CONCEPTS EXPLAINED

This reference is a companion to the Tutorials for the purpose of providing deeper explanations of concepts related to game designing and building. This reference will be updated with additional concepts as needed.

CONCEPTS (IN ORDER)

Conditional Statements and Logic Systems

In the opening unit of this course, you learned a core concept called a *hypothesis* (in Step 3 of the design process). In game design and programming, you always must always consider the events that you want and the actions you expect as a result. Hypothesis statements in computer programming, especially game design, can be written as *conditional statements*. A *conditional statement* is written so that an event (or events) can be connected to the actions that they produce. The event is what causes the action to occur. In logic, this is called *cause and effect*. An event may be what the player does on the *game interface*, such as the pressing of a key on the keyboard, such as the right arrow key. That event then could *cause* the effect of an object moving to the right, hence this would be a *cause and effect*. A conditional statement for this might be written as an *IF/THEN* statement such as: **IF the right arrow key is pressed, THEN the object will move to the right in the room.**

Once this object is being moved under the control of the player, other things could happen, which would connect this event and action to other events and actions in a *logic system*. An action may require two separate conditions to exist, in other words, two different events would both have to occur in order to see the action. This more complex conditional statement could be written as an *IF/AND/THEN* statement such as: **IF a right arrow key moves the object AND it collides with a wall, THEN it is destroyed.**

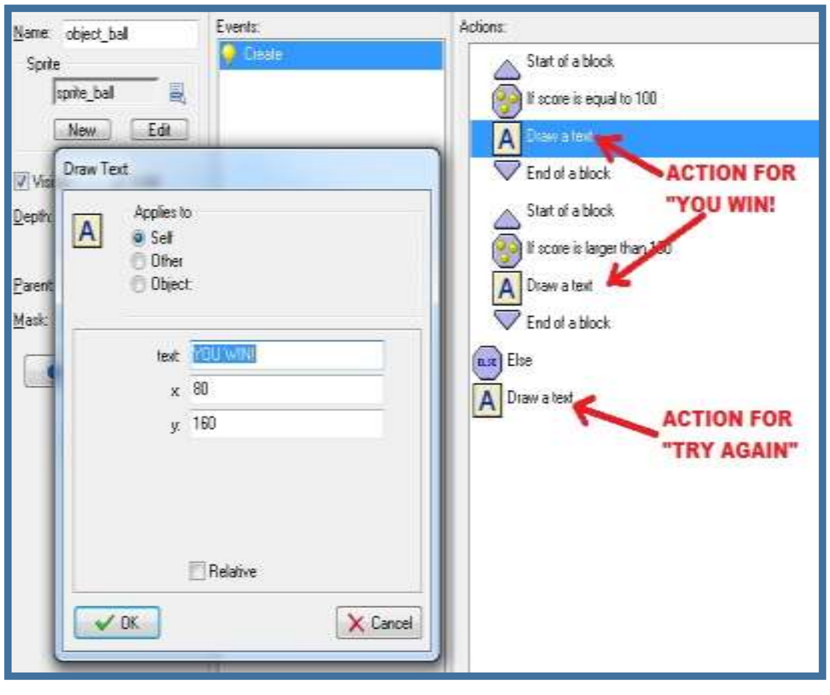
A statement may be written to describe an action based on two event options, requiring one or the other of the two events to occur so that action is created. This could be written as an *IF/OR/THEN* statement such as: **IF the object collides with a wall OR it stays in one place too long, THEN it will be destroyed.**

An event or events can also cause the effect of multiple actions to happen. For example, an event may be created to check, or "test", the game score and cause either one of two actions as an effect. Such a statement may be written in an *IF/THEN/ELSE* statement as: **IF the score is greater than 100, THEN the words "you win" are displayed, ELSE the words "try again" are displayed.**

Logic systems can be assembled in lots of ways to make a game challenging. Using *hypothesis* statements written in this manner will help you plan out the events and actions that you want in your game, especially when you have numerous events and actions that effect each other. Words like *IF, THEN, OR, AND, and ELSE* could be used in many different ways to help you think through the logic for your game programming. Here are other examples:

> **IF the score is equal to 100 OR greater than 100, THEN the words "you win" are displayed, ELSE the word "try again" are displayed.** The property window in the graphic below illustrates an event with actions that can be used to test this conditional statement (hypothesis). Parts of the *conditional actions* are programmed as *blocks*.
> **IF the score is greater than 100 AND the gameplay is in the second level, words "you win" are displayed, ELSE the word "try again" are displayed.**

By first writing *conditional statements* and treating them as *hypothesis* statements, *logic systems* using events and actions will become easier to program. After programming game properties based on your *hypothesis*, you can test the properties by running, or playing, the game to see if the hypothesis is *valid,* or "true". Testing, or checking, for a condition is done by adding a control action, usually shaped like a hexagon. Deeper explanations of various *control actions* can be found in in that section of *Concepts Explained*. The *control actions* in the blocks (see graphic to the right) set actions by checking on the condition "if the score is". You can review the concept of creating a hypothesis, testing validity, and other *Core Concepts of Technology* in the first course unit titled *Avatar Design Challenge.*



Name: object_ball
Sprite: sprite_ball
Events: Create
Actions:
Start of a block
If score is equal to 100
Draw a text
End of a block
Start of a block
If score is larger than 100
Draw a text
End of a block
Else
Draw a text

Draw Text
Applies to: Self
text: YOU WIN!
x: 80
y: 160
Relative

Control Actions

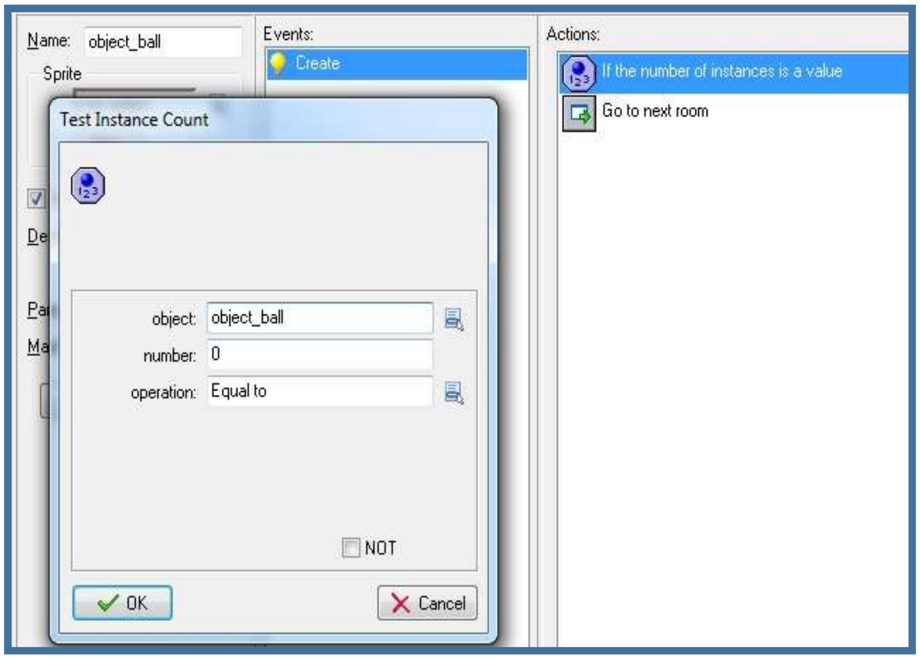
Control actions are included in programming properties when it is necessary to check to see if specific conditions are present so that other actions can occur. The games must be programmed so that events cause actions, or at a more complex level, events and actions cause other events and action (a concept explained further in *Conditional Statement and Logic Systems*). *Control actions* will "test" or "check for" the existence of conditions. Think of it this way: When active, the control action "asks questions" within the programming, and if the answer "yes" or "true", then other actions will occur. Read the conditional statement below, followed by an explanation of how a control action will check, or "ask", for the presence of the condition.

