University of Nevada

Reno

Large Scale Software Transitions:
A Case Study of the Second Half of MFIRE

A professional paper submitted in partial fulfillment of the
requirements for the degree of Master of Science
with a major in Computer Science

by

Lu-Chun Liao

Dr. Frederick C. Harris, Jr., advisor

May 2000

Abstract

This paper presents a case study of large scale software transitions. This is the second part of our virtual reality of mine ventilation systems project. This project is based on an existing software package called MFIRE. MFIRE is a computer simulation program that models a mine's ventilation system and its response to altered ventilation parameters, external influences such as temperatures, and internal influences such as fires. The program was written in FORTRAN 77. With the fast development of computer techniques, we can virtually display the result of MFIRE as a three-dimensional model.

The goal of the whole project is to use the output of MFIRE, and display the mine and its ventilation three-dimensionally on a computer screen. The purpose of this paper is to convert the FORTRAN code, Mfire2.for, to C++. This paper describes the conversion of and the debugging of the code.

Acknowledgements

I would like to express deepest and sincerest gratitude to my advisor Dr. Fred Harris, to take the time and effort to make suggestions and off opinions. His excellent class, CS 308 Data Structure in C++, make me really understand and become familiar with the C++ programming language. His interesting class, Advanced Computer Graphics, sparked my interest in Computer Graphics and Virtual Reality.

I also like to thank Dr. George Bebis and Dr. Bruce Johnson for being on my committee and their valuable time, and Dr. LingJiang Cheng for discussions and comments on various portions of this paper.

Contents

List of Figures

List of Tables

1. Introduction

MFIRE is a computer simulation program that models a mine's ventilation system and its response to altered ventilation parameters, external influences such as temperatures, and internal influences such as fires. The program was written in Fortran 77, and has three parts in it: Mfire0.for, Mfire1.for and Mfire2.for.

One of the goals of our research group is to use the output of MFIRE and display the mine and its ventilation three-dimensionally on a computer screen. This will allow us to provide a more realistic experience and provide us tools to train miners about the effects of the proposed alteration to the ventilation system of the mine. First we need to rewrite MFIRE, which is written in FORTRAN, to C++. Then we can modify it to make it more of a simulator. Once this is completed we want to design a Virtual Reality front end to the simulation and design training scenarios. This paper is the second part of the whole project, and its purpose is to convert the FORTRAN code, Mfire2.for, to C++. The code conversion is presented in Section 3, and debugging is covered in Section 4. The conclusion and future work are discussed in Section 5.

2. Code Conversion

The second half of the MFIRE code conversion is to rewrite the Mfire2.for in C++. In the following sections we will first present the main structure of the Mfire2.for, and then discuss the differences between FORTRAN and C++ in each kind of statement or structure, and finally describe how we did the conversion.

For the simplification of the code conversion, we need to know the main structure of the entire program. The main structure of Mfire2.for is given in Figure 1. There are three main loops, one is for loop and there other two are do while loop, in the main function.

