# Entertainment Everywhere: Programming Game on Graphical Calculator

Runjie Guan
University of California, San Diego
San Diego, United States
E-mail: guanrunjie@gmail.com

***Abstract*—I decided to program a game on the Casio fx-9750GII. The objective of the game *Defend Your Base* is attack the enemy using weapons programmed in the game. The programming language used in the computer is generally similar to BASIC but contains more calculator-specific functions and commands. This paper details the whole process from first designing to final debugging step-by-step and includes the code for review.**

***Keywords—programming; calculator; game; design***

## I. INTRODUCTION

### A. Disclaimer

The views and opinions of Casio fx-9750GII graphic calculator expressed in this paper are those of authors and do not necessarily reflect the official illustration of the product (unless explicitly referred to). Company and product names used in this paper may be registered trademarks or trademarks of their respective owners

### B. About Casio fx-9750GII graphical calculator and its programming language

Casio fx-9750GII graphical calculator is a versatile calculator with functions including but not limited to basic operations, equation graphs, statistics processing, programming and transferring data to the computer, which are all the functions I am going to use to program the game.

The screen of calculator is black-and-white, and the resolution is 128 pixels width by 64 pixels, which means the screen can only display 7 rows of 21 large characters or 10 rows of 40 small characters at maximum.

The calculator has two types of storage pool:

- letters, preferred storage choice
- *list*s, secondary storage choice.

These variables can only store number ranging from $10^{-100}$ to $10^{100}$ (both excluded) with a precision of 14 significant digits. There are 28 letters (A to Z, *r* and $\theta$) and 24 *list*s. The letters can be assigned a new value by using a right arrow "→" or read directly. The *list*s are secondary storage choices because the user cannot directly store values in the *list*s (in fact it is even NOT supposed to be used as variables; I am just taking advantage of it): the user has to initialize the *list* by `1→Dim List X` if the *list* is empty, and store the value by using `Fill(N,List X)`, and read the value by using `Sum List X`, where `X` refers to the ordinal number of the *list* and `N` to the value.

Besides the basic if-else-, while- and for-loop-statement, the calculator provides other commands which allow the programmer to draw lines or plot pixels on the screen.

Due to the diverse functions of the calculator and two ways of displaying (graphic screen and text screen), there are many commands similar to each other but completely different in its arguments, either required or optional. (e.g., when displaying massages on the screen, the programmer has two commands to use: `Locate X,Y,Txt` or `Text X,Y,Txt`, where `X` and `Y` refer to the coordinate of the display and `Txt` to the contents of the texts to display. But in the former function, `X` refers to the coordinate of horizontal axis and `Y` the one of vertical axis, and it displays larger character; in the latter function, on the contrary, `X` refers to the coordinate of vertical axis and `Y` the one of horizontal.)

When a programmer wants to use both commands like these interchangeably, it will take much effort to distinguish the differences between them.

### C. About my self-taught experience on programming

I have been teaching myself programming for more than two years, and by the time I touched the calculator, I had already learned Pascal, Visual Basic and Java, and was going to learn C++ myself. I had finished my program in Java, a program that helps me expand my vocabulary base the summer of the last year.

With previous experience on programming and basic knowledge of algorithms, I believe I can master this calculator-featured language, which has subtle differences from Basic but contains more calculator-specific functions. Unlike Visual Basic, this programming language requires the user to calculate the position of each graphical ingredient on the screen as accurately as possible. Unlike C++ or Java, this programming language does not support braces or indentation to clarify the structure of the codes.

But I felt confident to conquer this language. I believed I could make it like what I did to my handled languages before.

As I thought, entertainment exists everywhere as long as you try to find it.

## II. The Design of *Defend Your Base*

### A. *The initial idea of* Defend Your Base

Considering the limit of the screen size, calculator processor, the type of value variables can store (numbers only) and purely black-and-white screen (even without gray), it is almost impossible to write games of huge project like those we play on our personal computers, not even of "smaller" ones on mobile devices, e.g. *Angry Birds* or *Fruit Ninja*. It is also not likely to write, classic games such as *Snake*, *Mine-sweeper*, *Tetris* (or known as *Russian Box*), or complicated card games. My friends suggest me to try to write *Killers of the Three Kingdoms*, one of the most popular board games among my mates, but dissuaded me to do so when he found that there were no more than 30 variables available and the calculator does not support the use of array.

With so much limitation posed on my game, I listed all of the requirements should be met by my game. It should

- have few variables
- contain simple operations (otherwise the operations speed will be affected)
- have simple graphic design

Trying to be innovative, I also rejected to write classic game like *Whac-A-Mole* or *21 Points*. After several days of painstaking brainstorm, I was sitting in the classroom when a clutch of clouds passed by and planes went behind through them. Suddenly, the passing planes inspired me to the games something like *The Legend of Bombers*. In the game, the player can adjust the angle of releasing and the force of launching bombs at the both right time and height in order to gain higher scores; it also requires strategy and accurate operation! It is a perfect design, but unfortunately, the moving object is so difficult to display on the screen that this idea was dismissed immediately.

However, when I kept staring at the clouds, I suddenly realized something. It popped out like an epiphany. In Chinese, there is a phrase called "turn the guest into the host." That moving objects could not be displayed did not mean that the static ones could not, and since the targets could be static, it is possible that I turn the targeted objects into targeting ones. That is to say, rather than attack, the game could be something about defense: the "enemy" in the game of *The Legend of Bombers* could be the object controlled by the player, and the bombers were exactly what to be destroyed.

It was like a flow in my mind and I could not stop thinking. The bombers – now the enemies – could be static at the moment the player tries to aim at, but in each turn of the game, the bombers could be mobile – its position could be refreshed and randomly assigned by calculator, by which the difficulty varies. On the other side, I could increase the difficulty of defending by adding the elements that I previously thought for the bombers – the angle, the power, the height and so on – to the defending mechanism.

Which forms the initial idea of my first calculator game: a game named *Defend Your Base*.

### B. *The structure of* Defend Your Base

The next step is to think about the basic structure of the game.

The programming language, unlike other object-oriented programming (OOP) languages I had learned recently at that time, is a procedural-oriented programming (POP) language in order to solve mathematical problems. Knowing almost nothing about POP even though I had experience with programming in Pascal long time ago, I believed I should start my program from scratch like what I did with OOP.

I drew a flowchart of my game on the draft paper. It started with a game launcher `'LANCHR`, followed by a initializer `INIT` to initialize the display background of the game, followed by a main core program `MAIN` and a functional program `STORE` (the name explains its function well), followed by a program `FINAL` to evaluate the behavior of the player in the "combat." (The calculator only allows program name under 8 letters of upper case and other characters, and this launcher needed to be prefixed with a single quotation mark to be sorted first on the program list for players to launch the game more easily.)

In `'LANCHR`, the program only needed to offer three options: initialize game, new game, and load game. (The program can be terminated at any moment by pressing "AC/ON" button so that the game was unnecessary to add "exit.")

In `INIT`, big efforts should be made to draw enemy units and status bar of the player, which I will expand later in II.D.

As the most important part of the game, `MAIN` is the core of the program and plays pivotal role in interacting with player. The more time is spent on this single program, the more interesting this game would be.

In `STORE`, the player can purchase supplies as the game progresses. As `INIT`, this program also needs careful graphical interface design in addition to the appropriate price I set for the game because the balance of the game is one of the significant factors contributing to the playfulness of the game.

At the end of the game, `FINAL` will provide as much information and statistics of this round of the game as possible with a final score. With this final evaluation, any players will be able to compare their scores and the one with lower score will then try his or her best to beat another. And of course, the setting of the statistics in the evaluation needs comprehensive understanding on the game to determine which element in the game is more important than others to demonstrate the ability of the player. For instance, the number of enemies killed is more useful in showing the player is a good defender than the health he or she lost. And to create a standard for measuring those elements and statistics is another obstacle to overcome.

Then it comes to the details of `MAIN` and the basic structure of the game. In the final version of the design, there will be three units: a cannon fixed on the ground controlled by the player, an enemy plane in the air and an enemy tank on the ground whose positions of both are assigned randomly. The player could adjust the angle of cannon barrel and the power of launching bombs. I felt that the setting of the plane unit is necessary because,

although it would be difficult to test whether the plane is hit in the program, the existence of barrel and adjusting angle should help players get higher scores.

Besides, there will be three types of data of the player, and at least one of which determines whether the game continues:

- cash, which can be earned by destroying enemies units and can be used to purchase supplies in the store
- the number of available bombs to be launched
- the health point (HP) of the base, which determines whether the game is over – as soon as it reaches below zero, the player loses the game

A good game also needs diverse systems to increase its playfulness. As initially thought, *Defend Your Base* would include systems of rewarding, store (purchasing), final evaluation and archive, which I will explain further in section E. The archive system is quite necessary because if the player carries the calculator elsewhere, it will be a waste of battery for the players to keep the calculator awake to pause the game if the game is interrupted when, for instance, the bell rings or called by the teacher, or it will be a pity if you have to quit the game when your classmate comes to borrow your calculator for a complex mathematical operation, and it is bad to reject them, of course, because one of the purposes of my game is to bring classmates closer.

### *C. The general rule of* Defend Your Base

In the game *Defend Your Base*, the player plays a character of artillery who is trying to defend his base. The objective of the game is to defend the "base" and try to eliminate as many enemies randomly appeared on the screen in each turn as possible. Each turn contains one defense controlled and operated by the player and one attack from the calculator. Once the base is fallen, the game ends.

The player can adjust the angle of the barrel or power of launching the bombs: the greater the power, the farther the bomb hits. The player cannot adjust the cannon's height. There are two types of enemies, planes (or bombers) in the air and tanks on the land. With appropriate angle and power of launching, the player may probably hit both enemies at the same time, but it is also possible that the positions of enemies prevent such things happening.

At the beginning of the game, the player has 15 bombs (25 at maximum) and 100 HP (100 at maximum, once below zero, the game is over). Each launch consumes one bomb. The rewards (cash) of hitting the enemies, either planes or tanks, or both, vary according to the accuracy of hit: the closer the player hit to the center of the enemies, the more cash the player gets. Because planes are difficult to hit, their rewards are higher than tanks'. Each hit doubles the cash next turn, and this double is accumulative. (e.g., if you can hit enemies, either or both, in ten consecutive turn without missing a shot, you will have a bonus cash of 1024 times.) If you hit both enemies together in each turn, you will receive a reward of extra 50 HP and quadruple the cash next turn.

The cash can be used to purchase the bombs or repair the base later in the store. The store can be launched at any time.

After player's attempt to eliminate the invaders, it is now calculator's turn to attack. Both enemy units can cause damage to the base. The base damage depends on the distance of enemies from the base (the cannon) and will fluctuate randomly. If the enemy is destroyed by the player, it will not cause any damage. If you hit both enemies, congratulations on your success this turn and the base will not lose health point.

As more and more bombs the player launches, the difficulty level increases. The difficulty level will not decrease. The higher the difficulty level, the more damage the enemies will cause and the more cash the player has to pay for extra supplies to maintain his or her survival.

When the game ends, the calculator will give the player its detailed report and evaluation. The evaluation report includes the data of the total earned scores (cash), the total lost HP, the numbers of either type of enemies eliminated and the multiplier bonus based on the current difficulty level.