IF the number of cherries is zero, THEN go to the next level (room).

To get the action of going to the "next room", your program needs to be asking this question: "Are there zero balls?" You would need a control action called "Test Instance Count" to accomplish this. This action "tests" or "checks for" the number of ball instances in the current room. When there are no balls left (the value of zero), the action verifies that as "yes" or "true", and that condition then becomes the event that causes the action of taking the gameplay to the next room.

The graphic on the right illustrates the programming that could be created to perform the actions in the conditional statements example. The first action in the form is the **Test Instance Count**. When the number of object_ball instances Condition is "asked for" is validated as "true", and the then the **Go to next room** action is performed.

There are many control actions, most of which are found in the actions *Control tab* of the properties window. They have octagon shaped icons. You can check for scores, collisions, objects, locations of objects, and chances for random actions, just to list a few.


Name: object_ball

Events: Create

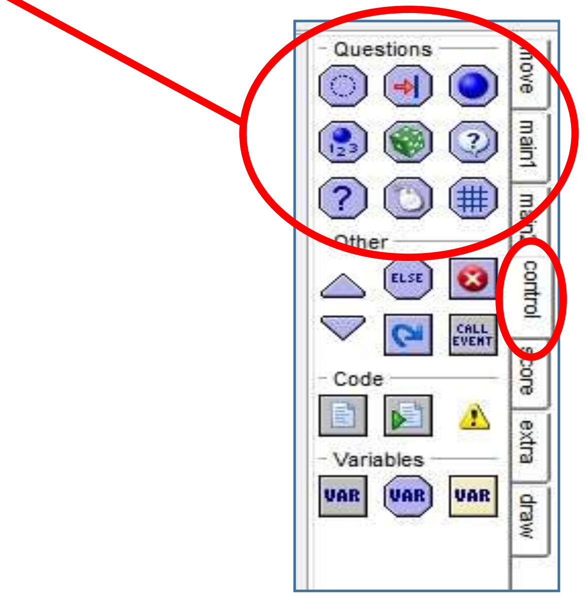
Actions: If the number of instances is a value; Go to next room

Test Instance Count

object: object_ball

number: 0

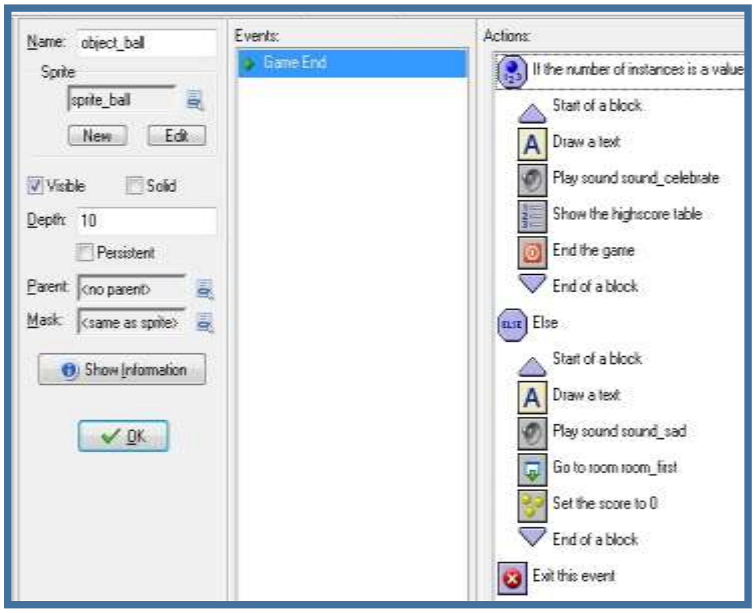
operation: Equal to



Blocks and Sub-procedures

Typically, when a condition checks as "true", the only resulting action will be the next one in list, appearing just below the *control action* in the properties window (see *Control Actions* in *Concepts Explained*). If you want more actions to be performed as a result of the one condition checked as "true", then all of those actions can be placed in a block. In the *Control tab* of the properties form, there are markers that indicate the start and end of the block, with the series of actions that you want in between the markers.

A condition can also result in options of actions, each option in a block of actions. These blocks need to be separated by an **Else** action. If a control action tests as "true", then the first of the two blocks will active. If it tests as "false", then second block following the **Else** action will activate. In computer programming, these blocks are referred to as *sub-procedures*.



Name: object_ball
Sprite: sprite_ball
Visible ✓ Solid
Depth: 10
Persistent
Parent: <no parent>
Mask: <same as sprite>

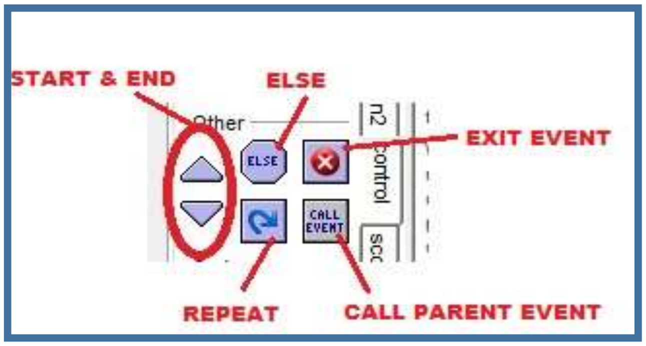
Events:
Game End

Actions:
If the number of instances is a value
Start of a block
Draw a text
Play sound sound_celebrate
Show the highscore table
End the game
End of a block
Else
Start of a block
Draw a text
Play sound sound_sad
Go to room room_first
Set the score to 0
End of a block
Exit this event

The graphic on the right illustrates properties using blocks as sub-procedures, starting with a control action and including an Else action in a Game End event. You may use programming like this at the end of a game. It also includes an **Exit** action that ends the entire process once it has executed. The following conditional statement describes a scenario for the properties in the graphic:

> **IF the number of ball object instances is zero, THEN display "you win" with happy music and show the top scores THEN exit. ELSE display "try again" with sad music, go back to the first level (room) and set the score to zero THEN exit.**