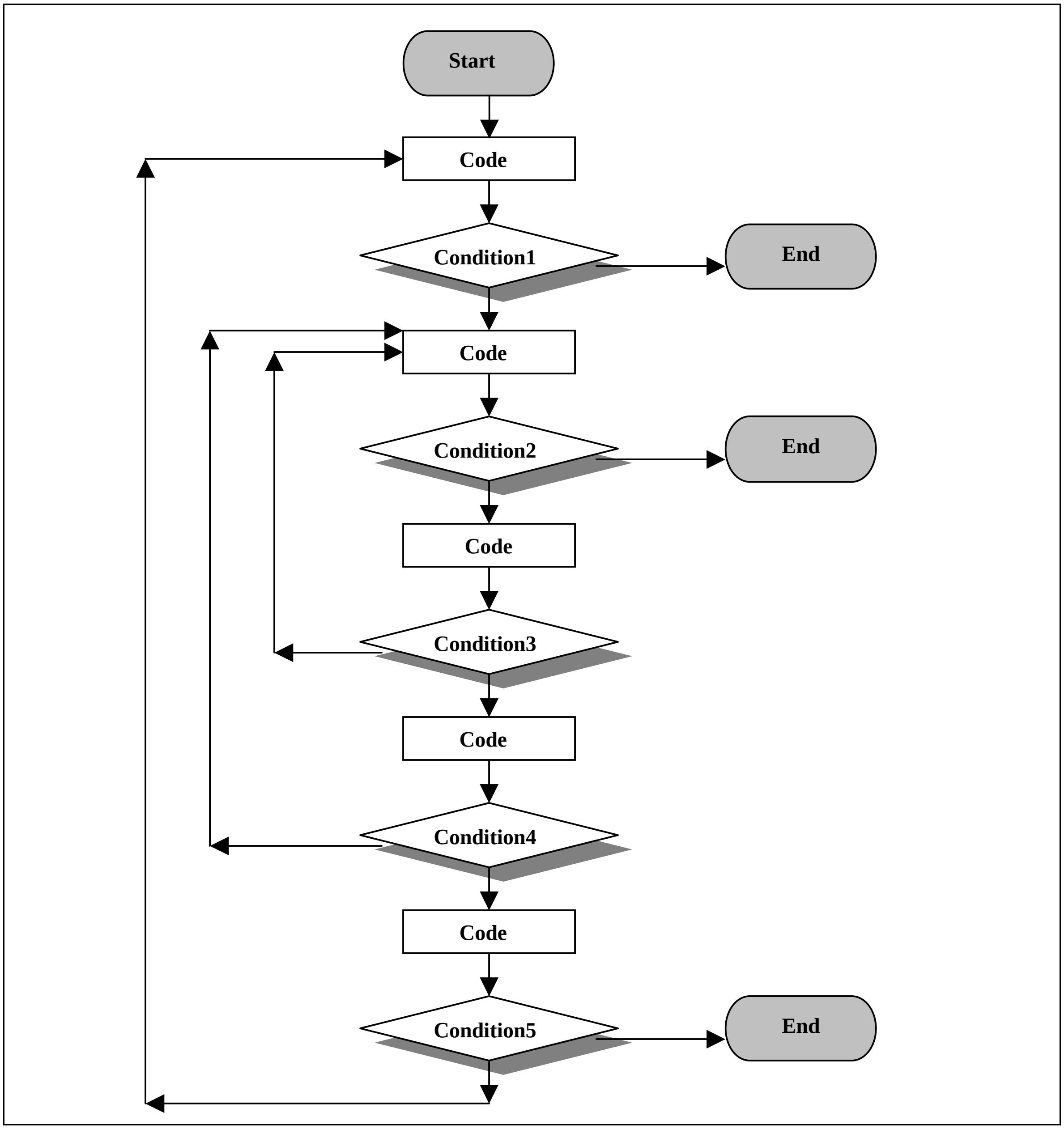

Figure 1: The main Structure of the Mfire2.for

2.1 Data Types

FORTRAN allows you to use several different types of values: INTEGERs, REAL values, COMPLEX values, CHARACTER values, etc. A memory location can contain only one type of value. The type of values stored in a variable is specified by two methods: implicit typing or explicit typing. With implicit typing, the first letter of a variable name determines the value type that can be stored in it. Variable names beginning with the letters I, J, K, L, M, or N are used to store integers. Variable names beginning with one of other letters are used to store real values. With explicit typing, FORTRAN data types (INTEGER, REAL, CHARACTER etc.) are used to specify the variable types.

In this code translation, we convert INTEGER in FORTRAN to be the data type int in C++, and REAL in FORTRAN to be double in C++.

2.2 Arrays

In FORTRAN, the DIMENSION *array(S)* statement is used to reserve memory space or storage for an array with size *S*. The type of values stored in an array can be specified implicitly through the choice of array name (arrays name begin with I-N are the integer array and the others store the real values) or explicitly with a type specification statement (INTEGER *arrays(S)*, REAL *array(S)*). The array subscript begins with the integer one. Using the statement DIMENSION *array(B::E)* means the range of subscripts associated with an array is from the integer *B* to the integer *E*.

In C++, the array is a collection of variables, all of the same type. The range of subscripts of an array is from zero to the number one less than the size of the array.

In Mfire2.for, there are only arrays with subscripts beginning at one. This made it easier to do the conversion by defining the array size in C++ one larger than the array size in FORTRAN, and not using the position with subscript zero. That means the code that uses the array will have the same subscripts after the conversion.

2.3 Arithmetic Operators

In FORTRAN and in C++, we use the arithmetic operators (=, +, -, *, and /) to do the assignment, addition, subtraction, multiplication and division. But we use the library function pow(*double, double*) in C++ instead of the exponentiation operator (**) in FORTRAN. The precedence order of arithmetic operators is the same in C++ and FORTRAN.

2.4 Logical Expressions

Logical expressions can be formed using relation operators both in FORTRAN and in C++. The relation operators in both languages are show in Table 1.

FORTRAN use two logical constants: .TRUE. and .FALSE. while in C++ you typically use the integer value one for the true and integer value zero for false.

FORTRAN	C++
.EQ.	= =
.NE.	!=
.LT.	<
.LE.	<=
.GT.	>
.GE.	>=
.OR.	||
.AND.	&&
.NOT.	!

Table 1: The relational operators in FORTRAN and C++

2.5 Control Structures

2.5.1 IF Structure

Several kinds of IF structures are used in FORTRAN code. The basic IF structures are given in Figure 2. There are many segment code made up with IF statement in Mfire2.for. We used the statements if(*condition*)…., if(*condition*)….else…., and if (*condition1*)….else if (*condition2*)….in C++ to convert the IF structure. Figure 3 is a code segment from Mfire2.for. Figure 4 shows the rewritten C++ code.

```
IF (logical expression) THEN
        Statements ….
ENDIF

IF (logical expression) THEN
        Statements ….
ELSE
        Statements ….
ENDIF

IF (logical expression) THEN
        Statements ….
ELSE IF (logical expression) THEN
        Statements ….
ENDIF
```

Figure 2: The IF Structure used in FORTRAN

```
IF (MARKS.GE.1) THEN
      CALL ADPT (IS,IQ,MARKS,I)
ELSE
      TMST=TACC+DELTAT
      CALL KALPHA (TMST,TM,FX,CP,I)
      COAGE=BI(I)-FX*BI(I)**2/(0.375+BI(I))
      DCOAGE(I)=COAGE
      XNEW(I)=HK(I)*O(I)**2.*COAGE/(120.*DR*QQ(I)*A(I))
      CALL ADPT (IS,IQ,MARKS,I)
ENDIF
```

