THINKING MODULE

This module handles but one task: the artificial intelligence for the Russian player. It has one entry point at $4700 and one exit point at $4C22. It includes several subroutines and data tables for its own use. Thus, this is the most direct and straightforward routine of the entire program. Unfortunately, it is also the most involved routine of the program. It is also the biggest, including about 1.5K of code. To make matters worse, it is almost devoid of comments. This module was one of the best-planned modules in the entire program. For this reason I felt little need to comment on it as I was writing the code. That just makes the task more difficult now.

The basic goal of this routine is to plan the moves of the units. This translates into the specific task of producing values of WHORDS and HMORDS for each Russian unit. Many factors must be considered in computing the orders for each unit. The routine must determine the overall strategic situation as well as the local situation that the unit finds itself in. This will tell whether the unit should think in terms of attack or defense. The overall situation is determined by computing the danger vector. The danger vector tells how much danger is coming from each of the four directions.

The unit must evaluate the four possible directions it can move in. Each direction must be evaluated in terms of the danger vector, the nature of the terrain, the impact of the move on the integrity of the Russian line, the possibility of traffic jams, and the presence of German units. All of the surrounding squares must be evaluated and the best one chosen.

The really difficult aspect of the decision-making process is the necessity of coordinating the moves of all Russian units. The problem is made vastly more difficult by the fact that we must coordinate each unit's possible move with the possible moves of all the other units. The possibilities multiply in a truly mind-boggling manner. My solution was rather esoteric. Imagine the Russian army lying in its positions at the beginning of a turn. Imagine now a ghost army of virtual Russian units, initially springing from the real army, but with each ghost army plotting a path of its own across the map. Each ghost plans its path based on the assumption that the other ghost armies represent the concrete reality that must be conformed to. Thus, each ghost in turn says, "Well, if you guys are gonna move there, I'm gonna move here." One at a time, the ghost army adjusts itself into new positions. This process can continue until each ghost can say, "If you guys are gonna be there, I'm gonna stay right where I am." In practice this situation is almost achieved after only about ten iterations. However, if the player presses the START button, the iterations stop and the ghost army becomes the destinations for the real army. In this way hypothesis is converted into plans.

OVERALL FORCE RATIO

The module begins at line 1680. The first task is to calculate the overall force ratio. This is the ratio of total German strength to total Russian strength, and is a useful indicator of the overall strategic situation. To calculate this number, we must first add up the total German strength and the total Russian strength. This calculation is made in lines

1730-1870. The upper byte of the total strengths is stored in TOTGS (total German strength) and TOTRS (total Russian strength).

The next problem is to calculate the ratio of these two numbers. This is a simple long division. Unfortunately, I was not prepared to do a long division. Such arithmetic takes many machine cycles to crunch and many bytes of code to do properly. The floating point arithmetic package provided in the Operating System ROM did not interest me. So I wrote my own special routine to handle the problem. This is an example of individual crotchetiness, not judicious planning. I probably should have used the floating point package, or at least a decent 16-bit integer arithmetic package, but I was too lazy and impatient.

The first problem I must solve arises from the high probability that the total German strength is going to be very close to the total Russian strength. If I take a straight ratio of the two I will very probably get a result of 1. Since I will have integer arithmetic, my result won't be very sensitive to changes in the total strengths. I solved this problem by arbitrarily multiplying the ratio by 16. It's my program and I can cheat on the arithmetic if I want to.

Unfortunately, multiplying by 16 creates a new problem. Should I multiply the quotient by 16 or divide the divisor by 16? Either approach will have the same effect, and both approaches have the simplicity of being executed with simple logical shifts. But dividing by 16 loses some precision in the quotient, and multiplying by 16 runs the risk of losing the whole number. For example, what if total German strength is 17 and I multiply by 16 by ASLing four times? I don't get 272 for an answer, I get 16. Check it out for yourself.

