



PART 4



# Multiplayer Games

**S**pread a little happiness by bringing a friend along for the ride. Then demonstrate your complete superiority by kicking their butt.



## CHAPTER 9



# Cooperative Games: Flying Planes

Up to now we've only created single-player games, but for the next two chapters we'll be creating games that are played with a friend. This chapter's game will require players to cooperate in order to succeed, while the next will make players compete against one another for ultimate supremacy. Cooperative multiplayer games challenge players to work together and sometimes even make sacrifices for each other in order to succeed in their common goals.

In this chapter we'll also use a number of new Game Maker features: we'll take an in-depth look at the use of variables and learn about using *time lines*.

### Designing the Game: Wingman Sam

We're calling this game *Wingman Sam* as it sets American and British fighter planes alongside each other in World War II. Here is a description of the game:

*At the end of World War II you and your wingman are part of an allied squadron with secret orders to intercept the dangerous General von Strauss. Unfortunately, your mission turns out to be less secret than you thought and you are soon engaged by wave after wave of enemy fighters. You'll need to work together to survive the onslaught and destroy the general's plane, so the mission will be aborted if either of your planes is destroyed.*

*One player will control their plane with the arrow keys and fire bullets with the Enter/Return key. The other player will control their plane with the A, S, D, and W keys, and will fire bullets with the spacebar. Enemy planes will appear from the front, the sides, and behind. Some will just try to ram you while others will shoot at you. Both of the player's planes can only take a limited amount of damage before they are destroyed.*

*The game consists of just one level that takes place over an ocean scene. It will present you with increasingly difficult waves of planes and end with a battle against the infamous general himself. The game is won only if both players survive this final battle. See Figure 9-1 for a screenshot.*



**Figure 9-1.** This is how the Wingman Sam game will look in action.

All resources for this game have already been created for you in the [Resources/Chapter09](#) folder on the CD.

## Variables and Properties

Before we begin creating the Wingman Sam game, let's take a moment to consolidate our understanding of variables in Game Maker. This simple concept can provide a very powerful mechanism for creating more interesting gameplay. We've already used them in several games, but what actually is a variable? Essentially, a variable is just a place for storing some kind of information, such as a number or some text. Most variables you will use in Game Maker store information about a numeric property of an instance. There are certain properties we can set when we define an object, such as whether it is visible or solid. Other properties store information that is different for each individual instance, such as its x- and y-position in the room. There are also a number of global properties, like the score, that are not related to individual instances. Each variable has its own unique name, which we can use to retrieve or change the value of that variable in Game Maker. Here are some important variables that every instance has—some of them should look familiar, as we have already used them before:

- `x` is the x-coordinate of the instance in the room.
- `y` is the y-coordinate of the instance in the room.
- `hspeed` is the horizontal speed of the instance (in pixels per step).

- `vspeed` is the vertical speed of the instance (in pixels per step).
- `direction` is the instance's current direction of motion in degrees (0–360 anticlockwise; 0 is horizontally to the right).
- `speed` is the instance's current speed in the `direction`.
- `visible` determines whether the object is visible or invisible (1=visible, 0=invisible).
- `solid` determines whether the object is solid or not solid (1=solid, 0=not solid).

---

**Note** Different variables employ different conventions for the meaning of the information that they store. A common convention for variables that either have a property or don't have a property is to use the values `0` and `1`. So if an instance is visible then its `visible` property will be set to `1`, whereas if it is invisible then its visible property will be set to `0`. This convention is so common that you can use the keyword `true` in place of the value `1` and the keyword `false` in place of the value `0`.

---

And here are some important global variables:

- `score` is the current value of the score.
- `lives` is the current number of lives.
- `mouse_x` is the x-position of the mouse.
- `mouse_y` is the y-position of the mouse.
- `room_caption` is the caption shown in the window title.
- `room_width` is the width of the room in pixels.
- `room_height` is the height of the room in pixels.

You can refer to an instance's variables from within its own actions by entering the names in their basic form shown here. Retrieving or changing an instance's variables in this way will only affect the instance concerned. To refer to variables in other instances, you need to use an object name followed by a dot (period/full stop) and then the variable name, such as `object_specialmoon.x` (the x-coordinate of the special moon object). When you use the name of an object to retrieve a variable in this way, Game Maker will give you the value of the first special moon instance's variable—ignoring any other instances there might be in the game. However, if you change the variable `object_specialmoon.x`, then it will change the x-coordinate of *all* special moon instances in the game! Game Maker also includes some special object names that you can use to refer to different instances in the game:

- `self` is an object that refers to the current instance. It is usually optional as `self.x` means the same as just `x`.
- `other` is an object that refers to the other instance involved in a collision event.

- `all` is an object that refers to all instances, so setting `all.visible` to `0` would make all instances of all objects invisible.
- `global` is an object used to refer to global variables that you create yourself.

---

**Note** You should only use the global object to refer to global variables that you create yourself. You do not use the global object to refer to built-in global variables, like `score` and `lives`. So `global.score` is not the same as `score` and would refer to a different variable. Global variables are particularly useful when you want to refer to the same variable in different instances or in different rooms, as they keep their values between rooms.

---

There are many, many more global and local instance variables, all of which can be found in the Game Maker documentation. There are actions to test the values of variables as well as manipulating them directly. You can even define variables for your own purposes as well. For example, the planes in Wingman Sam can only survive a certain amount of enemy fire, so each needs its own property to record the amount of damage it has taken. We'll create a new variable for this property called `damage`. The plane's **Create** event will set this variable to `0`, and we'll increase it when the plane is hit. Once the damage is greater than `100`, the plane will be destroyed.

---

**Caution** A variable's name can only consist of letters and the underscore symbol. Variable names are case-sensitive, so `Damage` and `damage` are treated as different variables. It is also important to make sure that your variable names are different from the names of the resources; otherwise Game Maker will become confused.

---

Two important actions that deal with variables directly are found in the **control** tab:



The **Set Variable** action changes the value of any variable, by specifying the **Variable** name and its new **Value**. Setting a variable using a name that does not exist will create a new variable with that name and value. Enabling the **Relative** option will add the value you provide to the current value of the variable (a negative value will subtract). However, the **Relative** option can only be used in this way if the variable already has a value assigned to it! You can also type an *expression* into the **Value**. For example, to double the score you could enter the **Variable** `score` and a **Value** of `2*score`.



The **Test Variable** action tests the value of any variable against selected criteria and then only executes the next action or block of actions if the test is true. The test criteria can be whether a variable is *equal to*, *smaller than*, or *larger than* a given value.

Don't worry if this seems like a lot of information to take in at once—we'll make use of many of these concepts creating the Wingman Sam game, so you'll have a chance to see how it all works in practice.

## The Illusion of Motion

The style of game we are creating in this chapter is often referred to as a *Scrolling Shooter*. This style takes its name from the way that the game world scrolls horizontally or vertically across the screen as the game progresses. Although the player's position on the screen remains fairly static, the scrolling creates the illusion of continuous movement through a larger world.

Game Maker allows us to create scrolling backgrounds by using a tiling background image that moves through the room.

### Creating a room with a scrolling background:

1. Start up Game Maker and begin a new empty game.
2. Create a background resource called `background` using the file `Background.bmp` from the `Resources/Chapter09` folder on the CD.
3. Create a new room called `room_first` and give it an appropriate caption.
4. Switch to the **backgrounds** tab and select the new background from the menu icon, halfway down on the left.
5. At the bottom of the tab set **Vert Speed** to `2` to make the background scroll slowly downward.

Run the game, and you'll see that the background continually scrolls downward. To enhance the look and feel, we're going to add a few islands in the ocean. An easy way to do this would be to create a larger background image and add the island images to this background. The disadvantage of this approach would be that the islands would appear in a regular pattern, which would soon become obvious to the player. Consequently, we'll choose a slightly more complicated approach: using island objects that move down the screen at the same speed as the background. When they fall off the bottom, they'll jump to a random position across the top of the screen again so that they look like a new island in a different position.

### Creating looping island objects:

1. Create sprites called `spr_island1`, `spr_island2`, and `spr_island3` using `Island1.gif`, `Island2.gif`, and `Island3.gif` from the `Resources/Chapter09` folder on the CD.
2. Create a new object called `obj_island1` using the first island sprite. Set **Depth** to `10000` to make sure that it appears behind all other objects.
3. Add a **Create** event and include the **Move Fixed** action with a downward direction and a **Speed** of `2`. This will make it move at exactly the same speed as the background.
4. Add an **Other, Outside room** event and include a **Test Variable** action. Set **Variable** to `y`, **Value** to `room_height`, and **Operation** to **larger than**. `room_height` is a global variable that stores the height of the room in pixels. Therefore, this will test for when the island has fallen off the bottom of the screen by checking whether the vertical position of the island is larger than this value (see Figure 9-2).







5. Add a **Jump to Position** action with X set to `random(room_width)` and Y set to `-70`. Using the global variable `room_width` with the `random()` command will provide a random number that does not exceed the width of the room. This will move the island to a random horizontal position just above the top of the room where it is out of sight.
6. Create objects for the other two islands and set the first island to be their parent.
7. Add one instance of each of the island objects to `room_first` at different vertical positions in the room.



**Figure 9-2.** This action tests whether the island is below the visible edge of the room.

Now test the game. The scrolling sea should contain three islands, which reappear back at the top of the screen after disappearing off the bottom. Because the island instances move at exactly the same speed as the background, they appear as if they are part of it.

## Flying Planes





Our scrolling background is complete, so it's time to create the planes that the players will control. These planes will have nearly the same behavior as each other, but there will need to be a few differences, such as the controls. To avoid duplicated work, we'll use the parent mechanism and three different plane objects. `obj_plane_parent` will contain all the behavior common to both planes, `obj_plane1` will be player one's plane, and `obj_plane2` will be player two's plane. Both of these last two objects will have `obj_plane_parent` as their parent so that they can inherit its behavior. Let's create the basic structure for these objects.

**Creating the plane objects:**

1. Create sprites called `spr_plane1` and `spr_plane2` using `Plane1.gif` and `Plane2.gif` from the `Resources/Chapter09` folder on the CD. Set the **Origin** of both sprites to **Center**. This is important for creating bullets and explosions later on.
2. Create a new object called `obj_plane_parent`. It doesn't need a sprite, and we'll come back to create events and actions for it later.
3. Create another object called `obj_plane1` and give it the first plane sprite. Set **Depth** to `-100` to make sure that it appears above other objects and set its **Parent** to be the `obj_plane_parent`.
4. Create a similar object called `obj_plane2` using the second plane sprite. Set **Depth** to `-99` to make sure that it appears above other objects (apart from the first plane). Also set its **Parent** to be `obj_plane_parent`.

The players will be able to move their planes around most of the screen using their own movement keys, but should be prevented from going outside of the visible area. When the player isn't pressing any keys, the plane will not move, but will still appear to be cruising along because of the scrolling background. We'll control the position of the planes manually (rather than setting their speed) so that we can easily prevent them from leaving the boundaries of the room.

**Adding movement keyboard events to player one's plane object:**

1. Reopen the properties form for `obj_plane1` by double-clicking on it in the resource list.
2.  Add a **Keyboard, <Left>** event and include the **Test Variable** action. Set **Variable** to `x`, **Value** to `40`, and **Operation** to **larger than**. Include a **Jump to Position** action. Set **X** to `-4` and **Y** to `0`, and enable the **Relative** option. This will now only move the plane to the left if its x-position is greater than 40, which means it must be well inside the left boundary of the screen.
3.  Add a **Keyboard, <Right>** event and include the **Test Variable** action. Set **Variable** to `x`, **Value** to `room_width-40`, and **Operation** to **smaller than**. Include a **Jump to Position** action. Set **X** to `4` and **Y** to `0`, and enable the **Relative** option. This will only move the plane right if its x-position is well inside the right boundary of the screen.
4.  Add a **Keyboard, <Up>** event key and include the **Test Variable** action. Set **Variable** to `y`, **Value** to `40`, and **Operation** to **larger than**. Include a **Jump to Position** action. Set **X** to `0` and **Y** to `-2`, and enable the **Relative** option. This will only move the plane up if its y-position is well inside the upper boundary of the screen.
5.  Add a **Keyboard, <Down>** event and include the **Test Variable** action. Set **Variable** to `y`, **Value** to `room_height-120`, and **Operation** to **smaller than**. Include a **Jump to Position** action. Set **X** to `0` and **Y** to `2`, and enable the **Relative** option. This will only move the plane down if its y-position is well inside the lower boundary of the screen.

Note that we only move the planes 2 pixels vertically in each step. Any more than this would make them move faster than the background, and it would look like the planes were flying backward! Also note that we have left a large area at the bottom of the screen where the planes cannot fly. We'll use this space later for displaying a status panel, but first we need to add similar events and actions to control the second plane.

#### Adding movement keyboard events to player two's plane object:

1. Reopen the properties form for `obj_plane2` by double-clicking on it in the resource list.
2. Add similar events with the same actions as before but this time using the A key for left, the D key for right, the W key for up, and the S key for down.

Your planes are now ready to fly! Place one instance of each of the planes in `room_first` and run the game. You should have control of both planes, within the bounds of the room, and get the illusion of passing over the sea beneath you. In case your game isn't working, you'll find a version of the game so far in the file `Games/Chapter09/plane1.gm6` on the CD.

## Enemies and Weapons

Well, there seems to be plenty of scrolling going on in our scrolling shooter, but not a lot of shooting yet! Let's rectify this by adding events and actions to make the planes fire bullets and create some enemies for them to shoot at while we're at it. We'll start by creating the bullet object.

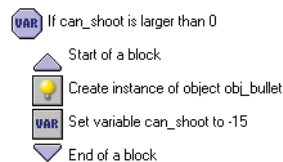
#### Creating the bullet object:

1. Create a new sprite called `spr_bullet` using `Bullet.gif` from the `Resources/Chapter09` folder on the CD. Set the sprite's **Origin** to the **Center** of the sprite.
2. Create a new object called `obj_bullet` and give it the bullet sprite.
3. Add a **Create** event and include the **Move Fixed** action. Select the up arrow and set the **Speed** to 8.
4. Add the **Other, Outside room** event and include the **Destroy Instance** action.

Deciding when to create a bullet instance is a little more complicated. In this game we want it to be possible for the player to keep the fire key pressed and create a continuous stream of bullets. Nonetheless, creating a new bullet every step would create way too many bullets (30 every second!). To limit the rate at which bullets appear we'll create a new variable called `can_shoot`. We'll only allow the player to shoot when `can_shoot` has a value larger than 0, and once a shot has been made we'll set `can_shoot` to -15. We'll then increase the value of `can_shoot` by 1 in each step so that it will be 15 steps (half a second), before the player can shoot again. As this behavior needs to be largely the same for both planes, we'll put most of it in the parent plane object.

**Adding shooting keyboard events to the plane objects:**

1. Reopen the properties form for `obj_plane_parent` by double-clicking on it in the resource list.
2. Add a **Create** event and include the **Set Variable** action (**control** tab). Set **Variable** to `can_shoot` and **Value** to `1`. Using **Set Variable** for the first time with a new variable name creates that new variable. Subsequent **Set Variable** actions may then use the **Relative** option to add and subtract from it.
3. Add a **Step, Step** event and include the **Set Variable** action with **Variable** set to `can_shoot`, **Value** set to `1`, and the **Relative** option enabled.
4. Now reopen the properties form for `obj_plane1`. Add a **Keyboard, <Enter>** event and include the **Test Variable** action. Set **Variable** to `can_shoot`, **Value** to `0`, and **Operation** to **larger than**. Include the **Start Block** action to make all the following actions depend on this condition.
5. Include the **Create Instance** action and select the bullet object. Set **X** to `0` and **Y** to `-16`, and enable the **Relative** option to create the bullet relative to the plane's position. Include the **Set Variable** action, with **Variable** set to `can_shoot` and **Value** set to `-15`.
6. Finally, include the **End Block** action. The event should look like Figure 9-3.
7. Repeat steps 4–6 for `obj_plane2`, this time using the **Keyboard, <Space>** event for the fire key.



**Figure 9-3.** These are the actions required for shooting a bullet.

To make the gameplay a bit more interesting, we'll also allow the player to shoot bullets more quickly if they repeatedly press the fire button. So when the player releases the fire button, we'll add 5 to the `can_shoot` variable.











**Adding key release events to the plane objects:**

1. Reopen the properties form for `obj_plane1`. Add a **Key Release, <Enter>** event and include the **Set Variable** action. Set **Variable** to `can_shoot` and **Value** to `5`, and enable the **Relative** option.
2. Reopen the properties form for `obj_plane2`. Add a **Key Release, <Space>** event and include the **Set Variable** action. Set **Variable** to `can_shoot` and **Value** to `5`, and enable the **Relative** option.

It might be wise to run the game now and make sure that this all works correctly before proceeding. Carefully check through your steps if there is a problem.


Now it's time to create our first enemy. This will be a small plane that simply flies down the screen and ends the game if it collides with one of the players (we'll add health bars later). The player's bullets will destroy the enemy and increase the player's score.