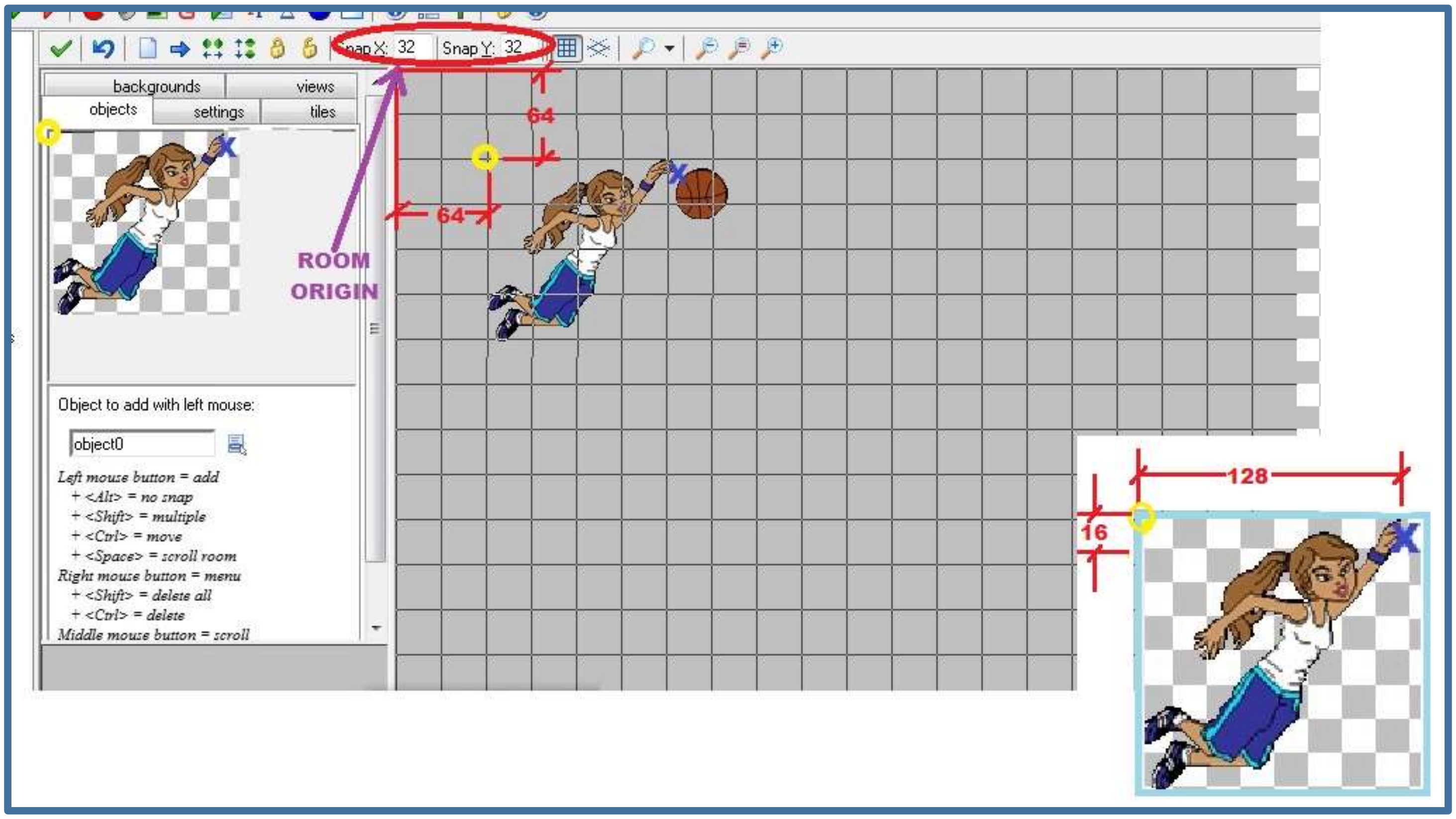
Even though the **Exit** action is the last one in the list, it will happen after either one of the block options takes place. Sub-procedure blocks may also include a **Repeat** action that will make the actions in the block repeat up to a specified number of times. The **Call Parent Event** is also useful to include as an action. It will run a sub-procedure, but the actions will be inherited from a different event. Whatever the actions are in the "parent" event, they will be used as a sub-procedure by the event you are programming, making the new event the "child". Look for examples of this programming in game build tutorials.



Coordinate Systems & Origins (Game Maker)

A coordinate system is a reference system used to plot the locations of game features, such as where object instances are located in a room. It can also place the location of object instances as they are related to, or *relative* to, other objects. In either case, you are using units that measure the distance of a point from an *origin* along a horizontal *axis* and a vertical *axis*. The locations from left/right (horizontal) are identified as *x*, while the up/down (vertical) locations are labeled as *y*. The *origin* is assigned a value of *x*=0 and *y*=0 (0, 0) which is the point at the upper left corner of the room. In *Game Make*r, locations in rooms are determined by the distance of the points from an *origin*. If you are locating an instance of an object to the right of the origin, the values of *x* increase as it moves further to the right of the origin. The *y* value increases as the object moves down. **NOTE: This is the opposite of what you are taught in math for locating points on a *y* axis. Typically, *y* values <u>would increase moving up from the origin</u>. DO NOT APPLY THE "GAMEMAKER RULE" IN YOUR MATH CLASS!**

The settings of the room grid lines are at a default value of 32 for each unit (*x* and *y*) although that can be adjusted by changing the Snap <u>X</u> and Snap <u>Y</u> settings (circled red in graphic below). Hence, you cut the distance between the lines in half by changing the values to 16. This doubles the number of gridlines and multiplies the number of squares by four. The instance of the "girl" object in the room below is at x=64, y=64 (64, 64), based on her object origin (yellow circle) as measured from the room origin (purple arrow). No matter where she moves in the room, the "ball" object must appear at her hand (blue "x"), so the instance of the ball must always will show at x=128, y=16 (128, 16), but *relative* to the location of her object origin, not the room origin. The properties for the girl that create the instance of the ball must be set to *relative* so that the ball always finds her hand.



Object to add with left mouse:

object0

Left mouse button = add
+ <Alt> = no snap
+ <Shift> = multiple
+ <Ctrl> = move
+ <Space> = scroll room
Right mouse button = menu
+ <Shift> = delete all
+ <Ctrl> = delete
Middle mouse button = scroll

Naming Conventions

When naming and saving files, regardless of the software application, start with your initials (or yours and a partner's) followed by an underscore (_). After the underscore, add a brief descriptive name based on the product you are creating. It may be a game file, a written document, sound, graphic, or even a spreadsheet for organizing information. Never use spaces and other punctuation in the file name. An underscore (_) can substitute for a space.

