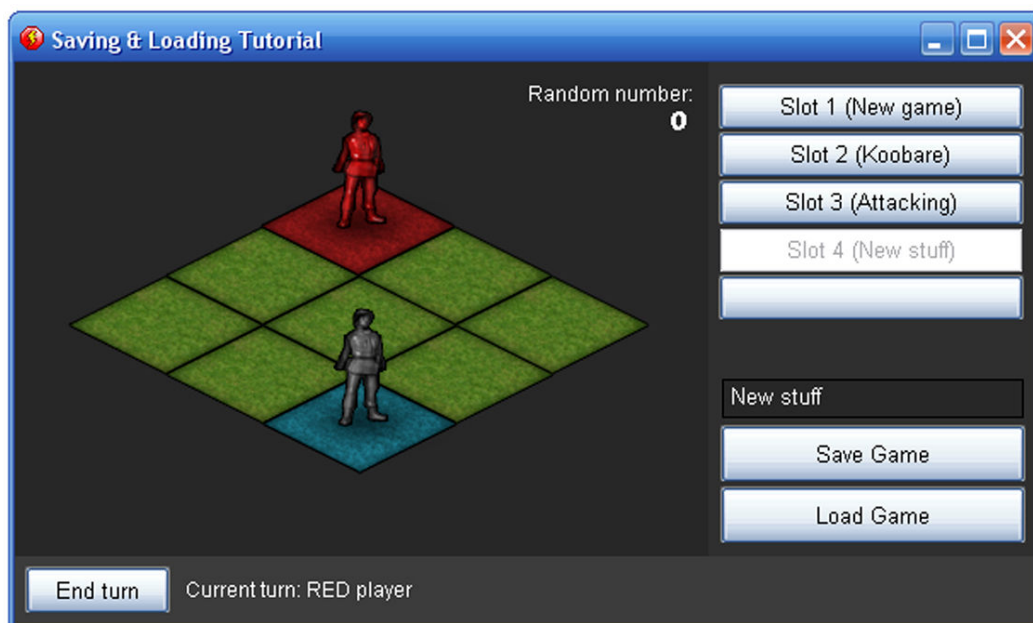




Engine Works: Saving & Loading



You may not use this tutorial for any other purpose than learning, working or having fun... In other words: You can use this tutorial for anything you'd like, as long as it doesn't involve both a hammer and a squirrel.

Koobare
marchewkowy@gmail.com

Hi there, all!

Welcome to another one of Koobare's tiny tutorials, teaching you how to effectively and efficiently work with the best multimedia authoring tool ever - [Multimedia Fusion 2](#) by Clickteam! The main purpose of this tutorial is to teach you how to easily create a fully-functional slot-based save & load system, by using a couple of Global Values, two arrays and a handful of slick tricks. But firstly – a story...

Once upon a time, there was a brave and fearless warrior that ventured into the Dungeon of Doom, to seek the evil dragon, slay him and rescue the beautiful princess from his ugly claws. After facing hordes of ghoulish crypt dwellers, flesh-eating undead, axe-throwing orcs and horrid lizardmen, he finally reaches the dreadful dragon's chamber, and... Eep, someone accidentally pushed the "reset" button! Mom, why the heck were you vacuum cleaning my computer?! Well, perhaps I like it being dusty n' all! And now my six hours of monster-slaying, level-advancing and door-bashing just went to the dustbin, thanks to you and this darn game designer that didn't even bother to input a way of saving the game!

Sounds familiar? No? Well, that's because most of the game designers DO bother to add a save & load system to their projects. And believe me – if you're thinking about creating anything bigger than the usual Pac-Man clone, you should really consider adding such a thing to your project too. Don't know how? Well, heck, that's why I'm here, right? Just follow the steps of this here tutorial to get yourself familiar with a nice and easy way of creating savegames and loading them from your hard drive... Oh yeah, and keep your eyes open for smaller details - perhaps you'll learn a trick or two that can come handy when creating an isometric game too.

So, are you ready to learn something new? Are you as brave and fearless as the warrior from our little story? Sure you are! Just grab your sword, adventurer, and let's get it on already!

If you have any problems with this tutorial, or notice that there are some mistakes present, please, contact me and I'll do my best to help you and replace all the errors with correct information.

