Lecture 9 - Dynamic resource loading and file and registry handling

Written by Carl Gustafsson

Goal of the lecture

The goal of this lecture is to teach the reader how to use Game Maker's dynamic resource handling, that is, how to load resources at runtime. The reader should also know about handling files and the registry after this lecture.

This lecture was revised for round 2 of the Game Maker programming course at www.gameuniv.net. Changes to the original document are shown with slightly greenish background. If you read this document for the first time, just ignore those markings and read it as if nothing was marked.

This lecture was also revised for round 3 of the Game Maker programming course at www.gameuniv.net. Changes to the previous revision of the document are shown with slightly blueish background. People reading this document for the first time could ignore the different background colors. Most screenshots are revised, but the change is very little, so they are not marked.

Introduction

When adding resources to Game Maker, all resource files, such as images and sounds are automatically "integrated" into the .gmd file (and also in the .exe file when such a file is made. There is however a way to make Game Maker load resources at runtime, i.e. when the game is already running. This is what I mean by "dynamic resource handling". Dynamic resource handling can come in handy in a number of circumstances. For example, if the developer wants the game to start faster and then have different sprite images, background images and sounds load during the game progress. One could also imagine the case where the player should be able to use a custom graphic for some game entity, or when it should be possible to draw an image in the game and then use it as a sprite. Then again, maybe you **never** will make a game that needs dynamic resource handling, but I think it is a good idea to have a bit of knowledge in the subject, just so that you know that it is possible to do.

Another thing that we will look into here is general file handling. To be able to load and save files is very important to some games.

Speaking of saving and loading, "dynamically loaded sprites" will unfortunately **not** be saved when you use Game Maker's default Save/Load Game feature. This means that you will have to manually store the sprite settings and then reload them if you plan to let the player save and load a game.

Lastly we will learn how to use the Windows Registry for storing game settings and similar small data.

Dynamic sprites

The first resource we will load dynamically is a sprite. As we know, a sprite consists of one or more images and a little data about transparency, origin, etc.

Still image

There are a little difference in how you add a new sprite depending on if you have a single image or an animation. If the image file is a **.gif** file, it makes no difference; Game Maker detects automatically if it is a still image or an animation. If the file is a **.jpg** or a **.bmp** file, it differs a bit though. We will look at that later on. First let us add a single **.gif** image.

So, create a new game file.

As I see it, the dynamic sprites handling can be made in two ways; either by having each object load its own sprite, or by having a certain sprite controller load all the sprites. I **think** it is the smartest to have a sprite controller object do all the loading of sprites. In this way, we can have all the files handling in a single script, and it is easier to change stuff in that script (as opposed to search through the code pieces of a large number of objects and change things).

Create a new object and call it **spr**. Um... "spr"? Why "spr"? Because it is going to contain sprite references and we can then refer to those sprites as e.g. spr.Clown, spr.Pie, etc. Make the object invisible.

Then create a new script. Call it **LoadSprites**. The command that is used for loading sprites is **sprite_add()**, which also takes a few parameters defining the file name of the sprite and other properties. Please look it up in the manual and read its description.

For a starter, enter the following code into the **LoadSprites** script:

```
// Sprite add syntax:
// sprite_add(fname, imgnumb, precise, transparent, smooth, preload, xorig, yorig)

// Load the clownsprite
clown = sprite_add("clown.gif", 0, true, true, false, true, 0, 0);
```

Make the script execute in the **Create** event of the **spr** object.

What this script does so far is just to load an image file called "clown.gif", , making it use precise collision checking, transparency and preload, setting it to **not** use "smooth" and setting its origin to (0, 0). The **imgnumb** parameter is not used when loading a **.gif** image. It is used if you have a **.bmp** or **.jpg** image that contains multiple images in a strip. Make sure that the file "clown.gif" is copied to the same directory that contains the .gmd file that you are now working on.