### *D. The graphical user interface (GUI) of* Defend Your Base

#### *1) The GUI of Program* `MAIN`

When a player is playing the game, he or she must know what he or she wants to know and needs to know, but sometimes if too much information is presented on the screen, it will be full of numbers and graphs, even worse, most of which are what the player probably do not care. On the other extreme, if little is presented, the players will have a bad playing experience as if they are finding their ways through a complex maze, too. Besides the adequate amount of information to be displayed, the way it displays also matters, because the programmer may not want to compromise his or her good design of codes and contents of game with poor works of GUI. Basically I thought that there are three ways of presenting information: text only, graph only, or both.

The screen of the calculator is smaller and limited, made out of 128 pixels width by 64 pixels height. At first I just considered about having two screens shown in turn: pure battlefield and pure statistics, presented alternatively. But as a fast-paced game, in which each turn of defending is less than five seconds, there was no need to separate two pictures apart; I had to integrate them to display both the information and the battlefield so that it is available all through the game.

If I want to save the space for larger combat, I may simply use plain numbers to indicate the cash, the HP of the base and the available bombs left, each followed or following by a label, but that is poor GUI design. I should not enhance the effect of the battlefield at the expense of poorly designed battlefield.

Then what about the ideas of combining the plain numbers with graphs? This idea sounds good, but it is almost impossible to draw a vivid figure on a calculator screen, whose pixels can only be either on or off, because that will take up so much space on the screen.

After trading off, I finally came up with my design of `MAIN`'s GUI (Fig. 1). At the top of the screen is the current cash followed by the result of cash earned after each turn (the figure of cash, however, looks like a gun to imply that it can be earned by killing

enemies). At the left side of the screen is the status bar of HP and available bombs with corresponding tiny figures next to each. The player's cannon and its barrel are at the left side of the bottom. The launch power is between the status bar of bombs and the cannon. The positions of enemy units are limited within certain coordinates so they will not interfere with the normal display of game data. The figures, I believe, are quite self-explanatory.

Fig. 1. The GUI of `MAIN`. Battle control standby.

During the trade-off, I encountered many unexpected problems, which I will explain further in section III.B.1).

*2) The GUI of Program* `STORE`

The purpose of the store is to sell supplies to the player, so there should not be too many figures to display on the screen. Besides, since there should be much information presented on the screen and the store should provide multiple kinds of packs of products, I finally decided not to put any figures in `STORE`. The goods in the store and their corresponding prices should be obvious and compacted to provide necessary information the player needs to know. Besides, I should consider the alignment of the products because the calculator does not have touchscreen or mouse, which means the player can only select what they want by:

- using an indicator like an arrow pointing to a commodity
- pressing the corresponding button beside the product
- inputting a specific number representing specific commodity and its amount of purchase

The first solution seems the most favorable because unlike solution 2, it will guarantee that the limit of buttons on the calculators will not pose a threat to the future update of more items in the store, and unlike solution 3, it will not contain difficult or unnecessary codes for the player to remember just in order to play the game. Otherwise the player turns to be the played; we are playing the game, not played by the game.

Whenever you go shopping, either in local stores or online, after paying the bill, you may hear a melodious sound indicating that your payment is successful. In the game, the player wants the same experience too, so `STORE` should also give feedback on the transaction to the player. And the feedback should be simple in order to save time and memory.

The player may also want to know the change of their game data after transaction to decide whether to buy more or to leave the store. Therefore, the information of the cash, HP, bombs and difficulty level should be visible throughout the transaction.

As shown in Fig. 2, which is quite self-explanatory, each information or data is presented in front of the player. After each transaction, the screen will be refreshed so the status can be up-to-date. The player can use the navigation pad on the calculator to select the items they want. Besides, I devise a way of using a common marketing strategy: the more the consumers buy, the more discounts they will enjoy.


Fig. 2. The GUI of `STORE`

There are two main problems when designing `STORE`'s GUI. I will explain both of them further in section III.B.4) and III.B.5).

*E. Diverse systems of Defend Your Base*

*1) Reward system*

Reward system is the most exciting component of the game not only because reward can stimulate the player to keep playing, but also because this system is "crazy".

*a) Double system*

As I mentioned in the section C, continuous hit will double the reward. And it is a power-law bonus: the more you hit the enemy (enemies) continuously, the more cash you will earn the next turn. Generally speaking, people may consider this will compromise the balance of the game because once you can make a sequence of 10 shots without a miss, you will earn at least 1 million cash! Compared to the price in the Figure 2, the player will become so wealthy that it is seemingly impossible to die!

Of course it is not likely for player to have such exquisite skills. Many players new to the game, as I found after the alpha version (the first version of my game that I only shared with my closet friends) of *Defend Your Base*, even was not able to eliminate *any* enemy before they lost the game. After getting used to the trajectory of the bombs, players were able to increase their hit rates up to 75% when the target was close enough (by *close*, it means that the tank is at the left third of the screen). When the tank appeared at the right third of the screen, the rates dropped below 25%. Even as a programmer, I cannot guarantee my hit rate is over 50%.

*b) Bonus system*

Besides cash earned possibly as rapidly as a rocket, there is another reward called bomb bonus. If a player can earn more than 500 cash in one turn (after multiplied by double multiplier), he or she will be rewarded with one bomb. For every further 500 cash the player earned, the player will be rewarded with an extra bomb. The maximum number of bonus is nine.

What's more, to counterbalance the countless cash and unlimited bomb bonus possibly caused by exponential explosion, I develop another system, difficulty level system, whose trick I will explain further in section 4).

In the future update, I may add another system to increase the difficulty: wind interference system. As the name of the system suggests, there will be an "invisible" force pulling or pushing the bombs off their targets. However, as none of my classmates had already conquered the game (I neither), this system was just an imagination.

*2) Store system*

As you can see in Fig. 2, there are three purchasing choices of every product. The player can decide whether to fill up all of their items, to buy five items, or to buy only one if he or she is short of money. The player needs strategy to make their choices in order to survive longer.

In the alpha version, the choices are a bit different. The player can choose to fill up, buy or sell a single item. When I tested the game myself, I found there were some problems with it. In each turn the player would loss HP of 10 points on average. Think about this. If the player could not afford to buy a fill-up and thus had to repair the base point by point, he or she had to select the item, press EXE (confirm button on the calculator), wait for refreshing status (which can consume about one second), press EXE, wait again and then press… The whole process of purchasing can consume at least ten seconds while the player keeps pressing the buttons. The same thing happened when I tried to convert my HP to cash to buy a bomb as a last ditch. However, when discovering I had to press the button for more than 20 times, I forfeited and thought about trying for a new game. The same thing will certainly happen on many players as well, so I needed to fix this problem.

And the selling function was useless as well. The price of buying a health point was pretty low and, ipso facto, the price of selling the health point is lower. Few, I supposed, would bother wasting half a minute to try a last ditch instead of quitting the game and starting a new one unless the player wants to break a record, but it rarely happens. With regards to sell bombs, I made accurate calculations to guarantee that the price of selling a bomb was lower than the price of minimum cash by hitting a tank. Most people would try a shot rather than to give a bomb away for cash.

As a result, I increased the basic unit of purchasing HP to 10 points. In addition, I replaced the selling with a pack of five items with proper discount – as the currentversion of store presented in Fig. 2.

*3) Evaluation system*

Fig. 3 is an example of poor evaluation of the game.


Fig. 3. The GUI of `FINAL`. A Poor Evaluation

In the figure, the player can clearly see his or her record throughout this defense: the total cash the player earned, the HP lost, the planes and the tanks being killed, and a multiplier based on the level the player reaches.

*4) Difficulty level system*

The difficulty level is based on the amount of launched bombs. The more bombs the player launches, the harder the game will be. It is dynamic. The difficulty level increases by one in five bombs.

The difficulty level directly affects two data:

- the damage caused by enemies, with a gradient of five percent increment
- the price of items available in the stores, with a gradient of ten percent increment

The difficulty level will keep increasing unless the game is over, which means the damage caused by enemy is unlimited – it is possible that you will be seckill-ed (killed in one shot) by enemies even though your HP is full.

However, the player does not necessarily need to worry about that. If that happens, the difficulty level will be at least 120 while almost 600 bombs have already been fired with the price of each single bomb at least 500. However, without any double combo, a player can earn only 280 at maximum. You can imagine how difficult it is to survive in that harsh condition before the player is finally killed in a single shot.

In addition, even though the player is wealthy enough to have the potential to survive to level 120, he or she may still lose the game before that. In other words, the player may not be able to use up all the cash before the game ends. The reason of it, along with some antidotes, both of which are quite interesting and thought-provoking as I discovered later, is described in Appendix 3.

*5) Encrypted archive system*

People can carry the calculator everywhere, so we can play the game everywhere. When they are waiting in the queue, or want to play something during the classes or feeling boring in the dormitory, they can turn on the calculator and play the game at any time. But if the players are interrupted and have to quit the game when the war is still raging, it is annoying because they have to either quit the game or put the calculator back into the pocket to pause at the risk of exiting the game by accidentally touching the "AC/ON" button and at the expense of wasting the battery because the game is still running.

The calculator does not support background running. When the player has to switch to other screens, he or she has to intercept the game and by so lose game process, except for some variables stored in local memory. And as long as the memory is available it is possible to load the game if necessary.

The `MAIN` program will reset every variable to be used later to the initial state, and it is possible to prevent the game from resetting process to load the previous game progress instead. Doing so is not difficult to implement this function by simply giving the user choice whether to load the available data from letters.

However, what is worrying is that it is then possible to cheat by modifying the data and loading the game. Therefore, I

designed a strategy to fight against it. The strategy, as explained further in section III.A.4), is to require the player to press "exit" button on the calculator to save the game progress, so he or she may resume the game at any time.

## III. The Difficulties of Programming *Defend Your Base* & Solutions

Knowing nothing about this language before, I encountered various problems during writing the code. These problems can be tricky, difficult or interesting. And solving them sometimes can be really time-consuming and annoying – I knew where the problem was and what it should be, but you just had no idea how to find the most optimal path or simply did not know which path to choose.

Most of the problems and difficulties have been solved by now. The rest of them, as presented in section VII, are either complicated or insolvable due to the limitation of the calculator.

### *A. Machine-based problems*

Machines-based problems refer to those related to the code design or the handling of the limitations of calculator itself. Here are several most thorny problems encountered.

#### *1) Limited variable storage*

Variables lay in the center of programming. It is the vein of the program. It is so essential that, I believe, a good design of variable can significantly affect the operation speed of the program. In the first chapters of almost every book for new programmers, the authors will take several examples of naming variables like

```
oh_gosh_this_is_an_extremely_long_variable_na
        me_and_that_is_not_the_end
```

to illustrate the importance of variable name. However, the calculator has only 28 letters to store the values, so each letter is precious and worth thinking twice before assigning a value.

Whenever programming, I would consider any possible future update. Therefore, although my current final version still has half available varieties of all, I will still take care to "reclaim" an unused letter. In addition, that the letter is the variable means I cannot name a variable. I had no idea how to map a letter to an object. It is difficult and unreasonable for the programmer to force myself to remember this meaningless "map" when programming.