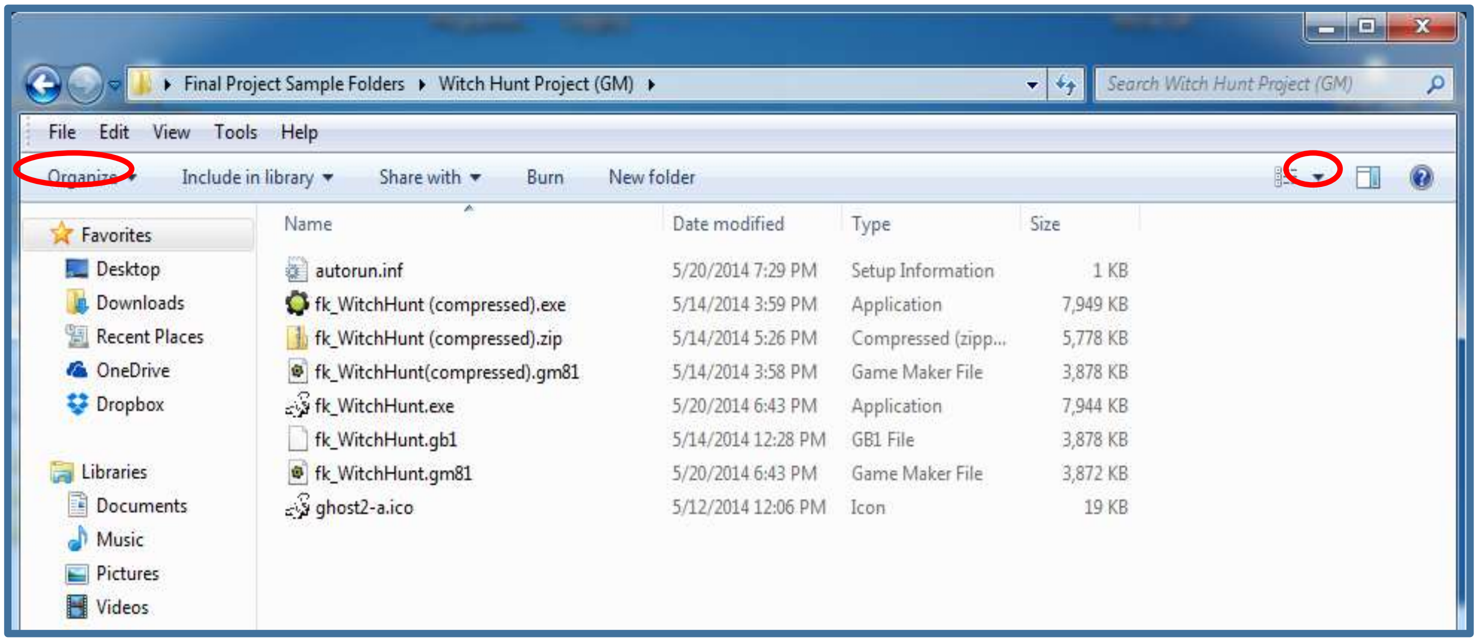
EXAMPLES: ***fk_clutches, fkrl_conceptdoc, rlfk_music, rl_soundcrash, fk_ballred, fk_alphadata***

File types (formats) and file extensions

A file is always created new, or saved, so that the information (bits and bytes of data) in the file is organized in way that is recognizable by computer software and applications. This usually occurs automatically when the file is created and saved. The file is also assigned a three digit *file extension* that identifies it as a specific file *type*, or *format*. For example, if you are doing a game build in Gamemaker8.1, it will be saved as a Game Maker file with a file extension *.gm81*. So a properly named game build file with a *file extension* might be named ***fk_clutches.gm81***.

ADDITIONAL EXAMPLES:

• A Scratch game build saved as a Scratch file with an extension of .sb or .sb2	***fk_clutches.sb***
• A Word document saved as a .docx	***fkrl_conceptdoc.docx***
• A sound asset saved as an .mp3 or .wav file	***rlfk_music.mp3*** or ***rl_soundcrash.wav***
• A graphic file saved as a .jpg, .bmp, .png, or .gif	***fk_ballred.png***
• An Excel spreadsheet saved as a .xlsx	***fk_alphadata.xlsx***


Final Project Sample Folders ▸ Witch Hunt Project (GM) ▸
Search Witch Hunt Project (GM)
File Edit View Tools Help
Organize ▾ Include in library ▾ Share with ▾ Burn New folder
Favorites
Desktop
Downloads
Recent Places
OneDrive
Dropbox
Libraries
Documents
Music
Pictures
Videos

Name	Date modified	Type	Size
autorun.inf	5/20/2014 7:29 PM	Setup Information	1 KB
fk_WitchHunt (compressed).exe	5/14/2014 3:59 PM	Application	7,949 KB
fk_WitchHunt (compressed).zip	5/14/2014 5:26 PM	Compressed (zipp...	5,778 KB
fk_WitchHunt(compressed).gm81	5/14/2014 3:58 PM	Game Maker File	3,878 KB
fk_WitchHunt.exe	5/20/2014 6:43 PM	Application	7,944 KB
fk_WitchHunt.gb1	5/14/2014 12:28 PM	GB1 File	3,878 KB
fk_WitchHunt.gm81	5/20/2014 6:43 PM	Game Maker File	3,872 KB
ghost2-a.ico	5/12/2014 12:06 PM	Icon	19 KB

An open folder with various file types, or formats, with proper naming conventions and extensions.

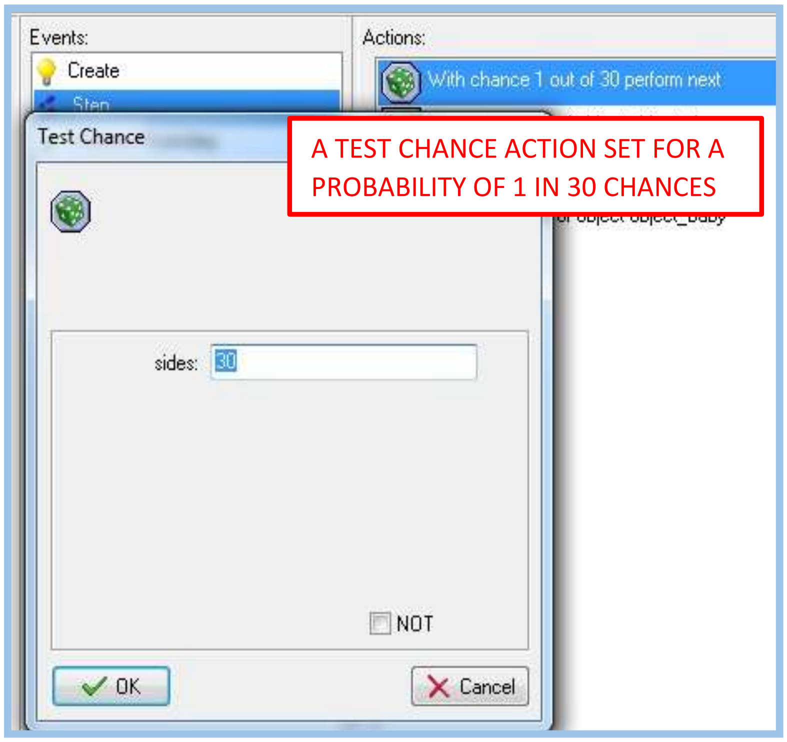
NOTE: If your file document window does not show the file details that you see in the above graphic, your teacher will show you how to set the window defaults using the **Organize** and **More Options** tabs (circled red).