#### Creating an enemy plane object along with its sprites and explosions:

1. Create sprites called `spr_enemy_basic`, `spr_explosion1`, and `spr_explosion2` using `Enemy_basic.gif`, `Explosion1.gif`, and `Explosion2.gif` from the `Resources/Chapter09` folder on the CD. Set the **Origin** of all the sprites to the **Center**.
2. Create sounds called `snd_explosion1` and `snd_explosion2` using the files `Explosion1.wav` and `Explosion2.wav` from the `Resources/Chapter09` folder on the CD.
-  3. Create an object called `obj_explosion1` and give it the first explosion sprite. Add a **Create** event and include the **Play Sound** action to play the first explosion sound.
-  4. Add the **Other, Animation end** event and include the **Destroy Instance** action.
-  5. Create an object called `obj_explosion2` and give it the second explosion sprite. Add a **Create** event and include a **Play Sound** action to play the second explosion sound.
-  6. Add an **Other, Animation end** event and include the **Sleep** action. Set **Milliseconds** to **1000** and set **Redraw** to **false**. Include the **Restart Game** action directly afterward.
-  7. Create an object called `obj_enemy_basic` and give it the basic enemy sprite. Add a **Create** event and include the **Move Fixed** action with a downward direction and a **Speed** of **4**.
-  8. Add an **Other, Outside room** event and include the **Test Variable** action. Use it to test whether **y** is **larger than** `room_height` and include a **Destroy Instance** action after it. This will destroy the enemy plane when it reaches the bottom of the room.
-  9. Add a **Collision** event with the bullet object and include a **Set Score** action with a **Value** of **10** and the **Relative** option enabled. Include a **Destroy Instance** action to make the enemy object destroy itself.
-  10. Now include the **Create Instance** action. Set **Object** to `obj_explosion1` and enable the **Relative** option so that it is created at the enemy's position. Finally for this event, add a **Destroy Instance** action to destroy the bullet (the **Other** object in this collision).
-  11. Add a **Collision** event with the parent plane object and include a **Destroy Instance** action to make the enemy object destroy itself. Include a **Create Instance** action with **Object** set to `obj_explosion1` and the **Relative** option enabled.
-  12. Include a **Destroy Instance** action to destroy the player's plane object (the **Other** object). Finally, include a **Create Instance** action with **Object** set to `obj_explosion2` and the **Relative** option enabled. Also select **Other** for **Applies to** so that the explosion is created at the position of the player's plane—not the enemy's.

This gives us a working enemy plane object, but we still need a mechanism to create enemy planes in the first place. To begin with, we'll do this in a controller object and randomly generate enemy planes about every 20 steps.

#### Creating a controller object:

1. Create a new object called `obj_controller`. It doesn't need a sprite.
2.  Add a **Step, Step** event and include a **Test Chance** action with **Sides** set to 20. Follow this with a **Create Instance** action, with **Object** set to `obj_enemy_basic`. Set **X** to `random(room_width)` and **Y** to -40. This will create an enemy plane instance at a random position just above the top of the room.
3. Add one instance of this controller object to the room.

This gives us the first playable version of our game. Recruit a willing volunteer to play with and check that everything is working okay so far. You can also find this version in the file `Games/Chapter09/plane2.gm6` on the CD.

---




**Note** You can easily change this into a single-player game by removing the second plane from the room.

---

## Dealing with Damage

The current version of the game ends as soon as one of the player's planes is hit by the enemy. We saw from the versions of Evil Clutches in Chapter 5 that this works better if we use a health bar that can absorb several hits instead. If you were curious enough to look and see how this was done, then you will have noticed that Game Maker provides simple actions to control and display the player's health. However, these actions only work for recording and displaying the health of one player, and in Wingman Sam we have two. Consequently, we'll create a new variable called `damage` for each player, and use this to record and display the status of their health independently. As the name suggests, each plane's `damage` will be set to 0 at the start of the game and increase slightly for each collision with enemy planes or bullets. If a plane's health gets larger than 100, then it explodes and the game is over. We'll update the `damage` variable in the parent plane object, because it will work the same for both players.

#### Adding a damage variable to the parent plane object:

1.  Reopen the properties form for `obj_plane_parent` and select the **Create** event. Include a **Set Variable** action, setting **Variable** to `damage` and **Value** to 0.
2.  Select the **Step** event and include a **Test Variable** action. Set **Variable** to `damage`, **Value** to 100, and **Operation** to **larger than**. Follow this with a **Start Block** action to begin a block of events.
3.  Next include a **Destroy Instance** action to destroy the player's plane. Also include a **Create Instance** action that creates an instance of `obj_explosion2` **Relative** to the position of the plane.

4. Finally, include an **End Block** action.
5. Reopen the properties form for `obj_enemy_basic` and select the **Collision** event with `obj_plane_parent`. Remove the last two actions that deal with destroying the plane and creating the second explosion.
6. Include a **Set Variable** action with **Variable** set to `damage`, **Value** set to `10`, and the **Relative** option enabled. Also select the **Other** object from **Applies to** so that it increases the player's `damage` variable (rather than the enemy's, which doesn't exist!).









If you try running the game now, each plane should take about 10 hits before it explodes and the game ends (it actually takes 11—can you think why?). Nonetheless, this is a little hard to keep track of in your head, so clearly we need to display each player's current damage for them to see. To this end we're going to add a panel at the bottom of the screen and draw over the top of it. The controller object will be responsible for showing the panel, which will look something like Figure 9-4. We'll use a number of Game Maker's drawing actions to draw the panel, damage bars, and the score over the top.



**Figure 9-4.** The information panel will display the damage of each plane and the combined score.





#### Creating the panel sprite and object:

1. Create a sprite called `spr_panel` using `Panel.gif` from the `Resources/Chapter09` folder on the CD.
2. Create a font resource called `fnt_panel` for displaying the panel text (just like you would any other resource). Set **Font** to **Arial** and **Size** to `14`, and enable the **Bold** option, or choose a completely different font if you prefer.
3. Reopen the properties form for the controller object and set its **Depth** to `-100`. This will make the panel appear in front of enemy planes.
4. Add a **Draw** event and include the **Draw Sprite** action (**draw** tab). Set **Sprite** to `spr_panel`, **X** to `0`, and **Y** to `404`. This will draw the panel at the bottom of the screen.
5. Include the **Set Font** action and set **Font** to `fnt_panel`. Actions that draw text will now use this font.
6. Include a **Set Color** action and choose a green color, as this is the color of the first player's plane. Actions that draw graphics or text will now do so in this color.
7. Include a **Draw Text** action, setting **Text** to `Damage 1:`, **X** to `20`, and **Y** to `420`.

8.  Include a **Draw Rectangle** action. **X1** and **Y1** refer to the top-left corner of the rectangle and **X2** and **Y2** refer to the bottom-right corner. Working down the form in order, set these to 130, 420, 230, and 440. Also set **Filled** to **outline**. This will draw a green box around the edge of the damage bar for player one.
9.  Before we use each plane object's **damage** variable, we need to check that it hasn't already died and been deleted; this would mean Game Maker couldn't access the variable and would produce an error. Include a **Test Instance Count** action. Set **Object** to **obj\_plane1**, **Number** to 0, and **Operation** to **larger than**. The next action will now only be executed if the first plane exists.
10.  Include a **Draw Rectangle** action. Set **X1** to 130, **Y1** to 420, **X2** to  $130 + \text{obj\_plane1.damage}$ , and **Y2** to 440, and set **Filled** to **filled**. This will draw a filled rectangle with a length equal to the **damage** variable of **obj\_plane1**.
11.   Now we'll do the same for the second plane. Include the **Set Color** action and choose a reddish color. Include a **Draw Text** action, setting **Text** to **Damage 2:**, **X** to 20, and **Y** to 445.
12.  Include the **Draw Rectangle** action. Set **X1**, **Y1**, **X2**, and **Y2** to 130, 445, 230 and 465, respectively, and set **Filled** to **outline**.
13.   Include the **Test Instance Count** action, setting **Object** to **obj\_plane2**, **Number** to 0, and **Operation** to **larger than**. Follow this with a **Draw Rectangle** action, setting **X1** to 130, **Y2** to 445, **X2** to  $130 + \text{obj\_plane2.damage}$ , and **Y2** to 465. Also set **Filled** to **filled**.

We'll also use the panel, rather than the window caption, to display the player's score, and get the controller object to show a high-score table when the game is over.

#### Displaying the score and high-score table:

1. Add a new background called **background\_score** using the file **Score.bmp** from the **Resources/Chapter09** folder on the CD.
2. Reopen the properties form for the controller object and select the **Draw** event.
3.   Include a **Set Color** action at the end of the list of actions and select a bluish color. Include a **Draw Score** action (**score** tab), setting **X** to 350 and **Y** to 430.
4.  Add a **Create** event and include the **Score Caption** action (**score** tab). Change **Show Score** to **don't show** to stop Game Maker from displaying the score in the window caption.
5. Reopen the properties form for **obj\_explosion2** and select the **Animation end** event.
6.  Include the **Show Highscore** action between the **Sleep** and **Restart Game** actions. Set **Background** to **background\_score**, set **Other Color** to yellow, and choose a nice font.

Now test the game to check that the damage and score are shown correctly on the new panel. This version can also be found in the file **Games/Chapter09/plane3.gm6** on the CD.



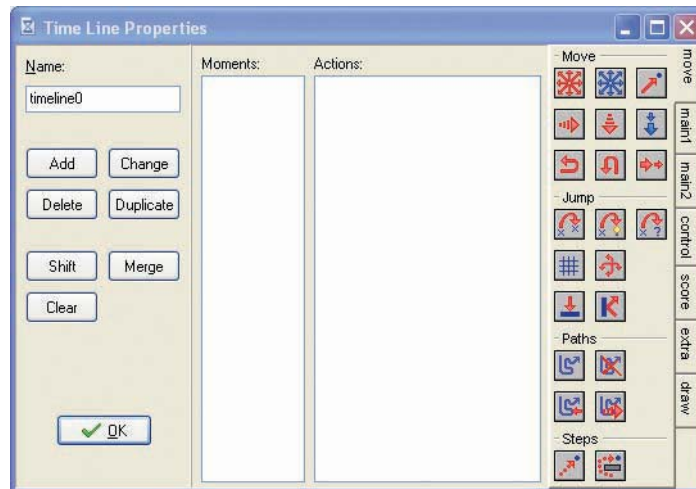
## Time Lines

So far all the enemy planes have been generated randomly, but this doesn't tend to create interesting gameplay. Therefore, we will use Game Maker's time line resource to create waves of enemy formations that start off easy and gradually grow more difficult through the level. A time line allows actions to be executed at preset points in time, so we can use them here to determine when enemy planes are created.

We'll begin with a very simple example of a time line. We'll need to create a new time line resource and indicate when we want actions to be performed. This should feel familiar as a time line resource has a list of **Moments** and their **Actions** similar to an object resource's list of **Events** and their **Actions** (see Figure 9-5). Then we'll need to set the time line running by using the **Set Time Line** action.

### Creating a new time line resource and starting it running:

1. From the **Resources** menu choose **Create Time Line**. A properties form will appear like the one shown in Figure 9-5. The buttons on the left allow moments to be added and removed from the **Moments** list in the middle of the form. Each moment can have its own list of actions, which can be added in the usual way from the tabbed pages of action icons on the right. Call the new time line `time_level1`.



**Figure 9-5.** A time line has a list of moments and actions.

2. Click the **Add** button to add a new moment. Type `60` in the prompt that appears and click **OK**. Moments are measured in steps, so this creates a moment that happens after 2 seconds and adds it to the **Moments** list.

3. Include the **Create Instance** action (**main1** tab) in the **Actions** list for this moment. Set **Object** to `obj_enemy_basic`, **X** to `80`, and **Y** to `-40`. This is the first of many new instances we will create in the time line, so from now on we will refer to their **X** and **Y** positions using the shorthand form: **(X, Y)**, which would be `(80, -40)` in this case.
4. Add similar actions in the same moment to create enemy planes at the following positions: `(200, -40)`, `(320, -40)`, `(440, -40)`, and `(560, -40)`. This will create a horizontal row of five planes at the top of the screen, two seconds into the level.
5. Reopen the properties form for the controller object and select the **Create** event. Include a **Set Time Line** action from the **main2** tab and set **Time Line** to the one we just created. Leave **Position** set to `0` so that the time line starts at the beginning. The action should now look like Figure 9-6.
6. Select the **Step** event and remove the two actions, as we no longer want to generate random enemies.

---

**Note** Each instance in the game can only have one time line set on it at a time. Setting a new time line replaces the old one and setting a time line to **no time line** stops the execution of the current time line.

---



**Figure 9-6.** The **Set Time Line** action is used to start a time line.

Now run the game, and you should find that a row of enemy planes appears after about two seconds. These are the only enemies that ever appear, so clearly we need to add a lot more moments to the time line to make an interesting level! By giving planes different movement directions, we can also make formations that fly horizontally or diagonally. But before we can do that we'll have to create some more types of enemy planes.

## More Enemies

In this section we'll create several types of enemy planes with varying behaviors. First, we'll create enemy planes that come from the left, right, and bottom of the screen. These are harder to avoid and more difficult to shoot.




### Creating new enemy plane objects:


1. Create sprites for the new enemies using `Enemy_right.gif`, `Enemy_left.gif`, and `Enemy_up.gif` from the `Resources/Chapter09` folder on the CD. Set the **Origin** of each sprite to the **Center**.
2. Right-click on `obj_enemy_basic` in the resource list and select **Duplicate**. Call the duplicate object `obj_enemy_right` and give it the right-facing enemy sprite.
3. Select the **Create** event for the new object and double-click on the **Move Fixed** action in the **Actions** list. Select the right movement arrow instead of the down arrow and close the action properties again. Select the **Outside room** event and double-click on the **Test Variable** action in the **Actions** list. Change **Variable** to `x` and **Value** to `room_width`, to test for when the plane is off the right edge of the screen.
4. Repeat steps 2 and 3 to create `obj_enemy_left`, which moves left and tests for `x` being **smaller than 0**.
5. Repeat steps 2 and 3 to create `obj_enemy_up`, which moves up and tests for `y` being **smaller than 0**.



Add some new moments and actions to the time line to test these new enemy planes. Leave an appropriate pause between waves so that the step for each moment is between 100 and 200 steps more than the last. You'll need to create instances of `obj_enemy_right` just to the left of the room (`X` less than 0 and `Y` between 0 and 360), `obj_enemy_left` just to the right of the room (`X` greater than 640 and `Y` between 0 and 360) and `obj_enemy_up` just below the room (`X` between 0 and 640 and `Y` greater than 360). As you will see, this last type of plane is particularly nasty and difficult to avoid.



The two other enemy planes that we're going to create will shoot bullets. One will shoot bullets in a straight line, while the other will direct bullets toward the player's planes. We'll start by creating three types of bullets: one that moves downward and two others that move toward each of the player's planes.

### Creating new enemy bullet objects:

1. Create a new sprite called `spr_enemy_bullet` using `Enemy_bullet.gif` and set the **Origin** to the **Center**. Create an object called `obj_enemy_bullet` and give it the enemy bullet sprite.
2.  Add an **Other, Outside room** event and include a **Destroy Instance** action to destroy the bullet.
3.   
 Add a **Collision** event with the parent plane object and include a **Destroy Instance** action followed by a **Play Sound** action to play `snd_explosion1`.

- 



4. Include a **Set Variable** action and select **Other** for **Applies to** (the plane object). Set **Variable** to `damage` and **Value** to `5`, and enable the **Relative** option.
- 5. Create an object called `obj_enemy_aim1` and give it the same bullet sprite. Set the **Parent** to `obj_enemy_bullet`. This bullet will move toward player one's plane, so we need to check that it exists as we did in the control panel.
- 




6. Add a **Create** event and include the **Test Instance Count** action. Set **Object** to `obj_plane1`, **Number** to `0`, and **Operation** to **larger than**. Follow this with the **Move Towards** action (**move** tab) so that it is only executed if player one's plane exists. Set **X** to `obj_plane1.x`, **Y** to `obj_plane1.y`, and **Speed** to `8` so that it targets player one's plane.
- 


7. Include an **Else** action followed by a **Move Fixed** action with a downward direction and a **Speed** of `6`. This will make the bullet move straight down if the player's object doesn't exist.
- 8. Finally, duplicate the `obj_enemy_aim1` object and call it `obj_enemy_aim2`. Select the **Create** event and edit the **Test Instance Count** action to set **Object** to `obj_plane2`. Click on the **Move Towards** action and change the references to `obj_plane1.x` and `obj_plane1.y` to `obj_plane2.x` and `obj_plane2.y`. This bullet will now target player two's plane instead.

The aiming bullets check to see if their target exists and start moving toward that plane's current position if they do. If it doesn't exist, then they start moving straight downward. Now we need to create the enemies that will shoot these bullets. Note that we haven't yet given the normal enemy bullet a direction and speed because we're going to do this when we create instances of it.

#### Creating enemy plane objects that shoot:

- 1. Create sprites called `spr_enemy_shoot` and `spr_enemy_target` using the sprites `Enemy_shoot.gif` and `Enemy_target.gif`. Set the **Origin** of both sprites to the **Center**.
- 2. Create a new object called `obj_enemy_shoot` and give it the shooting enemy sprite. Set **Parent** to `obj_enemy_basic` as its behavior is almost the same.
- 


3. Add a **Step, Step** event and include a **Test Chance** action with **40 Sides**. Include a **Create Moving** action with **Object** set to `obj_enemy_bullet`, **X** set to `0`, and **Y** set to `16`. Enable the **Relative** option, and set **Speed** to `6` and **Direction** to `270` (downward).
- 4. Create an object called `obj_enemy_target` and give it the correct sprite and `obj_enemy_basic` as its **Parent**.
- 






5. Add a **Step, Step** event and include a **Test Chance** action with **100 Sides**. Include a **Create Instance** action with **Object** set to `obj_enemy_aim1`, **X** set to `0`, **Y** set to `16`, and the **Relative** option enabled.
- 6. Repeat step 5 to include an equal chance of creating instances of `obj_enemy_aim2` in the same way.