At first I decided to use the capital of the name of the object to remember corresponding letters. When finding this complicated, I simply listed all the variables on a notepad (as you can see in Appendix 2). Although I had to refer to the notepad each time I forgot and to take several copies in case I lost the original list, I found this useful.

This problem is caused by hardware or complier that I was unable to handle. All I could do was to try to adapt it.

#### *2) The discovery of another storage pool*

However, when I was programming `FINAL`, another problem emerged. The program needed to count the total cash, the bombs launched (in order to calculate difficulty level) and the number of killed enemies to present data as listed in Fig. 3. I needed to find a special way because these data shared similar characteristics:

- They are the evaluation standards of the final game report
- They will only increase
- Their opportunities of being assigned are lower
- They are read at the same time
- New members may be added in the future update.

(The fourth item, however, was one of the mistakes I made during the whole program. At first I did not aware that anytime the store menu was opened or the enemy hit the player, the calculator had to read the value of difficulty level converted from the amount of launched bombs.)

They were like a family that the best way to group them is to put these variables into an array, even though, unfortunately, the calculator did not and would not support. I tried to go through every command of the calculator and read the manual for several times, but no sign showed that I could use array to store the values in my calculator.

Then I thought about other ways of representing arrays. By asking myself "Is there any storage system other than letters may help?", I went to the system menu in the calculator where I could view all the storage usage of each type. There were two potential candidates for this task. The one was *list*, the other one was *function*.

*list* is actually a spreadsheet, so it turned out to be a relatively favorable place to store the value. It would also be relatively convenient to access the data. On the other hand, *function* can help solve the problem by storing the values as a constant function. Each time you want to access the data, you may just ask for the value of *y* at any value of *x*.

Unfortunately, it seemed that the program did not provide any command to modify the equations of *function*s, but provided simple commands to modify the numbers in the *list*s.

Finally, it was obvious that *list* was qualified to be variables, and with the command like `Sum List X`, where `X` stands for the ordinal number of the *list*, it is possible that the program can itinerate through every data in one for-loop.

#### *3) Statement block confusion*

The nesting of statements is common in programming, and different programming languages have different ways to deal with it to make the organization of code clearer, such as indentation or braces.

However, with this distinct calculator language, I did not know how to solve this problem thoroughly. There was no indentation or brace, just simply `If`, `Then` and `IfEnd` (the `While` and `For`-loop is simpler because these loops contain only two commands to start or end). `If` the of statement is short, `Then` it is clear, `Else` it is fine, `IfEnd`. But if the nesting of statement was complicated, you might put an `Else` in a wrong position without a clear organization of structure. Finding such

bugs and then solving them was time-consuming, and you might not even know what the problem was until you realized that it was the problem.

This problem happened when I was testing `STORE`. No command like `case` in C++ or Java provided, I had to write if-then and else statements six times to determine the type of items the player purchases, and in each block nest another two hierarchies of if-then-else's to test if the player has enough money and if the items are full. The program would give appropriate message to tell the player whether the transaction is successful and if not, why not.

At first I was not sure whether it needed another `IfEnd` after an `Else If` to complete the code, so I tested the following code on a new program.

```
1  If 0
2  Then 0
3  Else If 0
4  Then 0
5  Else 0
6  IfEnd
```

Fig. 4. Source code of a simple test program

It turned out that this code works quite well, which made me believe that `Else If` did not need an extra `IfEnd`, so I felt free and confident to omit all the `IfEnd`'s corresponding to `Else If`s. I ran my `STORE` program, but the calculator gives me a *syntax error* message. I was surprised at first and later checked all the code but found nothing weird; all the `If`'s, `While`'s and `For`'s were perfectly matched.

I extracted the nesting if-else statement out of `STORE` and tested it in a separated program. It worked fine. That being the case, the problem must be in the rest of `STORE`, but when I tested `STORE` without nesting statements, it worked fine too! What was the problem? Was there anything wrong with the complier? Does that mean I had to separate two programs instead of an integrated one to finish the game? IS it really the calculator at fault?

I thought about the way the complier worked inside the calculator. I had a sense that it was something wrong with `Else If`'s. As I later discovered, the `Else If`, in a sense, would not be integrated by the calculator to be recognized as an integral command; on the contrary they were two separate commands. The complier or the processor of the calculator might be so limited that it could not combine any two commands into a single one, which might also explain why the user of the calculator has to select the commands from the menu in the calculator rather than to inputting those commands letter by letter directly as I could do later in the calculator's transfer tool on my laptop – the computer could afford to convert the file between plain and calculator-specific text much.

But why, then, it was possible that the code in Fig. 4 worked, even though I attached several extra `Else`'s? I rewrote that code, and ended it with an extra `IfEnd`. It worked fine too.

Then it was clear that the calculator must add omitted `IfEnd`('s) at the end of the program automatically when it detected the *syntax error* that I omitted the necessary command to complete one block, but then succeeded in "trying" to help me solve this error, so this syntax error went smoothly undetected. When it comes to the more complicated program `STORE`, it failed to do so because simply adding several `IfEnd` doesn't work when there are other blocks of code after the problematic nesting `If`s.

There goes, eventually, seven straight consecutive `IfEnd`'s, which can occupy the entire calculator screen when input, as you can see my full code in Appendix 1.

*4) Save and load game progress*

Although my programs were locked so the player could not access the source code and read them directly, my clever classmates could still guess which letter stands for, for example, the cash, by reading the value of each letter, and modified the value to go back to play the game like a boss.

Cheating in a game is a serious problem to deal with. As the police fight against the crime, I believed to fight against cheating is to understand the process of cheating. And if any link among sub-processes is broken, the cheat can be prevented.

The process of cheat is quite simple: the player discovers the data, modifies it, and loads the game from the modified data.

For the first step, if we want to prevent the user from discovering the data, encryption is an effective to prevent cheating happening because by no means could I prevent players discovering those variables unless the button is ripped off the calculator. The program can encrypt the value, store it, and decrypt it when needed. However, all the process would slow down the speed of the operation. It would take much time for the program to access and use those important encrypted data. And I would have to go through all the programs to deploy encryption and decryption on every variable. In addition, once the encryption key was cracked, I had to create another one (it is not the most difficult part actually because both simply increasing the value by 2 and doubling work) and go through the programs again (and this one is tedious).

For the second, it was not possible to stop my clever classmates modifying the value because the attribute of the data could not be changed to "read-only" after the game ends.

For the third, the program can discover the exception of data and recognize it as abnormal modification by checking the *security code* (SC) generated by user data. By using SC, the program can make sure that all the data was intact through the game. Once the game data is modified, the SC generated by modified data will mismatch the original one. If that happens, the program will reset all the values to start a new game instead of loading the available data. Besides, it is convenient for me, the developer, to switch between release mode (i.e. customer mode") and debugging mode by turning checking SC on and off.

The third way attracted me most because not only could SC protect the data, but it could also help me test the game. Now it came to the question about the working mechanism of SC. An ideal SC would be able to include all the user data and detect exception as long as the data are modified; a special operation among these values a little more complex than simple addition will be fine.

SC will stop cheating, on the other hand, when the player tries to cheat by loading the game repeatedly to get a better score or combo. Because once the player presses the key to adjust the launch power, the number of bomb decreases by one, and the old security code is no longer available for the new data. If the player finds the trajectory unsatisfactory or misses the target, it is impossible for the player to reload the game again by force stopping the game.

This method has a small drawback consequently. It needs the player to press a key to save the process; if the game was terminated inadvertently, all the data will be lost. But given the fact that few games I have played so far support instant archive right now, it should not be considered as a big problem.

*B. GUI-based problems*

The visualization of the program needs to position the components by using functions or commands rather than the programmer did in Visual Basic by dragging every window or component and adjusted their sizes directly to see their changes immediately. But in Java or this calculator-specific programming language, I must calculate the coordinate of everything and use the results as arguments in the function or command to place them in the right position (As a learner so I do not know for sure whether there is any tool like "Visual Java" to facilitate the visualization of Java program. Here I refer to the package `javax.swing`.). And since this calculator is a graphical calculator by nature, all the figures appeared on the screen are basically the graphs of different functions.

*1) The figuration of every unit*

In order to draw recognizable figure on the screen, I had to scratch them on the drafts and then convert them into pixel form to fit in the calculator screen. I opened Photoshop on my computer and set the canvas size as large as the calculator screen. Then it was convenient to draw the figures after I zoomed in the image, where every pixel was presented in its own grid.

I had to define the property (i.e. the range of *x* axis and *y* axis displayed on the screen) of the graph view-window so that it would be convenient for me to calculate the right positions of each figures.

When drawing in Photoshop, the coordinates of each pixel the mouse pointed is shown on the computer screen. With the help of those numbers, I could convert them using an equation developed by myself to the corresponding coordinates on the calculator. Whenever I wanted to adjust the positions of figures, I could nudge them to see the effects in Photoshop at first. That would save me a lot of time before I finally found the best way to present and organize all the figures in the GUI.

*2) Refreshing the screen*

Graphing on the screen of this calculator like drawing on the paper using a pen: once you write something on it, it cannot be modified, unless you change the paper for a new one. The same thing applies to the graphical screen: once you graph the function on the screen, the only way to clear the ink efficiently is to use command `ClrGraph` or `Cls`, which means every graph drew before is wiped out.

How could that happen if I needed my GUI fixed on the screen? Or how slowly my game would be if the program had to generate the GUI each time when only certain part of the screen needed to be refreshed?

Fortunately, this time the calculator provided commands of `StoPict` and `BG-Pict`, by which I could save the screenshot of the screen in the memory and set that image as a background picture. This background picture served as the pattern printed originally on the screen regardless of the use of screen-reset commands mentioned above in this paragraph. .

With the help of fixed background, it was faster to refresh the screen so that each time the player wanted to adjust the angle of cannon barrel, he or she did not need to wait for a long time any more. Everything other than cannon barrel is archived in the background picture.

Now the player can enjoy faster speed to prepare for their each turn of battle.

*3) The display and the calculation of the trajectory of the launched bombs*

When a bomb launches at a given speed, it is easyto calculate the equation of the trajectory before the bomb hits the ground.

Using knowledge learned in mathematics and physics in the first semester in my high school, and given the launch speed of $v$, the angle $\theta$ between the barrel and the time $t$ since the bomb launches, I calculated the parametric equation of the trajectory as (1), where $x_0$ and $y_0$ refer to the initial coordinates of the bomb and $g$ to the gravity constant.

$$\begin{cases} x = x_0 + vt\cos\theta, \\ y = y_0 + vt\sin\theta - 0.5gt^2. \end{cases} \tag{1}$$

It was OK to apply (1) to the program after necessary replacement. To fasten the speed of operation, I decided to use a single pixel to represent the bomb and its trace on the screen to represent the trajectory. I plotted on a single pixel at a regular interval on the screen. The pixels, when connected, still look like a recognizable perfect parabola.

