TI-Calculator Game Design

By Blast Programs, Inc.

Chapter One: An Introduction

Introduction

Your Texas Instruments graphing calculator is a very powerful device, with two built-in programming languages that it can understand. The first is z80 assembly, a low-level language that provides direct command line access to your calculator’s processor. This is very dangerous if you are unfamiliar with what you are doing. Oftentimes, the calculator will simply crash, without throwing an error of any kind. If you make a mistake, you can damage your device’s memory beyond repair.

There is a safer alternative to that. TI-Basic is a high-level language. It is an interpreted language. This means that the processor reads each line before it executing the command. If you make a mistake here, the calculator will return an error. Also, one command in TI-Basic is the equivalent of several lines of assembly code, thus making it a lot easier to use. TI-Basic is the only language to be discussed in detail in this tutorial.

For future clarity, I will now define several terms involved in the TI-Basic programming language. The first is command. A command is a word or symbol that directs a device to perform a certain task. An argument, otherwise known as a specification, is a series of one or more values or inputs that limit the execution of a command. Some commands are comprised of arguments.

Memory Structure of the TI Calculator

The TI calculator has two main sectors, the RAM and the Archive. The RAM is a portion of the memory that is available to programs for design and execution. As such, it is also most prone to crash. A crash is defined as a reset of memory data, due to some software malfunction. The RAM is relatively unorganized and unprotected. It is also the only part of the calculator’s memory that can truly be tinkered with. RAM stands for Random Access Memory.

A fixed part of the calculator’s RAM cannot be edited, used or modified (it is actually possible using assembly, but this is a very dangerous task). It is called System RAM. This is where the calculator’s operating system places important information about the calculator, its drivers, registers, ect. Also, there is another section of RAM that corresponds to the hardware of the calculator, that may be edited, but only through assembly language routines that call specific pieces of hardware. The rest of the RAM is free to use. Your calculator has about 32k bytes of total RAM, but of that, only 24k bytes are free for your average Basic programmer to use.

The structure of the Archive is much different. First off, the Archive is secured against crash because it is locked down to prevent user interference. The Archive is structured in blocks, into which Archived variables are placed. When a variable is placed into Archive memory, it must occupy one entire block of memory. So, let’s say, for instance, that a calculator user wants to install an application that uses one and a half blocks. The next application you install will start from the next full block, not the top half of the last. Thus, space may be wasted. It is not possible to write to or from the Archive memory using TI-Basic, though some assembly utilities install hooks that allow you to do so.

The Uses of TI-Basic

TI-Basic is the first language that a developing calculator programmer learns. First off, let me tell you that that because of hardware compatibility issues, the actual language is different between some calculator models. For instance, a program designed for use on a TI-83 Plus will not work on a TI-83 or lower. This game design tutorial will only cover the language that can be used on one of the following calculators:

-TI-83 Plus
-TI-83 Plus Silver Edition
-TI-84 Plus
-TI-84 Plus Silver Edition

If you are coming here to learn to code in TI-Basic, you are in the wrong spot. This tutorial is designed, for those already familiar with TI-Basic, as a crash course in game design. This tutorial will delve, first, into the mechanics of a game, ways to accomplish those mechanics in TI-Basic, and lastly, the game debugging process. Beginners in TI-Basic will have a migraine by the first page of Chapter 2.

Chapter Two: Mechanics of a Game

What is a Game?

I'm sure that you have at some point played a video game on some gaming system. Thus, you know what a game is. If you haven't then, a game is a program that provides challenge to a user, who performs some series of actions to accomplish a set goal. Games generally accept vast amounts of user input, from simple data entry to key presses. This tutorial will describe all of that.

Some games that we know of are played between two players. These are called multiplayer games. They can be further subdivided into two classes. There are multiplayer games where both players share one machine or they can be played on two separate machines, connected by some sort of linking device. The later method is substantially more complex, as it requires the two devices to be synced with each other.

Then, there are the single player games. In this class of games, a player battles an artificial intelligence generated by coding sequences or algorithms. These games can be further subdivided into two classes. Some of these games have an artificial intelligence that is randomized. The actions of the machine are completely unrelated to what the user does. In the second class of single player games, the artificial intelligence is programmed to respond to specific actions a user takes and the truth of various conditions increases the likelihood that the machine will take a certain action. The best and most challenging type of artificial intelligence (and by far the hardest to code) is one in which the machine constantly gathers information about how you play and adjusts its tactics accordingly.