Here's the clunky solution I came up with: ASL the dividend (line 1950) until a bit falls off the high end of the byte into the Carry bit (line 1960). Put it back where it belongs (line 1970) and then LSR the divisor (line 1980) the remaining number of shifts.

Now I am prepared to do a dumb long division (lines 2070-2140). Load the dividend into the accumulator. Keep subtracting the divisor from it until it is all gone. The number of times you subtract the divisor is the quotient. It's dumb, it's slow, but it works. More important, I can understand it. The final result is stored in OFR, the overall force ratio.

INDIVIDUAL FORCE RATIOS

The next task is to calculate the individual force ratios. The war might be going really well for Mother Russia, but the 44th Infantry Army may not find conditions as rosy if it is surrounded, out of supply, and being attacked by four Panzer Corps. It is necessary to supplement global planning with a local assessment of the situation. This is expressed in the individual force ratios. There are five individual force ratios: Four express the amount of German danger bearing down on a Russian army from the four cardinal directions. The fifth expresses the average of these four.

The fifth is called the individual force ratio (IFR). The other four are called the IFRN, IFRS, IFRE, and IFRW, for the directions they represent.

SUBROUTINE CALIFR

Subroutine CALIFR (lines 8390-9690) calculates the individual force ratios. This is an extensive computation which requires a great deal of time and memory. The fundamental idea behind this subroutine is that danger is a vector, having both a magnitude and a direction. This subroutine determines aggregate magnitude and the aggregate sum of the danger to the unit.

The subroutine begins by zeroing the local variables IFR0, IFR1, IFR2, IFR3, and IFRHI. These correspond to the IFRN, IFRE, IFRS, IFRW, and IFR tables, but are easier to use in the routine. After initializing some coordinate variables, the first large loop begins.

This loop, beginning with line 8520, extends all the way to line 9230. Its purpose is to calculate the directional IFRs, so it is really the meat of the subroutine. It sweeps through each unit, first checking if the unit is on the map (lines 8520-8540). If so, it determines the separation between the tested unit and the unit whose IFR is being computed. It measures this in terms of both the total distance between the two (ignoring Pythagoras) and the X-separation (TEMPX) and the Y-separation (TEMPY). Units further than eight squares away are considered to be too far to be of any local consequence (lines 8680-8690). The range to closer units is halved and stored in TEMPR.

The unit's combat strength determines the magnitude of the unit's threat. We must also calculate the direction to the unit. This is done in lines 8750-9020. These lines test the direction vectors to determine the overall direction to the unit. The result of these tests is a value in X of 0, 1, 2, or 3. This value specifies the direction of the threat.

In lines 9030-9150 we determine the magnitude of the threat. We get the combat strength of the tested unit, divide by 16, and check to see if the tested unit is Russian or German. If Russian, the result is added to the running sum of local Russian strength (RFR). If German, it is added to the running sum of local German strength in the direction specified in the X register. This done, program flow loops back to the next unit in sequence.

The next chunk of code, lines 9250-9320, add up all the danger values from all four directions and leave the result in the accumulator.

The next chunk of code, lines 9350-9570, calculates the final individual force ratio in much the same manner that the overall force ratio was calculated. The dividend is multiplied by 16 (lines 9350-9420), and then the divisor is subtracted from the dividend repeatedly until the dividend is all gone (lines 9450-9510). The count of the number of subtractions equals the quotient. This quotient is averaged with the overall force ratio (lines 9540-9560) and the result is stored in the IFR for the unit. The only remaining function is to move the local directional IFRs to the unit-specific

IFR tables (lines 9610-9680).

Subroutine INVERT is a simple absolute value routine. It takes a value in the accumulator and returns the absolute value of the number in the accumulator. You may have noticed that it was used heavily in the code. By JSRing to INVERT+2, we get the negative value of the accumulator returned.

