Guide to Good Programming and Game Making Practices with Game Maker

(For Game Maker 4.0 and above)

By Allen Cheung
(Allen_Cheung@hotmail.com)

1 Contents

2 Introduction

Credits

First, acknowledgements go out to Mark Overmars for creating a one-of-a-kind program that makes making games much, **much** easier than ever before. Also, much thanks is needed to give to the numerous tutorials, examples, and sample games/programs (too much to name, unfortunately) found online which serve as a welcoming embrace into the world of game making with Game Maker.

Program Information

The program that this document shall focus on is Game Maker 4.0, by Mark Overmars. It is a very powerful utility for programmers and non-programmers alike to create 2D games of old, classic games such as platformers, top-down shooters, puzzlers, RPGs, etc. – all of which existed in 2D form. This amazing program can be found here: http://www.cs.uu.nl/~markov/gmaker/index.html

My Position

Indeed, as pointed out above, it's a fairly easy task to make games with Game Maker 4.0 (GM), due to its simple drag-and-drop interface. A lot of the coding common to experienced programmers has been simplified into buttons that are used mostly in games – changing variables, adding sprites, etc. – has all been taken care of by the program. However, Mark has done one better by giving the user the ability to enter his or her own programming via the Game Maker Language (GML), which multiplies the usefulness of GM tenfold.

It is in the GML that I see the true power of GM emerging. While the visual interface is useful for non-programmers to jump right into the action, all of those actions can be successfully completed via code. Furthermore, the rich library of functions provided (and often unused, sadly) give the programmer a diverse set of tools to complete his or her task; he or she should certainly not be limited by the drag-and-drop interface.

Yet, as a Computer Science major, I often *cringe* at the code that is used by GM users. While it is certainly not expected of non-programmers (which this program is designed for) to program with perfection just from GM alone, there still exists much improvement in areas. This is not merely an aesthetic criticism; it is also a practical matter, for cleaner and better programming usually results in less confusion, reduced code, and general happiness for all.

Purpose

The purpose of this document is to give the user a greater understanding of GML, the techniques involved, and the untapped power within. I realize that some of this is covered in the manual of the program itself

(http://www.cs.uu.nl/people/markov/gmaker/doc/gmaker.htm), but these concepts are important enough to bear repeating, over and over again, until you get bored and tired of reading them. Then you may promptly close this document and flame me.

Level of Usage

To fully utilize this document, it would be best for you to already be familiar with GML; preferably, you have used some code in your programs, either to shorten the long list of actions and events or to accomplish a few functions not in the scope of those drag-and-drop techniques.

3 General Concepts

We begin by presenting a few general concepts that will vastly make the non-programmer more knowledgeable to the ways of programming.

Object Orientation

What is Object Orientation? What is Object-Oriented Programming (OOP)? GM is founded upon this system of programming, but what is it really?

OOP is a *style* of programming popularized with C++ and such languages of the past; it has become widespread with the emergence of a wave of visual development languages, such as Visual Basic, Visual C++, and Delphi. Basically, the concept behind OOP is that independent objects make up the program. Each object knows how to take care of itself, and nothing more; it holds all the procedures, routines, and functions needed to run its own purposes. This is seen in GM by the various sections of an object – it takes care of its own creation, destruction, movement, counter, alarm, and input.

Event-Driven

Yet another buzzword. This simply describes how OOP is run, which is by events – the program acts only when an event is detected, and is idle otherwise. For example, GM has events for keyboard commands; the system sits idle until a key is pressed, then looks to see whether that key triggers some snippet of code. Of course, the programmer can choose to have a **continuous** event so that the game is always moving – this is embedded in GM's Step events.

Inheritance

A second feature of OOP is the power of inheritance. Because of the nature of object orientation, it is easy to build a hierarchy of objects, a tree of parent-children couplets. The purpose of such hierarchies is *inheritance* – that is, the ability for a child object to inherit the properties of its parent. For example, suppose that I have an **electronics** object, which has the property that it is run by electricity. Now, a child of this object could be a **computer** object, which not only has the property inherited from its parent (that it uses electricity), but also has the property of having a monitor and keyboard. We can go even further and build a **Mac** object, with all the properties of a **computer** object (and thus all the properties of an **electronics** object) with the addition of having the ability to produce outstanding graphics.