Features of Games

There are several things that distinguish a game from any other program type, and some of them make game programming one of the more difficult types of programming. First off, more advanced games require that the program be able to respond to a key press, not just to data entry. This is part of the data interface. An interface is, simply, a communication. The communication between two machines in multiplayer mode is one form of interfacing. The communication between user and machine is another.

Saving progress is another feature of a good game. What good would a longer game be if you had to start over every time you relaunched the program? It would be much better if you could somehow revisit the place you left off.

Saving is not a hard concept to understand, but keeping it organized is another story. You can record game data in variables and recall them later. But, using fifteen variables can get confusing and can conflict with other programs. Luckily, most programming languages use arrays; single variables that can be created and can store more than one piece of data. Arrays will be discussed in a subsequent chapter. When you relaunch a program, the variable supplies information to the program, allowing it to "remember" the last point at which you saved. The next time you save, the old data is overwritten and the new data is saved.

Artificial intelligence **or** a live opponent is a requirement of all games, as you saw in the previous section. Otherwise, you don't have a game. A game requires that you have a set goal to achieve and several ways of accomplishing it.

Another quality of a game is the ability to perform different tasks in different levels or parts of the game. Unless you have been involved in programming, you probably don't even think about these phenomena: A certain weapon cannot be used until level five. You cannot use a flamethrower in water. An explosive detonates near gunpowder and the explosion is several times larger than normal. This involves the use of conditionals. Conditionals test for the truth of a statement. We will explore use of conditionals in game design later in this tutorial.

Game Coding

What makes a game do what a game does? The answer to that question lies in the programming of the game. As you probably already know, a program is a series of cryptic instructions given to a processor, resulting in a certain output. Game programs often become large and complicated. They occupy large amounts of memory, especially on devices that don't have much memory to begin with. There is a process called optimization, in which a program's code is edited to accomplish the same task, but use less memory. In the following chapters, we will discuss how to create games of all styles.

Chapter 3: The Game Design Process

Planning

The most important part of designing a game is in the planning process. Often, how well you plan you game determines how long it takes to finish, how large it is, how organized it is, and how much of it you will need to fix later. In planning a game, the first thing you should do is write down what you want to accomplish. For simple games, this can be one thing, and for more complex games, this can be many things. The more goals you want to include in a game, the larger the game will be.

The next thing you should do is think about what data you need to collect and choose variables to use. Remember, once you use a variable for one thing, it must not be used for anything else. You quickly learn that the standard system variables are very inefficient to use, and never to use *x* or *y* in games that use the graph screen. Arrays become your best bet, as you can create them within the program and give them a unique name, to ensure that no other program or function tampers with them.

Then, begin to format your program. Lay out the conditionals, such that you know where they all are and where they end, because, trust me, this gets annoying. Once, you know where everything is, you are ready to move on.

Coding

The next step in the design of a game is to actually code the game. This is the lengthiest part of the process, challenged in length only by the debugging process. In this process, you must write all conditionals, statements, and variable usage.

Sometimes, it can get hard to write one large program. When you need to reference one part of the code, longer programs require seemingly endless scrolling through lines worth of stuff that gets confusing very quickly. To make the programming easier, a game can actually be constructed as several smaller programs, and consolidated later.

Then, it is crucial to know what coding does what. The uses of TI-Basic in game design, the part we've all been waiting for, are in the next chapter. The next chapter will not discuss all portions of TI-Basic, only those commands that are relevant to game design.

Debugging

The final, and perhaps the most annoying, part of the game design process is debugging. In this process, any errors are removed from the program and it is optimized to run faster, and take up less memory. Not all programmers will find this easy. I often find myself taking longer to debug a game than I did to design it. While some errors are easy to find, because the calculator throws an error and points to its location, some errors are less visible. This is because there is nothing actually wrong with the code. Instead, there is something wrong with the handling of the variables or the placement of commands that are altering the results, without throwing an error. For example, a program using the variables *X* and/or *Y* and the graph will not work properly because the drawing commands modify those variables. It's issues like these that frustrate programmers.