Back in the main part of the module, we complete the IFR loop by setting the army's current position (CORPSX, CORPSY) to the objective position (OBJX, OBJY). OBJX and OBJY are the coordinates of our ghost armies. This completes the initialization loop. We now enter the main loop of the program.

MAIN LOOP STRUCTURE

The main program loop begins on line 2340 and extends all the way to line 7290. It is obviously a gigantic loop, and it takes a long time to execute. It is also an indefinitely terminated loop. It does not terminate after a specific number of passes. It keeps looping until the player presses the START key. The main loop sweeps over the entire Russian army. The inner loop sweeps over each unit in the Russian army.

The first task of the loop is to ignore militia armies and armies that are not on the map. Militia are not allowed to move. If an army does not fail these two tests in lines 2360-2420, then the local military situation for the army is evaluated. This is done by comparing the army's individual force ratio with the overall force ratio. If IFR=OFR/2, then the army must be more than eight squares from the nearest German unit. This conclusion can be made from the way that CALIFR calculates the IFR. If the army is far from the front, then it is treated as a reinforcement. If not, it is treated as a front-line unit, and a different strategy is used.

REINFORCEMENT STRATEGY

The job of a reinforcement is to plug weak spots in the line. This requires that the unit be able to figure out where the line is weak, no easy task. The trick is to use the existing Russian front-line units as gauges for the seriousness of the situation at any segment of the front. Where the front is solid, the IFRs of the front-line units will be low. Where the front is weak, their IFRs will be large. So we need merely examine the IFRs of all Russian units, select the largest, and head in that army's direction. Well, not quite. We don't want all the reinforcements heading for the same spot or the beleaguered Russian army will find himself trampled by his rescuers. More important, we need to take into account the distance between unit in distress and rescuer. There is no point in rushing to save somebody several thousand miles away.

The code to do all this extends from line 2470 to line 2870. The section starts by initializing BVAL to the value of OFR/2. BVAL stands for "best value" and is used to store the value for the most beleaguered Russian

army. Then a loop begins at line 2520 which sweeps through all Russian armies, rejecting off-map armies and calculating the separation between the tested army and the reinforcing army. This separation is divided by 8 (lines 2660-2680). I cannot now figure out the purpose of the branch in line 2690. It throws out the tested army if the separation had bit D3 set. A very strange test indeed. Lines 2700-2760 subtract the separation from the tested unit's IFR and compare the result with the best previous result. If the new result is bigger, then this unit has a better combination of proximity and (get this) beleagueredness. This unit becomes the preferred unit. Its value is stored in BVAL and its ID number is stored in BONE (best one). Then we move on to test another unit. When all units have been tested the best one is selected for support. Its coordinates become the objective of the reinforcing army. The job of planning that army's move is done and the routine jumps to the end of the loop (TOGSCN).

STRATEGY FOR FRONT-LINE ARMIES

Front-line armies have a very complex strategy. They must evaluate a large number of factors to determine the best possible objective square. These factors are: the army's IFR, its supply situation, the accessibility of the square, the straightness of the line that would result, the vulnerability to being surrounded, the danger imposed by nearby Germans, the possibility of a traffic jam, the terrain in the square, and the distance to it. Let's take it slowly.

In lines 2990-3050 we perform a simple test to see if the unit should take emergency measures. We ask, is the army seriously outnumbered? Is it out of supply? If either answer is yes, then this army is probably trapped behind German lines and it must escape to the east. It is given an objective square directly east of its current position. It will frantically crash eastward, regardless of the circumstances. It will even attack vastly superior German units in its haste.

This may strike you as pretty stupid. I gave a good deal of thought to the problem and I am convinced that this is the best all-round solution. My first solution was much more intelligent: I had such Russian units run away from the Germans. This normally meant that they ran to the west, straight for Germany. This is not very realistic. It also forced the player to assign large numbers of troops the boring job of tracking down and finishing off the forlorn Russian armies. I considered having cut-off Russians sit down and stay put, but then they would never have any chance of escaping. Quite a few Russians do indeed escape with this system, so I think it has proven to be a successful way of dealing with a difficult problem.

