VectorTable[IntegerTable[IntegerTable[6]]],$$

where the $\cdot$ operator represents the dot product operation between vectors. In addition, when accessing memory, data type conversion can be performed more flexibly between different data types. For example, the specific meaning of the instruction **MOV &4,?5** is:

$$IntegerTable[IntegerTable[IntegerTable[4]]] = \\ (\text{int})(StringTable[IntegerTable[IntegerTable[5]]])$$

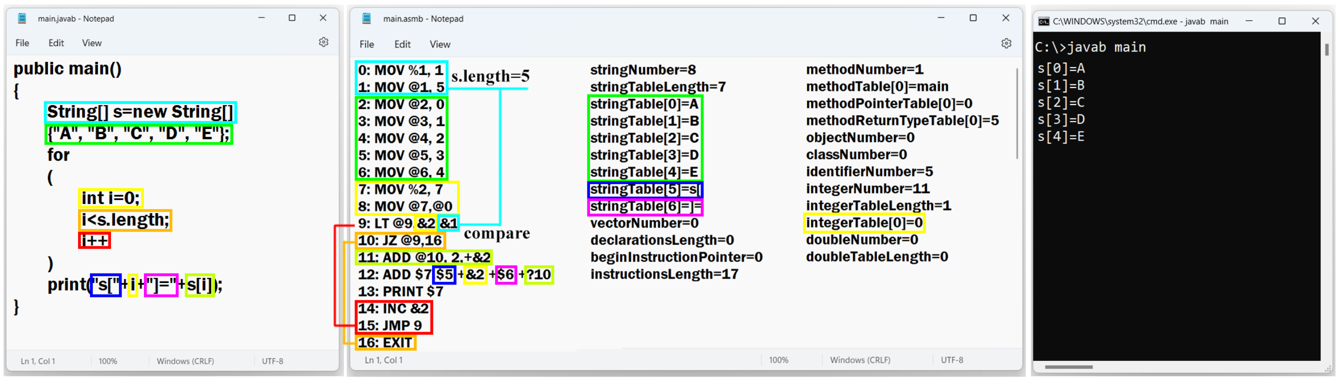
2.2 Instruction Categories

In general, there are eight different functional modules in the JVM$^\flat$. Each functional module corresponds to a group of ASM$^\flat$ instructions. Each instruction has different execution aproach. There are 108 kinds of instructions that the JVM$^\flat$ can execute. These functional modules can be divided into the following eight categories: arithmetic calculation, logic or Boolean operations, execution control, stack and queue operations, input and output control, mathematical calculation, parametric function, and Unity PM4VR Interactions. Figure 10 shows the ASM$^\flat$ instructions and their identifiers.



Categories	Identifiers
Arithmetic	MOV,ADD,MUL,VMUL,NEG,INC,DEC, ACC,PRO,CMOV
Logic | Boolean	AND,NOT,EQ,NE,LE,LT,GE,GT
Execution Control	EXIT,CALL,JZ,JNZ,JMP,RET,SWITCH, RUN,NEW,INVOKE
Stack and Queue	ENQ,DEQ,PUSH,POP
I/O Control	SCAN,PRINT,OPEN,READ,WRITE,CLOSE
Mathematical	MOVPI,SIN,COS,TAN,COT,SEC,CSC,ASIN, ACOS,ATAN,ACOT,ASEC,ACSC,ABS,POW, EXP,LOG,LG,LN.SQRT,RAND,FACT, PERM,COMB,SQU,SAW,SH,CH,TH,ARSH, ARCH,ARTH,GAU,TAY
Function	STEP,FUNC,PRINTF
Unity3D Interface	CHILD,PARENT,TRANSLATE,ROTATE,RO TX,ROTY,ROTZ,SCALE,CUBE,SPHERE,CY LINDER,PREFAB,PREFAB0,PREFAB1,MAT ERIAL,RANGE,SURF,ISOSURF,WIRE,COO NS,PIPE,CRUST,PRESURF,PRETRANS

Figure 10: ASM$^\flat$ instructions and identifiers.