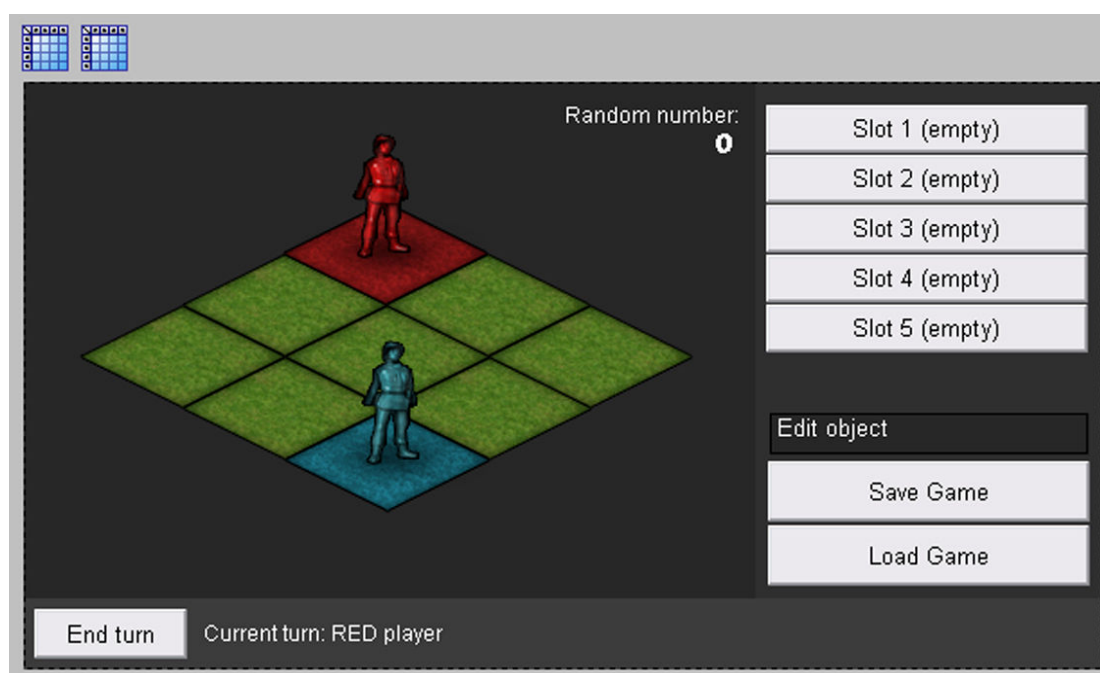
Contact me at: marchewkowy@gmail.com

Part I: Let's skip it!

Usually, we start with setting up our application's properties, click after click. We firstly create our app, save it, mess around with the Properties Toolbar, play a little with the Window size, import some objects, create a few more... Well, not this time, lad. The **"Engine Works"** tutorial series is really about creating a bit of mechanics that can be added to your game's engine – not a graphical effect, nor something that has to be precisely positioned in the Frame Editor. Nope – **it's all about the Event Editor here**. It's all about MMF2-coding, the programming part. And that's why we're going to skip the "setting up, importing & creating" part (which sometimes can be as interesting as watching paint dry) and go straight for the programming essence... If you'd like to learn more about using the Frame Editor, creating objects or playing with their settings – just download the *"Glob Wars"*, *"Smelly Claw"*, *"Fusion Player"* or *"Enhancing the Feel"* tutorials. You'll find a lot info covering the aforementioned topics there. If you require more of a basic approach – download the *"Interface Guide"* and read it thoroughly.

Now, let's start by opening our **empty_save&load.mfa** file (don't mistake it with the second one - **engine_save&load.mfa** – which already has all the events coded) – it should be zipped in the same archive as this PDF document was. We will use it as our basis for this project, as it has all the needed objects already set up and ready to use.

Once the file is opened, go to the **Frame Editor** (just double-click on the frame's thumbnail in the Storyboard Editor) and examine the available frame. It should look like this:



All TGF2 users: be advised!

- Please note that some of the objects created for this tutorial use **alpha channels**, a feature that is unavailable in *The Games Factory 2*. TGF2 users should use basic library objects or create their own graphics instead.

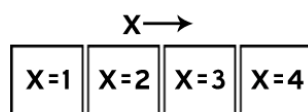
We've got 29 objects in general here: eight Button objects, two Quick Backdrops, two Strings, an Edit Box, two Array objects, couple of Actives... Looks pretty cool, right? Examine them if you wish – check their settings for yourself, notice that the “*blue_square*” and “*blue_soldier*” objects are two separate objects, that the “*string_current_turn*” object has two paragraphs set up, that the Edit Box object has a different background color... The two most important objects to look up here are our Arrays – the “*main_array*” and “*slots_array*” objects. Before we do that, though, let's have some theory, shall we? A little chit-chat about objects that really can come in handy, at least from time to time – about Arrays.

Part II: Introduction to Arrays

Arrays? What are they? What can you do with them? Those of you, dear Clickers, that haven't met with an Array yet may have such questions running between their ears right now. And I would be a backstabbing, vicious skunk if I haven't answered them. And as vicious and backstabbing as I am, I would prefer staying a *Homo sapiens*, rather than becoming a *Mephitis mephitis* stinkin' little creature.

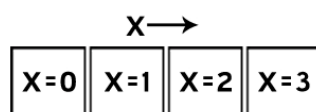
The Array object enables you to store data (values or text – you decide) in a simple, yet organized, way. Try to think of the Array as of a big table where all the cells have an “address” assigned to them. To access the data stored in the given box, you have to input the “address” that corresponds to that cell. “Addresses” can consist of a single number (if you're using a 1-dimensional array), two numbers (X and Y – if you're using a 2-dimensional array) or three numbers (X, Y and Z – when using a 3-dimensional array). You can specify how large should your array be (how many cells should it have) by accessing it's settings via the *Properties Toolbar* (the “Settings” tab, first from the left – you can set your Array's type there, set it's dimensions, make your Array global to the entire application and/or set it's index to be either 0-based or 1-based, determining whether the Array will start at 0 or 1).