Chapter 4: TI-Basic in Game Design

Laying the Frameworks

Usually, a program must have an underlying format, regardless of how complicated it may seem. Once the format is deciphered, the program suddenly becomes more comprehensible. It is up to you, when you are designing a game, to choose your format and stick to it. There are two options for your program core. By program core, I mean the code without any of the complicated commands or graphics

The Goto-Label Format

Some programs are constructed using the Goto-Label format. This construction makes a program simple to interpret, but very jumpy. It places the core of the program at the beginning, with any subroutines placed after it, usually in the order in which they are defined in the program. For some programs, this is the easiest method to use, and it is, by far, the easiest to code. But, for other, more complicated programs, it is highly impractical to use. As you move into more advanced games, the next method is the best to use.

In the Goto-Label format, a programmer assigns a label to a conditional. You can see an example below. If the conditional is true, the program moves to the Label. If the conditional is false, the program moves to another Label, or to another conditional.

The simplest game to make for the TI calculator is the "Guessing Game". Below is a variation of it, using the Goto-Label format. Because this program is common knowledge to programmers, I will not cite a source for it. Any programs listed in the Appendixes will be sourced appropriately.

The Guessing Game
prgmGUESS

```
ClrHome
randInt(0,100)->A
Lbl 1
Input "GUESS",B
If A<B:Goto 2
If A>B:Goto 3
If A=B:Goto 4
Lbl 2
```

```
Disp “TOO HIGH”
Pause
ClrHome
Goto 1
Lbl 3
Disp “TOO LOW”
Pause
ClrHome
Goto 1
Lbl 4
Disp “YOU GOT IT”
Pause
ClrHome
Return
```

Here you can clearly see the jumps from Goto to Label. Also, you should have gotten a good idea of how the commands work. *Goto* sends program execution to *Label* specified. Thus, *Goto 1* sends the program to *Label 1*, *Goto 2* sends the program to *Label 2*, and so on. Valid label specifications may be from 1 to 99 and from A to ZZ.

<u>The Conditional-Command format</u>

This format of programming involves following a conditional immediately with the code to be executed if the conditional is true. This saves the trouble of hunting for the specified label, as you do in the above program, but it makes the program so much more complicated. Below, you can see an example of the exact same Guessing Game, using the Conditional-Command format.

The Guessing Game
prgmGUESS

```
ClrHome
randInt(0,100)->A
While A≠B
Input “GUESS”,B
If A<B
Then
Disp “TOO HIGH”
Pause
ClrHome
End
If A>B
Then
Disp “TOO LOW”
```

```
Pause
ClrHome
End
End
Disp "YOU GOT IT"
Pause
ClrHome
Return
```

As you can see, this format has all of the subroutines inside of its code. Also, it uses a while loop to take the place of the Label 1, which, in the first program, serves the same purpose.

Interfacing

An "interface" is a system that causes some sort of communication between a user and a machine. Interfacing can range from being very easy to accomplish, in the case of simply asking the user for a number, to very difficult to accomplish, in the case of tracking key presses. A good gaming interface is timed so that the program doesn't somehow "miss" a key press.

There are several commands that can be used to create a rock-solid gaming interface. These commands are *Input*, *Prompt*, and *getKey*. The first two are used to receive and store data that the user inputs. The last is used to monitor key presses. The next section will discuss data storage techniques. This section will discuss the above three commands.

Input allows you to ask a user for data, with the prompt accompanied by a single-line string of text that explains to the user what to input. This is limited, as it is not easy to explain complicated things to a user within one line. I'll explain a way to work around this later.

Prompt functions in a similar manner to *Input*, except that *Prompt* will only display the name of the variable being asked for. This makes it even less reliable than *Input*, but it is possible to work around this as well.

GetKey is the final interfacing command. It is the best gaming command in the entire programming language. *GetKey*, simply put, returns the numerical value of the last key pressed. The value of each key on the calculator is shown in the chart below. For games in which a player needs to attack, defend, or perform multiple tasks within a short period of time, *getKey* quickly becomes one of the most frequently used commands within the program.