Add some additional moments to the time line in order to test the new enemy types. Note that you can duplicate moments. This makes it easier to repeat the same formation many times. A version of the game containing all the enemies and a simple time line to test them can also be found in the file `Games/Chapter09/plane4.gm6` on the CD.

## End Boss





Games of this type usually finish with some kind of end of level boss. This boss is often an extra strong enemy with its own damage counter and additional weapons. Our game only has one level, so defeating the boss is also the ultimate goal of the game. Our boss will be General von Strauss's plane—a large plane flying in the same direction as the players as they slowly catch up to it. It will let loose a barrage of bullets in different directions and must be hit 50 times to be destroyed. If the players survive this onslaught, then the general's plane will explode in a satisfying way and the game will end.





### Creating the boss plane object:

1. Create a new sprite called `spr_boss` using `Boss1.gif` and set the **Origin** to the **Center**.
2. Create a new object called `obj_boss` and give it the boss sprite. Set its **Depth** to `-10` so that it appears above other planes and bullets.
-  3. Add a **Create** event and include a **Move Fixed** action with a downward direction and a **Speed** of `1`. Also include a **Set Alarm** action to set **Alarm 0** to `200` steps.  

-  4. Add an **Alarm, Alarm 0** event and include the **Move Fixed** action with the middle square selected and a **Speed** of `0` (to make the boss stop moving).
-  5. Reopen the time line and add another moment at the end of the list. Include a **Create Instance** event to create `obj_boss` at `(320, -80)`.

Quickly run the game and check that the boss plane moves into sight and stops in the middle of the screen. Next we'll include a hit counter that indicates how many times the boss has been hit. We'll use a variable called `hits` to record the number of hits and destroy the boss when this reaches 50.



### Adding a hit counter to the boss object:

-  1. Select the **Create** event for the boss object and include a **Set Variable** action. Set **Variable** to `hits` and **Value** to `0`.
-  2. Add a **Collision** event with the `obj_bullet` and include a **Set Score** action with a **Value** of `2` and the **Relative** option enabled.
-  3. Add a **Create Instance** action and select the **Other** object from **Applies to** (the bullet). Set **Object** to `obj_explosion1` and enable the **Relative** option. Next include a **Destroy Instance** action and select the **Other** object from **Applies to** (the bullet).
-  4. Include a **Set Variable** action. Set **Variable** to `hits` and **Value** to `1`, and enable the **Relative** option.

-  5. Include a **Test Variable** action. Set **Variable** to `hits`, **Value** to `50`, and **Operation** to **equal to**. Include a **Start Block** action so that the next block of actions will be only executed once the boss has taken 50 hits.
-  6. Include a **Destroy Instance** action to destroy the boss object. Include a **Set Score** action with a **Value** of `400` and the **Relative** option enabled (to reward the players).
-  7. Next include five **Create Instance** actions to create instances of `obj_explosion2`, at the following **Relative** positions: `(-30, 0)`, `(30, 0)`, `(0, 0)`, `(0, -30)`, and `(0, 10)`.
-  8. Finally, include an **End Block** action.







While this does the job, the player has no way of knowing how many shots they have landed on the boss plane, or how close it is to destruction. Adding a bar to show this will help to make the player's goal and progress toward it much clearer.



#### Adding a bar to display the boss's hit counter:

-  1. Add a **Draw** event to the boss object and include a **Draw Sprite** action. Remember that object's sprite stops being drawn automatically if we add a **Draw** event, so we need to do this for ourselves. Set **Sprite** to `spr_boss` and **Subimage** to `-1` (which means keep the current subimage), and enable the **Relative** option.
-  2. Include a **Set Color** action and choose a dark red color. We will use this color to draw a bar that decreases in length by 4 pixels for each hit that the boss object takes. As it takes 50 hits to destroy it, we will need the bar to be 200 pixels wide to start with ( $50 * 4 = 200$ ). This can be achieved by including a **Draw Rectangle** action with **X1** set to `10`, **Y1** set to `5`, **X2** set to `210 - (4*hits)`, and **Y2** set to `15`.

Now it's time to make the boss fight back. Colliding with the boss should instantly kill players, and the boss itself should fire bullets in all directions. After a while we'll even make it send smaller ships out to target the player—just to make things interesting!

#### Making the boss object more challenging:

-  1. Add a **Collision** event with the parent plane object and include a **Set Variable** action. Set **Variable** to `damage` and **Value** to `101`, and select the **Other** object from **Applies to**. This will immediately destroy the plane and end the game.
-  2. Select the **Create** event and include a **Set Alarm** action for **Alarm 1** with `100` steps.
-  3. Add an **Alarm, Alarm 1** event and include the **Repeat** action (**control** tab). This will repeat the next action (or block of actions) a specified number of times. Set **Times** to `10` to repeat the next action 10 times.
-  4. Include the **Create Moving** action and set **Object** to `obj_enemy_bullet`. Set **Speed** to `6` and **Direction** to `random(360)`, and enable the **Relative** option.
-  5. Finally, include a **Set Alarm** action for **Alarm 1** with `30` steps. This makes the boss fire again in 1 second's time.
-  6. Select the **Create** event and include a **Set Alarm** action for **Alarm 2** with `250` steps.

-  7. Add an **Alarm, Alarm 2** event and include a **Create Instance** action. Create an instance of `obj_enemy_target` just below the boss's left wing by providing a **Relative** position of (-40, -10). Include another **Create Instance** action to create a second instance of `obj_enemy_target` just below the boss's right wing using a **Relative** position of (40, -10).
-  8. Finally, include a **Set Alarm** action for **Alarm 2** with 40 steps.




And that concludes the boss object! You should now be able to create a varied and interesting time line with many different waves of enemy planes. These should gradually get more and more challenging before the boss plane eventually appears on the scene for the final battle. If you want to play our version, then you'll find it on the CD in the file `Games/Chapter09/plane5.gm6`.

## Finishing Touches

All that remains is to add the final bells and whistles that turn this into a finished game. First, let's add some background music played by the controller object.

### Playing background music in the controller object:

- 1. Create a new sound resource called `snd_music` using `Music.mp3` from the `Resources/Chapter09` folder on the CD.
-  2. Reopen the controller object and select the **Create** event. Include a **Play Sound** action, with **Sound** set to `snd_music` and **Loop** set to **true**.

Now we're going to change some of the global game settings. You might have noticed that all games created so far use the standard Game Maker loading image and the same red ball icon when you create an executable. However, both of these can be changed very easily to create a more individual feel for your game. We can also make the game automatically start in full-screen mode and disable the cursor in the game.

### Editing the global game settings to change the loading image and game icon:

- 1. Double-click on **Global Game Settings** at the bottom of the resource list and select the **loading** tab.
- 2. Enable the **Show your own image while loading** option and click the **Change Image** button. Select the `Loading.gif` from the `Resources/Chapter09` folder on the CD.
- 3. Select the **No loading progress bar** option, as there is not much point in a loading bar for a game that loads so quickly.
- 4. Click the **Change Icon** button and select `Icon.ico` from the `Resources/Chapter09` folder on the CD.
- 5. Select the **graphics** tab and enable the **Start in full-screen mode** option.
- 6. Disable the **Display the cursor** option.
- 7. Click the **OK** button to close the **Global Game Settings**.

Test these changes by choosing **Create Executable** from the **File** menu. Save the executable on your desktop and you'll notice that it now has the plane icon. When you run the game, you should also see the new loading image and the game should start in full-screen mode.

There are many more useful options in the global game settings. You might want to take a look at the different tabbed pages and consult the Game Maker documentation on them. Before you finish, remember to add some help text (including the controls) in the **Game Information** section.

## Congratulations

We hope you've enjoyed creating a game that can be played with a friend. You'll find the final version on the CD in the file `Games/Chapter09/plane6.gm6`. It only has one level, so why not add some more of your own? You could create several more levels just by using the enemy planes we've already created, but obviously you can create new enemies too. You could create planes that fly diagonally, shoot more bullets, go faster, and so forth. You could also create some new end-of-level bosses as well. You might also want to add a title screen using `Title.bmp` provided for you in the resources directory. Finally, you could add some bonus objects that repair the damage of the plane or provide additional firing power.

Most of the graphics for this game come from Ari Feldman's collection, which you'll find on the CD. There is one big image called `all_as_strip.bmp` that contains many different images. In the **File** menu of the Sprite Editor, there is a command called **Create from Strip**, which can be used to grab subimages out of the big image (search for "Strips" in the Game Maker help for more details). The big image also contains a boat and a submarine that you can use to create ground targets.

The main new concept that you learned in this chapter was the use of time lines. Time lines are very useful for controlling the order of different events over the course of a game. We used just one time line to control the flow of enemy planes, but you can use many different time lines simultaneously (on different objects). For example, we could have used a second time line to control the attacking behavior of the boss plane. You can even use them to create little movies using Game Maker.

Wingman Sam is an example of a game in which two players must cooperate to achieve a common goal. However, multiplayer games aren't always this amicable, and in the next chapter we'll create a game in which two players must compete with each other by trying to blow up each other's tanks!





## CHAPTER 10



# Competitive Games: Playing Fair with Tanks

**C**ombat arenas are a popular theme in multiplayer games, because they create extremely compelling gameplay from very simple ingredients. This can often just be an environment filled with weapons that the players can use to wipe each other out. The game that we're going to create in this chapter is exactly that, with futuristic battle tanks. Although games like this are relatively easy to make, care must be taken in their design to ensure that both players feel they are being treated fairly. We'll discuss this more in Chapter 11.

This game will also introduce *views* in Game Maker to help create a larger combat arena. We will also use views to create a split-screen mode, where each player can only see the part of the arena around their own tank.

## Designing the Game: Tank War

We're calling this game *Tank War* for obvious reasons. Both players pilot a tank within a large battle arena and the winner is the last one standing. Here's a more detailed description of the game:

*Tank War is a futuristic tank combat game for two players. Each player drives his or her own tank through the walled battle arena with the aim of obliterating the other's tank. Once a tank is destroyed, both tanks are respawned at their start position, and a point is awarded to the surviving player. Most walls provide permanent cover, but some can be temporarily demolished to create a way through. There is no ultimate goal to the game, and players simply play until one player concedes defeat.*

*Each tank has a primary weapon that it can fire indefinitely. Pickups provide a limited amount of ammunition for a secondary weapon, or repair some of the tank's damage:*

- *Homing rockets: Always move in the direction of your opponent*
- *Bouncing bombs: Bounce against walls, and can be used to fire around corners*
- *Shields: Are activated to provide a temporary protective shield*
- *Toolbox: Repairs part of the tank's damage*

The game uses a split-screen view divided in two parts (see Figure 10-1). The left part is centered on player one's tank and the right part is centered on player two's tank. There is also a mini-map at the bottom of the screen for locating pickups and the other player.

Player one will move their tank with the A, D, W, and S keys and fire with the spacebar (primary) and Ctrl key (secondary). Player two will control their tank with the arrow keys, and fire with the Enter key (primary) and Delete key (secondary).

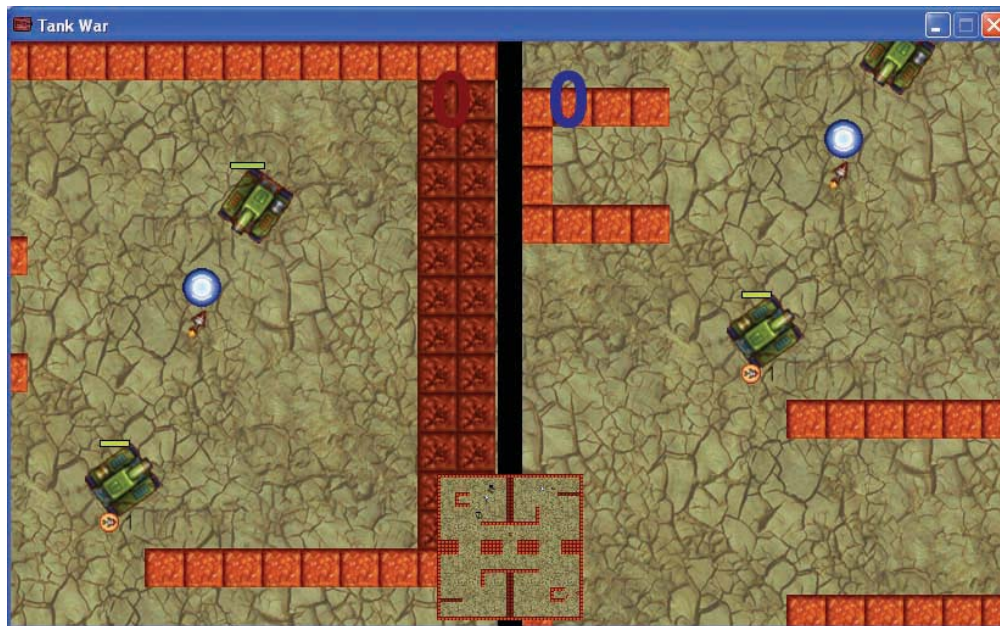


Figure 10-1. Tank War has a split-screen with a little mini-map at the bottom.

All resources for this game have already been created for you in the [Resources/Chapter10](#) folder on the CD.

## Playing with Tanks

Our first task is to create the battle arena. This will be a simple environment with two types of walls that will stop tanks and their shells. The first type of wall will be permanent, whereas the second type can be demolished by tank fire but will reappear again after a while.


### Creating the arena background and walls:

1. Launch Game Maker and start a new empty game.
2. Create a background resource called `background` using `Background.bmp` from the [Resources/Chapter10](#) folder on the CD.

3. Create two sprites called `spr_wall1` and `spr_wall2` using `Wall1.gif` and `Wall2.gif`. Disable the **Transparent** property for both sprites.
4. Create a new object called `obj_wall1` and give it the first wall sprite. Enable the **Solid** property and close the object properties. No further behavior is needed.
5. Create a new object called `obj_wall2` and give it the second wall sprite. Enable the **Solid** property and set **Parent** to `obj_wall1`.


Like most of the previous games, this game will have a controller object. For the time being, this will only play the background music but later it will also be responsible for displaying the score.



#### Creating the controller object and the room:

1. Create a sound resource called `snd_music` using `Music.mp3` from the `Resources/Chapter10` folder on the CD.
-  2. Create a new object called `obj_controller`, with no sprite. Set **Depth** to `-100` to make sure that the drawing actions we will give it later on are drawn in front of other objects. Add an **Other, Game Start** event and include the **Play Sound** action. Set **Sound** to `snd_music` and set **Loop** to **true**.
3. Create a new room and switch to the **settings** tab. Call the room `room_main` and give it an appropriate caption.
4. Switch to the **backgrounds** tab and select the background you created earlier.
5. Switch to the **objects** tab. In the toolbar, set **Snap X** and **Snap Y** to `32`, as this is the size of the wall objects.
6. Create a continuous wall of `obj_wall1` objects around the edge of the room. Also add walls of both types to the interior so that they create obstacles for the tanks (remember that you can hold the Shift key to add multiple instances of an object).
7. Add one instance of the controller object into the room.









Now we'll create our tanks. We'll need different tank objects for each of the two players, but most of their behavior will be identical so we'll create a parent tank object that contains all the common events and actions. In this game we're going to control the tank instances by directly changing their local `direction` and `speed` variables. Remember that the `direction` variable indicates the direction of movement in degrees (0–360 anticlockwise; 0 is horizontally to the right). The `speed` variable indicates the speed of movement in this direction, so a negative value represents a backward movement.

#### Creating the parent tank object:

1. Create a new object called `obj_tank_parent`, with no sprite.
-  2. Add a **Create** event and include a **Set Friction** action with **Friction** set to `0.5`. This will cause the tanks to naturally slow down and come to rest when the player is not pressing the acceleration key.

-  3. Add a **Collision** event with `obj_wall1` and include a **Set Variable** action. Set **Variable** to `speed` and **Value** to `-speed`. This will reverse the tank's movement direction when it collides with a wall.
-  4. Likewise, add a **Collision** event with `obj_tank_parent` and include a **Set Variable** action. Set **Variable** to `speed` and **Value** to `-speed` (you could also right-click on the previous collision event and select **Duplicate Event** to achieve this).