Probability and Randomness

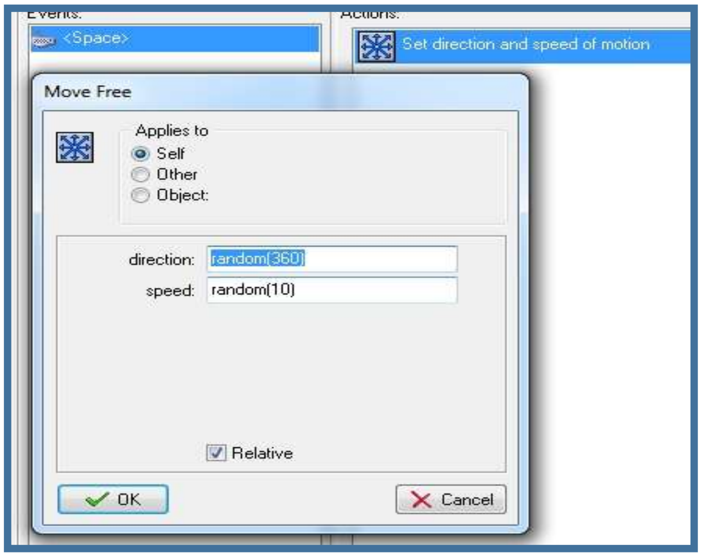
You can control the possibility that actions will happen in a game by designing *probability* into the actions. The **Test Chance** action, is a "chance to perform" action that creates a *random number generator* in your game. This is a virtual die that will generate a *random* number, just like a dice roll in any board game. You can decide how to use this concept to create actions that will occur based on a number of chances that you can set in the property. Game Maker programs will roll the die 30 times per second. Setting the number of chances to **1 in 30** creates a *probability* that for each of the **thirty rolls occurring in a second**, your chances of performing an action are **1 in 30**. Based on *probability*, you can expect your action to occur at least once per second, although it is possible that it can happen more than once per second. If you change the chance from **1 in 30** to **1 in 15**, you can expect the number of actions occurring in a second to double based on *probability*, although it also could be more. The die is still rolling 30 times per second, but you should get an action once every fifteen rolls, or 2 per second. If you change the chances from **1 in 30** to **1 in 60**, you may not see an action every second. In this case, *probability* says that you can expect an action to occur at least once in a two second period, since it will take two seconds to generate sixty rolls at rate of 30 rolls per second. Hence, you **increase the chances by lowering the second number** and **decrease the chances by raising the second number**. Managing *probability* in this manner can give your actions an element of *randomness*. When actions are *random*, they can be without the player knowing when or where they will occur.

The word *random* can also be used as a function property entered along with a value to create random speeds, locations of instances, and directions for movement. It can be written into property fields, or used as a command in scripts. You would enter the command random followed by a numerical value in parenthesis. If *random(10)* is entered in to a property field, it is creating a die with ten sides. If this were a speed property, than the speed in that action will be occur randomly between at a speed between 1 and 10. If it rolls a 2, the speed of that object will be slower than it would be if it rolls a 9. The speed will never be faster than 10. This same concept can be used to set directions, paths, and other actions.

There is also a *create instance* action that can be set to create an instances of random objects. They can appear at specific location coordinates, or at random locations. This is a great way to make objects *spawn.* This means that the object appears in the room randomly, or "made alive", often without player control.


Events:
Create
Step
Actions:
With chance 1 out of 30 perform next
Test Chance
sides: 30
NOT

A TEST CHANCE ACTION SET FOR A PROBABILITY OF 1 IN 30 CHANCES

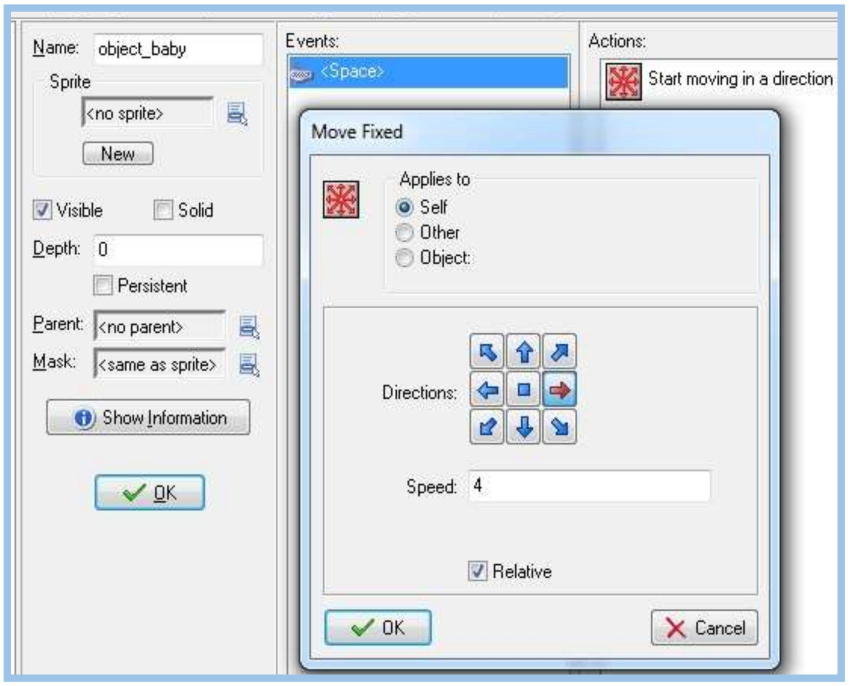

Events:
<Space>
Actions:
Set direction and speed of motion
Move Free
Applies to
[✓ Self]
Other
Object:
direction: random(360)
speed: random(10)
[✓ Relative]

A SPACEBAR KEYPRESS EVENT WITH MOVE FREE ACTION. THE OBJECT WILL MOVE IN A RANDOM DIRECTION FROM 1 TO 360^{O} AND AT A RANDOM SPEED OF 1 TO 10.

Relative

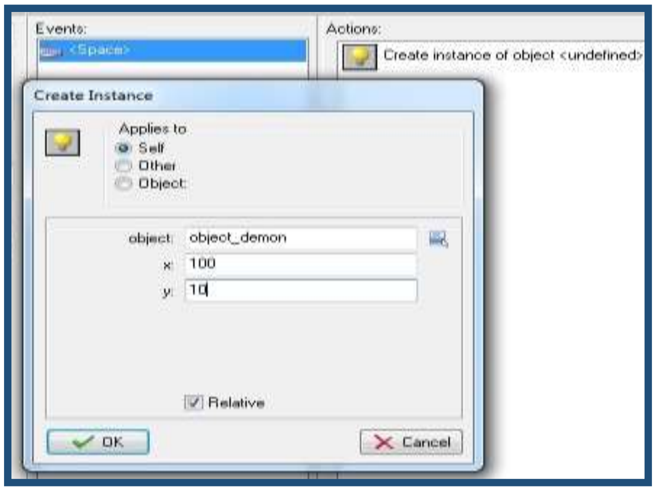
When *relative* is selected as an option in an action property, it sets that action to occur as it relates to the behavior of the object before the start of that action. For example, an object is in motion, based on properties moving it horizontally to the right at a speed of 8, meaning 8 pixels per step. You add a <right> key press event with a fixed move action set for right as a direction. In that action, you set the speed for 4 and select *relative* and choose *self*. The *self* option will compare the new action speed to its own speed that is already in action. With the object already moving to the right at a speed of 8, each time you press the right key, you add 4 to the speed that it already has. On the first key press you bump the speed to 12, because that is the speed relative, or as related, to 8. Your next key press will add another 4, making the speed 16. So the speed increases by 4, as relative to the existing speed as long as *relative* is checked. A value of -4 would reduce the speed by four, but it would continue moving in the same direction. Leaving *relative* unchecked would change the speed to the value entered in the property.