Key	**getKey value**
Y=	11
Window	12
Zoom	13
Trace	14
Graph	15
2nd	21
Mode	22
Del	23
Left Arrow	24
Up Arrow	24
Right Arrow	25
Alpha	31
X,T,theta,n	32
Stat	33
Down Arrow	34
Math	41
Apps	42
Prgm	43
Vars	44
Clear	45
x^{-1}	51
Sin	52
Cos	53
Tan	54
^	55
X^2	61
, (comma)	62
(	63
)	64
/ (division sign)	65
Log	71
7	72
8	73
9	74
X (multiplication sign)	75
Ln	81
4	82
5	83
6	84
- (subtraction sign)	85
Sto->	91
1	92

2	93
3	94
+ (addition sign)	95
On	---Interrupt key: UnMapped---
0	102
. (decimal)	103
(-) (negative sign)	104
Enter	105

Now, let's discuss ways to work around limitations with input and prompt. Let me remind you that the *Disp* command writes a line of text to whatever line the cursor is on currently, and then moves the cursor to the next line. Let me also remind you that the *Input* command places its one-line explanation on whatever line the cursor is on. Therefore, by using the *Disp* command, you can write out, say, the first three lines of a four line prompt, and then make the last line the text in the *Input* command. Or, you can forgo the one-line explanation altogether and use only *Disp* for the prompt. Both examples are shown below and will produce a similar result when run.

Example 1

```
Disp "Please input"
Disp "Your X and Y"
Input "Coordinates:",Str0
```

Example 2

```
Disp "Please input"
Disp "Your X and Y"
Disp "Coordinates:"
Input "",Str0
```

Saving Data

Anyone who has played a game before is familiar with another one of its features—the ability to store data and recall it later. This data is not lost when the program quits, and can be read by the program at a later time to recall some prior operating state. In games, the most common things to store are lives, health, level, and equipment. This section will detail, not only how to receive and interpret data and how to save it to a secure location, but also what variable types on the calculator may be used to retain the data.

Let's first discuss places where the data can be saved. There are two basic types of variables (locations in memory) that can be used, temporary variables and long-term variables, distinguished by their volatility when used to retain data over a long period of time.

The real variables are the most easily accessible and the most used places to hold data. This makes them bad for long-term storage, as any other program that is run is likely to alter its data. The real variables are great for holding data temporarily while a program performs some execution, but should not be relied upon outside the program. We need a more stable storage location.

The system variables are the next option. While they are not used as often as real variables, they are more unstable. For example, any data placed in the variable 'Xmin' will be altered any time the graph screen's range is changed.

GDB's and Pics hold data about the graph settings or any drawings on the graph screen, respectively. But, for numerical data, this still does not help.

Lists present a worthwhile solution and have a very low volatility. A list can be created by a program and given a five-character name to help identify it. The program can then destroy lists once they are no longer needed, or kept in memory if data needs to be retained. Plus, having a unique name ensures that the list will not be altered by another program or function. Lists can be as large as memory permits. They may also be moved to and from Archive in order to protect program data.

Collision Detect

Collision detect is the process of causing a program to 'know' when some sprite you are controlling has made contact with another sprite, or a border. Collision detect is used in games where you are moving some object around the screen. The classic Snake game is an example of this. In this game, there are two types of possible collisions—collisions with food, which cause your snake to grow longer and another food piece to appear elsewhere on the screen, and collisions with walls, which cause you to lose a life. This collision detect is done by using the *Pxl-test* command, which checks to see if the specified pixel is on. As the Snake moves forward, the *Pxl-test* command checks the square the Snake is about to move onto to see if it is already on. If it is, the program enters a conditional event script, that first checks to see if the square ahead matches the coordinates of food. If it does, it carries out the instructions associated with lengthening the Snake and redrawing a new food particle, and increasing your score. Then, it checks to see if you have sufficient points to advance to the next level. If there is no food ahead, the program interprets the square ahead as a wall and runs the kill routine that resets your level score, takes away one life, and restarts the level. Technical details about using collision detect will be discussed in the next section about graphics.

Chapter 5: Creating a Graphical Interface

Sprites

If you are unfamiliar with other programming languages, or any sort of game making software, you are probably asking what importance a type of soda has to graphics. But, I assure you that there is more than one type of sprite. The sprite with which we are concerned is an image, of varying size, which a program can call to and display. Games usually consist of multiple sprites, one for each major item.

Ti-Basic is very limited in its use of sprites. Drawing sprites that require more than a few characters need assembly libraries, routines, or applications installed in order to work properly. Those types of sprites will not be covered here. This tutorial will only address those very basic sprites that TI-Basic natively supports. Before we progress any further, let me first let you know that using sprites in any fashion, but especially in TI-Basic, requires large amounts of memory, anywhere between 150 and 800 kilobytes of free RAM per sprite. Even on the TI-84 Plus Silver Edition, there are only 24,000 kilobytes of RAM available in total.