#### Creating the two players' tank objects:









1. Create two sprites called `spr_tank1` and `spr_tank2` using `Tank1.gif` and `Tank2.gif`. Set the **Origin** of both sprites to **Center**. Note that these sprites have 60 subimages corresponding to different facing directions for the tanks.
2. Create a new object called `obj_tank1` and give it the first tank sprite. Set **Parent** to `obj_tank_parent` and enable the **Solid** option. Set **Depth** to `-5` to make sure it appears in front of other objects, such as shells, later on.
-  3. Add a **Keyboard, Letters, A** event and include a **Set Variable** action. Set **Variable** to `direction` and **Value** to `6`, and enable the **Relative** option. This will rotate the tank anticlockwise.
-  4. Add a **Keyboard, Letters, D** event and include a **Set Variable** action. Set **Variable** to `direction` and **Value** to `-6`, and enable the **Relative** option. This will rotate the tank clockwise.
-   5. Add a **Keyboard, Letters, W** event and include a **Test Variable** action. Set **Variable** to `speed`, **Value** to `8`, and **Operation** to **smaller than**. Include a **Set Variable** action, setting **Variable** to `speed` and **Value** to `1` and enabling the **Relative** option. This will then only increase the speed if it is smaller than 8.
-   6. Add a **Keyboard, Letters, S** event and include a **Test Variable** action. Set **Variable** to `speed`, **Value** to `-8`, and **Operation** to **larger than**. Include a **Set Variable** action, setting **Variable** to `speed` and **Value** to `-1` and enabling the **Relative** option. This will only reduce the speed (reverse) if the speed is greater than -8 (full speed backward).
-  7. Add a **Step, End Step** event. In this event we must set the subimage of the sprite that corresponds to the direction the tank is facing. Include the **Change Sprite** action, setting **Sprite** to `spr_tank1`, **Subimage** to `direction/6` and **Speed** to `0`. As in Galactic Mail, `direction/6` converts the angle the object is facing (between 0 and 360) to the range of images in the sprite (between 0 and 60).
-  8. We will draw the tank ourselves because later we want to draw more than just the sprite. Add a **Draw** event. Include the **Draw Sprite** action, setting **Sprite** to `spr_tank1` and **Subimage** to `-1` and enabling the **Relative** option.
9. Repeat steps 2–8 (or duplicate `obj_tank1` and edit it) to create `obj_tank2`. This time you should use the arrow key events to control its movement (**Keyboard, Left**, etc.)
10. Reopen the room and put one instance of each tank into it.

Now test the game to make sure everything is working correctly. In case something is wrong, you'll find a version of the game so far in the file `Games/Chapter10/tank1.gm6` on the CD.

## Firing Shells

Now the fun begins. In this section we'll create shells for the tanks to shoot at each other, but first we need a mechanism to record the tank's damage and scores. As in Chapter 9, we'll give each tank a variable called `damage` to record the amount of damage it has taken. It will start with a value of 0, and once it reaches 100 the tank is destroyed. We'll also use two global variables called `global.score1` and `global.score2` to record how many kills each tank has made. The controller object will initialize these variables and display their values.



### Recording the player's score in the controller object:

1. Create a font called `fnt_score` and select a font like Arial with a **Size** of 48 and the **Bold** option enabled. We only need to use the numerical digits for the score, so you can click the **Digits** button to leave out the other characters in the font. This will save storage space and reduce the size of your `.gm6` and executable game files.
- 

 2. Reopen the controller object and select the **Game Start** event. Include a **Set Variable** action with **Variable** set to `global.score1` and **Value** set to 0. Include another **Set Variable** action with **Variable** set to `global.score2` and **Value** also set to 0. This creates and initializes the global score variables that will store the player's score.
- 

 3. Add a **Draw** event and include a **Set Font** action. Set **Font** to `fnt_score` and **Align** to **right**. Include a **Set Color** action and choose a dark red color.
- 
 4. Include a **Draw Variable** action from the **control** tab. Set **Variable** to `global.score1`, **X** to 300, and **Y** to 10.
- 

 5. Include another **Set Font** action with **Font** set to `fnt_score`, but this time set **Align** to **left**. Include a **Set Color** action and choose a dark blue color.
- 
 6. Include a **Draw Variable** action with **Variable** set to `global.score2`, **X** set to 340, and **Y** set to 10.



If you run the game now, you should begin with a large 0–0 score displayed on the screen. Next we're going to create two explosions: a large one for when a tank is destroyed, and a small one for when a shell hits something.

### Creating the large explosion object:

1. Create a sprite called `spr_explosion_large` using `Explosion_large.gif` and **Center** the **Origin**.
2. Create a sound called `snd_explosion_large` using `Explosion_large.wav`.

-  3. Create a new object called `obj_explosion_large`. Give it the large explosion sprite and set **Depth** to `-10`. Add a **Create** event and include a **Play Sound** action, with **Sound** set to `snd_explosion_large` and **Loop** set to **false**.
-  4. Add an **Other, Animation End** event and include the **Restart Room** action.

#### Creating the small explosion object:






- 1. Create a sprite called `spr_explosion_small` using `Explosion_small.gif` and **Center the Origin**.
- 2. Create a sound called `snd_explosion_small` using the file `Explosion_small.wav`.
-  3. Create an object called `obj_explosion_small`. Give it the small explosion sprite and set **Depth** to `-10`. Add a **Create** event and include the **Play Sound** action, with **Sound** set to `snd_explosion_small` and **Loop** set to **false**.
-  4. Add the **Other, Animation End** event and include the **Destroy Instance** action.



Explosions in hand, we're now ready to create the damage mechanism. The parent tank object will be responsible for initializing the `damage` variable, checking the damage, and drawing the tank's health bar on the screen. It will also be responsible for blowing up the tank when its damage reaches 100, which is why we needed the explosion objects first.


This is all pretty straightforward, and putting this code in the parent tank object will save us some time. However, when the tank blows up we also need to increase the correct player's score—so how do we know which player's tank has died if we are working with the parent object? Fortunately, every instance has a variable called `object_index` that records a number corresponding to the type of object it is. Every object has its own unique number, which can be accessed by using the object name as if it was a variable (in this case `obj_tank1` and `obj_tank2`). So by comparing `object_index` and `obj_tank1` we can tell if the instance is an instance of player one's tank or an instance of player two's.


We'll check the tank's damage in the **Step** event of the parent tank object and increase the appropriate score if it is larger than 100. Then we'll create a large explosion and destroy the tank. The large explosion object will automatically restart the room once the animation is finished.

#### Adding a damage mechanism to the parent tank object:

-  1. Reopen `obj_tank_parent` and select the **Create** event. Include a **Set Variable** action with **Variable** set to `damage` and **Value** set to `0`.
-   2. Add a **Step, Step** event and include a **Test Variable** action. Set **Variable** to `damage`, **Value** to `100`, and **Operation** to **smaller than**. Include an **Exit Event** action so that no further actions are executed if the damage is smaller than 100.
-   3. Now we need to find out what type of tank we are dealing with. Include a **Test Variable** action with **Variable** set to `object_index`, **Value** set to `obj_tank1`, and **Operation** set to **equal to**. Include a **Set Variable** action with **Variable** set to `global.score2`, **Value** set to `1`, and the **Relative** option enabled. This will then increase player two's score if this instance is player one's tank.

- 



4. Include an **Else** action followed by a **Set Variable** action. Set **Variable** to `global.score1` and **Value** to `1`, and enable the **Relative** option. This will increase player one's score if this instance is player two's tank.
- 


5. Include a **Create Instance** action with **Object** set to `obj_explosion_large` and the **Relative** option enabled.
- 


6. Finally, include a **Destroy Instance** action.


Obviously, we need to draw some kind of health bar so that the players can see how well they are doing. It would be easiest to use the **Draw** event of the parent tank object to do this, but there is a problem. The two tank objects already have their own **Draw** events so they won't normally execute the **Draw** event of the parent object because their own takes priority. Fortunately, we can use the **Call Parent Event** action in the two tanks' own **Draw** events to make sure that the parent's **Draw** event is called as well.

#### Adding a draw event to the parent tank object to draw the health bars:

- 1. Add a **Draw** event for the parent tank object.
- 

2. Include a **Set Health** action (**score** tab) and set **Value** to `100-damage`. Damage is the opposite concept to health, so subtracting it from 100 makes this conversion (e.g., 80 percent damage converts to  $100 - 80 = 20$  percent health).
- 

3. Add a **Draw Health** action. Set **X1** to `-20`, **Y1** to `-35`, **X2** to `20`, and **Y2** to `-30`. Enable the **Relative** option, but leave the other parameters as they are. This will draw a small health bar above the tank. It may seem strange to be using the health functions here as they only work with one health value and we have two players. However, this technique works because we set the health in step 2 using the instance's own `damage` variable, just before we draw the health bar.
- 








4. Reopen `obj_tank1` and select the **Draw** event. Include the **Call Parent Event** action (**control** tab) at the end of the list of actions for this event. This will make sure that the **Draw** event of the parent tank object is also executed.
- 

5. Reopen `obj_tank2` and select the **Draw** event. Include the **Call Parent Event** action (**control** tab) at the end of the list of actions for this event.

With the damage and scoring mechanism in place, we can now create the tank shells. We only want the player's shells to damage their opponent's tank, so we will create a separate shell object for each tank and put common behavior in a shell parent object. We'll also use an alarm clock to give shells a limited life span (and therefore a limited range). Alarm clocks will also help us to temporarily demolish the second wall type when they are hit by shells. We'll move the walls outside the room and use an alarm event to bring them back to their original position after a period of time.





**Creating the parent shell object:**

1. Create a sprite called `spr_shell` using `Shell.gif` and **Center** the **Origin**. Note that like the tank sprite, this contains 60 images showing the shell pointing in different directions.
2. Create a new object called `obj_shell_parent` and leave it without a sprite (you can set it, but it isn't necessary for the parent as it never appears in the game).
-  3. Add a **Create** event and include the **Set Alarm** action. Set the **Number of Steps** to 30 and select **Alarm 0**.
-  4. Add an **Alarm, Alarm 0** event and include the **Destroy Instance** action.
-  5. Add a **Step, End Step** event and include the **Change Sprite** action. Set **Sprite** to `spr_shell`, **Subimage** to `direction/6`, and **Speed** to 0 (to stop it from animating).
-  6. Add a **Collision** event with `obj_wall1` and include a **Create Instance** action. Set **Object** to `obj_explosion_small` and enable the **Relative** option. Also include a **Destroy Instance** action to destroy the shell.
-  7. Add a **Collision** event with `obj_wall2`. This object must be temporarily removed. Include a **Create Instance** action with **Object** set to `obj_explosion_small` and the **Relative** option enabled. Include a **Jump to Position** action with **X** and **Y** set to 100000. Also select the **Other** object for **Applies to** so that the wall is moved rather than the shell.
-  8. Include a **Set Alarm** action and select the **Other** object for **Applies to** so that it sets an alarm for the wall. Select **Alarm 0** and set **Number of Steps** to 300. Finally, include a **Destroy Instance** action to destroy the shell.
-  9. Add a **Collision** event with `obj_shell_parent` and include a **Create Instance** action. Set **Object** to `obj_explosion_small` and enable the **Relative** option. Also include a **Destroy Instance** action to destroy the shell.




We now need to make sure that any removed `obj_wall2` instances are returned to their original position when the alarm clock runs out. We will also need to check that the original position is empty first, as we did for the locks in Koalabr8.

**Editing the destructible wall object to make it reappear:**

-  1. Reopen the `obj_wall2` object and add an **Alarm, Alarm 0** event. Include a **Check Empty** action with **X** set to `xstart`, **Y** set to `ystart`, and **Objects** set to **All**. Include a **Jump to Start** action.
-  2. Next include an **Else** action followed by a **Set Alarm** action. Select **Alarm 0** and set **Number of Steps** to 5. That way, when the position is not empty it will wait five more steps and then try again.







We can now create the actual shell objects.

**Creating the players' shell objects:**

1. Create a new object called `obj_shell1`. Give it the shell sprite and set its **Parent** to `obj_shell_parent`.
-  2. Add a **Collision** event with `obj_tank2` and include a **Set Variable** action. Set **Variable** to `damage` and **Value** to `10`, and enable the **Relative** option. Also select the **Other** object for **Applies to** so that the tank's `damage` variable is changed.
-   
 3. Include a **Create Instance** action with **Object** set to `obj_explosion_small` and enable the **Relative** option. Also include a **Destroy Instance** action to destroy the shell.
4. Repeat steps 1–3 to create `obj_shell2` using a **Collision** event with `obj_tank1` rather than `obj_tank2`.

Finally, we'll add the actions to make the tanks fire shells. Player one's tanks will shoot shells of type `obj_shell1` when the spacebar is pressed, and player two's tank will shoot shells of type `obj_shell2` when the Enter key is pressed. As in the Wingman Sam game, we'll limit the speed with which the player can fire shells using a `can_shoot` variable. To create bullets that face in the same direction as the tank, we will use the **Create Moving** action and pass in the tank's own `direction` variable.

**Adding events to make the tank objects fire shells:**

-  1. Reopen the parent tank object and select the **Create** event. Include a **Set Variable** action with **Variable** set to `can_shoot` and **Value** set to `0`.
-  2. Select the **Step** event and include a **Set Variable** action at the beginning of the list of actions. Set **Variable** to `can_shoot` and **Value** to `1`, and enable the **Relative** option.
-   
 3. Reopen `obj_tank1` and add a **Key Press, <Space>** event. Include the **Test Variable** action, with **Variable** set to `can_shoot`, **Value** set to `0`, and **Operation** set to **smaller than**. Next include the **Exit Event** action so that the remaining actions are only executed when `can_shoot` is larger or equal to 0.
-   
 4. Include a **Create Moving** action. Set **Object** to `obj_shell1`, **Speed** to `16`, and **Direction** to `direction`, and enable the **Relative** option. Also include a **Set Variable** action with **Variable** set to `can_shoot` and **Value** set to `-10`.
5. Repeat steps 3–4 for the `obj_tank2`, this time using a **Key Press, <Enter>** event for the key and `obj_shell2` for the **Create Moving** action.

That completes the shells. Test the game carefully and check yours against the one in the file `Games/Chapter10/tank2.gm6` on the CD if you have any problems.








## Secondary Weapons

We're going to include secondary weapons and pickups to increase the appeal of the game. Pickups will appear randomly in the battle arena and can be collected by driving into them. Each tank can only have one secondary weapon active at once, so picking up a new weapon

will remove the current one. Toolboxes can also be collected to repair some of the tank's damage, but these will remove any secondary weapons too. All the secondary weapons will have limited ammunition, so the players must take care to make the most of them.

We'll use just one object for all these different kinds of pickups and change its appearance depending on the type of pickup. We'll use a variable called `kind` to record what sort of pickup it is by setting its value to 0, 1, 2, or 3. The value 0 will stand for the homing rocket, 1 for the bouncing bomb, 2 for the shield, and 3 for the toolbox. We can then choose a pickup type at random by using the `choose()` function. To make things more interesting, the pickup will change its `kind` from time to time and jump to a new position. It will also jump to a new position when it is collected by a tank.







#### Creating the pickup object:

1. Create a sprite called `spr_pickup` using `Pickup.gif`. Note that it consists of four completely different subimages, representing each different kind of pickup.
2. Create a new object called `obj_pickup` and give it the pickup sprite.
-  3. Add a **Create** event and include the **Set Variable** action. Set **Variable** to `kind` and **Value** to `choose(0,1,2,3)`. This will choose randomly between the numbers in brackets that are separated by commas.
-  4. Include the **Set Alarm** action for **Alarm 0** and set **Number of Steps** to `100+random(500)`. This will give a random time between 100 and 600 steps or about 3 and 20 seconds. Finally, include a **Jump to Random** action with the default parameters. This will move the instance to a random empty position.
-   5. Add an **Alarm, Alarm 0** event and include the **Set Variable** action. Set **Variable** to `kind` and **Value** to `choose(0,1,2,3)`.
-  6. Include the **Set Alarm** action for **Alarm 0** with **Number of Steps** set to `100+random(500)`. Finally, include a **Jump to Random** action.
-  7. Add a **Collision** event with `obj_tank_parent` and include a **Jump to Random** action.
-  8. Add a **Draw** event and include the **Draw Sprite** action. Set **Sprite** to `spr_pickup`, **Subimage** to `kind` and enable the **Relative** option.

Now reopen the room and add a few instances of the pickup object to it. Test the game to make sure that the pickups have different images and that they change their type and position from time to time. Also check out what happens when you drive over one with your tank.




We'll also need to record the kind of pickup that has been collected by the tank so that it can change its secondary weapon. We'll use the variable `weapon` for this, where a value of -1 corresponds to no weapon. The variable `ammunition` will indicate how many shots the tank has left of this weapon type. Once `ammunition` reaches 0, `weapon` will be set to -1 to disable the secondary weapon from then on. We'll check the value of the pickup object's `kind` variable in the collision event, and use it to set the tank's `weapon` accordingly.



**Editing the parent tank object to record pickups:**

1. Reopen `obj_tank_parent` and select the **Create** event.
2. Include a **Set Variable** action with **Variable** set to `weapon` and **Value** set to `-1`. Include a second **Set Variable** action with **Variable** set to `ammunition` and **Value** set to `0`.
 
3. Add a **Collision** event with `obj_pickup` and include a **Test Variable** action. Set **Variable** to `other.kind`, **Value** to `3`, and **Operation** to **equal to**. A value of 3 corresponds to the toolbox. This needs to repair the tank's damage, so include a **Start Block** action to begin the block of actions that do this.
 
4. Include a **Set Variable** action with **Variable** set to `weapon` and **Value** set to `-1`. Include a second **Set Variable** action with **Variable** set to `damage` and **Value** set to `max(0, damage-50)`. The function `max` decides which is the largest of the two values you give it (more about functions in Chapter 12). Therefore, this sets the new damage to the largest out of `damage-50` and `0`. In effect, this subtracts 50 from `damage` but makes sure it does not become smaller than 0. Include an **End Block** action.
 
5. Include an **Else** action, followed by a **Start Block** action to group the actions that are used if this is not a toolbox pickup.
 
6. Include a **Set Variable** action with **Variable** set to `weapon` and **Value** set to `other.kind`. Include another **Set Variable** action with **Variable** set to `ammunition` and **Value** set to `10`.
 
7. Finally, include an **End Block** action.
 

Obviously, it will help players to be able to see the type of secondary weapon they've collected and the ammunition they have remaining for it. We'll display this below each tank using a small image of the pickup. These images have been combined into one sprite again, so we'll need to test the value of `weapon` and draw the corresponding subimage if it is equal to 0, 1, or 2. We can then also draw the value of the variable `ammunition` next to it.