Relative is also used to locate the creation of instances for an object. An event that creates an instance of another object can either be assigned grid coordinates, or object coordinates for the instance location. If you are creating an instance and check relative, the instance will appear at a location based on the x, y coordinates entered as related to, or relative to, an object origin. It can either be relative to its own object origin by choosing self, or the origin of other objects. Leaving relative unchecked will cause the instance location to create itself based on grid coordinates. As an example, a create instance action can be added to a tank object, creating a missile shooting from the tank's missile launcher. The tank's launcher is located at x=20, y=7 (20, 7) from the tank's origin (0, 0). To get the missile to fire from the launcher, a key press event with a create instance action can be added to the tank object, with the (20, 7) coordinates entered, the *self* option selected, and the *relative* box checked. This will happen no matter where the tank object is located in the room. If *relative* is not checked, the instance of the missile will appear at grid coordinates (20, 7). The relative concept and property is used in a variety of other speed, directional, and instance generating properties and can be used in code writing.



Name: object_baby
Sprite: <no sprite>
[✓] Visible
[ ] Solid
Depth: 0
[ ] Persistent
Parent: <no parent>
Mask: <same as sprite>
Events: <Space>
Actions: Start moving in a direction

Move Fixed
Applies to: [✓ Self] Other Object:
Speed: 4
[✓] Relative



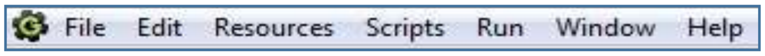
Events: <Space>
Actions: Create instance of object <undefined>

Create Instance
Applies to: [✓ Self] Other Object:
object: object_demon
x: 100
y: 10
[✓] Relative

EXAMPLES OF "RELATIVE" BEING APPLIED TO SPEED AND LOCATION PROPERTIES.

Tools and Resources in Game Maker: The Menu Bar

File Menu



The Menu bar is a horizontal bar located at the top of the screen below the title bar, containing drop-down menus accessing a variety of *tools and resources* needed for the Game Maker application.

New - Choose this command to start creating a new game. If the current game was changed you are asked whether you want to save it. There is also a toolbar button for this.

Open - Opens a game file. There is also a toolbar button for this command. You can also open a game by dragging the file into the Game Maker window.

Recent Files - Use this submenu to reopen game files you recently opened.

Save - Saves the game design file under its current name. If no name was specified before, you are asked for a new name. You can only use this command when the file was changed. Again, there is a toolbar button for this. See *Naming Conventions* for information on how to name files when saving.

Save As - Saves the game design file under a different name. You are asked for a new name. Again, refer to *Naming Conventions* for naming files.

Create Executable - Once your game is ready you will probably want to give it to others to play. Using this command you can create a stand- alone version of your game. This is simply an executable that you can give to other people to run.

Advanced Mode - When clicking on this command Game Maker will switch between simple and advanced mode. In advanced mode additional commands and resources are available.

Exit - Press this to exit Game Maker. If you mad edits to the current game you will be asked whether you want to save it.
The "G" – This is the icon to the left of the File menu. Tools in this menu can be used to close, minimize, or restore the Game Maker website that may open automatically in the main form when Game Maker is launched.

Edit Menu

The contents of this menu may vary based on the resource you want to edit (sprite, object, sound, etc.). Note that all these commands can also be given in a different way. Right- click on a resource or resource group, and the appropriate pop-up menu will appear.

Insert resource - Inserts a new instance of the currently selected type of resource before the current one. A form will open in which you can change the properties of the resource.

Duplicate - Makes a copy of the current resource and adds it. A form is opened in which you can change the resource.

Delete - Deletes the currently selected resource (or group of resources). You will see a prompt reminding you that deleting resources cannot be undone. Be sure this is what you want before deleting.

Rename - Gives the resource a new name. This can also be done in the property form for the resource. Also, you can select the resource in the resource explorer and rename it.

Properties - Use this command to bring up the form to edit the properties. Note that all the property forms appear within the main form. You can edit many of them at the same time. You can also edit the properties by double clicking on the resource.

Resources Menu
In this menu, you can create new resources of each of the different types. These include sprites, sounds, objects, rooms, backgrounds, scripts and much more. Note that for each of them there is also a button on the toolbar and a keyboard shortcut. Also you can change the game information and the global game settings.

Run Menu
This menu is used to run the game. There are two ways to run a game. They will behave as they would as executable files created with the File menu command.

Run normally - Runs the game as it would normally run. The game is run in the most efficient way and will look and act as in an executable game.

Run in Debug mode - Runs the game in debug mode. In this mode you can check certain aspects of the game and you can pause and step through it. This is useful when something goes wrong but is a bit advanced.

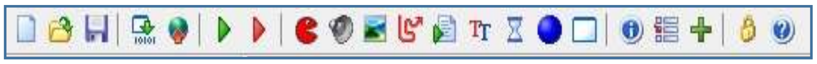
Window Menu
In this menu you find typical commands to manage the different property windows in the main form.
Cascade -Cascade all the windows such that each of them is partially visible.

Arrange Icons - Arrange all the icons property windows. This is useful when resizing the main form.

Close All - Close all the property windows, asking the user whether or not to save the changes made.

Help Menu
In this menu you find commands that open resources that will help you better understand Game Maker, and connect you to online resources that provide help and software upgrades. In this course, you will build “Your First Game”, which is a tutorial accessible under the Help menu.