NORMAL FRONT-LINE ARMIES

If an army is not in trouble then it must choose a direction in which to move. The computations for this choice begin in line 3130, with DRLOOP, the direction loop. The critical loop variable is DIR, the direction of movement being evaluated. For the purposes of this loop, DIR takes the following

meanings: 0=north, 1=east, 2=south, 3=west, FF=stay put. This loop answers the question, "Should this army move in direction DIR?" It first determines the square being moved into (lines 3160-3240). The coordinates of this target square are TARGX, TARGY. The square being left is a ghost army square at OBJX, OBJY. The value of this target square is SQVAL. After verifying that the square can be entered (lines 3290-3340), the primary logic begins.

LINE INTEGRITY COMPUTATIONS

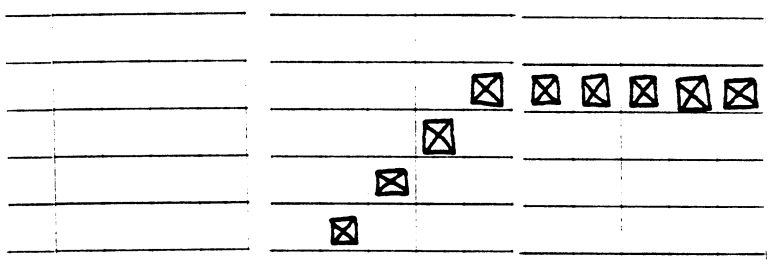
To figure whether a move will result in a solid line or a weak line, it is first necessary to give the computer some image of what that line looks like. I did this by creating two arrays. The first array is called the direct line array and is stored in LINARR. This array is 25 bytes long and covers a 5-by-5 square. The square being tested is always at the center of the big square. The routine will not evaluate the entire Russian line, for that task is impossibly large. Instead, it will treat it as a collection of short line segments and evaluate each segment for desirable configuration.

The big square is addressed by starting at the central square, whose coordinates are TARGX, TARGY, and stepping outward in a spiral from this square. The direction vectors for this spiral path are specified in a table called JSTP. The counter for the steps is called JCNT. The coordinate of a little square being considered within the big square is always SQX, SQY.

The contents of the big square are computed with two nested loops, LOOP56 and LOOP55 (lines 3450-3800). The outer loop steps through each of the 25 squares in the big square (except the central square, which we assume will contain the ghost army). The inner loops sweeps through all Russian armies to see if one's objective is in the square being tested. Note that we check not for the presence of the unit itself (CORPSX, CORPSY) but rather for the intention of the unit to go to the square (OBJX, OBJY). This is how we coordinate the plans of the different armies. If a match is obtained, the muster strength of that army is stored into the array element (lines 3760-3780). We then store the muster strength of the army whose plans are being made into the array element for the central square. When this task is completed we have an array, LINARR, which tells us how much Russian muster strength is in each of the 25 squares surrounding the square in question. We can now examine the structure of this configuration. We will examine it from four different directions: north, south, east, and west. We will keep track of which direction we are looking from with the variable SECDIR (secondary direction).

THE LINE VALUE ARRAY

A very useful tool for examining this two-dimensional array is to construct a one-dimensional representation of its most important feature. This one-dimensional representation will answer the question, "How far forward is the enemy in each column?" A picture might help:



LV ARRAY: 5, 5, 5, 5, 5 5, 4, 3, 2, 1 1, 1, 1, 1, 1

If a particular column is not populated at all, the value in the corresponding LV entry is five.