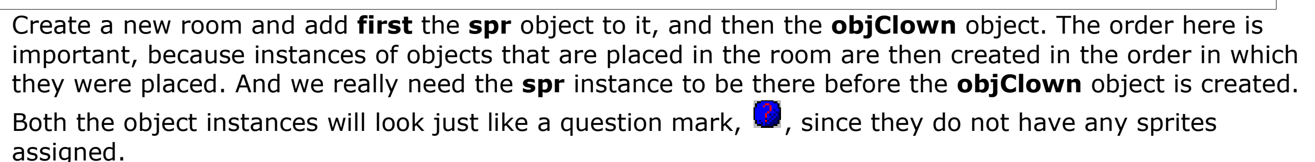
In case something unexpected happened when trying to add the image, such as the file is missine or unreadable, it is very wise to add some kind of error checking. Fortunately Game Maker has a way of telling us when something bad has happened. If an error occurs when trying to add an image, the number **-1** is returned instead of a proper sprite index. Thus, we should immediately check the returned value to see if it is **-1**. If it is, we display an error message and exit the game. Add the following code lines to the end of the **LoadSprites** script:

```
if (clown = -1) {
     show_message("Could not load the clown sprite");
     game_end();
}
```

This helps in finding errors if the game will not run.

Now, create a new object and call it **objClown**. In its **Create** event, do this:

```
// Set the sprite for the clown object.
sprite_index = spr.clown;
```

Create a new room and add **first** the **spr** object to it, and then the **objClown** object. The order here is important, because instances of objects that are placed in the room are then created in the order in which they were placed. And we really need the **spr** instance to be there before the **objClown** object is created. Both the object instances will look just like a question mark, , since they do not have any sprites assigned.

Save the game and run it to try it out. If everything is correct, you should now see the infamous clown from lecture two in the game window. Voilà! We have now loaded our first sprite dynamically.

In order to test this error trapping, try renaming the clown image file, or move it temporarily to another directory and then run the game. If everything works as it should, you should now see above error message and the game should exit.

Did you see the assignment to **sprite_index**? Like this: **sprite_index = spr.clown**. That means that we take the **clown** variable of the **spr** object and assign it to the **sprite_index** variable in out own object (**objClown**). That variable is actually just an integer number. The **sprite_add** function returns a number that is used to refer to the sprite that has been added. If this value is not stored (like in **clown** above), there is no known reference to the sprite and it can not be used. It still occupies memory though. This is somewhat similar to **pointers** in C/C++, if you know about them.

If you try and output the sprite reference as a string in a message box, you will see that it is **0** (e.g. **show_message(string(spr.clown));**). This means that this is the first sprite that is known by Game Maker. The next sprite added will get the reference **1**, etc. So, theoretically we could as well write **sprite_index = 0** to get this sprite. But the problem is that as we add sprites, it will get very hard to keep track of them all with just their reference number written literally like this, so that is why reference variables really is the only proper way to do it. One could of course also store these variables globally, like in **global.sprClown** or something. That is a matter of preference really. Personally, I grow tired of writing the word "global" all the time. ;)

Animation strip

An animation strip is a single image that contains the different frames of an animation. It could look for example like this:

(The real strip is actually longer. I have cut it short here to make it fit in the document. Check the resources for the entire strip).

When using an animation strip like this, you mention to Game Maker how many frames the strip consists of. Game Maker will then cut the strip in that many horizontal parts and add them to the game as a single sprite animation.

To try this out, we add a few lines to the **LoadSprites** script:

```
// Load the GM Logo sprite
GMLogo = sprite_add("LOGOStrip.jpg", 59, true, false, false, true, 0, 0);
```

As you can see, the main difference between this sprite addition and the clown sprite addition is that we provide a value for the number of frames in the animation. There are 59 frames in the animation and that is what we have to tell Game Maker when loading the image strip. The same error checking as above could be used here (but is omitted just because I am lazy. Or, perhaps I should add it as an assignment? ;-)).

The other thing that is different (well, apart from the file name) is that I have set "transparent" to **false** (the fourth parameter). That is because this image has a kind of gradient background and if transparency would be used, there would be funny "artifacts" along the bottom of the animation. You could try changing it to **true** to see the difference.

In order to use this logo sprite, we add a new object, **objLogo**. In the **Create** event of that object, set the **sprite_index** similar to the **objClown** object:

```
// Set the sprite for the GM Logo object.
sprite_index = spr.GMLogo;
```

Now what we also need to do is to add the **objLogo** object to the room. As you can see in the room, there are now three question mark-represented objects. So, working with dynamic objects in this way is starting to become a bit hard when it comes to handling the objects. Therefore it might be a good idea to perhaps use some kind of really low resolution sprites when building the room.

Dynamic backgrounds

So, now we know how to add sprite images dynamically to a game. Another resource that uses images is the background resource. We will see here that adding an image dynamically as a background is no harder than adding a sprite image.