OK, time for an example, let's keep this as clear as possible. Let's say that we've just created an action-adventure game, with little bits of RPG-like character development here and there. We want to store the current number of Health Points (HP) of the player's character, his Mana Points (MP), Experience Level (LEV) and his current number of Experience Points (XP). In other words: we have four different values to store. What should we do? Well, the easiest way (when it comes to Arrays) would be to create a **1-dimensional Array**: just create an Array object, go to it's properties, set the X dimension to 4, Y dimension to 1, Z dimension to 1, and set the whole thing to a *Number array*. Array objects are always invisible, so you won't be able to look at what you've just created, but if you'd have such an occasion, it would look like this:



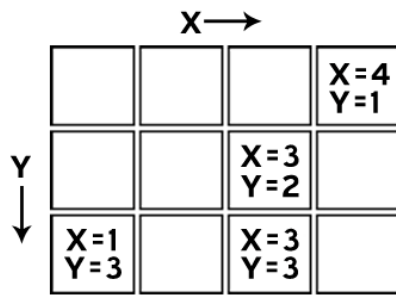
Having such an Array created, you can easily store our player's attributes in it, for example using X=1 as the storing point for player's HP, X=2 as the cell where we're going to store Mana Points, X=3 being used for LEV and X=4 for XP. This means that when our player's HP would be reduced from 100 to 57 (for example, the player could have just sustained a powerful blow from a ferocious troll), then our game should set the value in our Array's cell X=1 to 57. When the player would use a healing potion – set X=1 to 100 once more.

Oh, and remember me saying something about a “**1-based index**”? All newly created Arrays have the “**Base 1 index**” option turned **ON** by default. That means that our Arrays start counting their cells at “1”. What would happen if we'd turn this option OFF? Well, not much. Then our 1-dimensional array would look like this:



Not much of a change, is it? It's pretty much the same, you'll just have to remember that with the “Base 1 index” option OFF, our player's HP would be stored at X=0, not X=1.

Seems simple, right? And simple it is. Although, I must say, storing as little as four values in an object as powerful as an Array... That's just wasting it's power, man! The real fun starts when we want to store lots and lots of values – not only the basic stats of our game's protagonist, but also his inventory, spells that he learned, all the info about available quests, non-player characters running around, visited locations... Now that's a lot of data to store, my dear Clicker! And that's when the **2-dimensional Array** comes in!

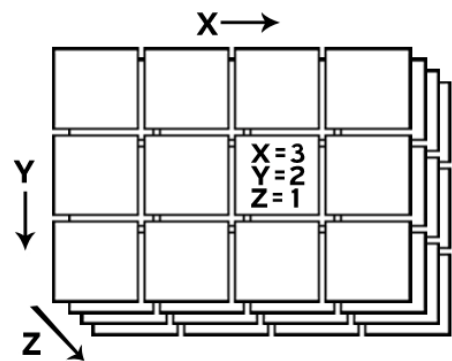


How to create a **2D Array**, like the one you can see to the left? Well, simply create a new Array object, change it's X dimension to 4, change it's Y dimension to 3, leave Z at 1... And you got it! Now you can not only store more data in a fancier way, but you can also keep it better organized! Just think about this: you could use the X=1 column (with it's Y=1, Y=2, Y=3...) as a place to store the

character's main attributes (for example: our character's Health Points could be stored at the X=1, Y=1 cell), the X=2 column could be used for secondary abilities, the X=3 column for keeping track of started and finished quests... And so on. Using a 2-dimensional Array gives you a lot more possibilities! The one thing that you should remember at all times when using a 2D array: the "addresses" leading to the specified cells consist now of two numbers – one for the columns (X) and one for the rows (Y).

There's not much to add when it comes to the **3D Array** (take a look at the image to the right if you need visual aid) – it basically just adds another dimension (Z – and that means that all cells now have "addresses" consisting of 3 numbers), which makes it a bit easier to control huge amounts of organized data. For example: a 3-dimentional Array could prove itself useful when our game's protagonist would be

joined by a band of adventurers, merging a party. Having each party member's attributes, abilities, etc. in the same Array, but on a different Z-level could speed up our creation process and simplify the expressions that we'll have to use to set, load and save all the needed figures. A 3D Array has it disadvantages, though: it can slow down your game on older computers, so be sure to use it wisely.



What's more to be said here? Three things. Firstly, be aware that **Arrays are a bit slower than Global Values and Alterable Values** – as their structure is more complicated. This won't show up in a typical game, but it's a thing to remember, especially if you're going for one of those bigger, resource-eating projects, where three fractions of a second here and there can really make a difference.