Lines 3920-4220 build the LV array from the LINARR. The variable POTATO (remember I told you I sometimes used funny variable names?) counts which column we are in. The Y-register holds the row within the column, and the X-register holds the LINARR index. The loop searches each column looking for the first populated square. When it finds one, the row index of the square is stored in the LV arrray. If it finds no populated square in the column, it assigns a value of 5 to the corresponding LV element. The sequence of CPX, BNE, LDX instructions in lines 4060-4220 translate the current row count in X into an index for LINARR and resume the loop. This is the clumsiest kind of code. It is special purpose code, code that is executed only once per condition. During program execution, much of the code is effectively useless, testing for conditions that do not exist. A more elegant solution is called for here. I was too lazy to be elegant; I just slopped the code together.

EVALUATING THE STRENGTH OF THE LINE (LPTS)

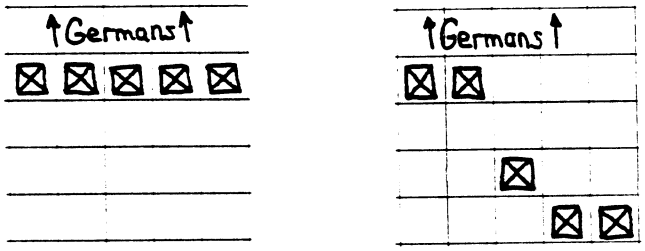
Now that the analytical tools we need are in place, we are ready to begin analysis of the position. We shall analyze the strength of a given line configuration by assigning points to it. We will assign various points for the various features we look for in a good line. These points will be stored in a variable called LPTS. Initially, we shall set this variable to zero and during the course of the evaluation we shall add to it or subtract from it.

The calculation begins on line 4240. We first evaluate the configuration for its completeness. Is there a unit in every single column in the array? For each populated column, we add $28 to LPTS (now in the accumulator). This is done in lines 4240-4320.

We then test if the contemplated presence of our army would fill an otherwise empty column. The test for this is simple and inelegant (lines 4360-4460). An easier way to have done this would have been LDA LV+2/CMP #$02/BNE Y95. It seems so simple and obvious now. In any event, if the condition is satisfied, we add $30 to LPTS.

We don't want to create a traffic jam, so we must evaluate the degree of blocking in this array. This is done by testing the frontmost unit in each column and looking behind it; if somebody is in that square the retreat route of the front unit and the attack route of the rear unit are both blocked. This is undesirable. Subtract $20 points for each such case (lines 4500-4730).

Our next concern is with penetrations. We do not want to create a line configuration which is easily flanked. A picture illustrates the problem.



GOOD BAD

The right arrangement is bad because it allows the enemy to easily penetrate to the rear of the most forward units before engaging the line. This places these forward units in jeopardy. We want a tight, parallel line as in the example on the left. It took me a lot of thinking to translate this concept into terms that the machine could execute. The final result was surprisingly easy to program. It is not so easy to explain. We have five columns in our square. We are going to take each column in turn, calling it OCOLUM, and compare its LV value with each of the other columns. While we are doing this test, we refer to the other column as COLUM. I know, the labelling seems backwards. Forgive me, I was feverish with effort. The comparison is made by subtracting the LV value of the one column from that of the other. If they are equal, there is no problem and we proceed to the next other column. If the latter column is more forward than the first, then we move to the next column; the discrepancy will be handled when the other column is directly tested. If the latter column is more rearward than the primary column, then a penalty must be extracted. The penalty I use is a power of two, one power for each row of discrepancy. The evaluation is done in lines 4880-4990.

We have now calculated the strength of the line and stored it in LPTS. However, the importance of this strength depends on the amount of danger coming from the direction in question. A line which is strong facing north will probably be weak to an attack from the west. We must therefore evaluate the strength of the line in light of the danger vector on the army. I do this by multiplying LPTS by the IFR value for the direction for which the line was evaluated. This multiplication is done in lines 5100-5370. The first 14 lines select the IFR to be used by some more inelegant code. The preparation for the multiplication is done in lines 5240-5280; the multiplication itself is done in lines 5290-5370. As with the long division,

this routine is a triumph of pedestrian programming. To multiply A by B, I add A to itself B times. It is a two-byte add, and only the upper byte (ACCHI) is important to me. I throw away the lower byte in the accumulator.