Figure 3: The IF…ELSE statement to be converted (FORTRAN)

```
if (marks >= 1) {
    adpt(is,iq,marks,i);
 }
else {
    tmst = tacc+deltat;
    kalpha(tmst,tm,fx,cp_cdjn,i);
    coage = bi[i] - fx*pow(bi[i],2)/(0.375+bi[i]);
    dcoage[i] = coage;
    xnew[i] = hk[i]*pow(o[i],2)*coage/(120*dr*qq[i]*a[i]);
    adpt(is,iq,marks,i);
}
```

Figure 4: The converted IF…ELSE code (C++)

2.5.2 DO loop

The structure of the DO loop in FORTRAN in shown in Figure 5. The for (*index = initial, index<=limit, increment*) loop in C++ is correspondence to the DO loop in the FORTRAN. Figure 6 shows a code segment in Mfire2.for using the DO loop. The corresponding C++ code, which is using the for loop, is given in Figure 7.

```
DO k index = initial, limit, increment
      . . . .
      . . . .
k continue
```

Figure 5: The DO loop structure in FORTRAN

```
       DO 3204 K=1,JAJ
          READ (20) JSTAR(K), TSTAR(K)
3204   CONTINUE
```

Figure 6: The DO loop to be converted (FORTRAN)

```
for(k=1; k<=jaj; k++) {
          fin20>>jstar[k]>>tstar[k];
}
```

Figure 7: The for loop (C++) corresponding to the DO loop in Figure 6

2.6 The GOTO Statement

2.6.1 The GOTO Statement as a DO WHILE loop

If the GOTO statement is in a loop and causes the flow of program control to transfer to statement n, which is before this loop, we need to break the loop and use the DO WHILE loop to convert that. The statements that were structured in the flow chart shown in Figure 8.

If the GOTO statement said that go to the line n before it, we can use a DO WHILE loop in C++ to rewrite it. If there are several GOTO statements that transfer control to line n, which is before them, we still can use a DO WHILE loop and one indicator to execute the same thing. In this conversion, we make the indicator to be an integer variable named *flag*, which initialized to zero and change it as the GOTO statement appears. A flow chart is presented in Figure 9 which shows this basic structure. Figure 10 shows a code segment for Mfire2.for. The corresponding C++ code is shown in Figure 11.