But how can the program test if the bomb hits the enemy? And this solution should apply to both tanks and planes. By testing whether the *y*-coordinate was negative, the program could determine whether the bomb hit the tank, but it is a problem for me to test whether the bomb hit the plane. I think about testing whether the product of difference between the *y*-coordinate of current bomb and plane and difference between the *y*-coordinate of previous bomb and plane is negative or not, but this constant (because both evaluations have to be done after each change of the bomb position) test could really slow down the speed of the display of the trajectory on the screen. Worse, even though the program could probably get the answer, the result may not perfectly accurate because the only way to get the result was to draw a line between two points and calculate the *x*-coordinate where both *y*-coordinates equal to each other.

Consequently, in order to calculate the amount of cash the player earns each turn, the program has to:

- determine whether the bomb hits the enemies

- if so, calculate the function equation of the line that goes through two points nearest to the target
- calculate the rough *x*-coordinate of the hitting points
- use a cash formula to calculate how much cash the player will receive finally

As it turned out, this worked. But the speed of traces appearing on the screen was so slow that it made me feel like the cannon was not launching a horrible bomb. What more surprised and worried me was that sometimes the rewarded cash did not seem legible. For example, in Fig. 5, if the plane is flying close to your base and you launched a bomb (whose traces are colored red) at a high angle in order to hit the plane, the bomb glances off the plane's top and misses it! The line (colored blue) formed by connecting two points nearest to the plane, surprisingly, intersects the bottom of the plane.

Fig. 5. A lucky shot

[a] This figure is not a screenshot of my calculator and is for illustration only, but it really happened in the game. This footnote also applies to Fig. 6

The occurrence of such problem probably means it was better to consider another version of algorithm. Even though it rarely happens, it did alert me to the error of the result or the "bug" my code generated. Then it was when Plan B came into play: to basically calculate the function equation of the trajectory and to find *x* where the value of equation equals to zero or the *y*-coordinate of the plane. Although it might be a bit more complicated, I believe that the programmer should try his or her best to reduce the number of known bugs.

And it takes me an entire evening to convert (1) into (2)

$$y = 0.9 + 0.6\sin\theta + (x - 1.4 - 0.6\cos\theta)\tan\theta - \frac{(x - 1.4 - 0.6\cos\theta)(x - 1.4 - 0.6\cos\theta - I\cos\theta)}{40(I\cos\theta)^2}. \quad (2)$$

[where $\theta$ refers to the launching angle and $I$ to the power of launching bombs, similar in (4), and I replace $g$ with 0.05] and to convert into the form of

$$y = ax^2 + bx + c. \quad (3)$$

to find the solution. I had to calculate the function that contains complex operations by hand, convert the letter into the variables, convert the variable version of the equation into linear function and test it on the calculator and debugged it several times, while the bug sometimes came from my careless conversion instead of equation itself.

$$y = -\frac{1}{40(I\cos\theta)^2}x^2 + \left(\frac{2.8 + (1.2 + I)\cos\theta}{40(I\cos\theta)^2} + \tan\theta\right)x + 0.9 - 1.4\tan\theta - \frac{(1.4 + 0.6\cos\theta)(1.4 + 0.6\cos\theta + I\cos H)}{40(I\cos\theta)^2}. \quad (4)$$

The way I tested whether my equation was correct was to graph it on the screen. If the graph fit the pixels plotted perfectly, the equation must be right. Now with equation, I considered switching the plotting crude pixels to graphing a parabola on the real game, but I did not do so in the end because graphing was slower and, more importantly, a complete parabola could not perform the beauty of the flying bomb.

With (4) in the form of (3)(2), the program could calculate the zero of the equation and choose the bigger one as the point where the bomb hit the ground. The smaller zero was useless unless the cannon "backfired". The answer could then be used to calculate how much cash the player got.

To find where the bomb hit the plane in the air (if so), the first thing was to determine which side of the parabola symmetry the plane was on by simply comparing *x*-coordinates of the plane with the parabola symmetry. If the plane was on the left side, find the smaller root of equation "*y* = {*y-coordinate of the plane*}"; if on the right side, larger one. However, the limited storage warned me to save variable to use. Is it possible, then, to combine the testing and getting the result together? It is, of course, because in the classical formula of root (5), the operator before the radical can be replaced by the plus or minus before the difference between the *y*-coordinate of the plane and the symmetry.

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}. \quad (5)$$

Now I met another problem: the calculator did not provide sign function to return 1 if the given value is positive, 0 if zero or -1 if negative. I had to write it myself. Therefore, I simply devised (6) to return the value. In (6), $\Delta x$ equals to the *x*-coordinate of symmetry minus the *x*-coordinate of the plane. The pair of square brackets "[*x*]", which is a type of floor function available in my calculator, returns the largest integer smaller than the given number *x* (similarly hereinafter).

$$f(x) = 2\left(\left[\frac{\Delta x}{50}\right] + 0.5\right). \quad (6)$$

It is now possible to get the answer equipped with (5) and (6). Its advantage over the previous method of testing whether the bomb hits the enemy is that the program can know whether the player hits the enemy as soon as the bomb is launched. This saves a lot of time, and makes the present of trajectory smoother and faster.

*4) The display of store: how to display the indicator*

There are numerous ways for the program to choose to design the GUI of the store, so are there various ways to present the items and for the player to select them.

As I finally thought, in order to support more available items sold in the store in future update, it was better to use something other than number as the indicator. Simple arrow (>) would work. In addition, the arrow would increase the interactivity when the player used navigation pad to select the items rather than to press the corresponding number.

But there was a problem if the program used an arrow as an indicator. In order to display the arrow on the screen, we had to specify its position in the command. However, it might be unnecessary to use two variables to represent the position. Instead, there must be some way to save the use of variables. And obviously, using only one variable would simplify the code for the program to translate the position of the arrow into the type of item the player purchases, or as an alternative, translate the type of item to the coordinate of the arrow. It would then save much more memory and time.

Because the six items are displayed regularly in the store in two rows by three columns, if I use the variable *t* to represent the type of item, the *x*- and *y*-coordinate of the arrow [presented in (7), where $x_0$ and $y_0$ refer to the coordinate of the item on the left top and $\Delta x$ and $\Delta y$ refer to the spacing between columns and rows, respectively. The function *mod* divides two given numbers and returns the remainder] can be realized by using calculator-specific functions.

$$\begin{cases} x = x_0 + \left[\frac{3}{t}\right]\Delta x, \\ y = y_0 + (t \bmod 3)\Delta y. \end{cases} \tag{7}$$

*5) The display of store: how to justify texts right*

The greatest problem I met in designing this GUI was how to justify the cash right. In section I.B, I mentioned two commands to display the texts on the screen, `Locate` and `Text`. But the arguments of these two commands only indicate the start position where the texts show, which means it is OK to justify the texts left. But as the program presents the amount of cash the player possesses in the store, it will look neat if the number was close to the right side of the screen regardless of its value.

I did not want to bother writing nesting if-else statements to do so because it was complicated and error-prone, and it had to change almost everything as the condition changed. I considered: the *y*-coordinate would not change as the value changed. In order to set the position of number dynamically according to its value, I only needed to work on the value of *x*-coordinate. The *x*-coordinate changed as the digits of the value presented changed. Their relationship was linear. And after thinking for a while, I could combine denary logarithm and floor function in (8) (where $x_0$ refers to the *x*-coordinate when the non-negative integer *n* is a single digit and $\Delta x$ to the width of a digit) to display justified text on the screen.

$$x = x_0 - [\log(n + 0.5)]. \tag{8}$$

The 0.5 next to *n* is necessary because *n* can be zero while the domain of denary logarithm is non-negative. And in this game, it is impossible that *n* will drop below zero.

*6) The display of final report of the game: another problem of aligning text right*

In the screenshot, all the products of equations are justified right to look better by using (8). But this equation only works when *n* is an integer. What if *n* can be either a decimal or an integer, when in either situation the amount of character can be different? For example, as you can see in Fig. 3, the relationship between the difficulty level and multiplier is linear. When the difficulty level ends with a zero, the multiplier is an integer; otherwise the multiplier is a decimal.

It showed that when it was a decimal, I had to move the number two more digits left. How could I test if the number is a decimal or an integer? The only difference between them was their fractional part. If I could find how to test the existence of the fractional part, I could then solve this problem.

The calculator supports the operation of Boolean algebra, and supports to use 0 to represent *false* and other non-zero real number to represent *true* directly. Discovering this convenient conversion, I can then turn the number into Boolean.

But I forgot an important thing. Even if the calculator could convert real number to Boolean directly, that did not mean the user could convert it back to the number 1 if *true* or 0 if *false* unless the user uses `True` or `False` in the calculator's library. I should find another equation to convert any non-zero number to 1, by which I could differentiate the decimals and integers. Thinking for a while, I came up with two interesting functions (9) with the same purpose to return 1 if *x* is a decimal, and 0 if an integer.

$$f(x) = \left[\frac{10x \bmod 10}{10} + 0.9\right] \quad \text{or } f(x) = [x - [x] + 0.9]. \tag{9}$$

This equation only applies when the fractional part of *x* only has one single digit. This function simply extracts the fractional part out of *x* then added 0.9. Since the fractional part varies from 0.1 to 0.9, and the result of adding 0.9 varies from 1 to 1.8, this function works perfectly in my program.

## IV. The Debugging and Testing of *Defend Your Base*

Because I will test the program after completing a block of statements, so after the game was done finally, I spent little time for further debugging. But there are still some minor glitches.

### A. Debugging and testing by myself

Besides many bugs and their corresponding debugging mentioned before, in this section only bugs or the process of testing as a whole will be elaborated. Some of them were subtle, but later were believed to played a pivotal role in the evaluation of this game. There was no evidence indicating whether these

changes are actually beneficial or not, but it did suggest that sometimes the programmer should both consider details and the entire thing to develop a comprehensive understanding on something.

*1) The balance of cash*

As the only currency in the game, the balance between gain and loss directly affects the difficulty and playfulness of the game. When I started to test the game at the very beginning, I realized that, my choice that the game should not contain punishment system if the player misses the target for many times was correct, because it is already difficult to earn the cash.

The plane, as I initially set, was only a bit more rewarding than the tank. Later I found that a bit did not compensate for the difficulty to hit the plane. Even if the player fixated the angle to attempt to hit the tank, the power he or she launched the bomb is not proportional to the distance from where the bomb hits the ground to the base, not to mention the difficulty when the player tried to hit the plane for a double kill.

After enhancing the reward of eliminating a plane, I thought that I should also change the price in the store. The average net cash the player is supposed to earn each turn should be a little lower than the price of bomb, because it is impossible to miss all the possible combos every 25 turns (the maximum number of bombs the player can have at once is 25). Besides, if the player can hit the enemies consecutively for at most 4 times, he or she will be priced with an extra bomb. If the player's skill is extraordinary enough, it is unnecessary to buy any bomb at all.

*2) The inaccuracy of trajectory trace: hit or not?*

The screen of the calculator is combined with numerous pixels, so when graphing the trace, the calculator cannot depict them accurately enough. Thus it is possible that the graph shows the bomb goes through the enemies but the program thinks it misses.

This is not to blame the calculator completely. It is not all of its fault. There is also something wrong with my algorithm to test whether the bomb hits the enemy. In order to save memory and speed up the operation, I simply test whether the distance from where the bomb hits the ground to the center of the tank chassis is less than half the width of tank to determine whether the bomb hits the tank. But the tank has height, so if the bomb goes through the tank and falls somewhere that does not meet the criteria set by the code, the player cannot get any reward at all.