NEXT SECONDARY DIRECTION

I have now calculated the line configuration value of the square from one direction. I must now perform the same evaluation for each of the other directions. First I increment the secondary direction counter (SECDIR). Then I rotate the array. It is easier to rotate the array in place and evaluate it than to write code that can look from any direction. My code is customized to look at the 25-square array from the north. To look at it from other directions, I simply rotate it to those directions. This is done with an elegant piece of code (at last!) in lines 5480-5580. First I store the array LINARR into a temporary buffer array (BAKARR). Then I rotate it by a pointer array called ROTARR. This array holds numbers that tell where each array element goes when the array is rotated 90 degrees to the right. Thus, the zeroth element of ROTARR is a 4; that means that the zeroth element of LINARR should now be the fourth element. With the rotation done, the program flow loops back up to the beginning of this huge loop.

In developing this code I made heavy use of flowcharts. When I was satisfied with these I then wrote a small BASIC program that performed most of the manipulations in this chunk of code. It took only a few hours to write and test the BASIC code and verify that the fundamental algorithms would work as I had intended. Only then did I proceed to write the assembly code. This shows the value of BASIC: it is an excellent language for tossing ideas together and checking their function. I firmly believe that almost any assembly language project on a personal computer should have several BASIC tools developed just for supporting the effort. I wrote four different BASIC programs as part of the EASTERN FRONT development cycle. They are no longer useful, so I have discarded them.

EVALUATING IMMEDIATE COMBAT FACTORS

It is not good enough to analyze the danger in a square in terms of some obscure danger vector. It is also necessary to ask the simple question, how close is the nearest German unit? The proximity of a German unit will be of great significance to a Russian unit, although the precise significance will depend on whether the Russian is pursuing an offensive or a defensive strategy. In considering the direct combat significance of a square, we must also consider the defensive bonus provided by the terrain in the square.

These factors are considered in lines 5620-6310. After storing the modified line points value into SQVAL (square value), we determine the range to the nearest German unit. This is done with a straightforward loop that subtracts the coordinates of each German unit from the target square's coordinates, takes the absolute value, and adds the two results together. If the resulting range is less than the best previous value, it becomes the new best value (NBVAL).

This range to the closest German unit, when multiplied by the IFR, will give us the specific danger associated with the square. However, IFR is not a signed value; it is always positive. If the Russians are doing well, then IFR will be small but still positive. In such a case the value of IFR*NBVAL would be a measure of the opportunity presented to the Russian, not a measure of danger. Thus, small values of IFR demand that IFR*NBVAL be interpreted differently. The logic to do this is managed in lines 5930-6050. The IFR is subtracted from $F; if the result is greater than zero it is doubled and stored into TEMPR to act as a fake IFR; NBVAL is replaced by 9-NBVAL. The effect of these strange manipulations is to invert the meaning of the code about to be executed. This succeeding code was intended to determine the importance of running from a square. With the inversion, it will also determine the importance of attacking the same square.

The fooled code (lines 6090-6250) begins by checking the square to see if it is occupied by a German. If so, it immediately removes the square from consideration; we don't go around picking fights with Germans when we are the underdogs. Note that this will never happen when the Russians are using offensive strategy. If the square is unoccupied, we add the terrain bonus to NBVAL; this is a crude way of including terrain into the computation. I now think that this was not the correct way to handle terrain.

In lines 6200-6250 I execute one of my disgustingly familiar Neanderthal multiplications. I then add this value to SQVAL (lines 6270-6310).

TRAFFIC AND DISTANCE PENALTIES

The final tasks are to include penalties for traffic jams and long-distance marching. The former is necessary to make sure that Russians don't waste time crowding into the same square. The latter reflects the brutal reality that things sometimes do not go as expected, and so plans that call for armies to march long distances in the face of the enemy are seldom prudent.