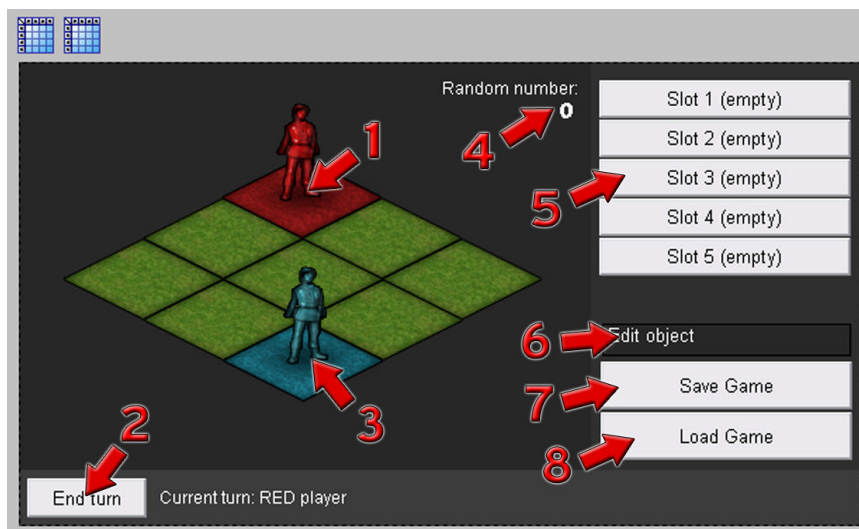
Secondly, the Array object itself doesn't have a "Compare to a value" condition. This can be a bit misleading, but there's an easy way around this – just use the "**Compare two general values**" condition from the Special Object's menu and then retrieve data from the Array.

And finally... Last, but not least (actually, I guess that's the most important thing about Arrays for us, when it comes to this tutorial): **Arrays are great when it comes to saving data to disk and retrieving it later**, as you can easily set them up to save all their cells in a fast and reliable way. Always remember: need to save something? An Array can help you with that.

And... That's all when it comes to Arrays. They're not as scary as they seemed at the beginning, right? Scary – no. Useful – yes. Better keep them in mind. You'll never know when one of them can come in handy.

Part III: Defining goals

It's time to ask the big question: **what are we trying to achieve here?** Heck, we know that we want to create a fully-functional save & load system, but you still don't have too much details on your hands, don't you? Time to change that. Time to define our goals! Come on, take a quick look at the image shown below:



Remember those two soldiers? Or that isometric battlefield? They will help us FAKE a little turn-based game. The whole thing goes like this: the “game” starts with the RED player’s turn being active. Our user can move his red soldier (marked as “1” in the upper image) by clicking somewhere on the battlefield. Once the move is done, the player has to click on the “End turn” button (2), which will deactivate the red soldier and activate the blue one. Once again a move can be made and the “End turn” button can be pushed. Every time a soldier moves, a random number is generated in the upper-right corner of the battlefield (4)... And... that’s it. No grenade-throwing, no shooting, no special effects – it’s as basic as possible, just to give us

something to save in our little Array, and that “something” is a randomly generated number, two pairs of coordinates and a “who’s turn is it?” control value.

OK, let’s say that we have already played a bit with moving our two figurines. What happens next? We have to select a slot to which the game will be saved (**5**), enter the name for our savegame (**6**) and click the “*Save Game*” button. Then – let’s move some more! Once we’re ready to load the previously saved data, we just have to select the savegame slot (**5**) and press the “*Load Game*” button (**8**). And – voila! – the savegame has been loaded!

To simplify and speed up the whole process of creating such a system, I’ve used – besides the aforementioned Arrays – a couple of Global Values (two, to be exact – named “*SaveSlot*” and “*WhichTurn*”) and Global Strings (five of them: “*SlotName1*”, “*SlotName2*”, “*SlotName3*”, “*SlotName4*” and “*SlotName5*”) – if you’d like to have a closer look at them, just go to the “*Values*” tab in the properties of our application. If you’d like to read more about Global Values, their usage and functions – be sure to download the “*Fusion Player*” tutorial, available at the “*Learning Resources*” corner of Clickteam’s website.

One more thing worth mentioning: our “pseudogame” will create 6 files at your hard drive (**drive C:**) - you can erase them as soon as this tutorial is over. All files are purely game data: “*test_save1*”, “*test_save2*”, “*test_save3*”, “*test_save4*”, “*test_save5*” and “*test_saveslots*”, they contain about 515 bytes of information. They are saved to [**C:**] just to make this tutorial a bit easier and skip the “*build into a stand-alone app and install*” part. If you’ll add this system to one of your games, be sure not to trash up someone’s main directory, make sure that all savegames will be created in the folder your game has been installed to. To do so, use this expression when giving the path to where your data should be saved:

appdrive\$ + appdir\$ + “name_of_your_file”

This little expression saves your savegame to the *name_of_your_file* file at the directory your game is installed at (It’s an expression worth remembering, so make sure that you memorize it, or at least scribble it down on a piece of paper, then attach it to your monitor). It has one disadvantage, though: it’s not the best thing if you’re still in the process of creating your game. It works best with a fully completed, built into an .exe and installed copy of your project.