Although players may lose the cash they should have earned, they may also discover they will earn unexpected cash when hitting the tank. However, this is not a glitch because the calculator plots a pixel at a regular interval, and the last pixel in the sequence of bomb's trace does not necessarily represent where the bomb falls on the ground. This happens more to the plane. If the vertex of parabola happens to be inside the plane, and the trajectory rises and falls smoothly (E.g. In Fig. 6, the pixel is colored red and the parabola is colored blue. The yellow line refers to the bottom of the plane, where its intersections with the parabola fall out of the plane), it is possible that the program may fail to consider it as a qualified hit (see VII.A for more information).

Fig. 6. Bad hit judgment

*3) Resurrection after death*

Once I was playing the game myself, I made a sequence combo of 9 times, which meant the last multiplier was 512. That incredible multiplier made me extremely rich and it seemed that I would never die. However, even so, I did not survive very long (the reason of which will be elaborated in section Appendix 3).

I could not reconcile myself to accept the fact that I died before I had much money yet to be used. I started to consider anyway I could resurrect by "cheating". It was impossible to load the game because the user data changed, and I did not want to bother calculating the complex security code.

Then I suddenly found I could run other programs even after the game ends. Although this discovery did not help me resurrect, such behavoir was inappropriate or dangerous to allow players to launch these subsidiary programs manually or accidentally even though it seemed no harm because fatal error can always be caused by unauthorized access.

The solution, as I later found, was to specify a letter as the "mode variable." This variable changes as the currently running program switches. I attached each program with a header and an ender to check whether the call of program was legal. If so, reset this variable to zero to prevent rerunning the program; if not, stop the program immediately. In addition, I also assigned the variable zero at the ending of each program and assigned the corresponding value of variable before calling another program.

From then on, the only access to launch the game is from `'LANCHR`.

*B. The advice from my classmates*

This game is designed for everyone, so it is shortsighted to limit the debugging and testing process to myself. I asked several of my friends to take a look at my first game. They were all surprised at first and felt excited about playing game on a tiny mobile device other than GBA or PSP. They took the calculator to their dormitories to use their spare time to test the game as I suggested that I did not want them to waste too much time testing. Here I listed several acceptable suggestions they gave to me, some of them are considered as future updates (in section VIII).

*1) Lower the difficulty level*

As an experienced player, I might feel nothing about the difficulty level because I had played this game several times and somehow got familiar with its trajectory. However, when my friends were playing the game, they said the difficulty level was set so high that sometimes it would make the player frustrating and discouraging. Any player new to the game needs time to familiarize themselves how the game works. However, an excellent game needs balanced challenge and its corresponding reward. Setting them too high and low will both compromise the playfulness and the duration the players play the game.

*a) More bombs at the beginning*

Cash means survival, and more bombs mean more cash. When I was testing my game, I set the initial amount of bomb at 10. Given that the hit received by the player at the average of 10 and filling up HP requires 400 cash, the player needs to earn cash at the average of 90 cash each turn in order to survive, not to mention the impact of difficulty level increment. Even with the help of double system, many players may not be able to survive this period. The objective of this phase is clear and obvious: to earn as cash as possible to repair the base and purchase the bombs.

Every friend who participated in the test complained about this. They said the scarcity of bombs at the very beginning limited their showing off exquisite skills and made the pause system meaningless (because they lost the game before they were intercepted). They suggested I set the initial amount of bombs higher to lower the difficulty. Thinking this suggestion reasonable and watching them playing my game so painstakingly, I decided to enhance the value from 10 to 15.

*b) Increase the cash from killing a plane*

My friends were still not satisfied about the cash they could earn by hitting a plane. They complained that the figure of planes was too tiny and the damage of planes was too aggressive.

The message my friends transmitted to me was clear. Even though I had enhanced the reward of the plane, it was still not enough. Basically I should also do something to compensate for the inaccuracy when testing the hit of plane.

Finally, I did not enhance the reward for another time; the upper bound of the prize was high enough. However, I extended the lower bound, which meant the qualified region recognized by the calculator as *hit* grew broader. This could also solve part of the problem I depicted in Fig. 6. In addition, I added a *second-step verification* to enhance the possibility of being recognized as qualified hit by testing whether the body of the plane just above it bottom is hit.

*c) Lower the prices of items in the store or slow the pace of rising prices*

In alpha version, the difficulty level increases by one every 10 bombs launched as the damage of enemy increases by 10% and the price by 25%. The value grows constantly, each time added to by a fixed number. Accordingly, each time the value rises, the player will feel the obvious change.

One of my friends suggested me to lower the growth rate and shorten the number of launched bombs required to level up. I thought for a while, and asked him about the plan that, the difficulty level incremented each 5 bombs launched as enemy damage increased by 5% and the price by 10%. He seemed delightful to hear this new plan and agreed, because 5 percent multiplies two equals 10 percent and 10 percent multiplies 2 equals 20 percent, less than 25 percent. Doubtlessly this change lowers the consumption of cash.

Not before long did I find this change interesting because it was not necessarily supposed to "save" the consumption. It simply depends on your shopping habit.

If you purchase 5 bombs each time 5 bombs are launched to keep balance, the functions of total cash you will spend, in both ways to calculate the difficulty level, are presented in (10), where *x* refers to the difficulty level.

$$f(x) = 50\left(\frac{1}{16}x^2 + \frac{7}{8}x + \frac{13}{80}(x \bmod 2)\right).$$
$$g(x) = 50\left(\frac{1}{20}x^2 + \frac{19}{20}x\right). \tag{10}$$

In this group of equations, $f(x)$ represents the original total amount of cash while $g(x)$ the changed total amount of cash. The changed way of calculating difficulty levels will not save money before level 7, while both functions equal at the level of 6. However, at level 7, you have already launched 35 bombs. The first period of combat, when your cash is the relatively the lowest and most worrying during a round of game, has already passed.

On the other hand, if you fill up bombs each time until you run out of bombs, you will purchase the bomb the first time at level 3, and the second time at level 8, when, roughly based on the information obtained in (10), the changed version saves more cash.

Both situations might be real because although the unit price of each bomb to buy fill-up is lower, it also cost more total money to do so. As I later discovered, this tricky "optimization" in the calculation mode of difficulty level actually increases the difficulty.

*2) Diversify the system*

Although it is a single program, and there have been already several interesting systems, my game still has much potential to become more playable and attractive. One way to do so is to add some new features into the game, so my friends suggested me several possible new systems included in the future update, which I will explain and elaborate my plan further in the section VIII.

## V. The Current Final Version of the Game: Several Screenshots of a Complete Round

Hera are several screenshots of a complete round of a real game to present a real combat on the screen.

Pressing [EXE] to launch the launcher to enter the program, the player will see Fig. 7. The large and bold texts on the screen clearly explain how to continue to start a new or load a game. After pressing 0 to initialize the background graph, the player might press 1 to start a new game.


Fig. 7. The GUI of launcher, the only entrance of the game.

The screen when the player first entered the game was the same as Fig. 1. In order to survive the first period, the player had to guarantee the cash is enough for him or her to purchase a HP fill-up and 5 bombs after this period. After several doubles, luckily, a considerable income was received while the base was only half-damaged.

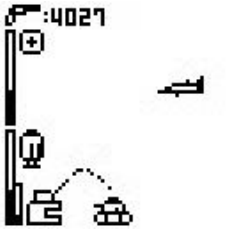


Fig. 8. A bomb is flying to a tank

It is when double system came into play. There were consecutive turns when either a plane or a tank was near my base, so hitting the enemies is easy to accumulate the multiplier. The cash increased exponentially until both enemies were too far and difficult to hit.

Much money was made by using the trick depicted in Fig. 9. This was considered acceptable to make the game easier. The screenshot shows the player earned 180 cash by hitting the plane while the highest reward of plane is 180, which meant the player hit exactly the center of the plane.

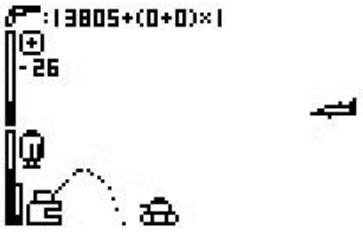


Fig. 10. An easy shot missed

It was usual when sometimes the player could hit every enemy no matter how far they were from you, and similarly, there were times when the player missed enemies even they were under your nose like shown in Fig. 10.

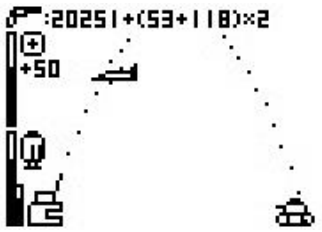


Fig. 11. An epic double kill

However, with calculated mind and experienced skill, the player hit both enemies in a row, and won an extra bonus of 50 HP. However, the multiplier was not high enough to win extra bombs.


Fig. 12. Purchasing supplements

The skill to survive through the late game (the game before it is over) is to control the consumption as low as possible while being extraordinarily sensitive to the HP bar. Once it dropped under half, the player should consider fill up or buy half of them in the store, because there is by no means to know how strongly the enemies grew and whether the player would miss all of them, which meant the base would be heavily attacked.


Fig. 9. A trick to hit the plane.


Fig. 13. A pitiful death in spite of fair evaluation

Being opportunistic has no place in this game, because sometimes it is believed the enemies were not strong enough to defeat the player completely, and in order to save money, the player might select to wait until next turn to fill up the bar. This idea was the most challenging obstacle that every player had to conquer in order to die short of cash rather than short of luck especially when the player is short of cash.

The game ended before the player ran out of money another time, again. Never underestimate your enemies.

## VI. The Feedback from My Classmates

### A. Positive impacts

When I was going to write this game, I told myself this game was meant for bring classmates closer. I hoped my game could be able to do something to pull my nerdy classmates out of excessive learning and inspired them to discover recreations around them. This game, of course, is impossible to be charming enough to affect every student in my class because some of them were so fond of studying that they would ignore anything other than study happening around them.

### B. Activate the atmosphere

After I finished my final version of calculator game, I allowed my calculator to be handed on and on among my classmates freely to let them take a look at my game and if interested, have fun. I had seen my some of my nerdy mates gathering together pointing to the screen and "instruct" who in charge of the calculator how to hit the enemies to get higher score. They were so amazed that the calculator turned out to be a game console. They would not believe entertainment could be really everywhere.

When the popularity of my game was at its peak, almost anyone who played the game would ask me any trick to get higher score between classes, some of whom I had seldom talked to before. Students also grouped together to discuss my game. I was so glad that there was finally some tie that could connect my classmates together when they were still not familiar with others (although it was the second year of our high school lives because many classmates are busy studying they spent less time having social communications), even though this game mania did not last for an entire semester. Still, I was so proud of myself.

### C. Spur the discussion on my code

When my classmates got more and more familiar with my game, they could not stop thinking about how my game worked, especially those main problems I described in section III. They grouped together and came up with ideas about the algorithms or the way that the game might work and asked me whether it was. I, on the other side, explained to anyone who was interested in those. During that time, we were also learning programming in information technology and computer science classes, so most of my classmates could provide some valuable thinking on my game. We would exchange our ideas and they might offer their useful opinions on my code even though some of them had no experience on coding before.

