GAME MAKER TRICKS

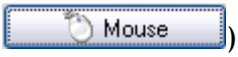
In this tutorial we will play with some of useful things inside of Game Maker that we haven't dealt with yet, and some tips on how to handle some common activities inside game maker.

MOUSE INTERFACE

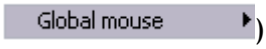
Handling the mouse is relatively easy in Game Maker because of the access to 2 very important system variables:

- mouse_x
- mouse_y

As you can probably guess, these variables allow you to know the X and Y coordinates of the mouse in the room at all times. Once you have the (X,Y) you can use conditionals to find the type of object and take action accordingly.

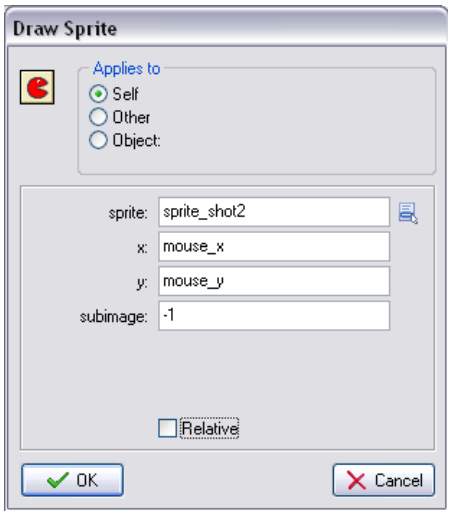
In combination with the X and Y is a collection of events under the **Mouse** (

) heading that have to do with clicking the individual buttons, scrolling the mouse wheel and the mouse entering and exiting the object. All the standard mouse events listed apply when the mouse is over the object, so a left click event deals with the mouse clicking ON the object only.

If you want to use the events *anytime* they happen, use **Global Mouse** (

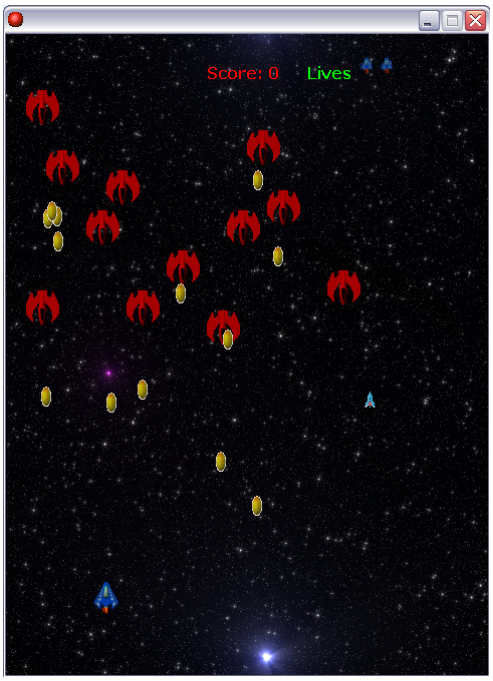
) events at the bottom. These events will fire whenever the buttons are pressed, regardless of which object it is over.

A quick example on how to use the mouse is by modifying Vacuum Marauders V 5.0 into a *Defender* type game. First we draw our own cursor on the screen using the **Draw** event in the controller and adding a **Draw Sprite** (
) action with **Sprite** = "sprite_shot2" at **X** = "mouse_x", and **Y** = "mouse_y" (with relative off):



We handle this in the controller because mouse movement is independent from any other object in the game.

Now we can place the destructive explosions anyplace on the screen by adding a **Global Mouse Left Button** event that has a **Create Instance** () action with **Object** = “object_exp_hurt” @ (“mouse_x”, “mouse_y”) (relative off). Now you should have a missle cursor and the ability to place explosions:



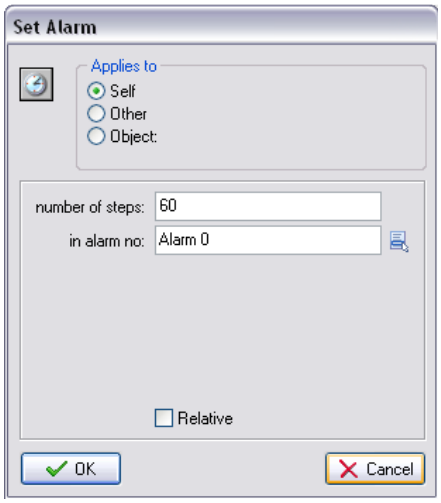
Notice on our game that you can keep creating explosions as long as you hold down the left button, this is because our left key event is like the keyboard event, it happens on every step the button is pressed. There are “pressed” events for the mouse that act like the “key press” event for keyboards.