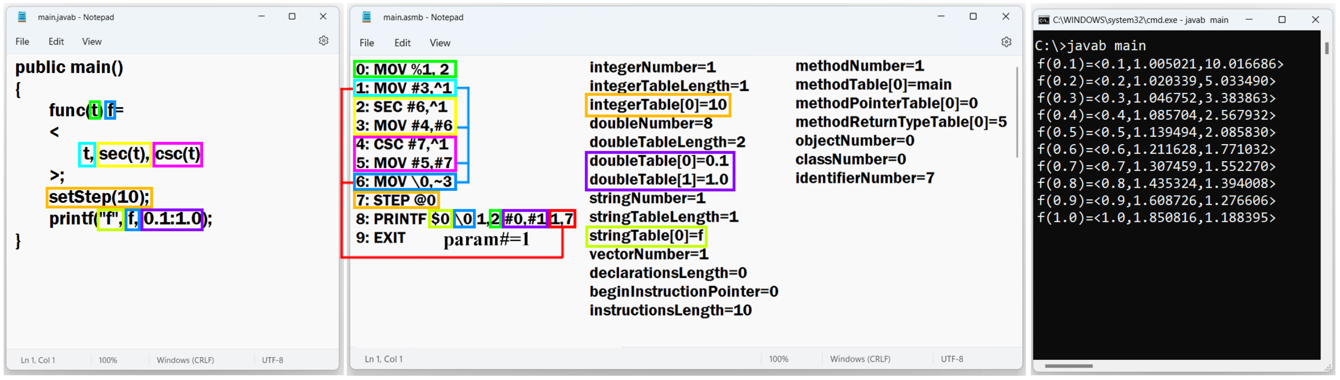
```
public main()
{
    String[] s=new String[]
    {"A", "B", "C", "D", "E"};
    for
    (
        int i=0;
        i<s.length;
        i++
    )
    print("s["+i+"]="+s[i]);
}
```

```
0: MOV %1, 1          stringNumber=8                methodNumber=1
1: MOV @1, 5          stringTableLength=7           methodTable[0]=main
2: MOV @2, 0          stringTable[0]=A              methodPointerTable[0]=0
3: MOV @3, 1          stringTable[1]=B              methodReturnTypeTable[0]=5
4: MOV @4, 2          stringTable[2]=C              objectNumber=0
5: MOV @5, 3          stringTable[3]=D              classNumber=0
6: MOV @6, 4          stringTable[4]=E              identifierNumber=5
7: MOV %2, 7          stringTable[5]=s[             integerNumber=11
8: MOV @7,@0          stringTable[6]=]=             integerTableLength=1
9: LT @9 &2 &1        vectorNumber=0                integerTable[0]=0
10: JZ @9,16          declarationsLength=0          doubleNumber=0
11: ADD @10, 2,+&2    beginInstructionPointer=0     doubleTableLength=0
12: ADD $7 $5 +&2 +$6 +?10   instructionsLength=17
13: PRINT $7
14: INC &2
15: JMP 9
16: EXIT
```

```
C:\>javab main
s[0]=A
s[1]=B
s[2]=C
s[3]=D
s[4]=E
```

(a) Java^{b} Script (b) ASM^{b} program (c) Output

Figure 11: ASM^{b} program sample of array declaration.



```
public main()
{
    func(t) f=
    <
        t, sec(t), csc(t)
    >;
    setStep(10);
    printf("f", f, 0.1:1.0);
}
```

```
0: MOV %1, 2          integerNumber=1               methodNumber=1
1: MOV #3,^1          integerTableLength=1          methodTable[0]=main
2: SEC #6,^1          integerTable[0]=10            methodPointerTable[0]=0
3: MOV #4,#6          doubleNumber=8                methodReturnTypeTable[0]=5
4: CSC #7,^1          doubleTableLength=2           objectNumber=0
5: MOV #5,#7          doubleTable[0]=0.1            classNumber=0
6: MOV \0,~3          doubleTable[1]=1.0            identifierNumber=7
7: STEP @0            stringNumber=1
8: PRINTF $0 \0 1,2 #0,#1 1,7   stringTableLength=1
9: EXIT               stringTable[0]=f
                      vectorNumber=1
                      declarationsLength=0
                      beginInstructionPointer=0
                      instructionsLength=10
```

```
C:\>javab main
f(0.1)=<0.1,1.005021,10.016686>
f(0.2)=<0.2,1.020339,5.033490>
f(0.3)=<0.3,1.046752,3.383863>
f(0.4)=<0.4,1.085704,2.567932>
f(0.5)=<0.5,1.139494,2.085830>
f(0.6)=<0.6,1.211628,1.771032>
f(0.7)=<0.7,1.307459,1.552270>
f(0.8)=<0.8,1.435324,1.394008>
f(0.9)=<0.9,1.608726,1.276606>
f(1.0)=<1.0,1.850816,1.188395>
```

(a) Java^{b} Script (b) ASM^{b} program (c) Output

Figure 12: ASM^{b} program sample of function declaration.