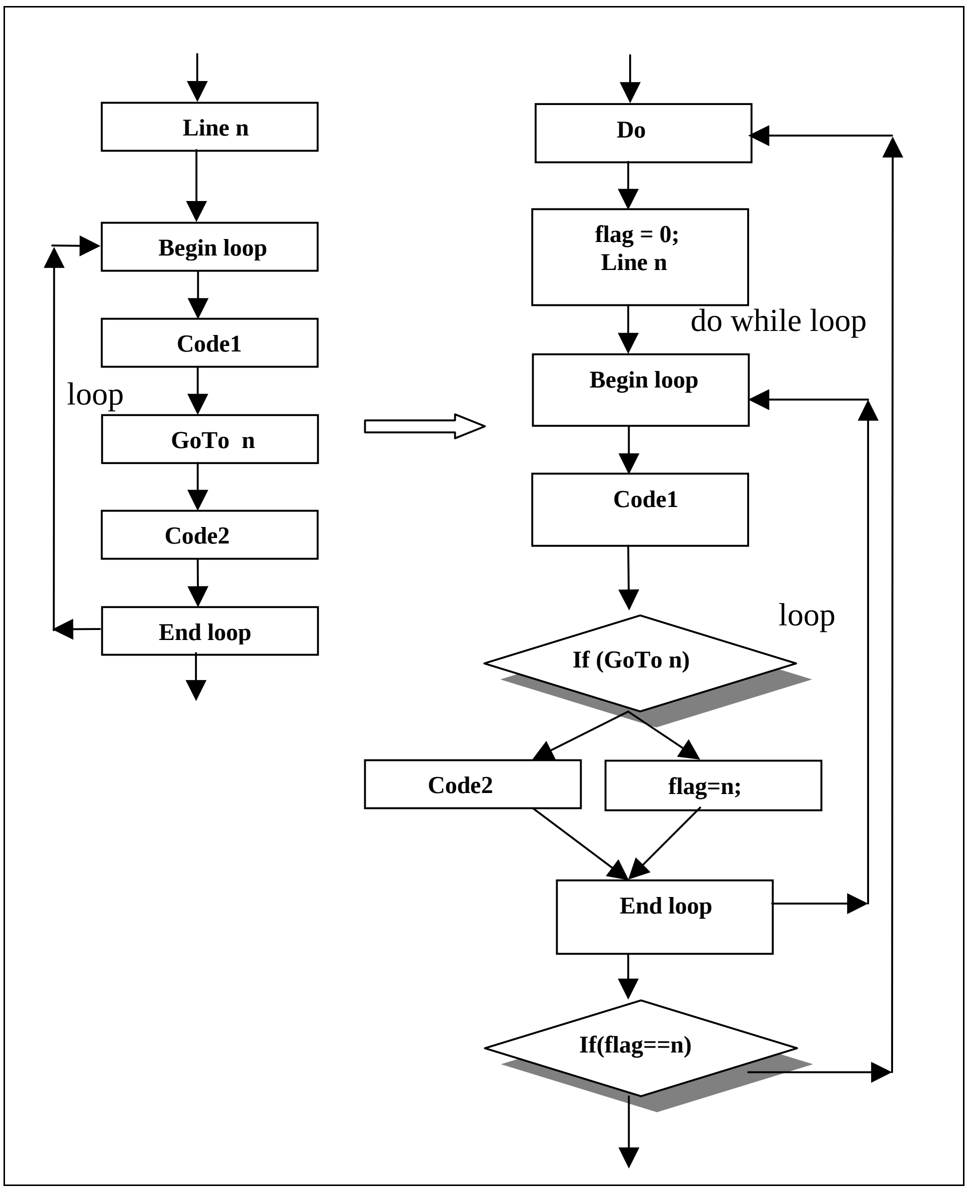

Figure 8: The GOTO statement and the correspondent DO WHILE loop

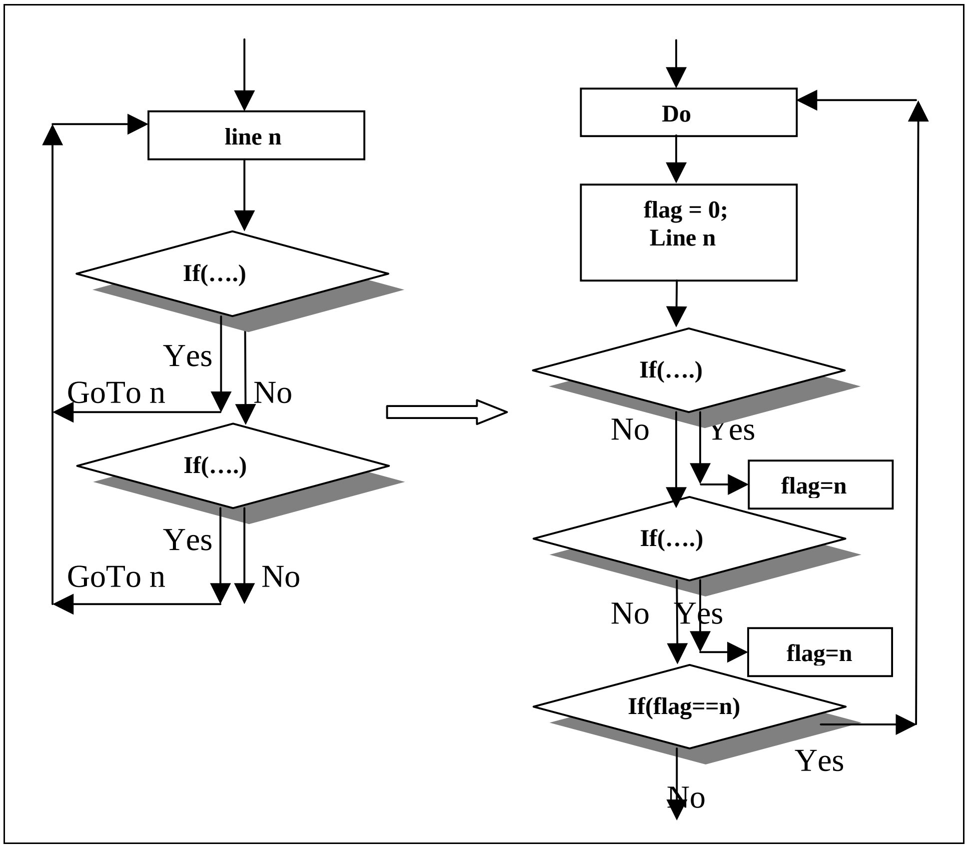

Figure 9: Several GOTO statements as the DO WHILE loop

```
4        CALL READIN (DAL,14,ISTOP,0)
         IF (ISTOP.EQ.1) THEN
              ICODE=DAL(2)
              WRITE (8,300)
              IF (ICODE.EQ.1) WRITE (8,310)
              IF (ICODE.EQ.2) WRITE (8,320)
              IF (ICODE.EQ.3) WRITE (8,330)
              IF (ICODE.EQ.4) WRITE (8,340)
              IF (ICODE.EQ.5) WRITE (8,350)
              IF (ICODE.EQ.6) WRITE (8,360)
              IF (ICODE.EQ.7) WRITE (8,365)
              WRITE (8,370) (ROW(IE),IE=1,80)
              NSTOP=5
             RETURN
        ENDIF
        IF (DAL(15).LT.(-1.E20)) GO TO 4
        IF (NCOMTS.GT.NCOMT2) THEN
             NCOMT2=NCOMTS
            GO TO 4
        ENDIF
        IREAD=0
```