You will also see **Joystick** events under the **Mouse** heading; these tie into the joysticks set up under Windows. There are events for the simple actions like pressing a button, but most of the advanced input for analog controls is handled through functions. Consult Game Maker help for more on joysticks.

ALARMS

We have dealt with timelines to handle things at a particular moment in time, but there is a much more flexible way to time things that allows a lot more flexibility: *Alarms*. *Alarms* are basically events that are put off until a certain time in the future. You set an alarm, give it the number of steps to wait before going off, then define in a separate **Alarm** event the series of actions to take. Alarms are bound to individual objects and you can set up to 12 alarms (numbered Alarm0 to Alarm11), each whit their own set of actions.

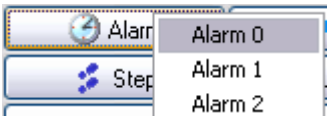
Take the point in Temple of Locks where we pause the screen at the end so the player can revel in their achievement, we use the **Sleep** action () to pause the game but we can just as easily use an alarm. Remove all the events from **Create** and add in a **Set Alarm** ( , in main 2 tab) for **Steps = 60** and **Alarm = Alarm 0**, (**Relative** off):



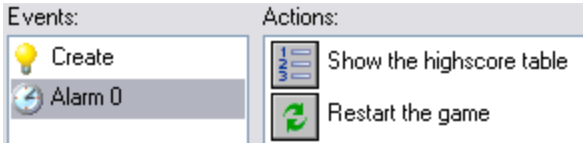
This will set Alarm0 to go off in 60 steps (~ 2 seconds at standard game speed). In this case, **Relative** can be used to add time to an alarm that is already set. The Alarm system in Game Maker works a lot like the timer that we used to kill the player if they don't get to the exit in time in Temple of Locks.

Now we will define what the alarm will do by adding a **Alarm** event (

) to the object set for Alarm0:



This event will fire when the timer for Alarm0 goes off in 60 steps. Add in a **Show Highscore** and **Rest Game** to the new event and you should have a system that works just as before:

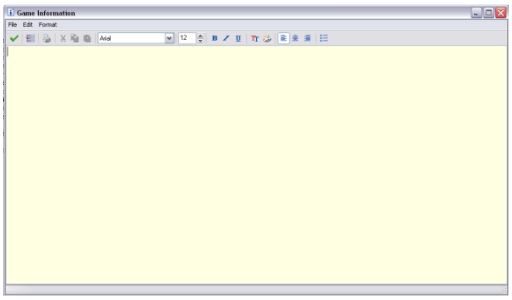


The advantage of this approach is that **Sleep** freezes all action in the game, while an alarms allows us to do other things while we are waiting, like playing a sound effect or animating an image.

GAME INFORMATION AND SETTINGS

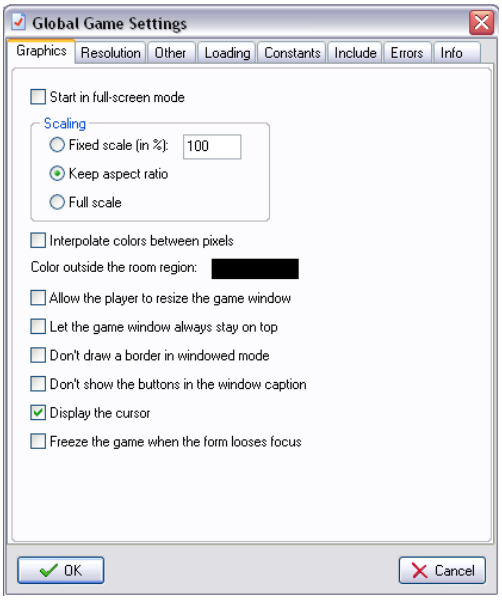
We have focused a lot on the internals of making games, but Game Maker allows us some "polish" for controlling some of the high-level game aspects. These are down at the bottom of the resource list:

Game Information is a useful place to put documentation about the game for the user and about the development of the game. This is roughly equivalent to the Help function in most games. I provides a simplistic word processor to put down all the long (yet important) parts of your game like citing the people that helped develop the game:



Game information is a great place to put help on functionality in the game. This information can then be displayed in its entirety using the **Show Info** action ( , in main2 tab). For shorter informational messages, **Display Message** ( , in main2 tab) is another good choice.