Anyways... Once you’re done admiring that nifty expression... Let’s continue. Part IV is awaiting us, and that means... Yep, that means it’s time for some coding, cadet! Open your **Event Editor** and let’s not waste any more time!

Part IV: Scripting the system

Koobare's MMF-to-paper coding system

I guess that some of you have already met my MMF2-to-paper event-recording system, that helped us a lot with my earlier tutorials. If not – read on, it's pretty simple to learn and really quite useful. If you know what it's all about, just skip this introduction and head on to the scripting. Anyway, here's what it's all about:

IF (Condition): [Object for the condition] > Condition group > *Condition*

THEN (Action): [Object for the action] > Action group > *Action*

All the conditions are marked in red color, while actions are written in charming blue. Object names are always put in [square brackets]. The final condition/action is always in *Italic*. If we'll have a multi-condition event, then we'll have:

IF (Condition 1): [Object for condition 1] > Condition group 1 > *Condition 1*

IF (Condition 2): [Object for condition 2] > Condition group 2 > *Condition 2*

THEN (Action): [Object for the action] > Action group > *Action*

If you'll have to input anything by keyboard (for example: a value to set the counter to, or a text that is going to be displayed with the alterable string option – or other things that you use the *Expression Editor* for) it will be indicated by coloring the text in green and using < angle brackets >, like in this example (note that sometimes the given text will be set *Italic* for easier detection – it doesn't really mean anything):

< Set the Global Value A to 32 >

Additional comments, info and instructions will be put in << double angle brackets >>, using a different color:

<< Select any wave sound from the MMF2's sound library >>

There's not much philosophy in it, you just have to go step-by-step through all the events and keep one eye on your Event Editor, and the second one on this tutorial. It's really as simple as that... So, are you ready? Let's move in, then!

The Save & Load system

Create all the events listed here, one by one:

1) Firstly, let's start with the usual **"Start of Frame"** event... This event will set up our frame for all the things to come, it also loads the *"test_saveslots"* savefile from the given location, if available. Note that you don't have to enable all those buttons (just a single *"Disable"* for *button_slot_1* would do), you don't have to set the *SaveSlot* Global Value here either – I'm just being pedantic here. What's important: **be sure that the sequence of actions is PRECISELY the same as listed here**. To do this, double-click on one of the actions ("checked boxes") and then drag & drop actions to match the given order:

IF: [Storyboard Controls] > *Start of Frame*

THEN: [slots_array] > Files > *Load array from file*

<< Click on the "Expression" button >>

< input: "C:\test_saveslots" >

<< In the expression above, be sure to input quotation marks as well! >>

THEN: [button_slot_1] > *Disable*

THEN: [button_slot_2] > *Enable*

THEN: [button_slot_3] > *Enable*

THEN: [button_slot_4] > *Enable*

THEN: [button_slot_5] > *Enable*

THEN: [Special Object] > Change a global value > *Set*

< Set *SaveSlot* to 1 >

THEN: [edit_box] > Control > *Limit text size*

< input: 15 >

THEN: [Special Object] > *Set global string*

<< Select the "SlotName1" Global String >>

< Either just input this: *StrAtX("slots_array", 1)* OR click on the "retrieve data from an object button, select the *slots_array* object, and then the "Read value from X position" and after that input: 1 >

THEN: [Special Object] > *Set global string*

<< Select the "SlotName2" Global String >>

< input this: *StrAtX("slots_array", 2)* >

THEN: [Special Object] > *Set global string*

<< Select the "SlotName3" Global String >>

< input this: *StrAtX("slots_array", 3)* >

THEN: [Special Object] > *Set global string*

<< Select the "SlotName4" Global String >>

```

    < input this: StrAtX( "slots_array", 4) >
THEN: [Special Object] > Set global string
    << Select the "SlotName5" Global String >>
    < input this: StrAtX( "slots_array", 5) >
THEN: [button_slot_1] > Change text
    < input this: SlotName1 >
THEN: [button_slot_2] > Change text
    < input this: SlotName2 >
THEN: [button_slot_3] > Change text
    < input this: SlotName3 >
THEN: [button_slot_4] > Change text
    < input this: SlotName4 >
THEN: [button_slot_5] > Change text
    < input this: SlotName5 >

```

Phew. Now that was something, wasn't it? There's gonna' be even a bigger one later on.