Figure 10: The GOTO statement that will be rewritten as a DO WHILE loop

```
do {
        flag = 0;
        readin(dal,14,istop,0); //4

        if(istop == 1) {
           icode = dal[2];
           fout8<<endl<<" THE FOLLOWING VARIABLES "<<
                          "WERE EXPECTED TO BE READ"
                   <<" IN:"<<endl<<endl<<endl;
           if(icode == 1) {
                   fout8<<" TS,1,NBRR,R"<<endl;
           }
           if(icode == 2) {
                   fout8<<" TS,2,NBRR,INDEX,"
                       <<"QF,PF,QF,PF,QF,PF,QF,PF,QF,PF"<<endl;
           }
           if(icode == 3) {
                   fout8<<" TS,3,NBRR,CONT,CONC,HEAT,02MIN,"
                       <<"SMPO2,HTPO2,QCENT"<<endl;
           }
           if(icode == 4) {
                   fout8<<" TS,4,NBRR"<<endl;
           }
           if(icode == 5) {
                   fout8<<" TS,5,DELTAT"<<endl;
           }
           if(icode == 6) {
                   fout8<<" TS,6,TOUT"<<endl;
           }
           if(icode == 7) {
                   fout8<<" TS,7,JDPP"<<endl;
           }
           fout8<<endl<<"THE CURRENT INPUT "
                <<"WAS:"<<endl<<endl;
           for(ie=1; ie<=80; ie++)
           fout8<<row[ie]<<' ';
           fout8<<endl;
           nstop = 5;
           return 0;
        }
        if(dal[15] < (-1e20)) {
           flag = 4;
        }
        else {
           if(ncomts>ncomt2) {
              ncomt2 = ncomts;
              flag = 4;
           }
        }
} while (flag == 4);

iread = 0;
```

Figure 11: The DO WHILE loop (C++) from the GOTO statement in Figure 10

2.6.2 Using IF Statement in C++ to convert GOTO Statement

If the GOTO statement is in a loop and causes the flow of program control to transfer to statement n which is after this loop. What we can do is break the loop and transfer the flow of control to the statement n. The flow chart is shown in Figure 12.

If several GOTO statements go to the same line n, which is after them, we can use several IF statement in C++ to convert this case. The flow chart for this case is shown in Figure 13. Figure 14 shows some GOTO statements in Mfire2.for that will be rewritten with several IF statement in C++. Figure 15 show the corresponding C++ code.