In any case, inheritance is a powerful concept that will potentially save a lot of programming time and possibility programming errors from duplication of code. More on this later.

Variables

Variables are the meat of any useful program; they are placeholders for pretty much everything that can change in a program. There are a number of variable types that GML uses:

Real: Real numbers, that is numbers with decimal places, such as 1.2345
String: A string of characters, such as "this is a string"
Boolean: Either 0 or 1, meaning false and true respectively

And truly, for making 2D games with such computing power, it is not necessary to include any more variable types (unlike the programming of yesteryear, where everything had to be conserved…**byte** types anyone?).

Scripts

Scripts are the equivalent of functions or procedures in other programming languages; they serve to separate chunks of code so that they can be reused or that the general program is simplified. Sadly, while the latter is sometimes utilized, I almost never see the former used. The power of functions cannot be ignored, for they provide the means to repeatedly do a task with different parameters – taking a number and squaring it, for example. As a matter of fact, they are powerful enough to invoke their own style of programming, known as functional programming and seen in languages such as LISP.

Loops

I think that this topic is general enough to warrant some discussion here. As an old CS teacher told me, computers are only good at doing boring things repeatedly, and I tend to agree with him. The existence of loops (and its manifestations in GM, as **while**, **for** and **repeat** statements) greatly simplifies a lot of code with some simple math, and they almost always work with variables to accomplish their tasks. Smart programmers will take time to learn these well – they make case-checking and otherwise tedious code much more bearable (I'm thinking of a specific code sequence found in a "Code Book" that will remain unnamed here).

4 Style

Well, enough about abstract concepts. Let's get down to the nitty-gritty stuff. The first topic, before any other, is the discussion of the style of programming. This is the most important factor to good programming practice.

Indentation

Indentation is the process of making code more readable by indenting chucks of code by spaces. By doing so, the programmer can easily check for incorrect bracket placements, bad function calls, and overall program intent. Compare two sample programs:

```
for (x = 1; x <= 10; x += 1) {
y = 5;
for (z = 5; z >= 1; z -= 1) {
if (z == 3) {
y += 1; s = "Big Bertha";
}
else
{
name = 'Bill';
}
y -= 1;
}
}
```

and

```
for (x = 1; x <= 10; x += 1) {
  y = 5;
  for (z = 5; z >= 1; z -= 1) {
    if (z == 3) {
      y += 1;
      s = "Big Bertha";
    } else {
      name = 'Bill';
    }
  y -= 1;
  }
}
```

The first one is left plain, while the second is properly indented; by simple inspection alone, we can conclude that the second one is much easier to read at least – we can easily tell which block (that is, the chucks of code that are bounded by { }) corresponds to which statement. I can even attest to the fact that it took me a while to actually check that all the brackets were in the right place.

I can't stress this point enough! I've seen enough GML code with no indentation whatsoever, and it is simply a mess to read. Even with small snippets of code, this can easily create programs that are confusing. (try, for example, organizing a triple for loop

with nested if statements) It does not take that much effort to properly indent in the first place, and the program and programmer will be much better off for it.

Comments

Comments were included in GML for a reason, and not just because they looked particularly nice either (although I hope that Mark will someday implement the often-used /* */ notation). Once again, they are utilized to make code readable, but for whom that really benefits is the reader. There will be times when you will stumble across code that you have written ages ago, or code written by others, and you will have *no idea* what it does. So, rather than frustrating your future self and readers, why not just comment obscure code? This is especially important in Scripts, where the name of the Script is usually not enough to tell what is happening, and it's rather hard to find where the game calls that Script. Just a // may save a lot of searching in the long run.