Global Game Settings controls a lot of the external aspects of the game that give it a more professional look:

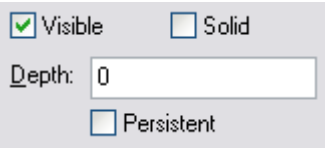


There are a lot of settings in here and most of them are self-explanatory, but here is a summary of what is in each tab:

- **Graphics**: This deals with some of the more subtle aspects of doing graphics in the game and effect visual performance. It can make full screen the default and control how the window containing the game is displayed.
- **Resolution**: Used to force a certain resolution on the screen when playing the game.
- **Other**: Binds Hotkeys to certain functions, allows for setting game process scheduling priority and setting version and copyright information.
- **Loading**: Controls the icon displayed with the game's executable and the look (or presence of) the loading bar when the game starts up.
- **Constants**: Allows you to add in custom global variables with a fixed value for use anywhere inside the game (i.e. CLUSTER_BOMB = 3)
- **Include**: Part of the Pro edition – used when plugging in extension packages.
- **Errors**: Handles how deal with errors, also allows for automatically initializing all variables to 0 to avoid errors with uninitialized variables.
- **Info**: Additional information that can be bound to the executable to be seen in the file information.

PERSISTANCE

One of the boxes you will notice on the objects and rooms is **Persistent**:



This controls whether an object or room maintains its state during room transitions. If we take Temple of Locks as an example, each time we move to a new room the value of all our custom variables (like object_controller.timer and object_exit.locked) are reset. As soon as a room loads all the objects manually placed in that room execute their **Create** events as though this is their first time popping into existence. This is fine when each room is kind of a separate game, but what if we want the player to track certain things between rooms?

Persistent allows us to keep the state of the object or room between room transitions. So if we want to use our rooms to act as a subsection of a larger map, we would check the **Persistent** button on our rooms. Persistence in rooms make any changes to the room (like the number of enemies) made during gameplay stick around after we move to another room. So if our room is persistent we could kill all the enemies in it, go to another room, and when we come back all the enemies will still be dead.

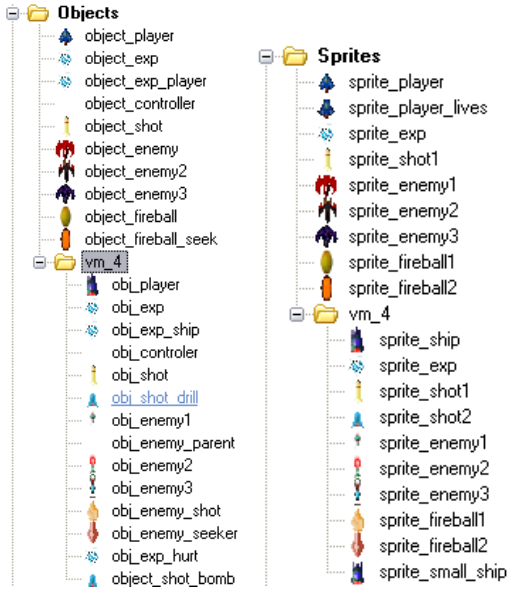
With objects, **Persistent** keeps all the custom variables intact. So if you wanted to keep the ammo gained from another room around for the next one, you would make the object **Persistent** and room transitions wouldn't affect the variable tracking ammo.

MERGING GAMES

When multiple people are working on a project in a group, combining their work together can be a logistical nightmare. The best way to avoid problems related to integrating code is using a solid design plan, but even then problems still arise. Professional development houses spend a large amount of time and effort making sure that their code management system is working correctly and effectively.

Game Maker has a much simpler system for integrating code which makes for less integration problems, but more manual configuration to get it to work. To merge a game, go into **File** and select **Merge Game …**:

Select the game to merge and hit OK. You won't see anything happen directly, but you will find a new folder with the same name of the file you imported in many of you resources:



This is basically all merge does – it imports all the object, resources and code from another game into your game. The folder keeps things the imported data separate from the regular game and keeps resources with the same name from being confused ("sprite_exp" is different from "vm_4/sprite_exp"). From here it is up to you to grab and modify whatever code you want from the merged game by using cut and paste, or replacing your own objects with the new object.

If you are going to use the new objects, make sure you go through them and change all their references to other resources, since they won't be referring to any of the original resources in the game.