The image we will use is one I just made in Terragen (another great freeware application, very useful for creating landscape backdrops):

We can load the backgrounds using the same principles as loading sprite images. Create a new object and call it **bgr** (for **b**ack**gr**ound). Add it to the room. Then create a new script, **LoadBackgrounds** and make the **bgr** object execute it in its **Create** event.

The command used for loading backgrounds at runtime is **background_add**. Check it up in the manual to see its parameters and how it works.

This is what we put in the **LoadBackgrounds** script:

```
// Load the landscape background.
landscape = background_add("background.jpg", false, false, false);

// Make the current room use this background.
background_index[0] = landscape;
// Make the background visible.
background_visible[0] = true;
```

The first statement, **landscape = background_add(....)**, loads the image file "background.jpg" into the backgrounds repository. It is not transparent, it is not preloaded, and "smooth" is not activated (three **false** values at the end of the function call). The result from this function call (or, "command" if you like) is the index number that Game Maker gives this new background resource. That number is stored in the new variable **landscape**.

This new variable can now be used to tell Game Maker to display the new background. That is what happens with **background_index[0] = landscape**. Each room can have up to **8** different background images displayed at the same time. Why would you want to display that many backgrounds at the same time? Well, if at least some of them are partly transparent it will make very much sense. This is what we used in the space shooter game to simulate the "parallax" phenomenon, remember? So, this new background resource is assigned to **background_index[0]** of the room. The only thing that is left to do then is to make sure that background 0 is made visible. It is invisible by default. I hope you recognize and understand these settings if you look at the settings of the **Room** editor.

So, running the game now would display the landscape as a backdrop to the other objects. There is no need for a single object representing each background. That is done by the room instead.

This feature, runtime background loading, can be used if you for example want to make some kind of drawing program or so in Game Maker. If you look through the Game Maker manual you will see that it is possible to save "screen-shots" from Game Maker. Loading these as backgrounds then will work very similar to a painting application.

Dynamic sounds

The only resource we have left now that can be loaded from a file is the sounds resource. Fonts can actually also be loaded from a file since GM 6.0, but I chose not to cover that here. The other resources **can** be manipulated at runtime, yes, but I will not go into detail of those here, because that would **really** be advanced programming and I have not any experience with that.

Let us not make this more complicated than necessary. We will just load the sound "boitt.wav" that was also used in the previous lesson. So, add a new object, **snd** and add it to the room. Then add a new script, **LoadSounds**, and make the **snd** object execute it in its **Create** event.

The **LoadSounds** script should look like this:

```
// Load the sound file "boitt.wav".
boitt = sound_add("boitt.wav", 0, true);
```

What it does is similar to the sprites and backgrounds loading. It loads the sound file "boitt.wav" into a new sound resource. It sets the sound type to 0 (normal) and sets "pre-load" to true, which means that the sound will be loaded immediately. Finally, the index of the new sound resource is assigned to the new variable **boitt**.

What we now need is an object that uses this new sound. Create a new object and call it **objSoundTester**. Add it too to the room.

In the **KeyPress / SPACE** event of the **objSoundTester** make it play the new sound with this code:

```
// Play the "boitt" sound
sound_play(snd.boitt);
```

If you tried to play a sound using the drag-and-drop **Play a Sound** action, you will see that naturally there are no sounds to choose from. That is because of the runtime addition of the sound. It simply does not exist at design time.

Running the game now and pressing space should, hopefully, play the "boitt" sound.

Exploring the Registry

In many games it would be useful if some of the player's settings and such could be stored in an easy way so that it could be retrieved the next time the game is played. The best and easiest way to store such things as a small collection of numbers and strings is in the Windows Registry.

To me, the Windows Registry at first sounded like some magic that only really professional programmers could handle, but I soon learned that is not hard at all to make use of this very useful resource. If you are completely new to the registry and want to have a look at it, bring up the Windows **Run** prompt (Start -> Run) and enter **regedit** (or, on some systems, **reged32**). This will bring up a hierarchical structure, not completely unlike the common file structure that you can view of your harddisk partitions. A word of warning here: "**DO NOT CHANGE ANYTHING IN THE REGISTRY UNLESS YOU KNOW EXACTLY WHAT YOU ARE DOING**". The registry stores a lot of important settings for Windows, and changing the wrong things here may, in very bad circumstances, lead to a system that can no longer start. Not good!