Some of their ideas were as good as mine. They offered another way to think about the game or how the data was processed and presented on the screen. Some of them were pretty cool, and I extracted their essence and made subtle adjustments on my code. It was also a process of learning.

Through various discussions, I felt that I learned a lot from my classmates. Their ways of thinking really inspired me to think about my algorithms again: previously I had been so proud of my "intricate" and seemingly perfect way of code but now was it *really* perfect? I believe, from then on, there were no best code and no best algorithm; there were just better ones. As a programmer, I should be aware of it and admit nothing is perfect. I should always keep moving forward.

## VII. Known Bugs & Unsolved Problems

### A. Inaccurate bomb hitting judgement

This bug is the type that I call "paradox" because I can solve the problem at the expense of slowing down the speed of the operation, and I had to work out a tradeoff.

As I mentioned before, if the bomb goes through the enemy figure without hitting its bottom, this firing is considered as futile because it misses targets. I have thought about testing the whether $y$-coordinate was between the upper and lower bound of the figure when $x$ varies from the left to the right of the figure (referred as *second test*), but it does have drawbacks because obviously it will significantly affect the speed of game. To save time, one may consider second test only if the first test (the only and present way the program does in testing the position of bomb hitting enemies) fails, but what if the second test yields higher reward? If that is possible and we have several results (not to mention that when $x$ varies we may get numerous results as well), it is another problem that the player should be rewarded the highest of them or average. And more importantly, it needs a lot of variables to store values.

To partly solve this problem, I implemented the program in line 167-174 of source code in `MAIN` (Fig. 16). I could not come up with any better idea to eliminate this bug even though it happens quite a few and it does not interfere with the progress of the game.

### B. Possible cheat by repetitive loading to get higher score

Although the security code may help in preventing from cheating, it is almost impossible to eliminating such behavior thoroughly, so is for any other game, even though this way of cheating is so prevalent among contemporary games that my game is not alone.

This bug can be simply described as: if the player can extract the data of archive file and restore it when intending to do so, he or she can also repetitively loading the game to get higher score. When this comes to computer, the archive files are always separated and can be overwritten by other archive files. When it comes to my game, the archive file – I admit it is complicated but cannot deny that nobody will try to do so – is various values stored in the letters and *list*. If the player bothers copying all of them down with security code, my encrypted archive system is useless.

If the game is designed on a computer, maybe I can test the time on machine to make the security check safer. However, the calculator does not record current time. Even so, it is still possible to cheat by rolling timeline forward or backward.

Maybe I am overstating the problem, or this problem should not be defined as a bug. But as long as this problem exists and there is possibility to cheat, it is impossible to ignore it. The discovery of this bug convinced me the fact that there are few complex programs without bugs, but as a programmer, I should try my best to prefect my program.

But beyond all of that, I realized something really important. **As a programmer or developer, if you cannot eliminate possibilities to cheat, then try best to eliminate the necessaries to cheat.**

### C. *Toggle between two screens*

The program has two screens to display. One screen is graphic screen, e.g. Fig. 8-Fig. 13. The other one is text screen, e.g. Fig. 7. In graphic screen, the user can graph and write small texts on the screen, while in text screen, the user can only write larger texts. Two screens are independent.

Here is the problem. It is easy to switch to graph screen from text screen by using any graphic command, but I tried several commands and all the possible ways and still could not switch from the graph to text screen.

The consequence of such problem is that after initializing the background necessary for the game, the screen cannot display the user interface of the launcher even though the calculator can still receive the command from the player. After initializing, the screen will be a complete blank except for a small solid square on the right top of the screen to indicate that the program is running.

It is crucial. I do not want the player to believe that the game sticks while the game is actually still running, so it is necessary to toggle the screen back to text screen to inform the player that the initialization is complete.

I finally came up with the idea to call the program in `INIT` (see the last line in Fig. 15) to return the launcher. However, this is a *temporary* and *unadvised* solution because it has the potential error to cause the calculator to raise a "nesting error." This error happens when the user uses `Prog` command to call programs for more than 10 times, and the suggested countermeasure is to avoid calling the program previously called by using `return` command to get back to the original program. [*1*] But this countermeasure does NOT work, at least for my program.

Although this is not possible that the player will initialize the program for several times and this bug is not quite likely to be triggered, it is still there. However, I will still try to solve this problem by other commands.

### D. *Unhandled dimension error*

This is another built-in error that will be arisen if the player tries to load the program when no data is in the list file. Then the calculator will display "dimension error" on the screen.

I believe this error is unavoidable and insolvable because the calculator does not provide error-catch-handle command (e.g. try-catch block in C++ or Java).

## VIII. Future Possible Update

Even though my game is still popular among my classmates, I am considering about update and diversify the game to enhance its playfulness. Here is some favorable update, either from my own ideas or my friends'.

### A. *Friendly fire*

I will place another friendly army (figure undecided) on the bottom of the screen somewhere other than the tank. If the player hits this figure, then he or she will lose some money at a certain percentage to pay for treatment.

### B. *Lucky box*

This system is to compensate for the possible lost and balance the increased difficulty from friendly fire or other update. The player should actively request this box in the store menu when the value of the cash contains same numbers and is larger than a thousand (e.g. 3 333, 55 555, 222 222). The reward contained in the lucky box includes activator (activate the friendly to force to attack the enemies), cash bonus, upgrading item, etc. This update can make the game more interesting when my nerdy mathematician classmates try to calculate the strategy of attacking or shopping in the store for a qualified cash value to win this box.

### C. *Item box*

Unlike lucky box, this kind of box is appeared randomly on the screen as supplies dropped by parachute. The reward in the box is still undecided.

### D. *Item upgrade (and/or collection system)*

Because as the game progressed, the health point of the base seems to be too low to defend the enemy, I decided to add the system of upgrading, which can increase the total amount of bombs the player can possess and the maximum value of HP.

What's more, I may probably add a system of item collection. The item can be gathered from killing enemies at a certain percentage, and can be used later as a requirement to upgrade the weapon or the base.

## Appendix 1: Full Code of Defend Your Base

The code pasted below is not necessarily the code displayed on the calculator. I have made several changes to fit the traditional display of code on the computer. Some style or format of code (e.g. indentation, colored text and comments) is changed deliberately to improve the readability of code.

*Code 1:* `'LANCHR`

```
1  //display user interface
2  Locate 1,1,"====Game Launcher===="
3  Locate 1,2,"--Defend your BASE!--"
4  Locate 5,3,"[0] INITIALIZE"
5  Locate 5,4,"[1] NEW GAME"
```

```
Locate 5,5,"[2] LOAD GAME"
Locate 2,6,"Press [DEL] to Quit"
Locate 12,7,"By Anoxic"
0→Z
//receive command from user
While Getkey≠44
    If Getkey=71
        Then -1→Z
        Prog "INIT"
        0→Z
    IfEnd
    If Getkey=72
        Then 1→Z
        Prog "MAIN"
        0→Z
    IfEnd
    If Getkey=62
        //test if security code is correct
        Then If Int ((AB+CL+MNO)(Sum List 1+Sum List 2+Sum List 3-Sum List 4+Sum List
5))+
          35=Sum List 20
            Then 1.5→Z
            Prog "MAIN"
        Else Locate 3,5,"DATA COMPROMISED"
        IfEnd
        0→Z
    IfEnd
WhileEnd
Stop
```

Fig. 14. Source code of 'LANCHR

*Code 2:* INIT

```
/*
 * the program will execute the statement after “»”
 * if the statement before it is true
 * /
Z≠-1»Stop
0→Z
//initialize graph background
Cls
Deg
GridOff
AxesOff
BG-None
ViewWindow 0,12.6,1,0,6.2
/****user data indicator****/
//cash figure
F-Line .3,6.2,1,6.2
F-Line .2,6.1,.3,6.1
F-Line .6,6.1,1,6.1
F-Line .2,6,.6,6
F-Line .1,5.9,.1,5.7
PlotOn .2,5.7
PlotOn .3,5.9
PlotOn .3,5.8
PlotOn 1.2,6
```

```
PlotOn 1.2,5.8
//data bar
F-Line .1,5.5,.1,2.9
F-Line .1,2.7,.1,.1
PlotOn .2,5.5
PlotOn .2,2.9
PlotOn .2,2.7
PlotOn .2,.1
F-Line .3,5.5,.3,2.9
F-Line .3,2.7,.3,.1
//health point figure
F-Line .6,5.5,1,5.5
F-Line 1.1,5.4,1.1,5
F-Line 1,4.9,.6,4.9
F-Line .5,5,.5,5.4
F-Line .7,5.2,.9,5.2
F-Line .8,5.3,.8,5.1
//bomb figure
F-Line .7,2.7,.9,2.7
F-Line .6,2.6,1,2.6
F-Line .5,2.5,.5,2
F-Line 1.1,2.5,1.1,2
F-Line .6,1.9,1,1.9
F-Line .8,2.3,.8,2
F-Line 1,2.3,1,2
PlotOn .7,1.8
PlotOn .9,1.8
F-Line .6,1.7,1,1.7
PlotOn .4,1.2
PlotOn .4,.1
F-Line .5,1.2,.5,.1
//draw cannon
F-Line .7,.6,.7,.1
F-Line .7,.1,1.6,.1
PlotOn 1.6,.2
F-Line 1.6,.3,1.2,.3
PlotOn 1.2,.4
F-Line 1.2,.5,1.6,.5
PlotOn 1.6,.6
F-Line 1.6,.7,.8,.7
F-Line .9,.8,.9,1
F-Line 1,1,1.3,1
F-Line 1.4,.8,1.4,.9
//store the picture into memory and clear graph
StoPict 1
ClrGraph
0→Z
//return to launcher
Prog "'LANCHR"
```