Naming

While not strictly a GML quirk, it's still a major topic that concerns a style of programming, and I will quickly note my point here. Simply put, make your names mean something. Calling sprites spr<name> is a good mnemonic to allow yourself to remember that it is indeed a sprite (especially within programs, where you're not given an easy menu that sorts out everything else), as is calling backgrounds back<name>, objects obj<name>, etc. If you are using four sprites in four directions, try calling them sprHeroLeft, sprHeroRight, sprHeroUp, and sprHeroDown…it certainly sounds more intuitive than s1, s2, s3, and s4.

5 Scripts

*Scripts, the way that they are represented in GM, are roughly analogous to procedures and functions. Pointless trivia: in the programming language Pascal, there are both functions and procedures; in most other languages, however (C, C++, Java, etc.), there are only functions. The difference? Simply that a procedure does not return a value, whereas a function does. C and its family of languages get procedures from defining functions with no return type (**void**). Since GM emphasizes Pascal and Delphi, we will make the distinction as procedures and functions. The importance of scripts, of course, is already highlighted in the section on general topics.*

One notational point: since scripts are closely related to functions in other languages, I will abide by the standard convention that scripts have a pair of brackets at the end, regardless of how many arguments are actually in the script. Hence, `useObjects` *is a variable while* `useObjects()` *is a script with some undetermined number of arguments.*

Scripts as Procedures

Scripts as procedures are simply scripts with no return value, so that they are ideal to perform visual operations such as displaying text and drawing graphics, as well as manipulating objects. One calls scripts analogous to procedures in code as follows:

```
Some_script (arg0, arg1, … , arg9);
```

And declaring that script:

```
{
   for (x = argument0; x = argument1; x += argument2) { … }
}
```

Note that scripts can have up to ten arguments when called within GML code, and that the script itself strives to only use its arguments and not anything else. Furthermore, there is no return statement at the end; the script finishes its operations and goes back to the main program.

The use of script-procedures can be further simplified into two categories: one-time procedures and multiple-use procedures. One-time procedures are created to make coding easier to understand; `set_global_variables()` and `display_graphics()` is certainly reasonable. On the other hand, multiple-use procedures are usually more general; examples include `moveObjectRight()`.

Scripts as Functions

Scripts as functions have an important difference, in that they require a return value. Because of this, most uses of functional scripts are to calculate and return some value – for example, some mathematical function. One usually seeks not to change the arguments themselves in a function, only to use them for calculations (although the key here is

usually, for there are uses of functions that manipulate their arguments and still return a useful value). They are used in very much the same way:

```
Ans = some_other_script (arg0, arg1, … , arg9);
```

And in the script itself:

```
{
  x = argument0 * argument1;
  …
  return x;
}
```

The important part is that there is a return value, although one is not obligated to use that value. For example, the provided library function instance_create() returns the id number of the instance, but the user certainly does not have to use that number if, say, the user is merely creating some graphical menu.

As with procedural scripts, function scripts can be largely separated into two categories. The first is already mentioned, and is mostly for calculating without actually manipulating the arguments given. Hence, good examples would be `cube()` and `findSmallestNumber()`. The other kind of functional script is less used, but still very useful…scripts like the above-mentioned `instance_create()` still manages to return useful information.

Portability

Once a script is written, if it is indeed written well, it can be applied to any program or game. Collections of such functions in other programming languages are known as **libraries**, and GM has its own library of useful scripts. Unfortunately, a lot of scripts that are being written are not truly adapted for other programs right off the bat, for they are hard-coded to a specific program and require some editing on the user's part to make work. This, however, defeats the purpose of scripts – ideally, one only needs to read the description of a script to figure out its usage.

To achieve this, the script-writer is advised to write his scripts using the arguments given, rather than hard-code variables that are used in the original program that the script is written for. For example, a line of code such as `{ y = ball.x; return y; }` will throw errors an any program that does not contain a ball instance; one must edit this script to fit with whatever the script is used for. However, if the script required one argument, such that `{ y = argument0.x; return y; }`, then as long as the user passes an argument, the script will work. This is known as "adding a level of redirection" by using a variable in place of actual objects.