A little useful information about the registry:
Normally, applications that store settings about themselves, will store them in the following key:
\HKEY_CURRENT_USER\Software\<Company name>\<Application name>

So, if you have a company called **Bogus LTD** and have made an application called **WCrascher Pro**, you would be advised to store settings about this application in the key:
\HKEY_CURRENT_USER\Software\Bogus LTD\WCrascher Pro

Thus, accordingly, Game Maker stores settings in the (company-less?) key of:
\HKEY_CURRENT_USER\Software\Game Maker

This is where our registry data will go later on.

The good thing about the **HKEY_CURRENT_USER** root key is that it is individual for each user of the computer.

We are going to make an example of registry storing and retrieving through storing the player's name in the registry.

Add a new script. Call it **SettingsHandler**. What it will do is to simply ask the player for a name. The default value for the textbox that we will use is the name that the player entered the last time the game was played. Thus, the name will be stored in the registry and retrieved the next time the player plays the game.

Enter this into the new script:

```
// Get the player's name from the registry, if it exists.
// Start with an empty string.
playerName = "";
// Check with the registry.
if (registry_exists("PlayerName")) {
    playerName = registry_read_string("PlayerName");
}

// Ask the player for a name, giving the last used name as default value:
playerName = get_string("Please enter your name: ", playerName);

// Store the entered name in the registry again.
registry_write_string("PlayerName", playerName);
```

Let us have a look at what happens in the script.

First, the variable **playerName** is set to an empty string. This is just to make sure it exists and to give it an initial value.

Then, a check is made to see if the registry key called "PlayerName" exists. The first time the game is run, this key does of course not exist, because no one has used it before. Thus, its value will not be read.

But, if you have run the game before, the registry key exists. Its value is then read and assigned to the variable **playerName**.

After reading the last value from the registry, the player is asked for a name. Note that the variable **playerName** is used twice in this statement. To begin with, the result from the question should be stored in that variable. **But**, the old name, that was read from the registry, should be used as the default name in the textbox. Therefore the variable is also used as the second parameter to the **get_string** function.

Finally, the string that the player entered as a name is written to the registry. The function that writes to the registry first wants the name of the registry key to write to, "PlayerName", and then the value that should be written. That value resides in the variable **playerName**.

You can use any registry key name as you like (Although there are **probably** some kind of restrictions on registry key names that I am unaware of). Note that registry names are referred to inside quotation signs ("), so they are actually strings.

Great! Now the script is done. We now only need to create an object to use it. Create a new object and call it **objSettings**. Make this script run in the **Create** event of that object (or in any other event of your liking). Then add an instance of the object to the room.

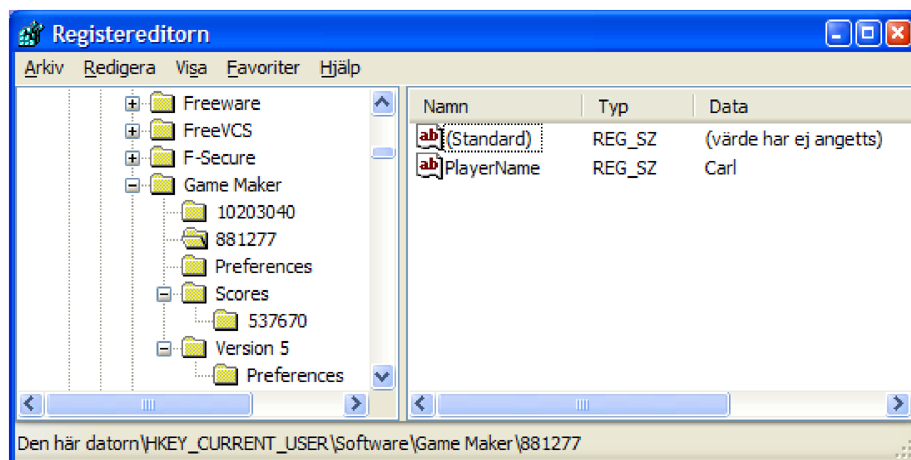
If everything is done correctly, starting the game should now display a textbox asking for your name. Enter a name and go on "playing the game". Stop the game and restart it again. This time, when the game asks for your name, it will display the last name you entered as the default value of the textbox. Isn't that fabulous? :)

"But?" you ask. "But what about all that HKEY_CURRENT_USER stuff?". Well, the thing is that Game Maker makes it very easy for you if you use the "normal" registry handling functions. All keys that you write to are contained in the following key:
\HKEY_CURRENT_USER\Software\Game Maker\NNNNNN

The "NNNNNN" above is the "unique game identifier". It is a number that Game Maker uses to uniquely identify your game. You can find out what your game id is by looking at the **Global Game Settings** and the tab **Loading**. There you will see a number after the label "Game Identifier". Take a note of the number (mine was 881277). Please do not **change** that number unless you really need to.