Fig. 15. Source code of `INIT`

*Code 3:* `MAIN`

```
Z≠1»Z≠1.5»Stop
Z-1→Z
AxesOff
ViewWindow 0,12.6,1,0,6.2
```

```
//if it is a new game, reset the data
If Z=0
    Then
    //reset all the user data
    40→H
    0→A
    100→B
    15→C
    1→L
    ClrList
    For 1→T To 5
    1→Dim List T
    Next
IfEnd
//if the base is healthy
While B>0
    BG-Pict 1
    Cls
    //display user data
    Text 2,15,A
    F-Line .2,2.9,.2,2.9+B÷40
    C>25»5→C
    F-Line .2,.1,.2,.1+.1C
    /****generate & draw the position of enemy****/
    //tank
    Z=0»2.5+9.4Ran#→M
    F-Line M-.4,.1,M+.4,.1
    F-Line M-.5,.2,M-.5,.3
    F-Line M-.2,.2,M-.2,.3
    F-Line M+.2,.2,M+.2,.3
    F-Line M+.5,.2,M+.5,.3
    F-Line M-.4,.4,M+.4,.4
    PlotOn M-.3,.5
    F-Line M-.5,.6,M-.3,.6
    F-Line M+.3,.5,M+.3,.6
    F-Line M-.2,.7,M+.2,.7
    F-Line M-.1,.8,M+.1,.8
    //plane
    Z=0»2.5+9.4Ran#→N
    Z=0»3+2Ran#→O
    F-Line N-.6,O,N+.6,O
    F-Line N-.4,O+.1,N-.3,O+.1
    F-Line N+.5,O+.1,N+.6,O+.1
    F-Line N-.2,O+.2,N+.6,O+.2
    F-Line N+.5,O+.3,N+.6,O+.3
    PlotOn N+.6,O+.4
    PlotOn N-.1,O-.1
    PlotOn N+.2,O+.3
    StoPict 2
    BG-Pict 2
    //reset program running status
    0→Z

    /****receive command from user****/
    //adjust angle of cannon
    F-Line 1.4,.9,1.4+.6cos H,.9+.6sin H
```

```
    While Getkey≠31
        //+ pressed, increase angle
        If Getkey=42 And H<80
            Then H+5→H
            Cls
            F-Line 1.4,.9,1.4+.6cos H,.9+.6sin H
        IfEnd
        //- pressed, decrease angle
        If Getkey=32 And H>10
            Then H-5→H
            Cls
            F-Line 1.4,.9,1.4+.6cos H,.9+.6sin H
        IfEnd
        If Getkey=48
            Then 2→Z
            Prog "STORE"
            1→Z
        IfEnd
        If Getkey=47
            Then 4→Z
            Prog "SAVE"
            0→Z
            Stop
        IfEnd
    WhileEnd
    //adjust the power of launching bombs
    1→J
    .05→I
    While Getkey=31
        If J=1
        //power increases
            Then I+.05→I
            PlotOn .4,.1+I
        Else
        //power decreases
            I-.05→I
            PlotOff .4,.1+I
        IfEnd
        //switch between increase and decrease if power reaches
        //maximum or minimum
        I=1»0→J
        I=.05»1→J
    WhileEnd
    //bomb launched, decrease the number of bomb immediately
    //to prevent cheating
    C-1→C
    //insufficient bombs warning
    If C<3
        Then F-Line .5,2.7,1.1,1.7
        F-Line 1.1,2.7,.5,1.7
            If C<0
            //automatically buy bombs when the player has no bomb
            Then If A>(50(1+.1(Sum List 2 Int÷ 5)))
                //afford to buy a bomb
                Then A-(50(1+.1(Sum List 2 Int÷ 5)))→A
                B-5→B
```

```
                0→C
                //if not affordable, lost 20 HP
            Else B-20→B
                0→C
            IfEnd
        IfEnd
    IfEnd
    Fill(Sum List 2+1,List 2)
    //track trajectory
    1.4+.6cos H→F
    .9+.6sin H→G
    Isin H→J
    //before bomb hits the ground
    While G>0
        F+Icos H→F
        G+J→G
        //gravity constant=.05
        J-.05→J
        PlotOn F,G
    WhileEnd
    //the function formula of trajectory
    //Graph Y=.9+.6sin H+(X-1.4-.6cos H)tan H-
    //.025(X-1.4-.6cos H)(X-1.4-.6cos H-Icos H)÷(Icos H)²
    //Graph Y=-.025X²÷(Icos H)²+((2.8+(1.2+I)cos H)÷(40(Icos H)²)+
    //tan H)X+.9-1.4tan H-(1.4+.6cos H)(1.4+.6cos H+Icos H)÷40(Icos H)²)
    /****game feedback****/
    //test if hitting tank
    -.025÷(Icos H)²→F
    ((2.8+(1.2+I)cos H)÷(40(Icos H)²)+tan H)→G
    .9-1.4tan H-(1.4+.6cos H)(1.4+.6cos H+Icos H)÷(40(Icos H)²)→I
    //x-coordinate where bomb hits the ground
    (-G-√G²-4FI))÷2F→T
    //calculate the score of hitting tank
    100-100Abs (T-M)→P
    Int (P+.5)→P
    //within qualified range
    If P<50
        Then 0→P
    Else Fill(Sum List 5+1,List 5)
    IfEnd
    //calculate the score of hitting plane
    //test if hitting plane
    I-O→I
    //test whether plane is on the left or right of symmetry axis
    0→T
    G²-4FI≥0»(-G+2(Intg ((-G÷2F-N)÷50)+.5)√(G²-4FI))÷2F→T
    180-180Abs (T-N)→Q
    Int (Q+.5)→Q
    If Q<100
        Then
        //test another possible hit position
        I-O-.1→I
        2(Intg ((-G÷2F-N)÷500)+.5)→S
        0→T
        G²-4FI≥0»(-G+S√G²-4FI))÷2F→T
        180-180Abs (T-N)→Q
```

```
        Int (Q+.5)→Q
        If Q<100
            Then 0→Q
        Else Fill(Sum List 4+1,List 4)
        IfEnd
    Else Fill(Sum List 4+1,List 4)
    IfEnd
    //display results
    Text 2,19+4Int log (A+.5),"+("
    Text 2,26+4Int log (A+.5),P
    Text 2,30+4Int log (A+.5)+4Int log (P+.5),"+"
    Text 2,34+4Int log (A+.5)+4Int log (P+.5),Q
    Text 2,38+4Int log (A+.5)+4Int log (P+.5)+4Int log (Q+.5),")×"
    Text 2,45+4Int log (A+.5)+4Int log (P+.5)+4Int log (Q+.5),L
    L(P+Q)→T
    A+T→A
    Fill(Sum List 1+T,List 1)
    //adjust multiplier
    If P Or Q
        Then 2L→L
        P And Q»L→L
    Else 1→L
    IfEnd
    //ammo bonus
    Int (T÷500)→T
    T>9»9→T
    T+C>25»25-C→T
    If T≠0
        Then Text 30,6,"+"
        Text 30,10,T
        C+T→C
    IfEnd
    /****enemy turn****/
    If P=0
        Then (.75+.5Ran#)(28÷3-4M÷9)(1+.05(Sum List 2 Int÷ 5))→P
    Else 0→P
    IfEnd
    If Q=0
        Then (.75+.5Ran#)(25÷3-4N÷9+O)(1+.05(Sum List 2 Int÷ 5))→Q
    Else 0→Q
    IfEnd
    If P+Q≠0
        Then Text 16,6,Int (-P-Q)
        B-P-Q→B
        Fill(Sum List 3+P+Q,List 3)
    IfEnd
    //double kill rewards
    If P+Q=0
        Then Text 16,6,"+50"
        B+50→B
        B>100»100→B
    IfEnd

    While Getkey=0
    WhileEnd
WhileEnd
```

```
229  3→Z
230  Prog "FINAL"
     1→Z
```

Fig. 16. Source code of `MAIN`

*Code 4:* `STORE`

```
  1  Z≠2»Stop
  2  0→Z
  3  BG-None
  4  Cls
  5  Text 1,1,"-=-=-=-=-=-=-STORE=-=-=-=-=-=-="
  6  Text 7,1,"YOU HAVE CASH                          "
  7  Text 7,122-4Int log (A+.5),A
  8  Text 13,1,"------------------------------"
  9  Text 19,40,"FILLUP"
 10  Text 19,70,"BUY 5"
 11  Text 19,100,"BUY 1"
 12  Text 25,1,"HP"
 13  Text 25,40,Int (400(1+.1(Sum List 2 Int÷ 5)))
 14  Text 25,70,Int (240(1+.1(Sum List 2 Int÷ 5)))
 15  Text 25,100,Int (50(1+.1(Sum List 2 Int÷ 5)))
 16  Text 31,1,"BOMB"
 17  Text 31,40,Int (2000(1+.1(Sum List 2 Int÷ 5)))
 18  Text 31,70,Int (450(1+.1(Sum List 2 Int÷ 5)))
 19  Text 31,100,Int (50(1+.1(Sum List 2 Int÷ 5)))
 20  Text 49,1,"[MESSAGE]"
 21  Graph Y=.8
 22  Text 57,1,"HP           "
 23  Text 57,18-4Int log (Int B+.5),Int B
 24  Text 57,22,"/"
 25  Text 57,26,100
 26  Text 57,40,"BOMB         "
 27  Text 57,63-4Int log (C+.5),C
 28  Text 57,67,"/"
 29  Text 57,71,25
 30  Text 57,81,"LEVEL"
 31  Text 57,104,Sum List 2 Int÷ 5
 32  0→T
 33  While Getkey≠47
 34      //left pressed
 35      If Getkey=38
 36          Then Text 49,40,"                                  "
 37          Text 25+6(T Int÷ 3),35+30MOD(T,3)," "
 38          T-1→T
 39          T<0»5→T
 40      IfEnd
 41      //right pressed
 42      If Getkey=27
 43          Then Text 49,40,"                                   "
 44          Text 25+6(T Int÷ 3),35+30MOD(T,3)," "
 45          T+1→T
 46          T>5»0→T
 47      IfEnd
 48      //up pressed
 49      If Getkey=28
 50          Then Text 49,40,"                                  "
```

```
        Text 25+6(T Int÷ 3),35+30MOD(T,3)," "
        T-3→T
        T<0»T+6→T
    IfEnd
    //down pressed
    If Getkey=37
        Then Text 49,40,"                                    "
        Text 25+6(T Int÷ 3),35+30MOD(T,3)," "
        T+3→T
        T>5»T-6→T
    IfEnd
    Text 25+6(T Int÷ 3),35+30MOD(T,3),">"
    If Getkey=31
        Then
        //prevent key stuck
        While Getkey
        WhileEnd
        //case T
        If T=0
            Then If A≥Int (400(1+.1(Sum List 2 Int÷ 5)))
                Then If B≠100
                    Then 0→J
                    A-Int (400(1+.1(Sum List 2 Int÷ 5)))→A
                    100→B
                Else 1→J
                IfEnd
            Else 2→J
            IfEnd
        Else If T=1
            Then If A≥Int (240(1+.1(Sum List 2 Int÷ 5)))
                Then If B≠100
                    Then 0→J
                    A-Int (240(1+.1(Sum List 2 Int÷ 5)))→A
                    B+50→B
                Else 1→J
                IfEnd
            Else 2→J
            IfEnd
        Else If T=2
            Then If A≥Int (50(1+.1(Sum List 2 Int÷ 5)))
                Then If B≠100
                    Then 0→J
                    A-Int (50(1+.1(Sum List 2 Int÷ 5)))→A
                    B+10→B
                Else 1→J
                IfEnd
            Else 2→J
            IfEnd
        Else If T=3
            Then If A≥Int (2000(1+.1(Sum List 2 Int÷ 5)))
                Then If C≠25
                    Then 0→J
                    A-Int (2000(1+.1(Sum List 2 Int÷ 5)))→A
                    25→C
                Else 1→J
                IfEnd
```

```
            Else 2→J
            IfEnd
        Else If T=4
            Then If A≥Int (450(1+.1(Sum List 2 Int÷ 5)))
                Then If C≠25
                    Then 0→J
                    A-Int (450(1+.1(Sum List 2 Int÷ 5)))→A
                    C+5→C
                Else 1→J
                IfEnd
            Else 2→J
            IfEnd
        Else If T=5
            Then If A≥Int (50(1+.1(Sum List 2 Int÷ 5)))
                Then If C≠25
                    Then 0→J
                    A-Int (50(1+.1(Sum List 2 Int÷ 5)))→A
                    C+1→C
                Else 1→J
                IfEnd
            Else 2→J
        IfEnd
        IfEnd
        IfEnd
        IfEnd
        IfEnd
        IfEnd
        IfEnd
        //transaction massage
        If J=0
            Then Text 49,40,"TRANSACTION SUCCEEDS   "
            //refresh user data
            Text 7,1,"YOU HAVE CASH                      "
            Text 7,122-4Int log (A+.5),A
            Text 57,1,"HP          "
            Text 57,18-4Int log (Int B+.5),Int B
            Text 57,22,"/"
            Text 57,26,100
            Text 57,40,"BOMB        "
            Text 57,63-4Int log (C+.5),C
            Text 57,67,"/"
            Text 57,71,25
            Text 57,81,"LEVEL"
            Text 57,104,Sum List 2 Int÷ 5
        Else
            J=1»Text 49,40,"ITEM FULL                 "
            J=2»Text 49,40,"INSUFFICIENT CASHES    "
            J=3»Text 49,40,"INSUFFICIENT ITEMS      "
        IfEnd
    IfEnd
WhileEnd
While Getkey
WhileEnd
//return and display combat screen
BG-Pict 1
Cls
```

```
Text 2,15,A
F-Line .2,2.9,.2,2.9+B÷40
C>25»25→C
F-Line .2,.1,.2,.1+.1C
//display tank
F-Line M-.4,.1,M+.4,.1
F-Line M-.5,.2,M-.5,.3
F-Line M-.2,.2,M-.2,.3
F-Line M+.2,.2,M+.2,.3
F-Line M+.5,.2,M+.5,.3
F-Line M-.4,.4,M+.4,.4
PlotOn M-.3,.5
F-Line M-.5,.6,M-.3,.6
F-Line M+.3,.5,M+.3,.6
F-Line M-.2,.7,M+.2,.7
F-Line M-.1,.8,M+.1,.8
//display plane
F-Line N-.6,O,N+.6,O
F-Line N-.4,O+.1,N-.3,O+.1
F-Line N+.5,O+.1,N+.6,O+.1
F-Line N-.2,O+.2,N+.6,O+.2
F-Line N+.5,O+.3,N+.6,O+.3
PlotOn N+.6,O+.4
PlotOn N-.1,O-.1
PlotOn N+.2,O+.3
StoPict 2
BG-Pict 2
F-Line 1.4,.9,1.4+.6cos H,.9+.6sin H
```