2) It's time for our second event – the **"Always"** one, always being true. This will make sure that everywhere the *"red_square"* object goes– the *"red_soldier"* object will follow. The same for the *"blue_square"* and *"blue_soldier"* objects:

```

IF: [Special Object] > Always
THEN: [red_soldier] > Position > Select Position
    << Set the coordinates to x=58, y=30 relatively to the "red_square" object >>
THEN: [blue_soldier] > Position > Select Position
    << Set the coordinates to x=58, y=30 relatively to the "blue_square" object >>

```

3) And here comes the third event... Pretty useful stuff for all you isometric-game-creators, even at it's basic form. This script brings our two soldier-figurines into correct order, based on a test that checks which one of them has a lower Y position (we'll return to ordering later on).

```

IF: [red_soldier] > Position > Compare Y position to a value
    << Set the comparison mode to "Lower" >>
    < input this: Y( "blue_soldier" ) >
THEN: [blue_soldier] > Order > Bring to front

```

4) Here's the same thing as in event #3, but this time the other way round:

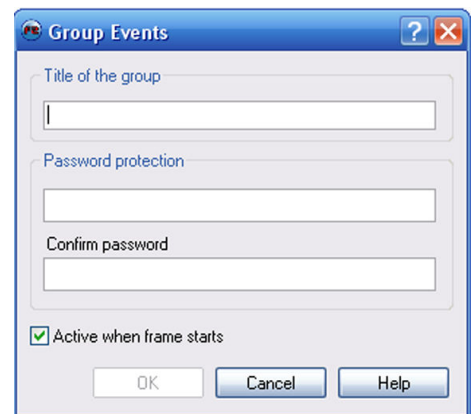
IF: **[red_soldier] > Position > Compare Y position to a value**

<< Set the comparison mode to "Greater" >>

< input this: **Y("blue_soldier")** >

THEN: **[red_soldier] > Order > Bring to front**

5) Create a new **group of events** and name it "**Who's turn is it?**" (this, once again, isn't really necessary, but will help us maintain some order and organization). While we're at it, create two more groups below, naming them "**RED player's turn**" and "**BLUE player's turn**" (we will use them to create the basic turn-based fake game). Make sure that both the "Who's turn is it?" and "RED player's turn" groups have the "**Active when frame starts**" option set **ON**, whereas the "BLUE player's turn" group should have this option turned **OFF**.



Inside the first group (the "*Who's turn is it?*" one), create the event given below – it will help us control which player (the red one or the blue one) should currently have his turn:

IF: **[Special Object] > Compare to a Global Value**

<< Select the "**WhichTurn**" Global Value >>

<< Set the comparison mode to "Equal" >>

< input: **0** >

THEN: **[Special Object] > Group of events > Activate**

<< Select the "**RED player's turn**" group >>

6) Another event, almost the same as the one listed above:

IF: **[Special Object] > Compare to a Global Value**

<< Select the "**WhichTurn**" Global Value >>

<< Set the comparison mode to "Equal" >>

< input: **1** >

THEN: [Special Object] > Group of events > Activate

<< Select the “BLUE player’s turn” group >>

Got it? Great! Time to sum it up a bit... Here, let’s take a look at what we’ve got by now (note that *it doesn’t have to look identical* – and most probably won’t, because I’ve hid a couple of unused objects, just to take a screenshot):

All the events Some objects hidden																					
1	• Start of Frame	✓									✓	✓	✓	✓	✓				✓	✓	
2	• Always					✓	✓														
3	• Y position of < Y()						✓														
4	• Y position of > Y()					✓															
5	Who's turn is it?																				
6	• WhichTurn = 0	✓																			
7	• WhichTurn = 1	✓																			
8	• New condition																				
9	RED player's turn																				
10	BLUE player's turn																				

7) OK, it’s now time to code what happens when the “RED player’s turn” group is active – in other words: what happens during the red player’s turn. Create this event in the “RED player’s turn” event group:

IF: [Special Object] > Group of events > On group activation

THEN: [Special Object] > Group of events > Deactivate

<< Select the “BLUE player’s turn” group >>

THEN: [string_current_group] > Set paragraph

<< Select the “Current turn: RED player” paragraph >>

THEN: [red_soldier] > Visibility > Change ink effect

<< Select “None” >>

THEN: [blue_soldier] > Visibility > Change ink effect

<< Select “Monochrome” >>

That’s the first event from this group! Three more events to go before moving on to the next one! Let’s rock & roll, yeah!

8) This event helps us to work out all that “who’s on top”, “who’s behind” ordering stuff:

```
IF: [blue_square] > Collisions > Overlapping another object
  << Select the “red_square” object >>
IF: [red_soldier] > Position > Compare Y position to a value
  << Set the comparison mode to “Equal” >>
  < input this: Y( “blue_soldier” ) >
THEN: [red_soldier] > Order > Bring to front
THEN: [red_square] > Order > Move in front of object
  << Select the “blue_square” object >>
```

9) A very simple, yet important, event that controls the player’s movement and makes the “random_number_counter” generate a random number from 0 to 999:

```
IF: [Mouse & Keyboard] > The mouse > User clicks on an object
  << Left button, single click >>
  << Select the “green_square” object >>
THEN: [red_square] > Position > Select Position
  << Set the coordinates to x=0, y=0 relatively to the “green_square” object >>
THEN: [random_number_counter] > Set counter
  < input: Random(1000) >
```

10) Here goes the last event in this group... A simple turn-switching:

```
IF: [end_turn_button] > Button clicked?
THEN: [Special Object] > Change a global value > Set
  < Set WhichTurn to 1 >
```

11) And now it’s time to code what happens when the “BLUE player’s turn” group is active (when it’s the blue player’s turn). Create this event in the “BLUE player’s turn” event group (and double-check if this group is **inactive** when the frame starts – that’s pretty important!):