2.3 ASM^{b} Program Samples

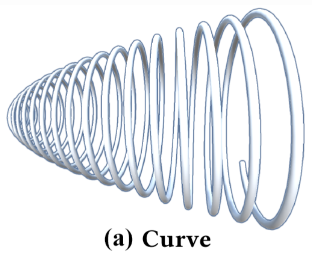
Figure 11 shows the ASM^{b} program sample of array declaration. In this case, a string array **s** is declared with initializations (**s**'s address is specified in IdentifierTable[1]=1) where IntegerTable[1]=5 is the **s.length** and IntegerTable[2..6]=[0..4] are the indices of each string element in **s**. When it executes **int i=0;**, the value of **i** is stored in IntegerTable[7] and it is initialized as 0 (**i**'s address is specified in IdentifierTable[2]=7). When it executes **i<s.length;** which is corresponding to **LT @9,&2,&1**, it will compare **i**'s value stored in IntegerTable[7] with **s.length**'s value stored in IntegerTable[1]. if **i** is less than (LT) **s.length**, IntegerTable[9]=1; Otherwise, IntegerTable[9]=0; Then, **JZ @9,16** instruction pointer will jump to line 16 (which is **EXIT**) if IntegerTable[9] is 0; Otherwise, it will execute **INC &2** to increase **i**'s value by 1, execute **JMP 9**, and jump to line 9 so as to simulate the behaviour of for loop.

Figure 12 shows the ASM^{b} program sample of the function declaration. In this case, a function **f**(t) is declared where (**t**'s address is specified in IdentifierTable[1]=2). According to **MOV \0,~3** the func(t) **f** returns to a vector initialized from DoubleTable[3][4][5] and store this vector into the VectorTable[0]. After executing **STEP @0** the value in IntegerTable[0]=10 is loaded in JVM^{b} as the step value. Once the ASM^{b} instruction of **PRINTF $0,\0,1,2,#0,#1,1,7** is executed, the JVM^{b} will do several steps: getting the string "f" in StringTable[0] as the function name, repeating update the parameter **t**'s value in DoubleTable[2] by uniformly sampling 10 times between the range specified in DoubleTable[0][1] (which are 0.1 and 1.0), executing the instructions between line 1 and line 7 (s.t. DoubleTable[3] = DoubleTable[2] = **t**, DoubleTable[4] = DoubleTable[6] = sec(**t**), and DoubleTable[5] = DoubleTable[7] = csc(**t**)), updating and printing VectorTable[0] accordingly.

3 UNITY^{b} AND PM4VR: MORE DETAILS

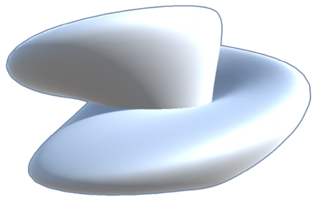
Figure 4 in the main paper shows an example of PM4VR's execution process of adding a wooden sphere with a radius specified by a range control named "s" whose current value is 0.5. As highlighted in yellow rectangles, when a user calls **setMaterial** ("Material Filename"), the user's predefined material in Unity's "Assets/Resources/Materials" folder will be automatically assigned to the sphere game object through the **JavabCompiler.cs**. When **getRange** ("Range Control Name") is called (as highlighted in cyan box), **JavabScriptBehavior.cs** will write its range controls' names and values into the Unity^{b} script before the JVM^{b} starts. Then, when JVM^{b} starts, it will load the range controls' names and values from the Unity^{b} script. This process is applied every few seconds according to the specification put in the "Interval" text box on Unity Editor. When the user calls **scale**(size) (as highlighted in red box), **JavabCompiler.cs** will setup the scale of the child until the child is created from the "[" symbol, this scale is applied onto that child. Figure 13 shows the Java^{b} scripts and their corresponding execution results for the examples included in the Main Paper (Section 3.5).

```
public main()
{
    func(t) f=[w=100, R=getRange("R"), r=getRange("r")]
    <2.0*r*t, R*sin(t*w), R*cos(t*w)>*t, g=0.02*t;
    setStep(<1000,20,0>); addPipe(f, 0:1.25, g, 0:1);
}
```



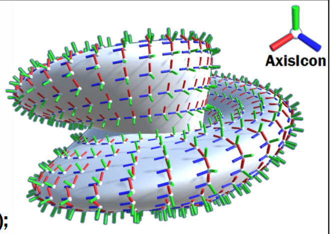
(a) Curve

```
public main()
{
    func(u,v) f=
    [
        R=getRange("R")*2, r=getRange("r")*2,
        m=cos(v/2)*sin(u)-sin(v/2)*sin(2.0*u),
        n=sin(v/2)*sin(u)+cos(v/2)*sin(2.0*u)
    ]
    <(R+m)*cos(v),r*n,(R+m)*sin(v)>;
    addSurface(f,0:PI,0:4.0PI);
}
```