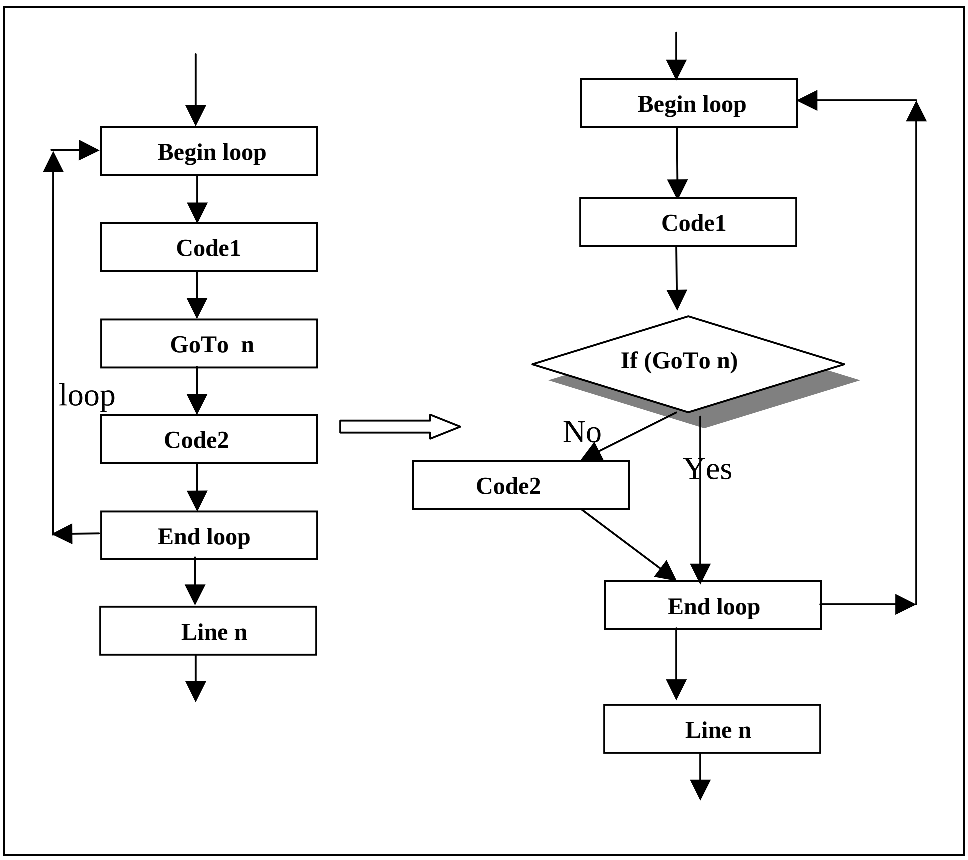

Figure 12: Using an IF statement to rewrite a GOTO statement

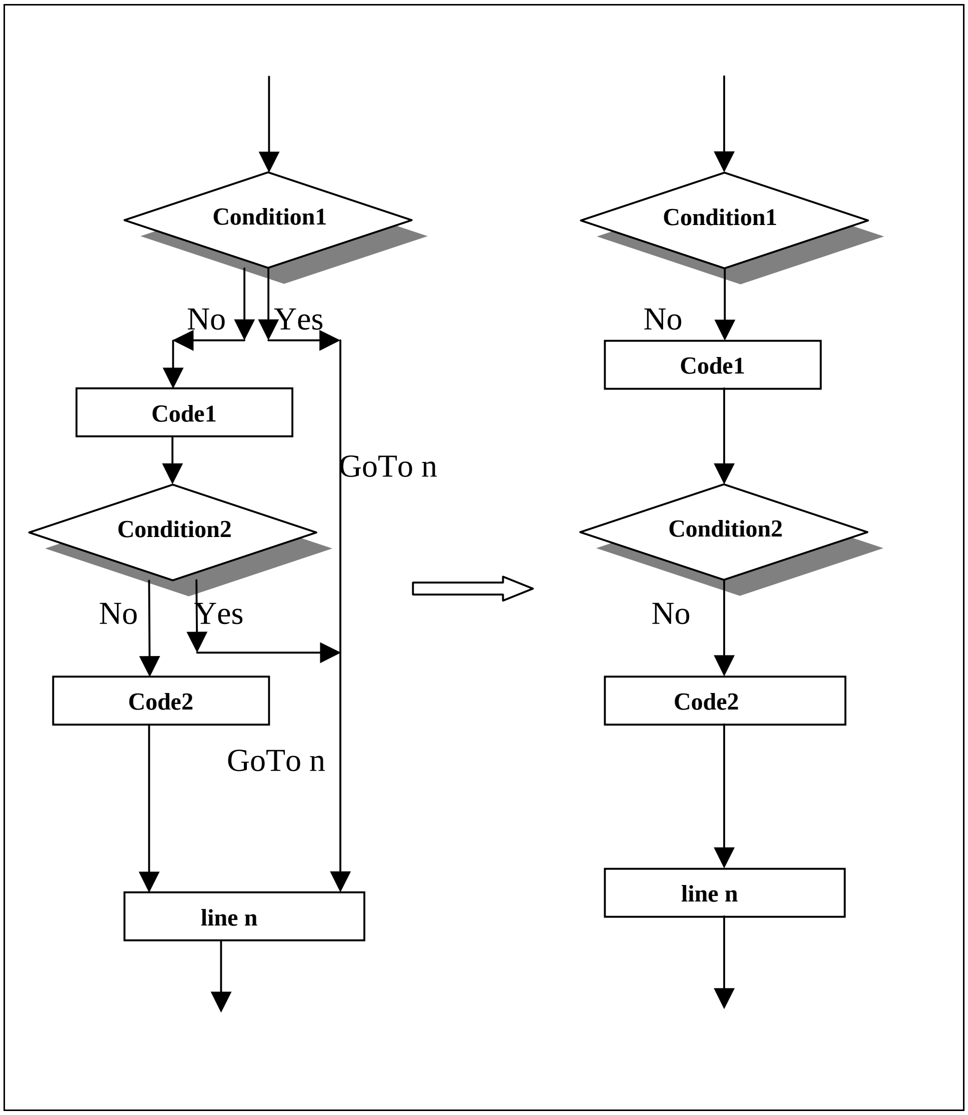

Figure 13: The flow chart for a GOTO statement and an IF statement

```
      IF (IS.LE.0) GO TO 50
      IF (DPPA(IQ,IS+1,1).GT.LA(I)) GO TO 50

      SEGZ=QQ(I)*DELTAT*(TMRD(I)+460.0)/(60.0*A(I)*(TR+460.0))
      FA=XNEW(I)*ABS(SEGZ)/CP
      FAS=0.0
      IF (FA.LT.25.0) FAS=EXP(-FA)

      DO 40 K=1,IS
        JA=IS-K+1
        SEG=DPPA(IQ,JA+1,1)-DPPB(IQ,JA+1,1)
        IF (ABS(SEG).LT.1.E-3) SEG=0.001
        VL=SEG/LA(I)
        T0=TAUXB(IQ,JA)
        T2=DPPB(IQ,JA,2)
        IRV=0
        IF ((TAUXB(IQ,JA).LT.DPPB(IQ,JA,2).AND.TAUXB(IQ,JA).GT.
     .    TROCK(I)).OR.(TAUXB(IQ,JA).GT.DPPB(IQ,JA,2).AND.
     .    TAUXB(IQ,JA).LT.TROCK(I))) THEN
          T0=DPPB(IQ,JA,2)
          T2=TAUXB(IQ,JA)
          IRV=1
        ENDIF
        T1=TROCK(I)+(T0-TROCK(I))*FAS-(DZRD(I)*VL/(2*778.26*CP))
     .    *(1.0+FAS)
        TAUXA(IQ,JA)=T1
        CALL COLUMN (T1,SEG1,I,IQ,JA)
        SEG=DPPA(IQ,JA+1,1)+SEG1-DPPB(IQ,JA,1)
        VL=SEG/LA(I)
        DPPA(IQ,JA,2)=TROCK(I)+(T2-TROCK(I))*FAS-
     .            (DZRD(I)*VL/(2*778.26*CP))*(1.0+FAS)
        IF (JA.EQ.1) TAUXC(IQ)=TROCK(I)+(TAUXD(IQ)-TROCK(I))*FAS-
     .            (DZRD(I)*VL/(2*778.26*CP))*(1.0+FAS)
        IF (IRV.EQ.1) THEN
          TBW=DPPA(IQ,JA,2)
          DPPA(IQ,JA,2)=TAUXA(IQ,JA)
          TAUXA(IQ,JA)=TBW
        ENDIF
        DPPA(IQ,JA,1)=DPPB(IQ,JA,1)+SEG
        SE=ABS(DPPA(IQ,JA,1)-DPPB(IQ,JA,1))
        DPPA(IQ,JA,4)=(DPPB(IQ,JA,4)*QQ(I)+CH4V(I)*SE/LA(I))/
     .            (QQ(I)+CH4V(I)*SE/LA(I))

        IF (DPPA(IQ,JA,1).GE.LA(I)) GO TO 50
40    CONTINUE

      VL=(DPPA(IQ,1,1)-DPPB(IQ,1,1))/LA(I)
      TAUXC(IQ)=TROCK(I)+(TAUXD(IQ)-TROCK(I))*FAS
     .                (DZRD(I)*VL/(2.0*778.26*CP))*(1.0+FAS)

50    ISS=NSAC(I,2)
```