**Displaying the secondary weapon in the parent tank object:**

1. Create a new sprite called `spr_weapon` using `Weapon.gif`. Note that it consists of three subimages (no image is required for the toolbox).
2. Create a font called `fnt_ammunition` and keep the default settings for it.
3. Select the **Draw** event in `obj_tank_parent` and include a **Test Variable** action. Set **Variable** to `weapon`, **Value** to `-1`, and **Operation** to **larger than**. This will ensure that we only draw something when there is a secondary weapon. Include a **Start Block** action to group the drawing actions.
 
4. Include the **Draw Sprite** action and select `spr_weapon`. Set **X** to `-20`, **Y** to `25`, and **Subimage** to `weapon`. Also enable the **Relative** option.
 
5. Include a **Set Color** action and choose black. Then include a **Set Font** action, selecting `fnt_ammunition` and setting **Align** to **left**.
 

-  6. Next include a **Draw Variable** action with **Variable** set to `ammunition`, **X** set to 0, **Y** set to 24, and the **Relative** option enabled.
-  7. Finally, include an **End Block** action to conclude the actions that draw the weapon information.





Test the game to check that the weapon icons are displayed correctly when you collect the different weapon pickups. However, so far only the repair kit actually does anything for the player, so let's start by sorting out the rocket. It will behave in much the same way as the shell but automatically starts moving in the direction of the enemy tank. We'll use the same structure of objects as we did for the shell, with common behavior contained in a parent rocket object (`obj_rocket_parent`) and separate rocket objects that home in on the different tanks (`obj_rocket1` and `obj_rocket2`). We'll also make `obj_shell_parent` the parent of `obj_rocket_parent` so that it inherits `obj_shell_parent`'s **Collision** and **Alarm** events. However, we don't want `obj_rocket_parent` to have the same **Create** and **End Step** events as `obj_shell_parent` so we'll give it new versions of these events that give the rocket a longer lifetime and draw the correct sprite.

#### Creating the parent rocket object:

1. Create a sprite called `spr_rocket` using `Rocket.gif` and **Center** the **Origin**.
2. Create a new object called `obj_rocket_parent` and set **Parent** to `obj_shell_parent`.
-  3. Add a **Create** event and include the **Set Alarm** action. Set **Number of Steps** to 60 and select **Alarm 0**.
-  4. Add a **Step, End Step** event and include a **Change Sprite** action. Select the rocket sprite, then set **Subimage** to `direction/6` and **Speed** to 0.





Next we create the two actual rocket objects.

#### Creating the actual rocket objects:

1. Create a new object called it `obj_rocket1` and give it the rocket sprite. Set **Parent** to `obj_rocket_parent`.
-  2. Add a **Create** event and include the **Move Towards** action. Set **X** to `obj_tank2.x`, **Y** to `obj_tank2.y`, and **Speed** to 8.
-  3. Add a **Collision** event with `obj_tank2` and include a **Set Variable** action. Select **Other** from **Applies to** (the tank), set **Variable** to `damage` and **Value** to 10, and enable the **Relative** option.
-  4. Include a **Create Instance** action, selecting `obj_explosion_small` and enabling the **Relative** option. Also include a **Destroy Instance** action.
-  5. Create `obj_rocket2` in the same way, but move toward `obj_tank1` in the **Create** event, and add a **Collision** event with `obj_tank1` for the actions in steps 3 and 4.




Finally, we need to make it possible for the tanks to fire rockets. We'll check whether they have the weapon and ammo in the **Key Press** event of the secondary fire key. If they do, then we'll create the rocket and decrease the ammunition. When it reaches 0, we'll set `weapon` to `-1` to disable it.




#### Adding events to shoot rockets for the tank object:

- 
 1. Reopen the first tank object and add a **Key Press, <Ctrl>** event. Include a **Test Variable** action, with **Variable** set to `can_shoot`, **Value** set to `0`, and **Operation** set to **smaller than**. Next include the **Exit Event** action so that the remaining actions are only executed when `can_shoot` is larger than or equal to `0`.
- 
 2. Include the **Test Variable** action, with **Variable** set to `weapon`, **Value** set to `0`, and **Operation** set to **equal to**. Next include a **Test Instance Count** action with **Object** set to `obj_tank2`, **Number** set to `0` and **Operation** set to **larger than**. Follow this with a **Create Instance** action for `obj_rocket1`, and enable the **Relative** option. This creates a rocket only when it is the current secondary weapon and the other tank exists (this avoids a rare error when the other tank has just been destroyed).
- 
 3. Next we need to decrease the ammunition. Include a **Set Variable** action with **Variable** set to `ammunition`, **Value** set to `-1`, and the **Relative** option enabled. Include a **Test Variable** action with **Variable** set to `ammunition`, **Value** set to `1`, and **Operation** set to **smaller than**. Follow this with a **Set Variable** action with **Variable** set to `weapon` and **Value** set to `-1`.
- 
 4. Finally, include a **Set Variable** action with **Variable** set to `can_shoot` and **Value** set to `-10`.
5. Repeat steps 1–4 for `obj_tank2`, using a **Key Press, Others, <Delete>** event and creating `obj_rocket2`.

Now we'll create the bouncing bomb secondary weapon in a similar fashion. It behaves in the same way as the shell except that it bounces against walls.






#### Creating the bouncing bomb objects:

1. Create a sprite called `spr_bouncing` using `Bouncing.gif` and **Center** the **Origin**.
2. Create a new object called `obj_bouncing_parent` and set its **Parent** to `obj_shell_parent`.
- 
 3. Add a **Collision** event with `obj_wall1` and include the **Bounce** action. Select **precisely** and set **Against** to **solid objects**.
- 
 4. Add a similar **Collision** event with `obj_wall2`.
- 
 5. Add a **Step, End Step** event and include a **Change Sprite** action. Select `spr_bouncing`, set **Subimage** to `direction/6`, and set **Speed** to `0`.
6. Create a new object called `obj_bouncing1` and give it the bouncing bomb sprite. Set its **Parent** to `obj_bouncing_parent`.

-  7. Add a **Collision** event with `obj_tank2` and include a **Set Variable** action. Select **Other** from **Applies to**, set **Variable** to `damage`, set **Value** to `10`, and enable the **Relative** option. Include a **Create Instance** action for `obj_explosion_small` and enable the **Relative** option.
- 
-  8. Include a **Destroy Instance** action.
- 9. Repeat steps 6 and 7 to create `obj_bouncing2` using a **Collision** event with `obj_tank1`.



Before we add actions to make the tank objects shoot bouncing bombs, we'll create the final special weapon: the shield. This is a bit more complicated as it allows the player to temporarily make their tank invincible. Activating the shield will set a new variable called `shield` to 40, and display a shield sprite. The value of `shield` will be reduced by 1 in each step until it falls below 0 and the shield is disabled again. We'll check the value of `shield` each time the tank is hit and only increase its damage when `shield` is less than 0.

#### Editing the parent tank object to support shields:

- 1. Create sprites called `spr_shield1` and `spr_shield2` using `Shield1.gif` and `Shield2.gif` and **Center** their **Origins**.
-  2. Reopen the parent tank object and select the **Create** event. Include a **Set Variable** action with **Variable** set to `shield` and **Value** set to `0`.
-  3. Select the **Step** event and include a **Set Variable** action at the start of the list. Set **Variable** to `shield`, set **Value** to `-1`, and enable the **Relative** option.
-  4. Reopen `obj_shell1` and select the **Collision** event with `obj_tank2`. Include a **Test Variable** action directly above the **Set Variable** that increases the damage. Select **Other** from **Applies to**, then set **Variable** to `shield`, **Value** to `0`, and **Operation** to **smaller than**. Now the damage will only be increased when the tank has no shield.
- 5. Repeat step 4 for objects `obj_shell2`, `obj_rocket1`, `obj_rocket2`, `obj_bouncing1`, and `obj_bouncing2`.
-  6. Reopen `obj_tank1` and select the **Draw** event. Include a **Test Variable** action at the start of the action list. Set **Variable** to `shield`, **Value** to `0`, and **Operation** to **larger than**. Follow this with a **Draw Sprite** action for `spr_shield1` with the **Relative** option enabled.
-  7. Repeat step 6 for `obj_tank2`, this time drawing `spr_shield2`.

Now all that remains is to adapt the tanks so that both the bouncing bombs and the shields can be used.

#### Editing tank objects to shoot bombs and use shields:

- 1. Reopen `obj_tank1` and select the **Key Press, <Ctrl>** event.
-  2. Include a **Test Variable** action below the **Create Instance** action that creates `obj_rocket1`. Set **Variable** to `weapon`, **Value** to `1`, and **Operation** to **equal to**. Follow this with a **Create Moving** action for `obj_bouncing1`, setting **Speed** to `16` and **Direction** to `direction`, and enabling the **Relative** option.
- 



3. Include another **Test Variable** action below this, with **Variable** set to `weapon`, **Value** set to `2`, and **Operation** set to **equal to**. Follow this with by a **Set Variable** action with **Variable** set to `shield` and **Value** set to `40`.
4. Repeat steps 1–3 for `obj_tank2`, adapting the **Key Press, <Delete>** event and creating `obj_bouncing2`.

This completes all the secondary weapons and the game should now be fully playable. We encourage you to play it a lot with your friends, to make sure everything is working as it should. You'll find the current version on the CD in the file `Games/Chapter10/tank3.gm6`.

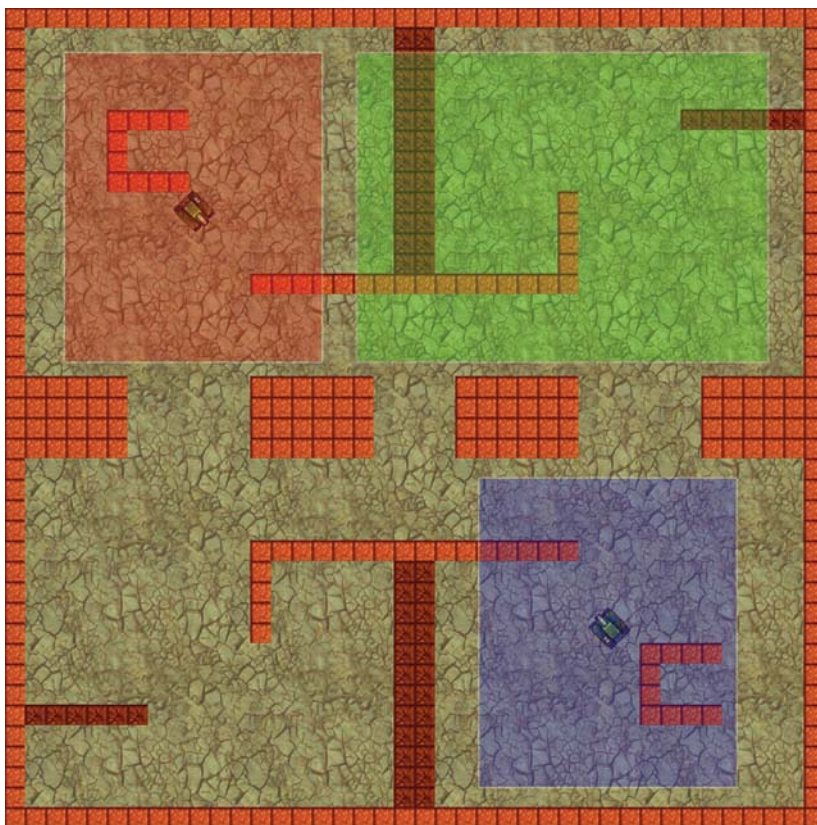
## Views

Currently, our playing area is quite small and both players can see all of it at once. However, we can create more interesting gameplay by giving each player a limited “window” into a much larger playing area. This can easily be achieved in Game Maker using *views*. We'll use two views to create a split screen, in which the left half of the screen shows the area around the first tank and the right half shows the area around the second tank. Later we use a third view to display a little mini-map as well.

To understand the concept of views, you need to appreciate that there is a distinction between a room and the window that provides a view of that room on the screen. Up to now, rooms have always been the same size as the window and the window has always showed the entire contents of the room. However, rooms can be any size you like, and views can be used to indicate the specific area of the room that should appear in the window. We're going to create a room that's twice the width of a normal room with an equal height (see Figure 10-2). The green rectangle shows the size of a normal room, and the red and blue squares show the size of the views we will give to each player in the room. To create these views, we will need to specify the following information on the **views** tab in the room properties:

- **View in room:** This is an area of the room that needs to be displayed in the view. The **X** and **Y** positions define the top-left corner of this area and **W** and **H** specify the width and height of it.
- **Port on screen:** This is the position on the window where the view should be shown. The **X** and **Y** positions define the top-left corner of this area and **W** and **H** specify the width and height of it. If the width and height are different from the size of the view area, then the view will be automatically scaled to fit. Game Maker will also automatically adapt the size of the window so that all ports fit into it.
- **Object following:** Specifying an object here will make the view track that object as it moves around the room. **Hbor** and **Vbor** specify the size of the horizontal and vertical borders that you want to keep around the object. The view will not move until the edge of the screen is closer than this distance from the object. Setting **Hbor** to half the width of the view and **Vbor** to half the height of the view will therefore maintain the object in the center. Finally, **Hsp** and **Vsp** allow you to limit the speed with which the view moves (–1 means no limit).





**Figure 10-2.** We'll create a large room, much bigger than a normal window (green rectangle), and provide views into it for each of the tanks (red and blue squares).

You can specify up to eight different views, but you'll probably only need one or two. Let's adapt our game's room to use two views.

#### Editing the room resource to provide two views:

1. Reopen the main room and switch to the **settings** tab.
2. Set both the **Width** and **Height** of the room to 1280, to create a much larger room.
3. Switch to the **objects** tab and add wall instances to incorporate the extra playing area. Start the tanks close to two opposite corners and add six pickup instances. Also don't forget that the room needs exactly one instance of the controller object.
4. Switch to the **views** tab and select the **Enable the use of Views** option. This activates the use of views in this room.
5. Make sure that **View 0** is selected in the list and enable the **Visible when room starts** option. We will use this view for player one.

6. Under **View in room** set **X** to 0, **Y** to 0, **W** to 400, and **H** to 480. The **X** and **Y** positions of the views don't really matter in this case as we will make them follow the tanks. Nonetheless, notice that lines appear in the room to indicate the size and position of the view.
7. Under **Port on screen** set **X** to 0, **Y** to 0, **W** to 400, and **H** to 480. This port will show player one's view on the left side of the screen.
8. Under **Object following** select `obj_tank1`, then set **Hbor** to 200 and **Vbor** to 240. The form should now look like Figure 10-3.
9. Now select **View 1** in the list and enable the **Visible when room starts** option. We will use this view for player two.
10. Under **View in room** set **X** to 0, **Y** to 0, **W** to 400, and **H** to 480.
11. Under **Port on screen** set **X** to 420, **Y** to 0, **W** to 400, and **H** to 480. This places the second view to the right of the first view with a little space between them.
12. Under **Object following** select `obj_tank2`, and set **Hbor** to 200 and **Vbor** to 240.

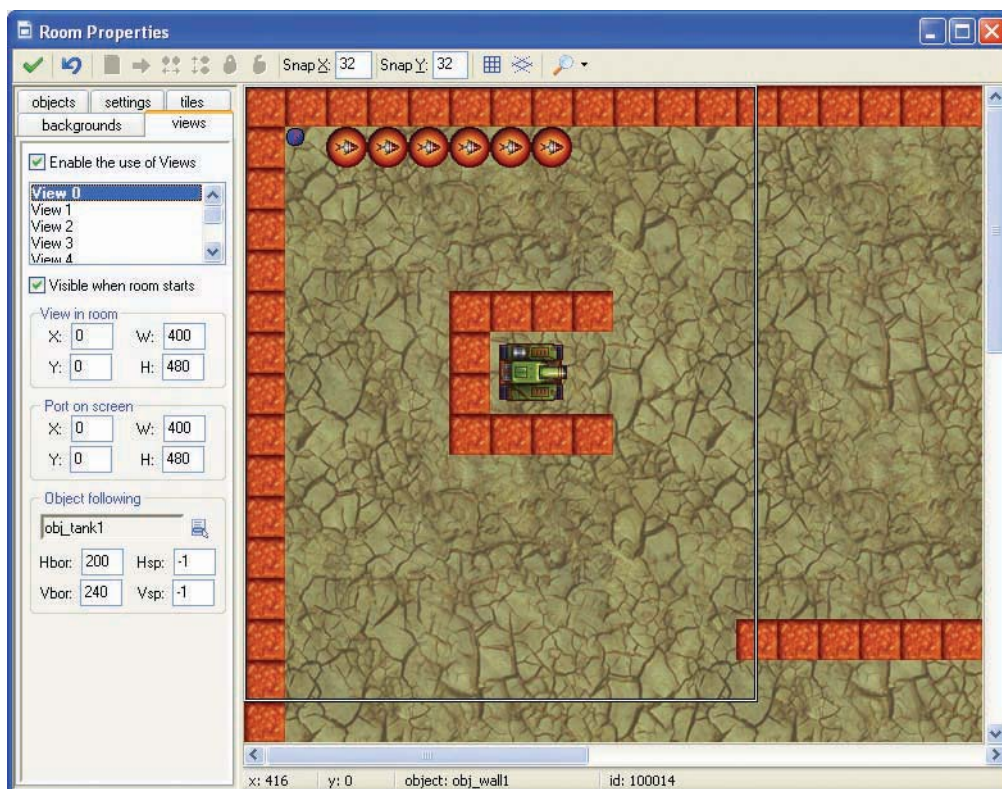


Figure 10-3. This is how the form should look when the values for View 0 have been set.