IF: [Special Object] > Group of events > *On group activation*
THEN: [Special Object] > Group of events > *Deactivate*
 << Select the “RED player’s turn” group >>
THEN: [string_current_group] > Set paragraph
 << Select the “Current turn: BLUE player” paragraph >>
THEN: [blue_soldier] > Visibility > *Change ink effect*
 << Select “None” >>
THEN: [red_soldier] > Visibility > *Change ink effect*
 << Select “Monochrome” >>

12) And here’s another event that deals with ordering – it’s almost the same as the 8th one, but this time it’s modified to suite the blue player’s turn:

IF: [red_square] > Collisions > *Overlapping another object*
 << Select the “blue_square” object >>
IF: [red_soldier] > Position > *Compare Y position to a value*
 << Set the comparison mode to “Equal” >>
 < input this: Y(“blue_soldier”) >
THEN: [blue_soldier] > Order > *Bring to front*
THEN: [blue_square] > Order > *Move in front of object*
 << Select the “red_square” object >>

13) Once again, this event should be placed inside the “BLUE player’s turn” event group:

IF: [Mouse & Keyboard] > The mouse > *User clicks on an object*
 << Left button, single click >>
 << Select the “green_square” object >>
THEN: [blue_square] > Position > *Select Position*
 << Set the coordinates to x=0, y=0 relatively to the “green_square” object >>
THEN: [random_number_counter] > Set counter
 < input: Random(1000) >

14) Not much left to go... Here’s the last event to put in the “BLUE player’s turn” group:

THEN: [button_slot_4] > *Enable*
THEN: [button_slot_5] > *Enable*
THEN: [Special Object] > Change a global value > Set
 < Set *SaveSlot* to 1 >

16) And here's another event, veeery similar to the previous one...

IF: [button_slot_2] > *Button clicked?*
THEN: [button_slot_1] > *Enable*
THEN: [button_slot_2] > *Disable*
THEN: [button_slot_3] > *Enable*
THEN: [button_slot_4] > *Enable*
THEN: [button_slot_5] > *Enable*
THEN: [Special Object] > Change a global value > Set
 < Set *SaveSlot* to 2 >

17) ...And another one...Am I'm starting to be a bit paranoid, or are they (almost) identical?

IF: [button_slot_3] > *Button clicked?*
THEN: [button_slot_1] > *Enable*
THEN: [button_slot_2] > *Enable*
THEN: [button_slot_3] > *Disable*
THEN: [button_slot_4] > *Enable*
THEN: [button_slot_5] > *Enable*
THEN: [Special Object] > Change a global value > Set
 < Set *SaveSlot* to 3 >

18) ...Perhaps if I won't look at it, it'll just go away?

IF: [button_slot_4] > *Button clicked?*
THEN: [button_slot_1] > *Enable*
THEN: [button_slot_2] > *Enable*
THEN: [button_slot_3] > *Enable*
THEN: [button_slot_4] > *Disable*

THEN: [button_slot_5] > *Enable*
THEN: [Special Object] > Change a global value > Set
 < Set *SaveSlot* to 4 >

19) ...And here's another one. Somehow, I'm not surprised. And I guess that neither are you.

IF: [button_slot_5] > *Button clicked?*
THEN: [button_slot_1] > *Enable*
THEN: [button_slot_2] > *Enable*
THEN: [button_slot_3] > *Enable*
THEN: [button_slot_4] > *Enable*
THEN: [button_slot_5] > *Disable*
THEN: [Special Object] > Change a global value > Set
 < Set *SaveSlot* to 5 >

20) Here's one that's a bit more interesting... And a lot more complicated. After pushing this button the whole saving process starts – and it's a real beauty. One thing to keep in mind: once again, the **sequence of the actions is very, very important**.

IF: [button_save] > *Button clicked?*
THEN: [main_array] > Write > *Write Value to XY*
 < input this: X("*red_square*") >
 < X index: 1 >
 < Y index: 1 >
 << this stores the X position of the "*red_square*" object within the Array >>
THEN: [main_array] > Write > *Write Value to XY*
 < input this: Y("*red_square*") >
 < X index: 1 >
 < Y index: 2 >
 << this stores the Y position of the "*red_square*" object within the Array >>
THEN: [main_array] > Write > *Write Value to XY*
 < input this: X("*blue_square*") >
 < X index: 2 >
 < Y index: 1 >
 << this stores the X position of the "*blue_square*" object within the Array >>
THEN: [main_array] > Write > *Write Value to XY*

```

< input this: Y( "blue_square" ) >
< X index: 2 >
< Y index: 2 >
<< this stores the Y position of the "blue_square" object within the Array >>
THEN: [main_array] > Write > Write Value to XY
< input this: WhichTurn >
< X index: 3 >
< Y index: 1 >
<< this stores the value that determines who's turn is it currently >>
THEN: [main_array] > Write > Write Value to XY
< input this: value( "random_number_counter" ) >
< X index: 3 >
< Y index: 2 >
<< this stores the value of the counter >>
THEN: [main_array] > Files > Save array to file
<< click on the "Expression" button >>
< input: "C:\test_save"+Str$(SaveSlot) >
<< this action saves the game in a file that's name is based on the selected
slot's number >>
THEN: [Special Object] > Set global string
<< click on the "Use Expression" button in the upper-right corner of the window >>
< input: SaveSlot >
<< click "OK" >>
< input: "Slot "+Str$(SaveSlot)+" (" +Edittext$( "edit_box" )+)" >
<< this one changes the text that is displayed on top of the selected saveslot button >>
THEN: [button_slot_1] > Change text
< input this: SlotName1 >
THEN: [button_slot_2] > Change text
< input this: SlotName2 >
THEN: [button_slot_3] > Change text
< input this: SlotName3 >
THEN: [button_slot_4] > Change text
< input this: SlotName4 >
THEN: [button_slot_5] > Change text
< input this: SlotName5 >