Go back to the registry editor now (**regedit**). View the following key:
\HKEY_CURRENT_USER\Software\Game Maker\881277 (or whatever your game identifier was).

It should then look something like this:



Although I now realize that I run the Swedish version of Windows XP and therefore many of the text strings in this image might look wierd to you "foreigners" :) :)

But, as you see in the right part of the window, there is now a value with the name "PlayerName" and the value "Carl". This is done from the game you just wrote!

Congratulations! You are now a "master" of the Windows Registry!
Writing numbers and such is done in a similar way.

General file handling

The last section of this lecture will deal with general file handling. With "general file handling" I mean the handling of files that are not of any of the resource types mentioned earlier in this lesson. The files the are handled by Game Maker in this way are plain ASCII files (as opposed to "binary files"). Actually, the newest version of Game Maker can read binary files too, but I have not had any use for that yet.

In order to make an example of this, we will make it possible to, in the game add text lines to a text file. It will also be possible to read the entire file and display it on the screen.

To make this possible, we create two new objects: **objWriteToFile** and **objReadFromFile**. Make them use the images **btnWriteToTheFile.png** and **btnReadFromTheFile.png** as sprites. (You may load them at design-time or at run-time as you please. :)). Place an instance of each button object in the room so that the player can click on them. I placed mine along the top of the room. That is because we will write some text below the buttons.

Then, make a third new object called **objTextViewer**. We will use this to display the text of the file that we will create. It does not need any sprite. Just add it to the room.

Displaying text

The text that we read from the text file will be stored as an array of strings. In the beginning we will need to init that array and tell **objTextViewer** how many lines of text we have. So, in the **Create** event of **objTextViewer**, do this:

```
// Init the text string array.
global.textStrings[0] = "";
// Init the line count.
global.nTextLines = 0;
```

Then we also need to draw the text on the screen. Add a new font resource of any kind you like. I used Arial and called the font resource **fntArial**. When that is done, add the following to the **Draw** event of the **objTextViewer**:

```
// Check if there is anything to draw. Exit if there is not.
if (global.nTextLines < 1) exit;

// Set the font.
draw_set_font(fntArial);
// Set the color.
draw_set_color(c_black);

// Go through all the lines of text and draw them on the screen.
for (a = 0; a < global.nTextLines; a += 1) {
    draw_text(10, 100 + a * 15, global.textStrings[a]);
}
```

The only complicated thing here I think is the calculation of the **y** position of the text. It is simply 100 + a * 15, which means that it will be 100 for the first line, 115 for the second line, 130 for the third, etc, since the variable **a** is increased with **1** every loop.

Writing text

Ah. Time to do the file **writing**. Make a new script and call it **WriteLineToFile**. Let it be executed by the **Mouse** event called **Left Pressed** of the object **objWriteToFile**. This means that it will only execute **once** every time the button is clicked. Write the following in the script:

```
// Ask the user for a line of text and write it to the text file.
newText = get_string("Enter a line of text to be written to the file", "");

// Open the file for appending data.
myFile = file_text_open_append("myTextFile.txt");
// The variable "myFile" now contains the "file id" of the file.
// Write the new text line to the file.
file_text_write_string(myFile, newText);
// Add a newline character (ENTER) to the file.
file_text_writeln(myFile);

// Finally, CLOSE the file. This is VERY important.
file_text_close(myFile);
```

So, what happens here is that first the player is asked for a line of text. Next, a file is opened. The file is called "myTextFile.txt". Note that there are basically **two** ways of opening a file for writing. If you use the **file_text_open_write** command, the file in question will be written from scratch, erasing all existing data. If you, on the other hand, use the command **file_text_open_append**, the file will be written starting from the **end** of the existing data. This is what we want here. Data that already exists in the file should be preserved. If the file does not exist, Game Maker automatically creates it. Neat!