And that's it. Easy, wasn't it? Run the game and you should be able to play in the new split-screen mode.

---



**Tip** The empty region between the views defaults to the color black. You can change this in the **Global Game Settings** on the **graphics** tab under **Color outside the room region**.

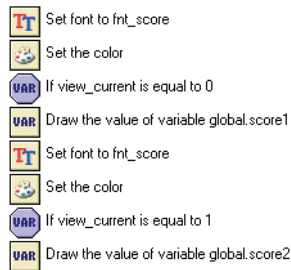
---

Have you noticed something strange? The score is displayed at a fixed position in the room so you can only see it if you drive up to it! To fix this we need to draw it at a changing position relative to the player's views. The score for player one needs to appear in the top-right corner of View 0 and the score for player two needs to appear in the top-left corner of View 1. Game Maker provides variables that we can use to obtain the positions of views. `view_xview[0]` and `view_yview[0]` indicate the current x- and y-positions of View 0 while `view_xview[1]` and `view_yview[1]` indicate the x- and y-positions of View 1.

Unfortunately, this does not solve the problem completely. To explain why, you'll need to understand what Game Maker is doing when you use views. For each view, Game Maker draws the whole room, including all the backgrounds, objects, and **Draw** events; clips the visible area to the size of the view; and then copies it to the required position on the window. This means the **Draw** event of the controller object (that draws the score) is called twice, once for drawing each of the views. So, to display the score in the correct place we need to know which view is currently being drawn. Game Maker allows us to check this using the variable `view_current`, which will be 0 for View 0 and 1 for View 1. Therefore, we can test the value of this variable in the **Draw** event of the controller object and draw the score of the appropriate tank relative to the position of the current view.

#### Editing the controller object to draw the score relative to the view position:

1. Reopen the controller object and select the **Draw** event.
-  2. Include a **Test Variable** action before the **Draw Variable** action that draws the score for player one. Set **Variable** to `view_current`, **Value** to 0, and **Operation** to **equal to**.
3. Edit the **Draw Variable** action that draws player one's score. Change **X** to `view_xview[0]+380` and **Y** to `view_yview[0]+10`.
-  4. Include a **Test Variable** action before the **Draw Variable** action that draws the score for player two. Set **Variable** to `view_current`, **Value** to 1, and **Operation** to **equal to**.
5. Edit the **Draw Variable** action for player two. Change **X** to `view_xview[1]+20` and **Y** to `view_yview[1]+10`. The action list should now look like Figure 10-4.



**Figure 10-4.** *These actions draw the scores correctly for each view.*

Run the game to check that the score is displayed correctly.

We'll now add a little mini-map to help the player see where they are. This mini-map shows the entire room, so that both players can see the location of their opponents and the pickups in the room. Creating a mini-map is very simple using views, as we can create an additional view that includes the whole room but scales it down to a small port on the screen.

#### Adding a view to create a mini-map:

1. Reopen the main room and switch to the **views** tab.
2. Select **View 2** in the list and enable the **Visible when room starts** option.
3. Under **View in room** set **X** to 0, **Y** to 0, **W** to 1280, and **H** to 1280 (the entire room).
4. Under **Port on screen** set **X** to 350, **Y** to 355, **W** to 120, and **H** to 120. No object needs to be followed.

And that finishes the game for this chapter. Run it and check that it all works. There are a few final improvements you might want to make. You should add some **Game Information** and you might want to change some of the **Global Game Settings**. For example, you might not want to display the cursor but might want to start in full-screen mode or add a loading image of your own for the game.

---

**Tip** To improve the mini-map and make it more “iconic,” you could make the different objects draw something different when the variable `view_current` is equal to 2. For example, the pickup object could simply display a red disk and the walls could draw black squares.

---

## Congratulations

That's another one complete! We hope you enjoyed making this game and playing it with your friends. The final version can be found on the CD in the file `Games/Chapter10/tank4.gm6`. You encountered some important new features of Game Maker in this chapter, including views, which can be used to create all sorts of different games.

There are many ways in which you could make Tank War more interesting. You could create different arenas for the players to compete in. Some could be wide and open while others could have close passageways. You could also add other types of walls, perhaps stopping shells but not the tanks, or even the other way around. You could create muddy areas that reduce your speed, or slippery areas that make it difficult to steer your tank. Of course, you can also add other types of secondary weapons, such as guns that fire sideways or in many different directions. You could even drop mines or create holes in the ground. You could also add a front-end to the game, displaying the title graphic that is supplied. You're the designer and it's up to you.

We'll be staying with our Tank War example in the next chapter as we explore the game design issues involved in creating multiplayer games. We've got some different versions of the game for you to play and you'll be balancing tanks, so you'd better go and find some king-sized scales!

## CHAPTER 11



# Game Design: Balance in Multiplayer Games

**M**ultiplayer games offer game designers many additional ways to create playing experiences that are fun. Even games that are not very enjoyable on your own can be very addictive when playing with or against other human players. Harnessing this power requires designing games that treat all players fairly but still allow them to make the kind of meaningful choices that make games interesting to play. In this chapter we'll discuss some of the strategies you can use to strike the right balance in your own multiplayer games.

## Competition and Cooperation

Competition and cooperation form the basis of all multiplayer games. No matter how convincing a computer-controlled character is, it never produces the same thrill as another human being taking part in the game. In this section we'll discuss some approaches for creating multiplayer game modes and the pros and cons of each.

### Independent Competition

The simplest way to create a competitive game mode is to make players take turns playing the single-player game and declare the player who does the best to be the winner. A long time ago, this was a common way of adding a multiplayer mode to a game, but these days players expect more. Another basic method is to split the screen so that both players play the single-player mode at the same time in different sections of the screen. Figure 11-1 shows how the Super Rainbow Reef game from Chapter 6 might look as a split-screen multiplayer game.

The important thing to note about both of these competitive modes is that neither creates interactions between the players—each player is expected to play independently, and their performance is compared at the end of the game. While such approaches offer an easy way to include a competitive mode, the results are generally not nearly as much fun as when there are interactions between the players.



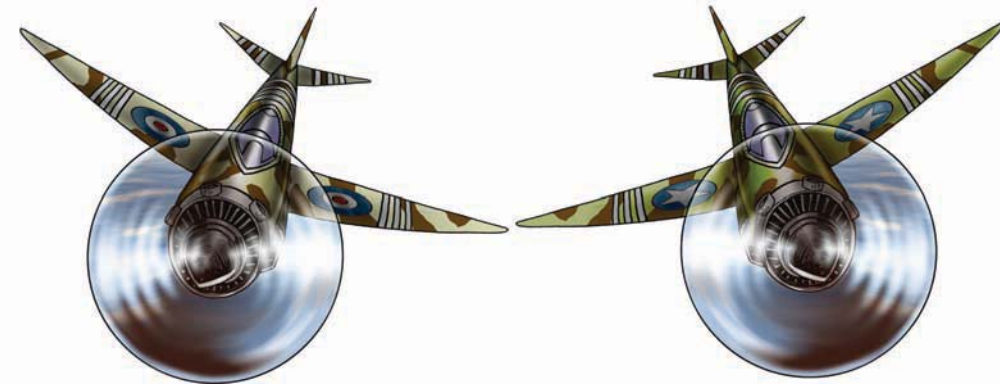
**Figure 11-1.** A split-screen multiplayer mode for *Super Rainbow Reef* might look something like this.

## Dependent Competition

Competitive games that create interactions *between* players provide each player with a game-play experience that is dependent on their opponent. Often this is achieved by setting the gameplay within a shared environment and providing competing goals (as in the Tank War game). However a shared environment is not always necessary: dependent competition could be added to the split-screen *Super Rainbow Reef* example by including power-up bricks that affect the opposing player's starfish or shell. This would also make each player's experience dependent on the other and add an extra competitive edge to the game.

Dependent competition may harness more of the competitive potential of multiplayer games, but it has its own drawbacks. Independent competition usually gives both players a decent chance to enjoy the game—even if one player is much less skilled than the other. Dependent competitions can often be over very quickly when players' skills aren't equally matched and soon stop being fun for both players. For this reason, many competitive games include features to rebalance the competition, such as handicap settings or weaker characters for more skillful players to use. Many even include hidden catch-up mechanisms, whereby the losing player is given better power-ups to try to even out the competition.

It's also worth remembering that competition is not everyone's idea of fun, and its appeal depends a lot on individual personality. For every player who craves the adrenaline rush of a head-to-head competition, there's another who hates the confrontation that this kind of gameplay creates. Fortunately, multiplayer games don't have to be competitive at all, and many players can get just as much enjoyment from cooperative game modes.



## Independent Cooperation

The game *Wingman Sam* provides a multiplayer mode with mostly independent cooperation. It allows players to work toward the same goal, but they are not necessarily required to work together or interact with each other to achieve it. This is usually the most practical way of including a cooperative game mode as it means that each of the players can survive on their own if one player dies. This is fairly essential for coin-operated arcade games, as players would feel cheated if their game ended just because their partner had run out of money! Fortunately, home computer games are free to encourage more collaborative forms of play that require players to interact with each other in order to survive.

## Dependent Cooperation

Dependent cooperation encourages players to interact or collaborate in order to achieve the game's goals rather than just both being on the same side. One way of achieving this is to give your players different roles or skills within the game. For example, in *Wingman Sam* we could make one player control a slow bomber with lots of forward firepower, while the other player has a smaller fighter plane with excellent maneuverability. This then forces players to find collaborative strategies that utilize their own strengths and weaknesses to complete the game's goals.

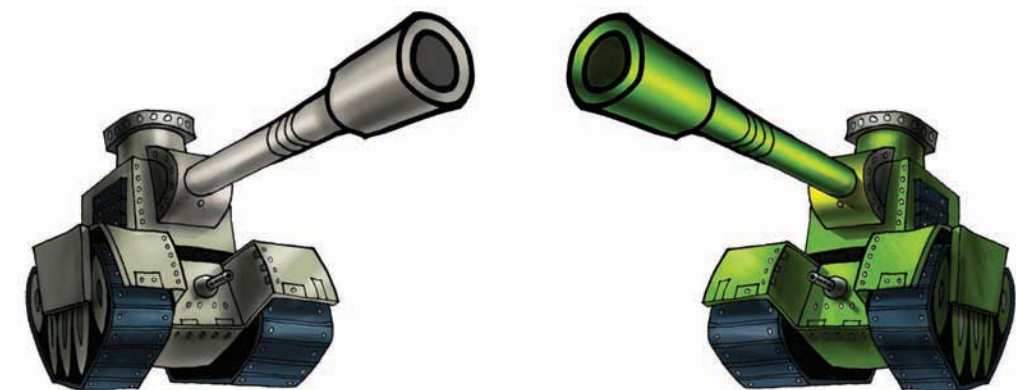
While multiplayer games that encourage collaboration are often more interesting than those requiring simple cooperation, you can't assume that players will naturally know how to collaborate effectively in your game. Modes in which collaboration is essential may be a culture shock for some players, and may require their own kind of training levels. In general, a game design that rewards players for collaborating rather than punishing them for not collaborating should have a broader appeal.

## Mix and Match

In practice, multiplayer games rarely fall neatly into one category. Cooperative games often turn competitive for a while when a health pickup appears on the screen, and competitive games often involve a whole set of unwritten rules that players cooperate to enforce. Even our *Wingman Sam* example encourages collaboration when one player is about to die, as this ends



the game for both players. Multiplayer games rely on human beings interacting with one another, so they are rarely simple or predictable! Nonetheless, they do provide a powerful way to motivate players, allowing designers to create some of the most enjoyable playing experiences around.



## Balanced Beginnings

Although not everyone plays multiplayer games just to win, nobody plays them to lose unfairly. Players need to feel that multiplayer games give them as much chance of winning as other players. Creating different sides with balanced abilities is not an easy task, and is a bit like trying to work out how many apples equal so many pears. In this section, we'll take you through a practical technique for ensuring that your characters are balanced using the Tank War example from the previous chapter.

## Equivalent Characters

Providing all players with directly equivalent features is a sure way to guarantee that no player has an unfair advantage. This kind of equivalence is obvious in Tank War, where, aside from their color, the red and blue tanks are identical. Nonetheless, games may often provide a number of characters to choose from that look completely different but play identically in the game. Equivalent characters can be a good option to include for die-hard gamers who want to prove that they're better than their opponents in a straight fight. However, carefully balanced differences between characters can often make a more interesting multiplayer game and increase the longevity of the gameplay.

## Balancing Differences

Including gameplay differences between characters provides players with meaningful choices right from the start of the game. However, having such choices soon becomes meaningless if players discover that one character always has an advantage over the others. From this point on, the game degenerates into a competition (or fight) to choose that character first as it usually determines the outcome of the game! The players might as well toss a coin to decide the

winner. Making sure that this doesn't happen requires careful planning and thorough play testing. We're going to create heavy and light tanks as balanced alternatives to the basic tank in Tank War. We'll start by making a list of tank characteristics that could be varied to make the game more interesting, as follows:

- Rate of fire—How fast the vehicle can shoot its main weapon
- Shot damage—The amount of damage caused by each shot of its main weapon
- Shot speed—The speed at which shells fly through the air
- Vehicle armor—The proportion of damage that is absorbed by the vehicle's armor
- Vehicle speed—The speed at which the vehicle moves forward
- Rate of turn—The speed at which a vehicle can turn on the spot
- Vehicle size—The size of the vehicle (the larger the size of the target, the easier it is to hit)

We probably won't want to change the vehicle size, but it's worth noting so that we don't accidentally change the gameplay later by making the light tank smaller than the heavy one. Next, let's create a table for each vehicle type and list the strengths and weaknesses we want them to have for each characteristic (see Table 11-1). The idea is to produce a different profile for each vehicle that balances out their abilities. There's not much point in trying to assign relative values to each strength or weakness at this stage. It's not really possible to say, for example, how slow one vehicle's speed should be to compensate for its high shot damage (it's like comparing apples and pears). It may seem safe to assume relationships between some characteristics; if you half the shot speed but double the damage, then it should have the same overall firepower. However, in practice they actually produce two very different weapons that favor different situations and strategies (as should become apparent).

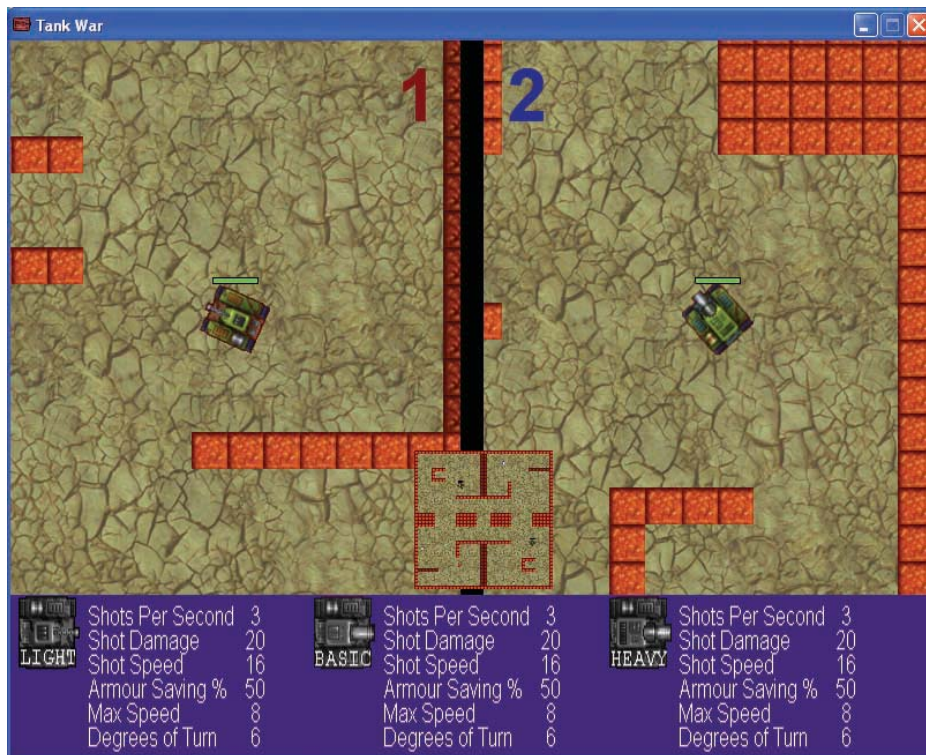
**Table 11-1.** *Tank Characteristics*

Characteristic	Heavy Tank	Basic Tank	Light Tank
Rate of fire	Weakness	--	Strength
Shot damage	Strength	--	Weakness
Shot speed	Strength	--	Weakness
Vehicle armor	Strength	--	Weakness
Max speed	Weakness	--	Strength
Rate of turn	Weakness	--	Strength

Now we need to create a test bed that allows you to alter all these characteristics for each vehicle while you are playing the game. A test bed is not something that the player gets to use, so it doesn't need to look very pretty or have a fantastic interface, as long as it is practical and doesn't crash. We've already created one for you in `Games/Chapter11/new_tank1.gm6` on the CD. This time you'll need to copy it into a directory on your computer along with the file `tankdata.txt`—that's so the game can read and write to the data file. Now load it up and run

the game. Clicking on either tank will cycle between the three different tank types available: light, basic, and heavy. Pressing the Shift key will toggle a test panel at the bottom of the screen, allowing you to tweak the characteristics for each tank (see Figure 11-2). Left-clicking on a value will increase it, and right-clicking will decrease it. The settings are automatically saved and loaded to the data file so that they're not lost next time you play the game. You can look at the contents of this file by double-clicking on it in Windows, and if you're very careful, then you can edit it this way too.