Examples

Since scripts are so important, I want to present a few examples as to good script programming. Perhaps the best way to practice this is by writing scripts by themselves as I am doing here; you will be forced to remove yourself from the context of a program and thus generalize your scripts. Feel free to copy and paste them into your code, using them as per the comments.

```
// Returns the square of arg
{
  return argument0 * argument0;
}

// Concatenates two strings, returns it
{
  return argument0 + argument1;
}

// Moves object arg1 spaces on a diagonal
// Arg0 is the object, Arg1 specifies distance
// Arg2 is direction - 1 = NW, 2 = NE, 3 = SW, 4 = SE
{
  localx = argument0.x; locally = argument0.y;
  if (argument2 == 1 || argument2 == 3)
    localx -= argument1;
  else localx += argument1;
  if (argument2 == 1 || argument2 == 2)
    localy += argument1;
  else localy -= argument1;
  argument0.x = localx; argument0.y = localy;
}

// Find the angle between two objects, in degrees
{
  disX = argument0.x - argument1.x;
  disY = argument0.y - argument1.y;
  tanRadians = tan(disY / disX);
  return tanRadians * 180 / (2 * pi);
}
```

I may create more scripts later if demand is great, but hopefully these few examples will serve to show the clarity that a good script can provide.

6 Variables

It is fair to say that a lot of programming and video games revolves around variables and their usage. They are what make programs interactive – the user can input data, have his input stored as variables, and the program outputs meaningful data. As such, it is worthwhile to explore and explain the value and good use of variables.

Data Structures

What are data structures? They are merely *types* of variables; for example, while **S** may be a variable, the type `string` can be considered a data structure. As explained in Section 2, General Concepts, GM provides three easy basic data structures: strings, real numbers (reals), and booleans.

Arrays

One important data structure that GM provides is an array, up to two dimensions. An array is a collection of similar data structures that can be easily accessed; it is an orderly progression of data. They are declared as follows:

```
Arrayname[number]
```

So that arrayname is name of the array, and number is the counting number (starting from zero) that contains a variable, also known as an index. In a way, an array can be thought of as a row of variables:

```
[0]   [1]   [2]   [3] …
15    178    3    84
```

In the example above, `arr[0]` would return 15, and `arr[3]` would give back 84.

The reason arrays are useful is that they can be accessed via simple counting, which is perfect for loops. By going from 0-9, for instance, one can quickly go through 10 similar variables and assign values to them. Consider the code:

```
{
  s = ''; // '' is the null string, kind of like the number 0
  for (n = 0; n <= 9; n += 1) {
    s += chr(n + ord("A"));
    arr[n] = s;
  }
  for (m = 0; m <= 9; m += 1)
    draw_text(10, m*10+10, arr[m]);
}
```

Looks intimidating, but it really isn't so bad. All the code does is first set the string variable s to the null string, then enter a loop that repeats ten times. Each time, s adds another letter to the end (remember that s += n is really shorthand for s = s + n), with that letter being the next letter in the alphabet. (a quick explanation: ord() gives the numerical

ASCII code of a character, so adding to that code will produce further letters down the line, and chr() will convert that number back to a character) It then stores that string into an array arr, which will have 10 elements (0-9) by the end of this loop. The last loop will then print the contents of that arr, which should be:

```
A
AB
ABC
…
ABCDEFGHIJ
```

With correct spacing from draw_text(), of course. The point here was that because of the nature of arrays, you can easily run through the entire thing without inventing ten variables to store ten variables that differ only by their value.

GML also allows the use of two-dimensional arrays, formatting as follows:

```
TwoDimensionalArray[num1, num2]
```

Where num1 and num2 are the two indices of the array. Just as a 1D array can be thought of as a row of data types, a 2D array is merely a table (rows and columns) of data. Since GM is in the business of creating 2D games, such arrays may come handy in, for example, representing the contents of a nxn grid (in a sliding puzzle game). Unfortunately, this is as high-dimensional as GM goes, but should be enough for all purposes.