When opening the file, a file id is returned. This is stored in the variable **myFile.** After the file has been opened, it can be written to. That is done with the **file_text_write_string** command. After the string has been written, a newline is added to the file with **file_text_writeln** ("write line"). That is because otherwise strings that would be added later would be added to the same line as the first string, resulting in a file with just one line, albeit a very long one. :)

Finally, something **very** important: **Closing the file!** This is important, because otherwise data might be lost. Also, on shakier operating systems (but probably not on modern ones), the file system could be corrupt if a file was not closed correctly. Yuck.

So, if you now run the "game" and click on the **Write to the file** button, you should be asked for a text line. Enter something there and click **OK**. You could then check with the Windows Explorer in the folder where your .gmd file resides. There should now be an additional file, called "myTextFile.txt". Open it up in Notepad and view your recently written text string. Great. Now we can write to a text file! Time to take look at how to read from it.

"- Mum! I can write! - That's good, honey. What are you writing? - I don't know, I can't read yet!"

Reading text

Reading the textfile would in our example mean taking each text line of the file and put it in the array of strings that is then displayed by the **objTextViewer**. We will do it with a separate script, just as with the text writing.

Create a new script and call it **ReadTextFile**. Make it execute in the **Mouse Left Pressed** event of the **objReadFromFile** object. Here is what the script should contain:

```
// Check if the text file exists.
if (not file_exists("myTextFile.txt")) exit;

// Open the file for reading.
myFile = file_text_open_read("myTextFile.txt");
// As long as there is data in the file, read it and put it in the string array.
// Reset the line count.
global.nTextLines = 0;
// Check if there is any data left in the file (End Of File is NOT true)
while (not file_text_eof(myFile)) {
    // Read a string from the file to the string array.
    global.textStrings[global.nTextLines] = file_text_read_string(myFile);
    // Jump to the next line in the file.
    file_text_readln(myFile);
    // Increase the line count.
    global.nTextLines += 1;
}

// Finaly: CLOSE THE FILE!
file_text_close(myFile);
```

First we check that the file exists. If it does not, we just exit from this script.

If, however, the file exists, it is opened for reading. The line count variable is reset to 0. Then a while loop is entered.

For each iteration the while loop checks if there is more data in the file, that is, the file is not in **End Of File** state. This is checked with the **file_text_eof** command. It returns false as long as there is more data in the file.

As long as there is data in the file, a line is read and assigned to the next variable in the text strings array. The **file_text_readln** command just jumps to the next line in the text file. At the end of the while loop, the variable that keeps track of the number of text lines is increased by one.

Finally, and very importantly, the file is closed.

Now, start the "game" and give it a try. Click the **Write to the file** button a few times and enter some text. Then click the **Read from the file** button in order to update the text display. I hope everything works now. If it does not, check that you have added the **objTextViewer** to the room. It is easily missed. :)

Conclusion

You now know how to use dynamic resources in Game Maker. You can use the registry and you can write to and read from text files. Is it not wonderful?

The dynamic resource handling can be combined with **including files** in the **Global Game Settings** in Game Maker in order to "hide" the resources from the player. You will then of course have a large **.exe** file again, but I think it will start faster and you have dynamic control over the resources. Be careful though! It is very easy to make mistakes when loading the resources at run-time. It is a good idea to implement a solid error-handling procedure to take care of the eventuality that something goes wrong when loading the resources.

Good luck!

Carl

Assignments

Assignment 9 - Store numbers in registry

Due date: Tuesday, 1 November 2005, 08:00 AM
Maximum grade: 100

The assignment for lesson 9 is to alter the text writing and reading parts of the game file so that instead of text, the scripts can read and write numbers from a text file.

Also, when you have written a few numbers to the text file, check it out in notepad or similar program. (Actually, my text editor of preference is PSPad, www.pspad.com). The numbers might not look as you expect them to do. This is called "scientific notation" if I am not completely lost. The meaning of this notation is left for the student to find out. ;) (You need not know about it at all for using it in Game Maker).

Good luck!

Carl

Read part of the GM Manual

Due date: Tuesday, 1 November 2005, 08:00 AM
Maximum grade: 0

Please read the following sections from the Game Maker 6.1 manual:

The Game Maker Language (GML), subsections:

- RESOURCES
- CHANGING RESOURCES