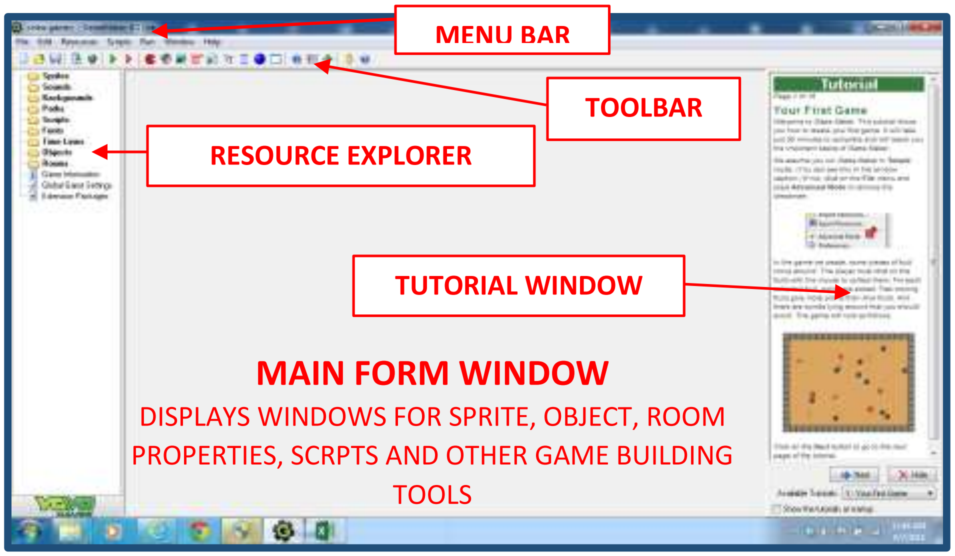
Tools and Resources in Game Maker: The Toolbar



The toolbar is a horizontal bar located below the menu bar containing graphic buttons or icons which open frequently used *tools and resources* in the Game Maker application. Commonly used tools and resources available from the Menu bar can be accessed with these shortcut buttons. These include create sprite, create object, create sound, and more.

Global User Interface in Game Maker

The *Game Maker* software engine has many tools for the creation of computer games. Your favorite computer games probably have many features that make it fun, including objects that you can control, various levels of challenge, awesome visual and sound effects, and much more. *Game engines* like *Game Maker* have many of the game building tools needed to build such fully inter-active games. *Game Maker* tools and resources are accessed and used in five different areas of the *Global User Interface* as labeled on the right.


MENU BAR
TOOLBAR
RESOURCE EXPLORER
TUTORIAL WINDOW
MAIN FORM WINDOW
DISPLAYS WINDOWS FOR SPRITE, OBJECT, ROOM PROPERTIES, SCRPTS AND OTHER GAME BUILDING TOOLS

Animation in Game Maker

There are numerous ways that *animation* can be incorporated into your games. One is to use or create assets that have animated properties built into the asset itself. When sprites and objects are made from these assets, the object instances will animate as they show in a room.

Animated Strips

For example, you can create a *PNG file (Portable Network Graphic)* with an animated strip that has many sub-images, or frames, with your object design in a different position for each frame. This is similar to making drawings of stick figures on small pieces of paper and flipping through them to create movement.
Game Maker has the capability of seeing the different images and flipping through them automatically to create the effect of natural movements like walking, jumping, flapping of wings, etc. When you see a PNG strip in an asset folder, all of the small sub-images will be visible in the icon. The sub-images can be viewed and edited in Game Maker's sprite editor, with numbering starting at image 0, then image 2, 3, and so on based on the total number of sub-images. *See PNG graphic on next page.*

Graphic Interchange Format (GIF)

If you use or build an asset that is a *GIF file* type *(Graphics Interchange Format)*, the animation itself is coded within the asset, unlike PNG which is a single graphic with images in a strip. Sub-images are compressed into the file (more like a video), along with all of the control data needed to open it and play the animation. A GIF icon will only show one image instead of a strip. However, like PNG images, the sub-images can be viewed and edited in Game Maker's sprite editor. *See GIF graphic on next page*.

Events and Actions for Animation

Animations of both of GIF and PNG file types can be made into sprites and objects and then given properties for events and actions. The example below uses a PNG strip to make a sprite and object, with Change Sprite actions to change the object from a static (unanimated) appearance to an animation. The first step is to add a Create event with a Change Sprite action to create the unanimated, or static (standing still), appearance of your object. In the Change Sprite properties, you select the sprite that you want and then the sub-image for the static object. Enter the sub-image number for the static position that you want. If you have eight sub-images, they are typically numbered 0 through 7 when you inspect them in the sprite editor. Typically, you want the static image to be the first image in the strip, which would be image 0, although you can choose another sub-image number. When the instance appears in the room, it will only show the entered sub-image number and the rest will be skipped. You won't want any movement so the speed should be set to 0.

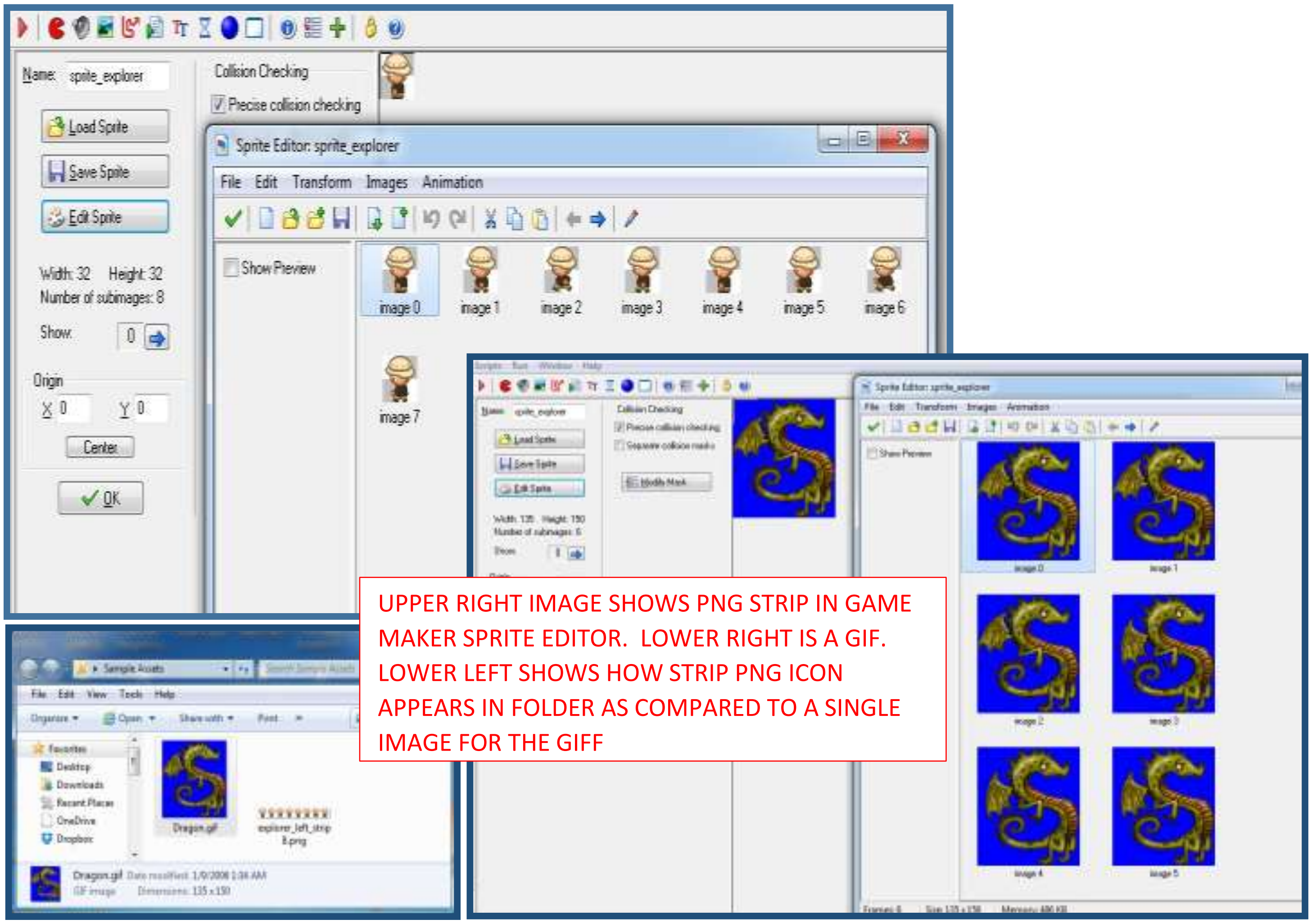
Next, you would have to add a Key Press (or Keyboard) event with another Change Sprite action. You would then select the same sprite, then enter -1 for the sub-image, which will cause the action to animate through all sub-images in the set. The speed should be set to a value greater than 0, although setting it to 1 will allow the animation to capture all of the sub-images. If the speed is greater than 1, sub-images will be skipped, although the object will animate faster. If it the speed is smaller than 1, sub-images will be shown multiple times slowing down the movement. Never use a negative speed. You can add a Move Fixed or Move Free action to your key events so that the object will move left, right, or other directions while animating. You may even want to change the sprite depending on the direction in which it moves, in other words facing left, right, up, down, jumping up, etc. based on the assigned arrow key. This can be achieved by making different sprites for each of the (four) directions. This same animation technique can be used with other events such as collisions, step, intersect boundaries, etc. *See properties graphic on next page*.