Arrays of Objects

An excellent use of arrays would have to be in storing similar objects and instances. GML itself already provides one such grouping for you – `instance_id[n]` is an array of all the instances of an object. You may also manually store instances as a part of an array as well – for example, in an RPG with a large party, one can use something like:

```
…
wholeParty[0] = objHero;
wholeParty[1] = objEvilHero;
wholeParty[2] = objNinja;
…
```

Then, when the time comes to do something to the entire party (like taking damage from a mass-destruction spell), we need not stress ourselves over the code:

```
for (n = 0; n < numPartyMembers; n += 1)
  with (wholeParty[n]) {
    damage = random(defense) + enemyMagic;
    HP -= damage;
  }
```

And voila, our entire party has just taken damage from the spell, where each individual has calculated his/her own damage levels according to their own stats. Mark also notes

that with instances of the same object, they are already stored in an array (the `instance_id[n]` above is that array), and that to cycle through all the instances, the command `with` is provided. More on this command can, of course, be found in the official manual.

Constants

We now move onto the discussion of constants in a program. Constants are global variables that are declared in the beginning of a program; they are there usually as general bounds to certain parameters. Taking our RPG example above, we can set the maximum damage to no more than 9999, and max health the same number in the beginning of game:

```
global.MAXDAM = 9999;
global.MAXHEALTH = 9999;
```

Then in your actual code:

```
if (HP > global.MAXHEALTH)
  HP = global.MAXHEALTH;
if (damage > global.MAXDAM)
  damage = global.MAXDAM;
```

Programming naming convention usually gives constants all capitals. The point here is that by declaring these variables in the beginning, you can easily look up what the numerical values are. Furthermore, if your code uses a certain constant multiple times, when time comes to change the number (from play-testing, perhaps, when you find that 9999 isn't enough health), all that is required is a quick change in the beginning of the code, and everything else will fall in line.

Variable Usage

As you begin to make more advanced games or simply just more games in general, you will begin to realize that not a lot of numbers need to be explicitly given (this is not true for strings, however) – explicitly using a "magical number" is known as "hard-coding". You will also begin to notice that your program will behave just as well with one value as well as another, given that your data structures and overall organization is reasonably solid.

You can take advantage of this by providing the gamer with options. From experience, most games created in GM tend to be started linear – one loads the game, reads the introduction/instruction screen, then one begins the game itself. Unlike most professional games, an options screen is not provided, when having one does characterize quality in that game. They do not even need to be tedious; a platform game, for example, can have the player choose between easy, medium, or hard, and assign lives 5, 4, and 3 respectively to the gamer. Or he can up the speed of his monsters with increasing difficulty. Perhaps the main character will walk slower/faster, and jump lower/higher.

The numerical nature of games and programming almost *begs* for manipulation, so why not add a professional polish to your game?

Global Variables

GM allows for global variables in the form of:

```
global.var = x;
```

Where the keyword `global` signifies that the variable can be used, called, and changed anywhere in the program. While this certainly sounds great, the old adage "too much of a good thing will ultimately reduce you to the shadow of a man you once was and make you beg for mercy" (paraphrased) is certainly applicable. In programming, overuse of global variables is shunned, and GML is no different in this regard.

While GML avoids the issue of *namespace* (it's just a fancy term for the available names that you may give your variables…makes a difference in complicated and large programs) by requiring the use of `global`, the practice is still bad because of the potential to make your program less object-oriented (this is explained in further detail in the next chapter). By all needs, you should not even need global variables – **controller** objects (also explained in the next chapter) should be enough to keep track of all variables that are needed in your game. In other words, they are provided as a convenience, and there is often the temptation to use them boldly to avoid otherwise clean code (for example, to avoid passing arguments to scripts by setting a few global variables and changing those). They are not completely horrible, *per se*, but extensive usage simply shows that you have organized your program poorly.

Custom Data Structures

As mentioned above, it does not appear to be the case that one can make one's own data structures in the context of GML. However, there are ways around this, though they may not be so obvious to the casual GM user. The general idea is to create a custom **object** that houses all the data that you need.