The simplest way to draw a sprite is to draw it on the buffer (graph screen), then store it to a picture, using the *StorePic* command. But, this creates a problem, as the designer must makes sure he supplies the picture along with the program, or else the calculator will throw an error. If the program requires several sprites, it becomes difficult to manage all the pictures. On top of this, there may not be enough free RAM, once all the sprites are extracted, to run the program. It is also possible to make the program draw the sprites out and save them to picture files when it starts, then delete them at the end. That removes the problem of grouping all the picture files and the errors caused if you missed one. But, if you want to conserve memory, this also is a bad idea.

The fact of the matter is that there really is no way around this fact: drawing sprites is very demanding on memory. But, there are good methods, and one of them is used in the Snake game I have previously mentioned. In this game, the data about where the walls will be is placed in a list, named L1. The data is entered in pairs of two, with an x-coordinate first, followed by a y-coordinate. Once this is done, the drawing routines begin. The routines move through the list, at increments of two. It will store the first in the pair to X, and the next to Y, and then turn on the point (X,Y). Then, it will jump to the next pair. This process will repeat until the list is finished.

A similar system can be used to display any kind of sprite, whether by line, by pixel, or by point. Once the image is displayed, the list used can be destroyed.

A string can also hold a sprite. The following string displays, simply, a border of X’s:

“XXXXXXXXXXXXXXXXX00000000000000XX00000000000000XX0000000000
000000XX000000000000000XX000000000000000XX000000000000000XXXXX
XXXXXXXXXXXX”

*0’s represent empty spaces.

It would, of course, need to be read by a drawing routine that would read it in increments of 16, and output the result to the screen:

```
XXXXXXXXXXXXXXXX
X              X
X              X
X              X
X              X
X              X
X              X
XXXXXXXXXXXXXXXX
```

Using this method, objects can be placed almost anywhere on the screen. But, how to we get our game to distinguish different objects and to respond accordingly? The next section details that.

Events and Event Scripts

An event, in game design, is an instance that is triggered when some series of conditions is met. An event will only occur when all of those conditions are true. Then, a series of commands is carried out. These things together make up an event script. It consists of one or more conditional statements, called the trigger, and a series of commands to carry out if the trigger is achieved. There is one event script per object that “does something” in the game.

Event scripts are the juice that makes any great game run. Without them, you cannot make truly interesting and engaging games. The construction of an event script requires a profound understanding of what you want the object to do.

Every event must have a trigger. There are three types of triggers: a key press, a collision, or both. A key press trigger, as you may have guessed, requires a key press. A collision trigger requires a collision detection to return positive. Some triggers require both a key press and a collision in order to execute. These require more advanced coding. While some people may be more comfortable with nesting conditionals (placing one conditional inside another), there is a much more efficient method of testing the truth of multiple conditionals in a single If-Then statement. Suppose you wanted to have the value of A increase by three if A is negative or decrease by 3 if A is positive, you can write this:

```
If A<0
A+3→A
If A>0
A-3→A
```

But, the next single line does exactly the same thing:

```
A+3(A<0)-3(A>0)→A
```

Here's why…

If A is less than 0, A cannot be greater than 0. Thus, here is what the calculator does:

```
A+3(true=1)-3(false=0)→A
A+3(1)-3(0)→A
A+3-0→A
A+3→A
```

Using the same logic, you can see why A>0 being true would subtract three.

Collision Detect

Apart from sprites and events, collision is one of the most crucial elements of game design. In most challenging games, especially those with hits and attacking and defending, the game would not be the same without collision. As expected, collision is one of the more difficult concepts (as if the rest of this tutorial isn't). I will be discussing the two easiest methods of achieving a collision detect.

The first method involves the variables assigned to represent the positions of the two sprites between which we are testing for a collision. In this method, you compare the x value of Sprite A to the x value of Sprite B, then compare the y values in much the same manner, through the use of conditional statements. If both the X and Y values coincide, there has been a collision. Provided that Sprite A is located at (A,B) and Sprite B is located at (C,D), the following conditional statement is perfect: *If A=C and B=D*. You may place parentheses around the A=C and B=D, but they are not required and waste memory.