The code for making these tests is simple (lines 6350-6870). The first test (lines 6350-6540) is a loop that tests all the other Russians, looking for one that has already chosen this square as an objective. If so, a penalty is extracted from SQVAL. The second test (lines 6580-6870) calculates the range from the army's current position to the target square. If it is greater than 6, the target is unreachable and the square is ruled out; SQVAL is set to zero. If not, 2 raised to the power of the range is subtracted from the SQVAL. With this work done, we have completed our calculation of the value of this square.

FINAL SQUARE EVALUATION

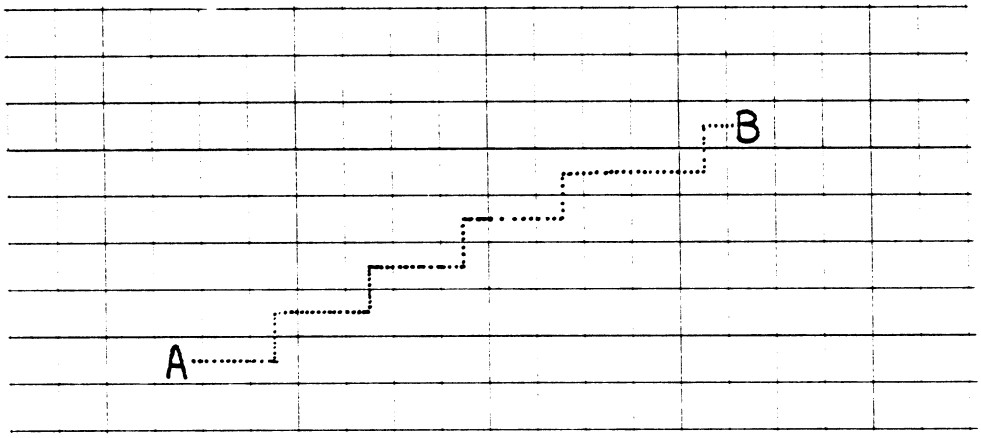
We now compare the value of this square with the best value we have obtained so far (lines 6910-6970). If our new value is better, it becomes the best. If not, we forget it. In either event, we go to the next square

and loop back to the far beginning (lines 6980-7020).

Upon completion of this gigantic process we have obtained a best square. In lines 7040-7150 we make this square our newest objective for this army. We then look at the START key to see if the human player has finished his move. If not, we continue our analysis, evaluating more and more squares without end. If so, we jump to a completely different section.

TRANSLATING TARGETS INTO ORDERS

If the human has pressed the START key, we must convert the targets figured by the previous routines into orders for execution in the mainline routine. This task in done in the remaining section of the module. The fundamental problem solved in the module is a very standard problem in computer graphics. It is depicted in the following diagram:



Starting at square A, what is the straightest path to square B? Specifically, what sequence of single steps will take you from A to B in the straightest possible line? For reasons of computational efficiency, we desire to find the answers without resorting to multiplications or divisions. The problem has been solved in its most general case, and the solution is so powerful that it is easily adapted to circles, ellipses, parabolas, and other curves. Unfortunately, I was unaware of this solution when I wrote these routines, so I had to make up my own, and thereby hangs a tale.

The obvious solution is to compute the slope of the line joining A and B, and then walk from A towards B, measuring the slope generated by each proposed step and comparing this resultant step with the desired slope; if the slope resulting from a proposed step is the closest that can be obtained, then that step is the best. Unfortunately, calculating a slope requires dividing a delta-y by a delta-x, and division is not allowed.

I found my solution in the calendraic system of the Mayan Indians. They never developed the concept of the fraction, and so they had a terrible time expressing the length of the year. Do you know how hard it is to measure the length of the year when you have no number for one-quarter day? They developed a novel solution: instead of declaring that one year is 365 and 1/4

days long, they declared that 4 years are 1461 days long. They refined the method to state that 25 years are 9131 days long. This procedure can be extended to arbitrary precision. Indeed, the Mayans did just that; their measurement of the length of the year was more accurate than the contemporary European value.