At the moment, the settings for each tank are exactly the same as the original. Your first job is to change them so that they have the strengths and weaknesses given in Table 11-1. It's difficult to know where to start, but just try adding or removing values and seeing what difference they make to the way the tank handles. Changes should take effect immediately, but you may need to close the debug panel for the game to run at full speed again (just press Shift again). Begin by driving and firing each tank on your own until you're fairly happy with the changes that you've made.



**Figure 11-2.** The new version of Tank War features different types of tanks and a panel for changing their characteristics.

Next, find yourself an opponent (preferably someone who is about the same standard as you) and start playing the game. Systematically play every combination of tanks (see Table 11-2) and battle it out in order to establish whether any of the tanks has an unfair advantage over

the others. When you find that one does, make some tweaks to balance things out and try again. Make sure you stick broadly to the original strengths and weaknesses in the table, though—it's no use slowly changing all their settings to be equivalent again! You may have to go through the table several times in order to make sure that your changes haven't unbalanced tanks that you tested earlier. Nonetheless, if you (and your opponent) are prepared to put the effort in, then you should eventually reach a stage where all the tanks are fairly equally matched without being the same.

**Table 11-2.** *All the Combinations of Tanks*

<b>Your Tank</b>	<b>Your Opponent's Tank</b>
Light	Light
Light	Basic
Light	Heavy
Basic	Light
Basic	Basic
Basic	Heavy
Heavy	Light
Heavy	Basic
Heavy	Heavy

Well done—you will have probably learned more about game balance from this exercise than we could ever teach you from just reading about it, but here are a few things that you may have thought about during this process:

- Which characteristics are the most/least important?
- How does this depend on the player's individual abilities and strategies? (Try always driving backwards, if you haven't done so already).
- How much do the characteristics required for balanced tanks change as players get better at the game? (Try watching some beginners play the game now that it's "balanced.")
- Do you still have a preference for one type of tank? Does this matter?
- Would these same settings be useful for creating progression in a single-player mode against the computer?
- Are your settings realistic, and does it matter if they're not?

Give yourself a pat on the back if you already found yourself asking some of these questions when you were playing the game. Asking yourself questions like this is an important part of expanding your own understanding of game design. Unfortunately, there are often no right answers to questions like these—in fact, there are few answers in game design that apply to every situation you'll come across. Becoming an expert at game design (or anything really) is not about learning a set of "answers" from a book, but using your knowledge and experience to ask the right questions in a given situation. Does it matter if the tank settings are realistic?

Well, it depends on who your target audience is. It probably doesn't for the kind of game we're creating here, but if we were making a combat simulation, then realism might be more important to the player than balance.

Okay, so this is one type of balance—where the characteristics of each player are balanced to provide an equal chance of winning from the start of the game. However, there are other ways to achieve balance in games that involve balancing the range of choices that the players get to make throughout the game.

## Balanced Choice

Just because players start off on an equal footing doesn't necessarily mean that the game will remain balanced. In some ways, this is expected—after all, one player has to win the game at some point by gaining an advantage. Nonetheless, it is still important to offer players a fair and balanced opportunity to make choices within the game. In this section, we'll discuss some simple and more advanced techniques for doing this.

## Weighting Choices

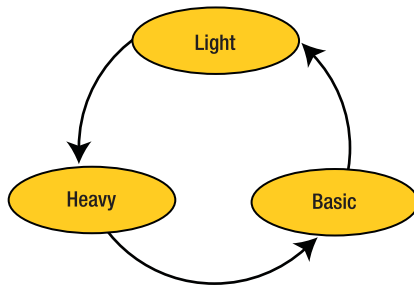
Imagine Tank War as a strategy game, where—rather than driving a single tank—players have to train squads of tanks and send them off to fight battles and conquer territory. As time progresses, each player accumulates resources that they can spend on creating light, basic, or heavy tanks. However, if all the tanks are equally balanced in combat, then deciding which ones to create is not a very meaningful choice. In this situation, it would be better to have a hierarchy of tanks going from light to heavy and weight the choices accordingly. So a heavy tank may be as good as three basic tanks, but it takes three times as long to accumulate the resources needed to build it. Alternatively, we might make it only take twice as long to build but require the player to invest resources building a heavy armor factory first. In this way, we encourage the players to choose between making an early investment for a delayed reward or making an early decisive strike on the enemy.

Weighting choices in this way helps to make the options that players have in the game both interesting and fair. However, once one player has built an army larger than their opponent, then the outcome of the game becomes fairly predictable. Once a player knows they can easily win outright with their superior might, the remainder of the game becomes less interesting for both players. Including cyclic relationships between the tank types can remove this predictability and ensure that even one-sided battles have to be fought with strategies in order to come out on top.

## Cyclic Relationships

If you've ever played rock, paper, scissors, then you've experienced a cyclic relationship (and if you haven't, then you must be from another planet). This ancient game has a balance and simplicity to it that can be applied to more complicated games as well. So far we have discussed a linear hierarchy of tanks where heavy tanks are better than basic tanks, and basic tanks are better than light tanks. This is like rock beating scissors and scissors beating paper. However, paper also beats rock, ensuring that rock doesn't become the dominant choice for

players to make. Admittedly, players might get a bit annoyed if their hard-earned heavy tanks could be beaten outright by light tanks, but that doesn't mean that they couldn't be more vulnerable to light tanks than they might otherwise be. After all, heavy tanks are slow and clumsy compared to light tanks that can hit and run more easily. So we could deliberately build in cyclic vulnerabilities into the relationship so that light tanks punch beyond their weight (in many ways) against heavy tanks, heavy tanks do the same to basic tanks, and basic tanks do the same to light tanks (see Figure 11-3).



**Figure 11-3.** This diagram shows the cyclic relationship in tank types.

Adding this kind of relationship forces a player to fight more tactically, bringing in different tanks to support each other depending on the type of enemy they are facing. This creates a more interesting game, and one in which a clever tactician can turn the tables on his opponent even when fighting against superior numbers.

This same principle can be even applied to our original single combat version of Tank War. Copy the file `Games/Chapter11/new_tank2.gm6` from the CD to the same directory as before. You'll need to make a backup copy of `tankdata.txt` if you want to keep your old characteristics settings, as this program will write over your old file. If you run the game, you'll see that we've changed things a bit; both tanks now start the game as weaponless tank bodies. The weapons now appear as pickups on the map that give you all the characteristics of the appropriate tank until you collect a different one. This means that you can roam around the map switching tank types more or less as you please. While this is already an interesting variation on the original game, you can improve on it by creating a cyclic relationship between the tanks. Try playing the game with a friend and tweaking the characteristics again until the basic tank has an advantage over the light tank, the heavy tank over the basic tank, and the light tank over the heavy tank. One way to achieve this is to make the light tank so nimble that the heavy tank can't catch it, while making its bullets slow enough that they linger around the map after the light tank has moved on. Combined with moving backward and firing, this makes the light tank a nifty opponent—although you'll need to make sure that the medium tank can beat it!

You'll know when you have the right balance; you should notice it turning into a game of cat and mouse, where the cat and mouse keep switching as players try to gain an advantage. When player one picks up the heavy gun, player two goes for the light gun, then player one goes for the basic gun, and so on. This style of play is a lot of fun and illustrates well how cyclic relationships can improve the playability of your multiplayer games.

## Balanced Computer Opponents

Many multiplayer games include the option for computer-controlled opponents to make up the numbers in games. Unsurprisingly, players expect the same level of fairness against computer opponents as human ones. In Chapter 14 you'll make a game with some very simple artificial intelligence (AI), but in this section we'll briefly discuss some of the broader implications of AI for game balance.



### Artificial Stupidity

For computer games, the term *artificial intelligence* is deceptive as it suggests that computer opponents should behave as intelligently as possible. In practice, a balanced computer opponent is one that behaves as humanly as possible, which is as much about human failings as human intelligence. A computer-controlled player without any failings is even more frustrating to play against than a human player with an unfair advantage. To a computer, making the perfect shot is child's play—you're in its world and playing by its rules. It has instant access to everything there is to know about the playing world and can process it a million times faster than you can!

Ideally then, computer opponents in multiplayer games should play by the same rules as you. They should refrain from using their god-like knowledge of the world to see past your diversionary attack, they should go to pieces under pressure, and they should occasionally just screw up because they got distracted! However, while these are good aims for a professional AI programmer, there is nothing wrong with using the computer's natural advantages to help you take your first steps in AI. Make use of every advantage the computer has, but remember to add a few random commands to make it look less than perfect. A player only needs to *feel* that the computer doesn't have an unfair advantage, and what they don't know can't hurt them! Given half a chance, players will naturally project human characteristics onto the behavior of computer opponents. An enemy that occasionally doesn't shoot when there's a clear shot can look like they haven't spotted the player. An opponent who randomly spins out on the corner of a racetrack can be seen as caving under pressure. A liberal use of the random command can hide a multitude of programming sins and give your artificial intelligence the apparent human fallibility that it needs.

## Summary

In this chapter we've looked at different kinds of balance in multiplayer games, but many of the same principles apply to balancing computer opponents in single-player games as well. Balance is ultimately about ensuring that players have a fair chance of winning the game, and that's what players want in single-player games too. Here's a summary of the main points discussed in this chapter to help you with designing your own multiplayer games:

- Make multiplayer games more fun to play by
  - Including competition and cooperation.
  - Making players' interactions with the game dependent on each other.
  - Balancing the game for players.
- Multiplayer games can be balanced by
  - Providing equivalent characters (less desirable).
  - Providing balanced characters with different strengths and weaknesses.
  - Weighing choices to provide interesting trade-offs.
  - Including cyclic relationships to provide richer gameplay.
- Characteristics can be balanced by
  - Creating a test bed that allows characteristics to be tweaked in real time.
  - Play testing.
  - Play testing.
  - More play testing.

Well, that's it for another part of the book. This is the last chapter on game design, but there's still plenty more to learn about game programming. Prepare yourself for a few frights as you go exploring for ancient treasure in haunted Egyptian tombs. On the way, you'll learn about the deeper workings of Game Maker and discover the power behind the icons—in the form of the programming language GML. Good luck!







PART 5



# Enemies and Intelligence

**You've come a long way, but Game Maker's greatest treasure still lies undisturbed . . .  
Dare you unleash the arcane powers of GML?**



## CHAPTER 12



# GML: Become a Programmer

**S**o far we've controlled the behavior of the different objects in our games using events and actions. These actions let the instances of the object perform tasks when certain events occur in the game. In this chapter we are going to define those tasks in an alternative way: by using programs. Programs define tasks through lines of text called *code* that use *functions* instead of actions. This extends the scope of Game Maker considerably as there are only about 150 different actions but close to a thousand *functions*. These *functions* give you much more control than actions, allowing you to define precisely how tasks should be performed in different circumstances.

The text in a program needs to be structured in a very particular way so that Game Maker can understand what you mean. Communicating with Game Maker in this way is like learning a new language with its own special vocabulary and grammar. The programming language Game Maker uses is called GML, which stands for Game Maker Language. If you have written programs before in languages like Java or C++, then you will notice that GML is rather similar. However, every programming language has its own peculiarities, so you will need to watch out for the differences.

Before we go into more detail about GML, you need to know how to tell Game Maker to execute a program using a script resource. Script resources are similar to other resources like sprites and backgrounds in the way they are created and accessed through the resource list. You create a script resource by choosing **Create Script** from the **Resources** menu, and then typing your program into the text editor that appears. Once you've created your script, you can include an **Execute Script** action to call the script in a normal event just as you would for any other action. We'll see how this works in more detail next.

---

**Note** You can also use an **Execute Code** action for including GML. Dragging this action into an event will cause the editor to pop up so that you can type the program directly into the event. Although this method is sometimes easier, it is often better to use scripts because they can be reused more easily.

---

To help you come to grips with GML, in this chapter we're not going to create a whole game but just small examples. Unfortunately, because of its size, we cannot cover all of GML, but we will discuss many key aspects. You can always refer to the Game Maker documentation for a complete overview.

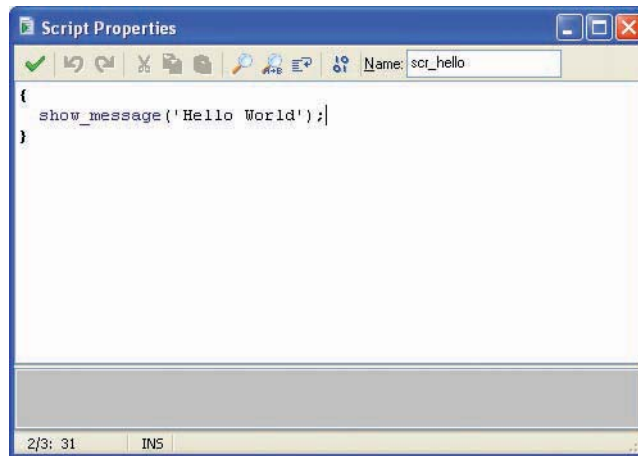
## Hello World

We'll start with a traditional program to demonstrate how to write some simple code. We're going to create a script that shows the message "Hello World" on the screen.

### Creating a simple script:

1. Start a new game.
2. Choose **Create Script** from the **Resource** menu. The script editor shown in Figure 12-1 will appear.
3. In the **Name** box in the toolbar, give the script the name `scr_hello`.
4. In the editor, type the following piece of code:

```
{
    show_message('Hello World');
}
```
5. Press the 10/01 button in the toolbar. This will test the program and display an error message if you made a mistake.
6. Close the editor by clicking the green checkmark in the toolbar.



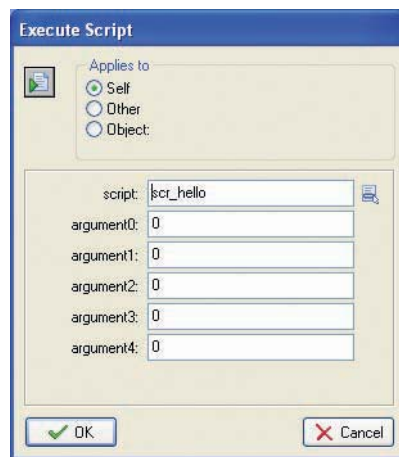
**Figure 12-1.** Enter this code in the script editor.

Note that Game Maker shows parts of the code in different colors. This color-coding helps you know when your code is written correctly. For example, we know that `show_message` is the correct name for one of Game Maker's built-in functions because it has turned blue. If we had made a spelling mistake, then it wouldn't turn blue and we would know something was wrong. It is also particularly important to give your scripts meaningful names; that way, you can remember what the script does when you use it in an action or some other code.

Before we can see what this code does, we need to execute it. To do so, we must create a new object with a key press event that executes the script.

#### Executing the script:

1. Create a new object and add a **Key press**, **<Space>** event to it.
2. Include the **Execute Script** action (**control** tab) and select the `scr_hello` script from the menu. The arguments can all be left at 0 since we do not use arguments in this script (more about these later). The action should look like Figure 12-2.
3. Create a room and place one instance of the object in it.



**Figure 12-2.** This action executes the `scr_hello` script.

Now run the game and press the spacebar. If you did everything correctly, a message box should pop up containing the text “Hello World”. If you made a mistake in your script, then Game Maker will report an error when the game loads or when the script is executed. If you do get an error, you should check the script carefully for typing errors. Even using an uppercase letter rather than lowercase can cause an error in GML—so take great care. You’ll also find this short program in the file `Games/Chapter12/hello_world.gm6` on the CD.

Now let’s consider what this script does. The first and last lines contain curly brackets. Different kinds of brackets signify different things in GML, and curly brackets mark the beginning and end of a block of code (a bit like the **Start Block** and **End Block** actions). However, in GML every program must start with an opening curly bracket and must end with a closing bracket. Curly brackets enclose a block of code. Such blocks of code will also be used later at other places.

The program consists of just one command. Such commands are called *statements*. A program consists of one or more statements. A statement ends with a semicolon. In this way, Game Maker understands where one statement ends and the next one begins. Don’t forget the semicolons!

The statement in our program is a call to the function `show_message()`. Functions can be recognized because they have a name and then (optionally) some arguments between the parentheses. We have already used such functions before as arguments in actions. For example, we've used the `random()` function in a number of places. Much like actions, functions perform certain tasks. The `show_message()` function has one argument, which is the text to be displayed; `'Hello World'` is that argument. Take note of the single quotes around it as they indicate that this is a string (text). Also note that to make functions easier to recognize and to indicate that you typed their name correctly, they are displayed in a dark blue color.

So when the script is executed, the one statement in it is executed, which shows the alert box containing the text that is provided as an argument. Of course, we could have achieved the same thing using the **Show Message** action. But as we will see later, by using scripts we can do many new things.

## Variables

We have used variables in previous chapters. In some cases, we have used the value of a variable (such as an object's position or speed) as a parameter to an action to change its behavior. Sometimes we have used an action to change the value of a variable. Let's now look in more detail at the use of variables, and see how to use them in scripts.