Figure 14: Using an IF statement in C++ to rewrite a GOTO statement

```
if(is <= 0)  flag = 50;
else {
        if(dppa[iq][is+1][1] > la[i])  flag = 50;
        else {
           segz = qq[i] * deltat * (tmrd[i]+460.0)/(60.0*a[i]*(tr+460.0));
           fa = xnew[i] * fabs(segz) / cp_adpt;
            fas = 0.0;
           if(fa < 25.0)  fas = exp(-fa);
           for(k=1; k<=is; k++) { //do 40
                   ja = is - k + 1;
                   seg_adpt = dppa[iq][ja+1][1] - dppb[iq][ja+1][1];
                   if(fabs(seg_adpt) < 1e-3)  seg_adpt = 0.001;
                   vl = seg_adpt / la[i];
                    t0 = tauxb[iq][ja];
                   t2 = dppb[iq][ja][2];
                    irv = 0;
                   if((tauxb[iq][ja]<dppb[iq][ja][2] && tauxb[iq][ja]>trock[i]) ||
                      (tauxb[iq][ja]>dppb[iq][ja][2] && tauxb[iq][ja]<trock[i]){
                         t0 = dppb[iq][ja][2];
                          t2 = tauxb[iq][ja];
                         irv = 1; }
                   t1 = trock[i] + (t0-trock[i])*fas
                         - (dzrd[i]*vl/(2*778.26*cp_adpt))*(1.0+fas);
                   tauxa[iq][ja] = t1;
                   column (t1,seg1,i,iq,ja);
                   seg_adpt = dppa[iq][ja+1][1] + seg1 - dppb[iq][ja][1];
                   vl = seg_adpt / la[i];
                   dppa[iq][ja][2] = trock[i] + (t2-trock[i])*fas
                                           - (dzrd[i]*vl/(2*778.26*cp_adpt))*(1.0+fas);
                   if(ja == 1)  tauxc[iq] = trock[i] + (tauxd[iq]-trock[i])*fas
                                          - (dzrd[i]*vl/(2*778.26*cp_adpt))*(1.0+fas);
                   if(irv == 1) {
                         tbw = dppa[iq][ja][2];
                         dppa[iq][ja][2] = tauxa[iq][ja];
                         tauxa[iq][ja] = tbw; }
                   dppa[iq][ja][1] = dppb[iq][ja][1] + seg_adpt + 0.1;
                   se = fabs(dppa[iq][ja][1] - dppb[iq][ja][1]);
                   dppa[iq][ja][4] = (dppb[iq][ja][4]*qq[i]+ch4v[i]*se/la[i])
                                           / (qq[i]+ch4v[i]*se/la[i]);
                   if(dppa[iq][ja][1] >= la[i]) {
                         break;
                         flag = 50;
                    }
           } //40
           if(flag != 50) {
                   vl = (dppa[iq][1][1]-dppb[iq][1][1]) / la[i];
                   tauxc[iq] = trock[i] + (tauxd[iq]-trock[i])*fas
                                 - (dzrd[i]*vl/(2.0*778.26*cp_adpt))*(1.0+fas);
           }
     }
}
iss = nsac[i][2]; //50
```

Figure 15: Using IF statements (C++) to convert GOTO statements

2.7 Intrinsic Functions

There are some intrinsic functions in FORTRAN, and they all have corresponding library functions in C++. The relationship between these intrinsic FORTRAN functions and the C++ library function, and the data type of their return values is presented in Table 2[2][3].

FORTRAN	Function Type	C++
SQRT(x)	Real	sqrt(x)
ABS(x)	Real	fabs(x)
IABS(x)	Integer	abs(x)
LOG(x)	Same as x	log(x)
ALOG(x)	Real	log(x)
LOG10(x)	Same as x	log10(x)
EXP(x)	Real	exp(x)

Table 2: Intrinsic functions as C++ library functions

2.8 User-Defined Subroutines

There are 23 subroutines in Mfire2.for. They are called by the main program or other subroutines. The relationship between these subroutines are shown in Table 3 on page 16. The subroutines with no return statement in it are converted to the C++ functions which return void, and the others are converted to the C++ functions which return integer.

The data in common DATA BLOCK are converted to the global variables and initialized in a header file.