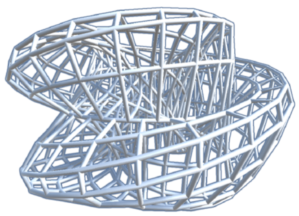
(b) Surface

```
public main()
{
    func(u,v) f=
    [
        R=getRange("R")*2, r=getRange("r")*2,
        m=cos(v/2)*sin(u)-sin(v/2)*sin(2.0*u),
        n=sin(v/2)*sin(u)+cos(v/2)*sin(2.0*u)
    ]
    <(R+m)*cos(v),r*n,(R+m)*sin(v)>;
    addSurface(f, 0:PI, 0:4.0PI); setStep(<20, 80, 0>);
    addPrefabSurface("AxisIcon",0.05,f,0:PI,0:4.0PI);
}
```



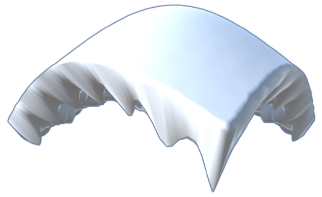
(c) Prefab Surface

```
public main()
{
    func(u,v) f=
    [
        R=2.0,r=1.5,
        m=cos(v/2)*sin(u)-sin(v/2)*sin(2.0*u),
        n=sin(v/2)*sin(u)+cos(v/2)*sin(2.0*u)
    ]
    <(R+m)*cos(v),r*n,(R+m)*sin(v)>/10;
    setStep(<10,50,20>); setMaterial("Red");
    addWireSurface(f,0:PI,0:4.0PI,0.0035);
}
```



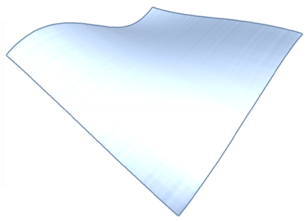
(d) Wire Surface

```
public main()
{
    func(x, y) f=[u=0, d=0.4, w=50.0]
    0.7*gauss(x, u, d)*gauss(y, u, d),
    g=0.05*sin(w*x)*sin(w*y),
    q=0.1; double r=0.4; setStep(50);
    addCrust(f,-r:r,-r:r, g,-r:r, -r:r, q,-r:r, -r:r);
}
```



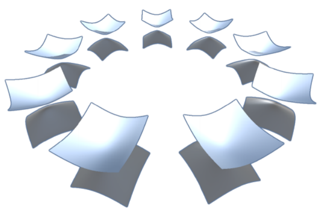
(e) Crust Surface

```
public main()
{
    double A=0.1,r=PI/8,wU=1.0PI,wV=2.0PI;
    func(u) pU0=<-r,A*sin(wU*u),u>,pU1=<r,-A*sin(wU*u),u>;
    func(v) p0V=<v,A*sin(wV*v),-r>,p1V=<v,-A*sin(wV*v),r>;
    addCoonsSurface(pU0,-r:r,pU1,-r:r,p0V,-r:r,p1V,-r:r);
}
```



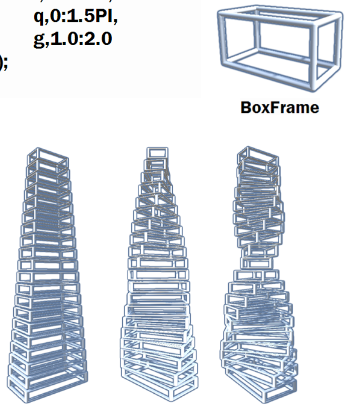
(f) Coons Surface

```
void isoSurface()
{
    func(x,y,z) f=[a=0.3,b=0.5,c=0.4]x*x/(a*a)-y*y/(b*b)-z*z/(c*c);
    setStep(30); double r=1.5; addIsoSurface(f==1.0,-r:r,-r:r,-r:r);
}
public main()
{
    newPrefab("P"){isoSurface();}
    int length=10; scale(0.3);
    double a=2.0PI/(length-1);
    double R=getRange("R")*4,r=getRange("r")/2;
    [
        for(int i=0;i<length;i++)
        {
            rotate(<0,1,0>,-a*i);
            translate(<R*cos(a*i),0,R*sin(a*i)>);
            [
                rotZ(0.5PI); [addPrefab("P", r);]
            ]
        }
    ]
}
```



(g) Isosurface

```
public main()
{
    func(t) f=
    [
        r=getRange("r"),
        k=getRange("k")
    ]
    <0, -r*2*t, 0>,
    q=<0, t*k, 0>,
    g=<t, r, t>;
    setStep(20);
    addPrefabTransform
    (
        "BoxFrame",
        f,0:4.0PI,
        q,0:1.5PI,
        g,1.0:2.0
    );
}
```



(h) Tilt Prefab

Figure 13: Java$^{\flat}$ script for the examples in the Main Paper (Section 3.5).