```

Wow. Now that was a real record-breaker. I guess that it's pretty easy to get lost within something like that – so, if any problems occur, double-check all these actions, just to see if you've typed them in correctly. Sometimes a small mistake can take it all apart.

21) OK, the “saving” part is now officially ready. It’s time to set up the loading system... But, first of all – save your work. Got it? Great, let’s move on then. As it was said – let’s do the loading part!

```
IF: [button_load] > Button clicked?
THEN: [main_array] > Files > Load array from file
    << click on the “Expression” button >>
    < input: "C:\test_save"+Str$(SaveSlot) >
    << this action loads the selected savegame >>
THEN: [red_square] > Position > Set X Coordinate
    < input: ValueAtXY( "main_array", 1, 1) >
    << sets the “red_square’s” X coordinate to the saved value >>
THEN: [red_square] > Position > Set Y Coordinate
    < input: ValueAtXY( "main_array", 1, 2) >
    << sets the “red_square’s” Y coordinate to the saved value >>
THEN: [blue_square] > Position > Set X Coordinate
    < input: ValueAtXY( "main_array", 2, 1) >
    << sets the “blue_square’s” X coordinate to the saved value >>
THEN: [blue_square] > Position > Set Y Coordinate
    < input: ValueAtXY( "main_array", 2, 2) >
    << sets the “blue_square’s” Y coordinate to the saved value >>
THEN: [Special Object] > Change a global value > Set
    < Set WhichTurn to ValueAtXY( "main_array", 3, 1) >
    << restores the turn order from the savegame >>
THEN: [random_number_counter] > Set counter
    < input: ValueAtXY( "main_array", 3, 2) >
    << restores the “random_number_counter’s” saved value >>
```

22) And here’s the final event – yep, dear Clickers, the last one in this tutorial! Just add this one to your code, save it and test the whole thing!

```
IF: [Storyboard Controls] > End of Application
THEN: [slots_array] > Write > Write String to X
    < input this: SlotName1 >
    < X index: 1 >
    << this stores the name that was given to savegame slot 1 >>
THEN: [slots_array] > Write > Write String to X
    < input this: SlotName2 >
    < X index: 2 >
```

```
<< this stores the name that was given to savegame slot 2 >>
THEN: [slots_array] > Write > Write String to X
  < input this: SlotName3 >
  < X index: 3 >
  << this stores the name that was given to savegame slot 3 >>
THEN: [slots_array] > Write > Write String to X
  < input this: SlotName4 >
  < X index: 4 >
  << this stores the name that was given to savegame slot 4 >>
THEN: [slots_array] > Write > Write String to X
  < input this: SlotName5 >
  < X index: 5 >
  << this stores the name that was given to savegame slot 5 >>
THEN: [slots_array] > Files > Save array to file
  << click on the "Expression" button >>
  < input: "C:\test_saveslots" >
  << this saves a file containing the names of the savegame slots >>
```

Ready for a test drive? I surely am. Let's test it then! I dunno' how your project ended up, but mine's working perfectly, just the way it's supposed to. All I can say is this: success!

Need encryption?

- Want to beef up security of your savegames? Use the *Blowfish* object to encrypt them! In just a few simple clicks you can use *Blowfish*'s binary encryption algorithm to make sure that no one is getting any extra HP thanks to a little hacking. You'll find more about *Blowfish* and other MMF2 objects in the upcoming tutorials.

And that's the way the cookie crumbles, folks!

Congratulations, lad, you've just completed the "Engine Works: Saving & Loading" tutorial! Hope that you found it both educative and enjoyable. There's a lot more to MMF2 than just the features presented here – and you can be sure that we'll soon venture on yet another expedition to the wonderful world of multimedia-fusioning! And in the meantime – if you haven't done that yet – check the other tutorials on Clickteam's website, visit the forums or just

drop me an e-mail with any questions you might have! And always, always, always remember:
practice (really) makes perfect!

Thanks for your time and see you again soon!

Cheers!

Koobare
marchewkowy@gmail.com

*If you have any questions, suggestions or just need help –
mail me at marchewkowy@gmail.com*