The main calls	Called by subs
INIT	
CDCH	- READIN
DTTR	
BASE	
MSLIST	
CDJN	- COLUMN
ADPT	- COLUMN
KALPHA	
PARAM	
NVP2	
ITR1	- SPLINE - LSFAN - ARR - BASE - MSLIST - MBLNC - NVP1 - NVP2
ITR	- SPLINE - LSFAN
EVHT	- DTTR - BASE - MSLIST
OUTPUT	
DISP	- BASE - MSLIST
FWCTA	

Table 3: The relationship between all the subroutines and the main function

2.9 I/O Statements

FORTRAN use the OPEN statement to open a file, READ statement to input the data from the file, WRITE statement to output to a file or a terminal screen, and CLOSE statement to close a file. The OPEN statement must precede any READ or WRITE statements that use the file. There are two types of statements that allow us to perform I/O operations in FORTRAN. The list-directed input/output statements are easy to use but give us little control over the input and output form. The formatted input/output statements allow us to control the input and output forms with greater detail. The FORMAT statement specifies the spacing to be used in input or output information.

Through the header file iostream.h, C++ use the iostream instance cin to do the standard input, and cout to do the standard output. To do the file input and output, we use two data types, ifstream and ofstream, which are defined in the C++ standard library through the header file fstream.h. The C++ standard library also supplies many manipulators to do the specific formats. In our conversion, we only used some of these manipulators: endl, setw, setprecision, showpoint etc.

3. Debugging the Code

Through the compilation process, our syntax errors in the C++ code were pointed out and fixed. We ran the C++ code after compilation, but the result from it was not the same as that from the FORTRAN code. It took more time to fix these logic errors than to do the code conversion.

We used an additional function void *prnt( )* for C++ code and SUBROUTINE PRNT for FORTRAN code to help locating errors. This function output all the global arrays, and with it we were able to find the rest of the errors. The output formats of this C++ function were very close to the output formats of FORTRAN SUBROUTINE. And when this function is executed, the program exits. A description of *prnt()*'s usage is given later in this section.

There are three parts in the MFIRE code: Mfire0, Mfire1, and Mfire2. To execute this project, we should run the file Mfire0 first, then Mfire1, then Mfire2, and then Mfire1 again. So the input data for Mfire2 are the output from Mfire1. This output was save in the file MFCTL3. At first, we thought that maybe we could run the FORTRAN code Mfire0.for and Mfire1.for to get the correct inputs for the C++ code for Mfire2. This would make it easier for us to do the debugging. But while trying this, we realized that this cannot be done. Because the C++ code cannot read the output data in file MFCTL3 from the FORTRAN Mfire1 because it was output as a binary file. We were only able to use the file MFCTL3 that came from the C++ Mfire1 to be the input file for the C++ version of Mfire2.

After the C++ Mfire0 and Mfire1 ran correctly[1], we began our work on Mfire2. The function *prnt( )* was insert right after the code read in all data from the file MFCTL3 and initialized the variables, then we were able to check if all the inputs were correctly given to the right variables. If correct, we then moved the function *prnt( )* to right before the first function call (the function *init( )*). If the output values of the arrays of C++ code match those of the FORTRAN code, the C++ code before the insertion point is correct. If not, there is something wrong with the code up to this part. Move the function *prnt( )* back and check every part to locate the errors and fixed them. If the code before the insertion point is correct, move the function *prnt( )* right after this function call. If the output from the function *prnt( )* doesn't match the output from the SUBROUTINE PRNT. That means something must be wrong in this function. We can move the function *prnt( )* in this function to locate the errors and fix them.

For the loop error checking, if the function *prnt( )* give the correct output before the loop but incorrect after the loop, we know that there must be something wrong in the loop. To locate this kind of errors, we defined an integer variable to be the loop counter. So we can use this counter and one IF statement to let the function *prnt( )* check all the arrays every time we run the loop.

We used the function *prnt( )* until all the errors were located and fixed. We then applied all the test data files to our code and used the same technique to make sure the C++ code worked correctly for all of them. At this point the output of our C++ code is exactly the same as the original FORTRAN code.

4. Conclusions and Future Work

4.1 Conclusions

MFIRE is a large and important piece of software to calculate mine ventilation. The output of it is in form of tables of numbers. With that we can only read the data and analyze them. There are current computer techniques that allow us to do Virtual Reality displays of data. This kind of simulator will let us visualize the real situation and analyze that, but the difficult connection between FORTRAN and C++ was stopping us from doing this. This is why we needed to do the code conversion first. My role in this project was to do the conversion and make sure the performance of the FORTRAN code and of the C++ code are exactly the same. Then, it will be ready for the design of the Virtual Reality simulator.

4.2 Future Work

This paper is the second part of our big project of creating a virtual reality mine ventilation system. Running our C++ code, we can get the values of arrays at each stage of calculation. Those will be the input for our VR program, which will contain two parts: the static mine system and the dynamic ventilation system. The static part would not change once the program begins, while the dynamic part changes according to the results of the ventilation calculation. The change of the concentration of different type of gases can be indicated with different colors. The critical situation can then be display on the screen.

References

[1] Cheng, Lingjiang, “Large Scale Software Transitions: A Case Study Of The First Half Of MFIRE”, Master paper, Department of Computer Science, University Of Nevada, Reno, NV, May 2000

[2] Dale, Nell, Weems, Chip, Headington, Mark, “Programming and Problen Solving with C++”

[3] Etter, D.M., 1993, “Structured FORTRAN 77 For Engineers And Scientists Fourth Edition”

[4] U.S. Bureau of Mines, 1995, “U.S. Bureau of Mines training workshop on the “MFIRE” mine fire and ventilation simulator, MFIRE users manual version 2.20,” August 1995 U. S. Bureau of Mines, Twin Cities Research Center, Minneapolis, MN