You may be at a loss to figure out why anyone would need such a structure, so let's give a quick example. Suppose that you want to write a script that returns two distinct strings. Normally, this is impossible, as a script can only have one return value. Hence, you would need some sort of *container* or *data structure* to house both strings. The answer? Create a dummy object that has two local variables of strings, then in the script, stuff those two strings in there and return that.

For the most part, however, you probably will not need these custom data structures. As Mark has so carefully informed me, GM is not based on dynamic scope, but rather seems to be more lexically scoped (for those that don't have any idea what I'm talking about,

ignore it), so variables tend to stick around, it seems. In any case, this little trick is here for those that would use it.

7 Object Orientation

Being object-oriented means much more than just having objects; as discussed in General Concepts, it also provides a number of useful properties that we can use to the fullest. I present them here.

Organizing Objects

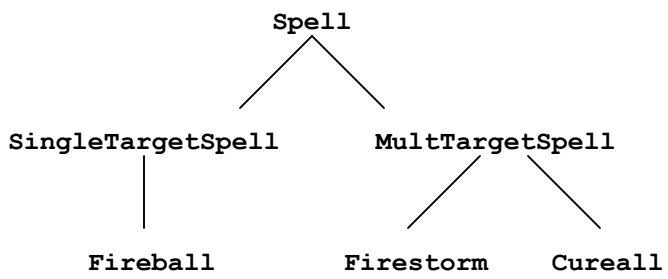
The name "objects" really should notify you of the fact that they are intended to simulate (remember that all computer programs are simulations) a single object in reality. As such, organize objects wisely and logically; not only can sloppy handling of objects be potentially confusing, but it will probably eat up resources as well, which will ultimately slow down your game and make it much less appealing to play.

One such example concerns, say, having four sprites for four directions of a character. The logical thing to do here is to create a single object and have it switch between the four sprites, rather than have four separate objects each with a separate sprite, so that objects are destroyed and created when the player moves around. Similarly, when one creates a fighting game, one creates one instance of an object, and allows that instance to change its sprite frequently (it would probably be a good idea to make sure that the object had precise collision detection too). In any case, getting exotic with objects and instances is not recommended, although by now I probably should not have to tell you that explicitly.

Inheritance

One of the big topics discussed was inheritance, and that is indeed a great timesaver in a lot of situations. All inheritance means is that a hierarchy of objects exists, in the form of parent-child pairs. A child "inherits" or automatically possesses all the properties of the parent, plus whatever the child wants to add and overwrite. A quick example: a **spell** object would have a child object **fireball**, which qualifies as a spell but has the additional property that it does damage. A **cure** would also be a child object of **spell**, but instead would have the property to heal.

The power of inheritance comes into play when we define parents that have attributes that all children share; we then only have to define that property once in the parent, and it will automatically be reflected in all the children. A more involved example:



In the hierarchy above, we notice that there are actually just three spells – **Fireball**, **Firestorm**, and **Cureall**. However, they are grouped very systematically via parents as well, all the way up until they all fit under the grouping of **Spell**. Why is this useful? Let's have some sample code:

```
// In objSpell
{
  MP -= MPused;
}

// In objSingleTargetSpell
{
  enemy[targeted].HP -= damage;
}

// In objMultTargetSpell
{
  for (n = 0; n < numEnemies; n += 1)
    enemy[n].HP -= damage;
}

// In objFireball
{
  damage = 50;
  MP = 15;
}

// In objFirestorm
{
  damage = 100;
  MP = 50;
}

…
```

We can see that with each layer, we simply add to the properties. We know that all spells consume MP, therefore that property is placed under **Spell**. Then we notice that the damage taken differs between **SingleTargetSpell** and **MultTargetSpell**, so we make different code bodies for those as well. Finally, we give those variables actual values with **Fireball**, etc. (note that I have not specified where one would place this code; you would have to fool around and figure that out; remember that there is an action for calling a parent's action/event) When all is said and done, a completed hierarchy of parents makes it very easy to create new spells; just designate the damage, the MP usage, and place it in the right place.