The second method involves the use of the *Pxl-test* command. In this command, you are required to specify what pixel(s) you want to test. The *Pxl-test* command returns 1 if the pixel is on and 0 if it is off. Assuming that Sprite B is a stationary sprite, and Sprite A is in motion, you can set *Pxl-test* to check the pixels ahead of the moving sprite.

It is important to know which method to use in which type of situation. While both methods can be used in exactly the same situation, I have found it easier to use the first method when dealing with two or more moving sprites and the second when dealing with one moving sprite and one or more stationary ones.

Combining this in a Game

In most games, these three topics are combined to produce the graphical interface. Usually, the flow of events within a game is as follows:

A user-controlled sprite is moving about the screen, or launching some sort of attack. When this attack is completed, or two sprites collide, the program triggers an event, and delivers control of the game to an event script, which executes its instructions, before terminating and restoring control to the user-controlled sprite. Event scripts may be in-game or external subroutines. There are pros and cons to both. Having your event scripts in-game increases the size of your game, thus making it more demanding on your calculator's memory. However, having them externalized makes your game unstable in the event of a crash, as some of these routines may be lost. In the end, it comes down to a matter of preference.

In the next chapter, we will discuss the final stages of game design, the optimization process, the debugging process, and then, the release process.

Chapter 6: Releasing Your Game

The Optimization Process

Now, let’s talk a little about games. We have already seen that their size can be quite large. There is something a game designer can do to help decrease the demands of his program on your calculator. Such a designer may chose to swap out routines or functions with ones that require less memory or remove parts that are unnecessary. This is called optimization.

Let’s start with simple examples. Repetitive coding can be replaced with some sort of a loop. Take, for instance, the program below:

```
ClrHome
Circle(0,0,3)
ClrHome
Circle(0,0,3)
ClrHome
Circle(0,0,3)
```

ClrHome is a 2-byte token and Circle(is a 1-byte token. That makes the above program 9 bytes long. We shall ignore, for now, the entries “0,0,3)”, which do, also, consume memory.

Now, here’s the optimized version:

```
For(A,1,3)
ClrHome
Circle(0,0,3)
End
```

The For(token uses 1 byte, as does End. Thus, the new program is 5 bytes large, a saving of 4 bytes. While this may not seem like much, let’s apply that ratio to a typical game size, about 2000 bytes.

4/9 x 2000 = 890 bytes. 890 bytes is large enough for one or two small programs.

Optimization can be a daunting task for someone who is not familiar with the intricate nature of the command options available. For this reason, I will provide some rules to optimization. Remember, that there are two main reasons to optimize, for speed, and for size. Oftentimes, these are redundant, but not always.

Rules of Optimization

1. 99% of the time, if the program is smaller, it is also faster.
2. Within a program that contains any type of loop, if the loop is smaller, it is also faster.
3. If you find yourself repeating an action, create a separate Label for it and use that label as a subroutine, calling it with the Goto command when necessary.
4. If have a Label that is only used once, eliminate it and place it into the area where it is used.
5. Play around with different ways to accomplish a task. Go with the one that takes up the least memory.

The optimization process technically never ends. Any time you find something that you can change to make your program smaller and faster, you are still in the optimization process, even if you have released an official version before then.

The Debugging Process

The debugging process is the most grueling part of the game design process. It involves finding and correcting issues that cause the program to malfunction. The easiest of the issues to resolve are the ones that trigger errors. But some of the issues we will be looking for will not yield errors. These are the trickiest to catch, but we will look at them in due time.

Run-Time Errors

There is a vast array of run-time errors, issues that can arise from improper code. TI-Basic, our language of choice here, is great for this because the calculator will tell you where the error is. Let’s review types of errors that commonly occur when running TI-Basic programs:

Syntax: A syntax error indicates that you typed something invalid into the program, such as an extra comma or period, or placed a quotation mark into a string. Pressing “Goto” in the error menu will scroll to the line of the program on which the error occurred.

For(A,0,,3): Will yield a syntax error.
“Hello” World”->Str1: Will yield a syntax error.

Argument: There are too many or too few arguments for a function, or one function is invalid. Pressing “Goto” in the error menu will scroll to the line of the program on which the error occurred.

For(A,0): Will yield an argument error, as a For loop requires at least 3 arguments.

Archived: The variable you have called cannot be read because it is in Archive. Pressing “Goto” in the error menu will scroll to the troublesome call. Fix this error by unarchiving the variable.