Naming conventions for animated strips should include the total number of sub-images. This sample has eight.

Eg. ***explorer_left_strip8.png***

Animation in Game Maker continued on next page

Animation in Game Maker *(continued)*



Name: sprite_explorer

Collision Checking

☑ Precise collision checking

Width: 32 Height: 32

Number of subimages: 8

Show: 0

Origin X 0 Y 0

Sprite Editor: sprite_explorer

image 0, image 1, image 2, image 3, image 4, image 5, image 6, image 7

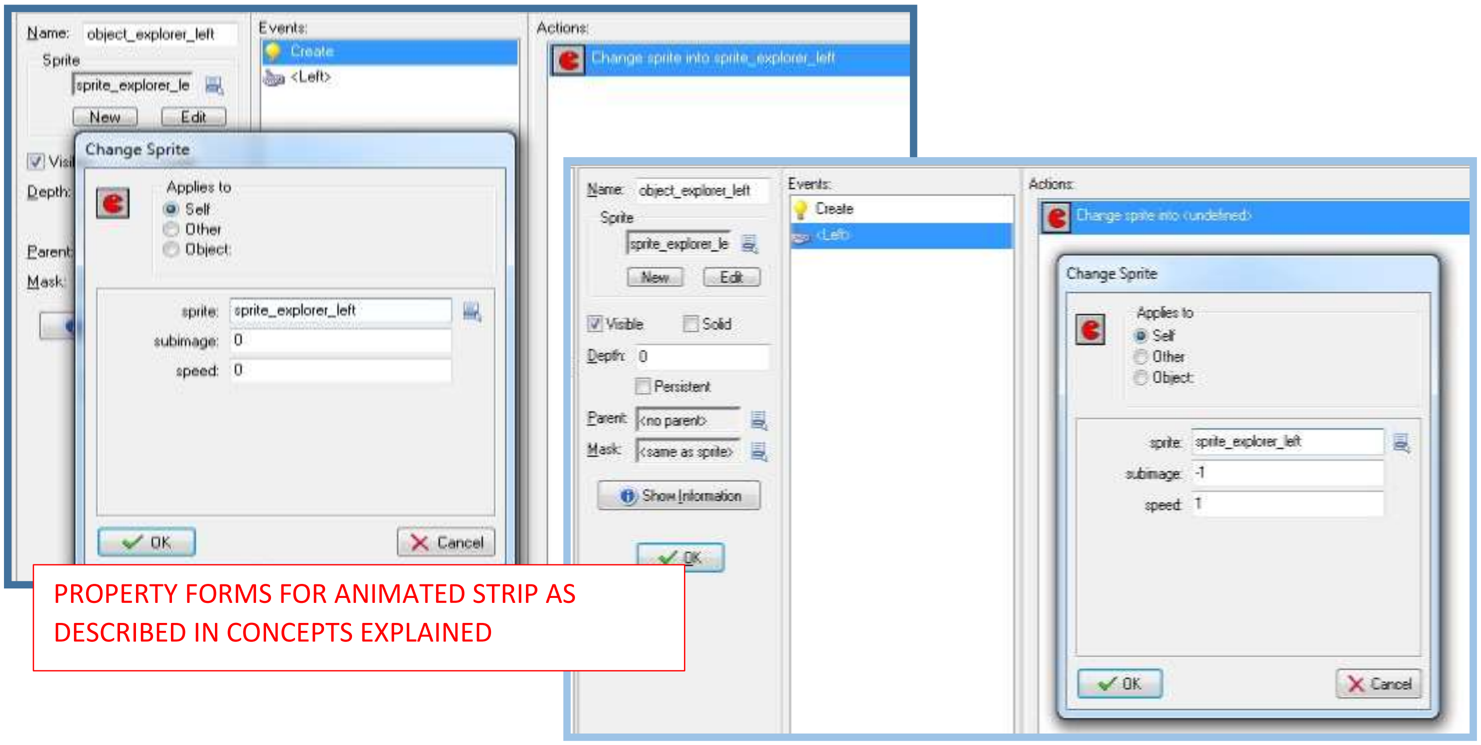
Width: 135 Height: 150

Number of subimages: 6

Dragon.gif explorer_left_strip8.png

Dragon.gif Date modified: 1/9/2008 1:36 AM GIF image Dimensions: 135 x 150

UPPER RIGHT IMAGE SHOWS PNG STRIP IN GAME MAKER SPRITE EDITOR. LOWER RIGHT IS A GIF. LOWER LEFT SHOWS HOW STRIP PNG ICON APPEARS IN FOLDER AS COMPARED TO A SINGLE IMAGE FOR THE GIFF



Name: object_explorer_left

Sprite: sprite_explorer_le

Events: Create, <Left>

Actions: Change sprite into sprite_explorer_left

Change Sprite

Applies to: ● Self ○ Other ○ Object

sprite: sprite_explorer_left

subimage: 0

speed: 0

Name: object_explorer_left

Sprite: sprite_explorer_le

☑ Visible ☐ Solid

Depth: 0

☐ Persistent

Parent: <no parent>

Mask: <same as sprite>

Events: Create, <Left>

Actions: Change sprite into <undefined>

Change Sprite

Applies to: ● Self ○ Other ○ Object

sprite: sprite_explorer_left

subimage: -1

speed: 1

PROPERTY FORMS FOR ANIMATED STRIP AS DESCRIBED IN CONCEPTS EXPLAINED