If that was too complicated and too specialized, then I'll give an easier example as well. Suppose that you want your character to take damage whenever he touches a monster. Now, that sentence alone should clue you in on what kind of a hierarchy you should build – just a parent **monster**, with children being the actual monsters. All that you need to be defined is when the character collides with the parent **monster**; every single child will inherit this property, and will behave properly thus.

Events

Another property of OOP is that all action is driven by events. Unlike other types of programming such as procedural (where you will plow through the main program), OOP operates on the premise that action happens only when an event occurs to trigger some action; otherwise, the system sits idle.

A mistake that some GML users make here is that they want to make the language procedural – perhaps they are used to the ways of C, Pascal, and BASIC. The closest thing to procedural programming in GM would be the **Step** event, which occurs in short regular intervals as the program is running. This event was designed for *continuous* operations – taking input from the player, for example – but for actions that occur less often!

I regularly see code that will attempt to draw a sprite in its step event, repeatedly for each step, when it is definitely not necessary to redraw the same graphic over and over again. Instead, why not draw when you know the graphic will be different? For instance, say that a lifebar needs to be drawn and updated. A smart (meaning one that has read this document thus far, of course) programmer would only redraw the graphic when the life total changes; if he has learned his lesson from this guide, he would bundle that code to change the life variable and redraw the lifebar into a script and call that. A not-so-smart programmer might try to draw the lifebar in the **Step** event, resulting in a lot of wasted CPU power as the same image gets drawn over and over again.

Of course, while my emphasis here is on graphics (which is almost always the most CPU-taxing portion of a game), it certainly applies anywhere else. This is not to say that using the **Step** event is bad by any means; rather, it cautions abuse and overuse of the event, which will bog down your game more than you would really want. Just be conservative about its usage, and you will be fine.

Controllers

In an entirely object- and event-driven world, a backstage is still needed for everything to play in; in GM, that would be the special object known as the **controller**. The controller should be familiar to most GM users; for those that are not familiar to this concept, it is merely an object that oversees everything in its domain, be that a group of enemies, a **Room,** or the entire game. Within the controller is stored all the vital variables that do not belong anywhere else – the formation of ships, for instance.

In contrast to the last chapter's topic about global variables, controllers are indeed a good programming practice. It keeps your games object-oriented (as intended by GM), and it gives you very precise control over each area that you want to address (different controllers for different rooms). Fortunately, this is one of the ideas that is well-used in the GM community, so there is not much to reprimand.

8 Efficiency

As games and programs become increasingly larger, they will inevitably become more complex, which means that more CPU power, more memory, and more time is required to run them. With small games, inefficient use of code is excusable, but in cases where every drop of power is savored, perhaps it is a good idea to be conservative in your resources, especially for older computers. Here are a few issues that may arise.

Loading

When GM loads a game and every subsequent room, it must first load all the graphics, sounds, backgrounds, and objects into the room first, along with their accompanying code, actions, and events (unless, of course, you specified that they to be loaded at runtime). The trick here is to try to minimize all the stuff that is being loaded in, and this is done by carefully choosing which things are needed when. For the most part, this process is obvious; if you have an option to choose a song, load that song at runtime. If you have rare special effects in the form of animations, unless they are truly grand and gigantic in size (which risks slowing down the actual game), have them load on runtime too.

Another idea is to limit yourself on the size of the files themselves. If permissible, use MIDIs, not .mp3s. Try to use .jpgs as much as possible, unless GM forces you to use .bmps. Keep rooms small in size, preferably under 1000x1000; split a room up if you have to (as large rooms usually occur in adventure games). An interesting idea is to make use of the custom drawing functions provided in GML – true, they tax CPU resources (as with most things in computer programming, advantages and disadvantages are often traded off), but they may be a good alternative to having a large collection of sprites if those sprites are mostly geometric (hence easily reproduced by simple drawing functions).

Memory

This topic should be a no-brainer to most GM users, as it parallels much of the tips given above. Simply put, the less loading, the less memory required (in terms of RAM requires, page file sizes, and overall program size, as they will be mostly obtained from downloads on the web); keeping things compact is the way to go in this situation as well.