Undefined: The variable you have called does not exist. Pressing “Goto” in the error menu will scroll to the troublesome call.

Label: You have called a Label that does not exist. Fix this error by adding the undefined label. This error cannot be scrolled to. You must locate the sector of code manually, locate the undefined label and determine what to add yourself.

<u>Logic Errors</u>

There are no separate classes of logic errors. This is because all logic errors have the same things in common and have the same effects. Logic errors are especially prevalent in game programming, where the variety of variables and the length of the coding used can lead to often-unnecessary mistakes.

A logic error occurs when the program does not throw an error, but some improper coding causes it to act improperly. A logic error is hardest to pinpoint, as it occurs when variables intermingle in a way that throws one or more of them off. For instance, using *x* and *y* in games that utilize drawing commands will cause a logic error. Logic errors cannot be located via error. To fix them, mentally follow your coding and determine what needs to be changed. The best method of doing this is to place pauses in between segments and then running the program. Use that to break the execution on the segment that is causing the error. Triggering a break takes you to the section of the program that was running when it broke. From there, search for the problem.

The Release Process

Well, we are finally here. You have written and coded your game. You have optimized it so that it takes up as little memory as possible. You have run it through several phases of debugging, fixing errors as you encounter them. But, you are still a little bit away from being done. You need to pass through the release candidates.

A release candidate is the first release of a game. It is an experimental release. It enables others to test your game and report any errors. Every time you fix a bug and release again, you upgrade the version number by one tenth (0.1). Once you have a solid, stable release, you add one (1.0) to the current version number and it becomes the official release. For instance:

Program X takes three release candidates before it is officially released. Then it takes two more to reach its next version. The version history would look like this:

Version 2.5
Version 1.5b
Version 1.4b
Version 1.3
Version 0.3b
Version 0.2b
Version 0.1b

Note that the version history is written in reverse-chronological order. Next to the version number, a true version history should indicate what was changed, added, or upgraded between versions. Also, note the *b* after the version numbers of the release candidates. This indicates that the version is a beta release. A beta release is a full version of the program, but still has unstable features. There is one other form of program version, the alpha version. This indicates that the version is not a complete version, but is complete enough to use.

A version history should be provided with every program for release. I often enclose it within its own document. It gives a user an idea of how stable the program is.

The second thing every good program needs is a disclaimer. This document should provide legal information about the program. It should contain the release date, whether the program is closed or open source, and distribution rights. It should also state what warranty is provided in the event that the software is defunct. Finally, it should cite contact information.

The final thing every good program needs is a help file, enclosed with the program. This file should cite basic information about the program, like how much memory it needs to install and run. It should provide troubleshooting information, information about special features, and a run-through of basic features. Finally, it should provide a technical section, so that other designers can avoid conflicting with it.

Once you have a stable program version, a version history, a disclaimer, and a help file, you are prepared for your final release. Zip all the contents into a compressed file and upload it to your hosting site of choice. You have just released your first game. A program's first release is a huge milestone in a programmer's career. Remember that a programmer makes money only on an official release that sells. There are some programs that are released as freeware or shareware. These programs are all open-source, which means they may be modified and redistributed so long as the one distributing does not profit from doing so. A closed-source program means that it may not be modified but it may be distributed. Again, these specifics may vary from program to program. Check the disclaimer before doing anything expressly forbidden.

After the Release

Your work does not stop after the first official release. Part of being an active developer means that you honor feature requests. The most loved developers tailor their programs toward what the community asks for. That means that you, the programmer, should be active and interact with your users, finding out what they like in the program, what they don't, and what they want added.

There is a stigma out there that programming is an anti-social field, but I can tell you, from experience, that it is not. Let it be noted that this tutorial was worked upon by a group of developers, interacting with each other. Most modern programmers feed this negative stereotype by spending time in little cubicles, mass-producing software with a myriad of features that no one will ever use, or even care about. But, a true programmer is actively involved in the community he is designing for. He talks to users, and alters his software to satisfy them. Remember, that every feature you include in a program consumes memory. When you add useless features, you waste valuable space that can be used toward stuff that people want.

When adding features to a program, you cycle through the beta/official phases again. Your software is beta until your beta testers give it a clean bill of health. Every time you release a beta, you up the version number by one-tenth (0.1). Once you get a clean bill of health, add one (1.0) to the version number. That becomes the official version. This cycle continues until you choose to stop that line of software.