Fig. 17. Source code of STORE

*Code 5:* SAVE

```
Z≠4»Stop
0→Z
1→Dim List 20
Fill(Int ((AB+CL+MNO)(Sum List 1+Sum List 2+Sum List 3-Sum List 4+Sum List 5))+35,List
20)
ClrText
Locate 1,1,"=====Game  Saved====="
Locate 1,3,"Do NOT Modify Values"
Locate 4,4,"or Clear Lists"
Locate 6,5,"Press [EXE]"
Locate 1,7,"====================="
While Getkey≠31
WhileEnd
Stop
```

Fig. 18. Source code of SAVE

*Code 6:* FINAL

```
Z≠3»Stop
0→Z
BG-None
Cls
Text 1,1,"-=-=-=-=-=-GAME OVER-=-=-=-=-=-"
Text 7,1,"---------FINAL REPORT-----------"
Text 13,1,"EARN CASH"
Text 13,122-4Int log (Sum List 1+.5),Sum List 1
```

```
Text 19,1,"LOST HP"
Text 19,60,Int Sum List 3
Text 19,81,"×5  ="
Text 19,122-4Int log (5Sum List 3+.5),5Int Sum List 3
Text 25,1,"KILL PLANES"
Text 25,60,Sum List 4
Text 25,81,"×200="
Text 25,122-4Int log (200Sum List 4+.5),200Sum List 4
Text 31,1,"KILL TANKS"
Text 31,60,Sum List 5
Text 31,81,"×100="
Text 31,122-4Int log (100Sum List 5+.5),100Sum List 5
Text 49,1,"LEVEL"
Text 49,60,Sum List 2 Int÷ 5
Text 49,69,"×.1 + 1="
Text 49,118-4Int log (1+.1(Sum List 2 Int÷ 5+.5))-8Int (.1MOD(10(1+.1(Sum List 2 Int÷
5)),10)+.9),"×"
Text 49,122-4Int log (1+.1(Sum List 2 Int÷ 5+.5))-8Int (.1MOD(10(1+.1(Sum List 2 Int÷
5)),10)+.9),1+.1(Sum List 2 Int÷ 5)
Graph Y=.8
Text 57,1,"FINAL SCORES"
(1+.1(Sum List 2 Int÷ 5))(Sum List 1+5Int Sum List 3+200Sum List 4+100Sum List 5)→T
Text 57,122-4Int log (T+.5),Int T
ClrList
While Getkey=0
WhileEnd
ClrText
Stop
```

Fig. 19. Source code of `FINAL`

## APPENDIX 2: VARIABLE LIST

TABLE I. LETTER VARIABLE EXPLANATION

| Letter | Refers to |
|---|---|
| A | Cash currently owned by the player |
| B | Health point (HP) of the base |
| C | Possessed bomb |
| F | x-coordinate of launched bomb |
| G | y-coordinate of launched bomb |
| H | Angle between cannon barrel and horizontal line |
| I | Power of launching bombs |
| L | Multiplier of double system |
| M | *x*-coordinate of tank |
| N | x-coordinate of plane |
| O | y-coordinate of plane (height) |
| P | Score of hitting / damage from tank |
| Q | Score of hitting / damage from plane |

TABLE II. ENUMERATION TYPE

| Letter | Explanation | Value | Refers to |
|---|---|---|---|
| Z | Running program indicator | 0 | Program terminated / running launcher |
| | | 1 | MAIN running as a new game |
| | | 1.5 | MAIN running as a loaded game |
| | | 2 | STORE running |
| | | 3 | FINAL running |
| | | 4 | SAVE running |
| | | -1 | INIT running |
| J | Transaction information | 0 | Success in transaction |
| | | 1 | Fail: Item full |
| | | 2 | Fail: Insufficient cashes |
| | | 3 | Fail: Insufficient items |
| *list* | Store evaluation information in final report | 1 | Cash earned |
| | | 2 | Bombs launched (divided by 5 equals difficulty level) |
| | | 3 | HP lost |
| | | 4 | Planes killed |
| | | 5 | Tanks killed |

## APPENDIX 3: ANTIDOTES AND REFLECTION

It is my own reflection on how difficulty level works and how it reflects the player's psychology as the consequence of preferring to take petty advantages.

Almost every player, when reading the game rule and noticing the double system, will doubt the fairness or balance of the game. They will wonder whether the double system will compromise this game. However, the reality is definitely no.

### *A. Why the player fails to use up all the money before they die even though wealthy*

When a player is hitting the enemies, consecutively in a row without missing, the multiplier increases exponentially and the bombs must be filled up. Just in several turns, a poor guy turns into a millionaire. Now the player does not have to worry about the scarcity of bombs and HP for a while, and all he or she has to do is to boost the multiplier as high as possible. Even interrupted, the combo is still high enough to support the survival. Maybe the player only needs to focus on health point and remember to fill up it when necessary. But the game is still over before the player can use up all of the money.

This usually happens when the difficulty level is high enough to increase the damage of enemies to a level that, if not carefully enough noticed, will cause damage several times higher than that of enemies at the beginning. Thirty health points, for example, may be enough for surviving two turns even though in both turns you miss all the targets, but absolutely insufficient to bear the damage at the difficulty of twenty, when a hundred bombs have already been launched. In addition, it does not take much time to launch a hundred bombs, generally speaking, when the average time of a turn is five seconds, and this time is short enough to make the players ignore the effect of difficulty level. When the players only need to care about the condition of base because the amount of bombs are all "charged" up during last consecutive hit, it is quite possible for them to forget the enemies are growing stronger at the same time even with the help of difficulty level displayed in the store. After all, the difficulty level shown in store just offers a rough image of the power of enemy because the damage varies greatly to the distance the plane or tank from the base and players have no idea what the values of basic damage are. Since players only focus on aiming at targets before to earn cash, they did not know to which extent the enemies are becoming stronger, and by the time they came aware of it, it has been too late.

It is ironic, but it is not hard to explain. People tend to focus on what is urgent rather than on what is important. When they have finally solved the urgent problems, the important ones emerge and become another group of urgent problems. When they solve this group of problems, another group occurs and so on. This is a vicious cycle, but people can prevent this by solving both types of problems at the same time while it cost far more less energy and time than that when it is too late to do so. Mapped in this game, it is urgent to earn the cash, but it is important to notice what happened on enemy and the condition of the base. Worse, players will underestimate their enemies when they became rich. And there is another lesson to learn: never underestimate your enemies.

### *B. Opportunists have no place in this game*

Even though many players, fortunately, gradually grow to be aware of the problem I mentioned in previous section, they may still fail with plenty cash. This time, however, it is not caused from their carelessness, but from their ideas of being opportunistic.

In the store, the player can enjoy a discount of 20% when buying a fill-up, but as the game progresses and the difficulty level becomes higher, this discount becomes less economical when the damage rises. The more economical way becomes to buy 50 health points. However, many "wealthy" and generous players will still prefer to buy fill-up to save time.

When the health point drop below 50, players will notice it and consider repairing the base at the best time to save cash. By *best time*, they mean to buy a fill-up when the health point is low but not low enough to lose the game – the price of buying a fill-up does not depend on your current health point, so the lower the HP, the more economical. But doing so risks losing the game unexpectedly because you really have no idea how much damage the enemies can cause. Besides, there will be times when targets are so close to you that you suppose it will be an easy shot, so it is unnecessary to repair the base immediately, but you miss it. The damage can be much higher when the enemies are too close to your base.

When opportunists are playing this game, they will think about to save as much cash as possible to survive longer and buy the items at a price as low as possible, but they cannot guarantee their judgment is reasonable. If the opposite happens, then poof! Game over.

### *C. Risk homeostasis*

Is it really better when you have a lot of cash and thus you need not to worry about that? Not necessarily. This is the comprehensive understanding of my game when I discovered what I described in two sections above. When possessing countless cash, players may be less alert as they were before when they were poor. Before, they pay a great amount of attention to every number on the screen, and think twice before making decisions on purchasing any supply. Every step they take was careful and they know for sure that a bad decision will cause something irrevocable. However, if they are fortunate enough to win a large number of cash through double system, they will not care what they cared anymore because cash means survival. Cash brings a sense of fake safety that "I will not die as long as I have sufficient money."

This is called risk homeostasis. Previously their risk is not having enough money and now it is having too much money, because what should have meant to be a safety that prevents them losing the game now turns to be something causing their treating the enemies lightly. This is also the reason why the antilock braking system (ABS) was useless in controlling the number of accident in Germany [2].

[1] L. Casio Computer Co. fx-9750GII - Calculators - Manuals - CASIO. [Online]. http://support.casio.com/en/manual/manualfile.php?cid=004009083

[2] M. Gladwell, "Blowup," in *What the Dog Saw: And Other Adventures*. Little, Brown and Company, 2009, pp. 355-356.