CPU Resources

This one should be a biggie. Having a slow game, no matter how pretty with how many sound effects and exceptional controls, is a drag. Unfortunately, even with the relatively simple 2D premise of GM, it is not easy to create a large game and still have it run at decent speeds, at least not on a run-of-the-mill computer (at the time of this writing, such a computer in the GM community seems to be a PII 350 or so, perhaps lower).

The idea here is to make one's code as streamlined and efficient as possible. I have already discussed a little of this above, under Events of Section 7, and I urge again not to go crazy on the **Step** event and creating a lot of unnecessary graphical updates. Other ideas echo the suggestions above about keeping memory and loading requirements to a minimum. For instance, I have played a shooter that I believe tried to simulate snow by having hundreds of snow objects fall in the foreground – this is obviously not a good idea! In another case, I played another shooter where it ran exceptionally slow, which I suspect is due to the huge amounts of animation that it required to be displayed on screen at once. In such a situation, it is advisable to just simply cut some of the animation; since the game is going to be choppy anyway, cutting out a few frames of animation will usually make the game go faster while keeping the same animation. Remember, just because your Athlon XP 1900+ is able to run the game well doesn't mean that your target players with PIII 450's (for the record, this is what I'm using to run the games) are going to do as well.

Algorithms

Don't let that scary-looking word strike fear into your vocabulary – it's just a technical term for a series of steps needed to accomplish some task. Whenever you write a script, chances are that you are writing an algorithm; in life, we use algorithms to conduct daily activities all the time.

The reason I bring this up is that as code gets more complex, it becomes less and less obvious what is going on. This is especially true for a programming idiom known as **recursion** (although the average GM user will not need to use this at all), but it just might pop up in less intuitive cases as well. The point here is to write efficient and fast code, but don't go overboard. As an old programming professor once told me, it's probably better to trade off extreme efficiency for some legibility and understanding. For instance, say that you have a list of items in a character's inventory, and you want to give the player a nice, sorted list. Now, there are a lot of ways to do this (or, in computer lingo, lots of **algorithms**), but some ways work better than others in certain situations, so knowing what the sorting algorithm does and why it works is usually preferred before using it. Obviously, the average GM user doesn't care about this, and really, such in-depth studies are irrelevant unless you have a list thousands of items long anyway, so why not just go with an easy one that you can understand?

Final Words

Summary

Well, you're certainly read a lot, and I've certainly written a lot. In any case, if you have gotten this far and you have learned a few things from this document, I would say that you're going to make your GML code a whole lot better. From style, to the ideas of orient-oriented programming all the way to having efficient programs, we have touched lightly upon some ideas that make for good programming, which ultimately makes for better games. If nothing else, these concepts are not restricted to GML alone; my numerous references to other, more prominent programming languages should hint to you that they are applicable in real life also…well, as real as the cyber world, anyway.

Corrections

As I'm well aware, I'm hardly perfect, and I have indeed made a few blatant assumptions here and there through the guide. As such, if anybody has corrections to anything here, please do not hesitate to e-mail me and tell me what it is, whether it is a typo, a grammatical error, a coding problem, a misconception, or a request for clarification.

Another Guide?

After having finished this one and thoroughly enjoying writing it (although, sad to say, it's only now during break that I have the time and energy to expend for such endeavors), I'm thinking of writing another guide on tips for making a game more professional, along the lines of making lifebars, allowing the user to customize his keys, etc. Let me stress, however, that if I ever decide to write such a guide (which will be determined on whether this guide will be read and be useful or otherwise), it will be a conceptual guide and not a code guide. Simply put, my philosophy is to understand, not to blindly follow, so those looking for a place to rip code will sadly be out of luck. There are other places for that.

"If you like it, let the author know. If you hate it, let the author know why."

- *Open Source Programming Adverb (but applicable anywhere)*

Allen Cheung
Berkeley, California, USA
Allen_Cheung@hotmail.com