The basic idea of the technique is simple: your quantity is divided into a whole part and a fractional part. You can't get your hands on the fractional part, so you keep a running sum, adding both whole and fractional parts until the accumulated fractional parts add up to one; then the whole part will tick over an extra time. That's the event to watch for; it tells you what the fractional part is.

The first step in implementing this algorithm is to calculate some intermediate values. These are HDIR and HRNGE, the horizontal direction from A to B, and the horizontal range (delta-x). VDIR and VRNGE are the corresponding values for the vertical separation. These four values are calculated in lines 7370-7540.

Next we calculate the larger range LRNGE and the smaller range SRNGE, as well as the corresponding directions LDIR and SDIR (lines 7550-7690). Then we prepare some counting variables by setting them equal to zero: RCNT, the number of steps taken, and RORD1 and RORD2, the actual orders assigned. RANGE is the total distance from A to B in non-Pythagorean measure. CHRIS (I was getting desperate for variable names) is the rollover counter. I initialized myself to half of LRNGE.

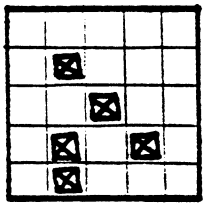
We now begin the walk from A to B. On each step we shall assume that we should take a step in the larger direction (LDIR). In so doing we add SRNGE to CHRIS; if CHRIS exceeds RANGE then we must instead take a small step in direction SDIR. The figuring for this is done in lines 7830-7940. The code runs fast. The orders that result from this are folded into the main orders in lines 8110-8140, another case of that weird code that first popped up in the interrupt module. If you didn't figure it out then, you might as well figure it out now.

A few more manipulations loop back to finish the walk to point B; then the army's orders are stored and the next army is given its orders until all armies have been taken care of. With that, the routine is complete and it returns to the mainline routine in line 8340. That was simple enough, wasn't it?

Point System for Artificial Intelligence

A. Line Points - LPTS

(Values for this example)



LINARR = 0,0,0,0,0,0,M1,0,M2,M3
0,0,M4,0,0,0,0,0,M5,0
0,0,0,0,0

LV = 5,1,2,3,5

+ 40 points for each occupied column — LPTS = 120
+ 48 points if central column is otherwise empty — LPTS = 168
- 32 points for each front unit whose retreat is blocked — LPTS = 168
- 2^{Δ} points for each column pair, where Δ is the difference in LV (iff $\Delta > 0$)

Δ values for this example:

Lead Column \ Lag Column	1	2	3	4	5
1	–	<	<	<	0
2	4	–	1	2	4
3	3	<	–	1	3
4	2	<	<	–	2
5	0	<	<	<	–

Hence total penalty in this example is:
$2^4 + 2^1 + 2^2 + 2^4 + 2^3 + 2^1 + 2^3 + 2^2 + 2^2 = 64$

LPTS = 104 final

B. Accumulated Points [ACCLO, ACCHI]

$$ACC = \sum_{SECDIR=0}^{3} LPTS_{SECDIR} * IFRX_{SECDIR}$$

C. Computation of Square Value [SQVAL]

Start with SQVAL = ACCHI
Determine NBVAL, range to nearest German

If IFR≥16 (defensive strategy):
    If NBVAL=0 (i.e., German in square), then SQVAL=0, exit?
    Add IFR* (NBVAL+defensive bonus) to SQVAL
If IFR<15 (offensive strategy):
    Add 2*(15-IFR)* (9-NBVAL+defensive bonus) to SQVAL

If somebody else has dibs on this square, SQVAL= SQVAL-32

SQVAL= SQVAL- 2^R where R is range from unit to objective

TUMBLECHART FOR RUSSIAN MOVE (CENTRAL PORTION)