Variables are containers that store values—these values can be changed throughout the course of a game. Such a value can either be a number or a string. A variable is given a name that we can refer to, and we use the name to inspect the current value and to change it. There are a number of built-in variables, like `x` and `y`, which indicate the position of an instance, or like `speed` and `direction`, which indicate the speed and direction in which an instance is moving. Changing a variable in a script is very simple. We use the `=` symbol to assign a new value. For example, to set an instance in the middle of the room, we could use the following piece of code:

```
{
    x = 320;
    y = 240;
}
```

The program starts and ends with curly brackets, as any program must do. There are two statements here, each ending with a semicolon. These both are *assignment statements*, or *assignments* for short. An assignment assigns a value to a variable. The first assignment assigns the value `320` to the variable `x`, which is the horizontal position of the instance. The second assigns the value `240` to the variable `y`, which is the vertical position of the instance. Note that when you type in the piece of code, the variable names will become blue. This color is used for built-in variables.

Rather than assigning a simple value to a variable, we can write a complete expression involving operators (`+`, `-`, `*`, `/`, and a couple more) and values and other variables. For example, the previous piece of code assumes that the room is 640×480. If we don't know this, we can also write the following:

```
{
    x = room_width/2;
    y = room_height/2;
}
```

Here, `room_width` and `room_height` are two variables that indicate the width and height of the room; by dividing them by 2, we get the middle of the room. We can also use functions in the expressions. For example, as you have seen before, the function `random()` gives a random number smaller than its argument. To move our instance to a random position on the screen, we can use the following code:

```
{
    x = random(room_width);
    y = random(room_height);
}
```

Rather than using the built-in variables, we can also create our own. You do so simply by picking a name and assigning a value to it. Names of variables should only consist of letters and numbers and the underscore ( `_` ) symbol. The first character cannot be a number. You should always make sure that your variable names differ from the names of resources or from existing variables or function names. It is vitally important to realize that variable names are case-sensitive—that is, `Name` is not the same as `name`.

Let's make our "Hello World" example a bit more personal. We will ask the player for their name and then greet them in a personal way. You can find the program in the file `Games/Chapter12/hello_you.gm6` on the CD.

```
{
    yourname = get_string('Who are you?', 'nobody');
    show_message('Hello ' + yourname + '. Welcome to the game.');
```

We use a new function here called `get_string()`. This function asks the player to enter some text. The function has two arguments. The first one indicates the question to display, and the second one is the default answer. Note the single quotes around the strings. Also note that the two arguments are separated by a comma. Arguments in functions are always separated by commas. This function returns the text that the player typed in. The assignment assigns this value to a new variable we call `yourname`. Next we use the function `show_message()` again to display a message. The argument, though, might look a bit weird. We include three strings, one of which is stored in the variable `yourname`. When used on strings, the `+` operator will simply put the strings next to each other—that is, it concatenates the strings. Better try this out in the same way we did for the "Hello World" example.

As you might have realized, the program we just wrote does something that cannot be achieved when only using drag-and-drop actions. It asks the player for some information. Scripts are a lot more powerful than drag-and-drop actions. That is why it is very useful to spend some time learning to write scripts.

You might wonder what happens to the variable `yourname` after the script has finished. Well, it is actually stored with the instance. Later on, in other scripts for this instance, we can still use it. So the user has to enter their name only once. But other instances of the same object or other objects cannot access it directly. It belongs to this particular instance.

There are actually three different types of variables. By default, the variable is stored with the instance and remains available. Each instance can have its own variable using that name. For example, each instance has its own `x` and `y` variables. Sometimes, when you use a variable in a script, you don't want it to remain available when the script is finished. In this case, you need to indicate that it is a variable inside the script, as follows:



```
{
    var yourname;
    yourname = get_string('Who are you?', 'nobody');
    show_message('Hello ' + yourname + '. Welcome to the game.');
```

By adding the line `var yourname;` we indicate that the variable `yourname` is defined only in this script. We call it a *local* variable. You can put multiple variables here, with commas separating them. We strongly recommend that you make a variable local whenever possible; it speeds things up and avoids errors when you're reusing the same variables.

Sometimes, though, you want the variable to be available to all instances of all objects. For example, multiple instances might want to use the name of the player once it has been asked using the `get_string()` function. In this case, we want to make the variable global to all instances. As we have seen in previous chapters, we can achieve this by adding the keyword **global** and inserting a dot in front of the variable name, as follows:

```
{
    global.yourname = get_string('Who are you?', 'nobody');
    show_message('Hello ' + global.yourname + '. Welcome to the game.');
```

Note that the word **global** appears in boldface. It is a special word in the GML language; such keywords are always shown in bold. The word **var** is also a keyword. We will see many more of these in this chapter. In some sense, the curly brackets are also keywords and appear in bold as well.

To summarize, we have three different ways in which we can use variables:

- Belonging to one instance, by simply assigning a value to them
- Local to a script, by declaring them inside the script using the keyword **var**
- Global to the game, by preceding the name with the keyword **global** and a dot

---

**Note** Global variables remain available when you move to a different room. So, for example, you can ask the player for his name in the first room, store the name in a global variable, and then use that variable in all other rooms in the game.

---

## Functions

Functions are the most important ingredients of a program; they are the things that let Game Maker perform certain tasks. For every action there is a corresponding function, but there are many, many more. In total, close to a thousand functions are available that deal with all aspects of your game, such as motion, instances, graphics, sound, and user input.

We have already used a few functions earlier. Also, you saw two different types of functions. Functions, like the `show_message()` function, only perform certain tasks. Other functions, like `random()` and `get_string()`, return a value that can then be used in expressions or assignments.

When we use a function, we say that we *call* the function. A function call consists of the name of the function, followed by the arguments, separated by commas, in parentheses. Even when a function has no arguments, you must still use the parentheses. Arguments can be values as well as expressions. Most functions have a fixed number of arguments, but some can have an arbitrary number of arguments.

Let's look at some examples. GML provides a large number of functions for drawing objects. These functions should normally only be used in the **Draw** event of objects. There are functions to draw shapes, sprites, backgrounds, text, and so on. If you have registered your version of Game Maker, you also have access to lots of additional drawing functions for creating colored shapes, rotated text, and even three-dimensional objects. To use some drawing functions, create a script with the following code:

```
{
    draw_set_color(c_red);
    draw_rectangle(x-50,y-50,x+50,y+50,false);
    draw_set_color(c_blue);
    draw_circle(x,y,40,false);
}
```

This piece of code first calls a function to set the drawing color. `c_red` is a built-in value that indicates the red color. Next, this code draws a rectangle with the indicated corners. The fifth argument, which has the value `false`, indicates that this must not be an outlined rectangle. Next, the color is set to blue, and finally a filled circle is drawn. To use this script, create an object (it does not need a sprite), and then add a **Draw** event and include an **Execute Script** action to call the script. Add a number of instances of the object to a room and check out the result. You can find the program in the file `Games/Chapter12/draw_shapes.gm6` on the CD.

As a second example, let's consider the creation of instances. Often during games you want to create instances of objects. The function `instance_create()` can be used for doing just this. This function has three arguments. The first two arguments must indicate the position where you want to create the instance and the third argument must indicate the object of which the instance must be created. Assume we want to create a bullet and fire it in the direction of the instance that creates it. This can be achieved with the following piece of code (which assumes that an object `obj_bullet` exists):

```
{
    var bullet_instance;
    bullet_instance = instance_create(x,y,obj_bullet);
    bullet_instance.speed = 12;
    bullet_instance.direction = direction;
}
```

In this code we first declare a local variable `bullet_instance` that is only available in this piece of code. Next, we call the function to create the instance. This function returns an ID of the new instance, which we store in the variable `bullet_instance`. An ID is simply a number that can be used to refer to the particular instance. To set the speed of this instance, we change the variable `speed` in the instance to `12`. We do this by using `bullet_instance.speed`, similar to how we addressed variables in objects. In the same way, we set the direction of instance `bullet_instance` to the direction of the current instance.

## Conditional Statements

As you know, in Game Maker there are conditional actions that control whether or not a block of actions is executed. When using scripts, you can use *conditional statements* in a similar way. A conditional statement starts with the keyword `if` followed by an expression between parentheses. If this expression evaluates to `true`, the following statement or block of statements is executed. So a conditional statement looks like this:

```
if (<expression>
{
    <statement>;
    <statement>;
    ...
}
```

The statements between the curly brackets are only executed when the expression is true. In the expression, you can use comparison operators to compare numbers as follows:

- `<` means smaller than.
- `<=` means smaller or equal.
- `==` means equal (note that we need two `=` symbols, to distinguish the comparison from the assignment).
- `!=` means unequal.
- `>=` means larger or equal.
- `>` means larger than.

You can combine comparisons using `&&` (which means and), `||` (which means or), or `!` (which means not). For example, to indicate that the `x` value should be larger than `200` but smaller than the `y` value, you can write something like this: `(x > 200) && (x < y)`.

---

**Note** The use of parentheses (like in the previous example) is not always strictly necessary, but it can help make things a lot clearer.

---

Let's consider an example. We want an instance to wrap around the screen—that is, if its x-position gets smaller than 0 while it is moving to the left, we change the x-position to 640, and when it gets larger than 640 and it is moving to the right, we change it to 0. Here is the corresponding piece of code:

```
{
  if ( (x<0) && (hspeed<0) )
  {
    x = 640;
  }
  if ( (x>640) && (hspeed>0) )
  {
    x = 0;
  }
}
```

Note the use of opening and closing brackets. It is important to carefully check whether they match. Forgetting a bracket is a very common error when writing programs. This code could even be a bit more compact. Note that in a program the layout of the code is up to you. We used a rather standard layout style in this example; we used indents to show which bits of code belong together. Although employing a clear layout style is useful, in this case we've overdone it a bit. Also, when only one statement follows a conditional statement, we do not need the curly brackets. So, we might as well use the following piece of code, which does exactly the same thing:

```
{
  if ( (x<0) && (hspeed<0) ) x = 640;
  if ( (x>640) && (hspeed>0) ) x = 0;
}
```

As with conditional actions, the conditional statement can have an `else` that is executed when the condition is not true. The structure then looks as follows:

```
if (<expression>)
{
  <statement>;
  ...
}
else
{
  <statement>;
  ...
}
```

For example, assume that we have a monster that should move horizontally in the direction of the player. So, depending on where the player is, the monster should adapt its direction. If the player is an instance of the object `obj_player`, we can use the following piece of code:

```

{
    if (x < obj_player.x)
    {
        hspeed = 4; vspeed = 0;
    }
    else
    {
        hspeed = -4; vspeed = 0;
    }
}

```

We test whether the x-coordinate of the current instance (the monster) is smaller than the x-coordinate of an instance of object `obj_player`. If so, we set the motion to the right; else, we set the motion to the left. As we saw earlier, we can address the value of a variable in another instance by preceding it with the ID of that instance and a dot. This time we do it slightly differently, and indicate the object that the instance belongs to. This will work equally well, assuming just one instance of `obj_player` exists. If there are several player objects, however, only one of them is chosen for comparison and the code may not function as you expect.

## Repeating Things

Often in code you want to repeat certain things, and there are a number of ways to achieve this. We call such a piece of code a *loop*. We will start with the easiest way to repeat a piece of code a given number of times: the `repeat` statement. It looks like this:

```

repeat (<expression>)
{
    <statement>;
    <statement>;
    ...
}

```

The expression must result in an integer value; the block of code following it is executed that number of times. Let's look at an example. In the following program we ask the player for a value and then create the given number of balls at random positions. It assumes that an object `obj_ball` exists.

```

{
    var numb;
    numb = get_integer('Give the number of balls:',10);
    repeat (numb)
    {
        instance_create(random(640),random(480),obj_ball);
    }
}

```

The program asks the player for an integer number, using the function `get_integer()`, and stores it in the local variable `numb`. Next it repeats a piece of code `numb` times, each time creating a new ball at a random position. The example `Games/Chapter12/balls.gm6` on the CD uses this script.

A second way of repeating code is when we want to repeat something as long as some expression is true. For this we use the `while` statement:

```
while (<expression>)
{
    <statement>;
    <statement>;
    ...
}
```

The `while` statement is similar to the `if` statement, but there is an important difference: when the expression is true, the block after the `if` statement is executed exactly once. However, the block after the `while` statement is executed for as long as the condition is true. If you are not careful, this code can loop forever, in which case the game will no longer respond.

In this example, we are going to draw 40 concentric circles in different colors:

```
{
    var i;
    i = 0;
    while (i<40)
    {
        var color;
        color = make_color_rgb(random(256),random(128),random(64));
        draw_set_color(color);
        draw_circle(x,y,2*i,true);
        i = i+1;
    }
}
```

---

**Note** A color is represented as a number. Such a number consists of three parts, one to indicate the blue component, one to indicate the green component, and one to indicate the red component. Each of these components is a number between 0 and 255. Here we use the function `make_rgb_color()` to combine these three components (each a random number) into a single color. In order to make the effect prettier, we have biased the numbers toward red and orange by using a larger red component than green and blue. It may seem confusing to use `random(256)` rather than `random(255)` to get a number between 0 and 255, but the `random()` function always returns a number less than the parameter passed in, so if we used 255 we would not get the full color range.

---

In the program we first initialize the local variable `i` with a value of 0. Next, we run a loop as long as `i` is smaller than 40. In the loop we create a random color using the function `make_color_rgb()`, and set the drawing color. Next we draw a circle with radius `2*i`, and finally we increase `i`. After the loop has been executed 40 times, the condition becomes false and the loop ends. The example `Games/Chapter12/draw_circles.gm6` on the CD uses this script to create a rather funny effect.

This type of `while` loop—in which we first initialize a variable, test the value in the `while` statement, and increase the variable in every execution of the loop—occurs very often. Hence, there is an easier way to write it: the `for` loop. It looks like this:

```
for (<initialize>;<condition>;<increment>)
{
    <statement>;
    <statement>;
    ...
}
```

The `initialize` statement is executed once before the loop starts. The condition is then checked at the beginning of each loop execution to see whether the loop must still be executed. The `increment` statement is executed at the end of each loop execution. So our circle example can also be written as follows:

```
{
    var i;
    for (i=0; i<40; i=i+1)
    {
        var color;
        color = make_color_rgb(random(256),random(128),random(64));
        draw_set_color(color);
        draw_circle(x,y,2*i,true);
    }
}
```

Note that loops can contain further loops. For example, to create 10 columns comprising eight balls each, we can use the following piece of code:

```
{
    var i,j;
    for (i=0; i<10; i+=1)
    {
        for (j=0; j<8; j+=1)
        {
            instance_create(40*i,40*j,obj_ball);
        }
    }
}
```

Note that we use the expression `i+=1`. This is a shortcut for `i = i+1`. It adds 1 to the variable `i`.

---

**Warning** During the execution of a `repeat`, `while`, or `for` loop, no events or actions are processed. So make sure the loop always ends and does not take too long as this might interrupt the game play.

---

## Arrays

Variables are infinitely useful in scripts, but they can only store one value each—in a number of situations you'll want to store a whole collection of values, and *arrays* allow us to do this. An array can store an arbitrary number of values. To get a particular value out of an array you have to specify the index. To this end, you put the index between square brackets. For example, if `aaa` is an array, you can obtain the fifth element in it by typing `aaa[5]`. Using arrays in GML is very easy. There is no need to declare them or to specify their length. Only when you want a local array should you declare it using the `var` keyword. Let's create a simple example in which we compute and store the squares of the numbers 1 through 30, and then draw them on the screen:

```
{
    var i, squares;
    for (i=1; i<=30; i+=1)
    {
        squares[i] = i*i;
    }
    for (i=1; i<=30; i+=1)
    {
        draw_text(10,15*i,string(squares[i]));
    }
}
```

The first loop fills the array with the values, and the second loop draws them on the screen. Note that the function `string()` turns the integer value into a string, which can then be drawn. Clearly this script is a bit useless; there is no need to first store the values in the array. You might as well have drawn them directly.

---

**Warning** Make sure that you first store a value into an element of an array before you try to retrieve it.

---

Let's now consider a slightly more interesting example. Assume we want to make a game containing a number of math questions. The game should ask a random question and then check the answer that the player gives. We can store all questions and answers in an array. We need two scripts: one that fills the array and one that asks the questions. The first script looks like this:



```

{
    question[0] = 'How much is 12+34?';
    question[1] = 'How much is 4*6?';
    question[2] = 'How much is 72/9?';
    question[3] = 'How much is 56-23?';
    question[4] = 'How much is 7*11?';
    question[5] = 'How much is 71+24?';
    question[6] = 'How much is 84/7?';
    question[7] = 'How much is 69-37?';
    question[8] = 'How much is 45+74?';
    question[9] = 'How much is 12*12?';
    answer[0] = 46;
    answer[1] = 24;
    answer[2] = 8;
    answer[3] = 33;
    answer[4] = 77;
    answer[5] = 95;
    answer[6] = 12;
    answer[7] = 32;
    answer[8] = 119;
    answer[9] = 144;
}

```

Note that we do not use local variables. The arrays are now stored in the instance. This script should be called just once when the instance is created. The second script asks a question and checks the answer:

```

{
    var ind,val;
    ind = floor(random(10));
    val = get_integer(question[ind],0);
    if (val == answer[ind])
        show_message('That is the correct answer.')
    else
        show_message('That answer is WRONG!');
}

```

We create a random integer index between 0 and 9. Next we ask the corresponding question and store the result in the variable `val`. Finally we compare `val` with the correct answer and display the appropriate message. You can call this script, for instance, in a mouse press event. For an example, see [Games/Chapter12/quiz.gm6](#) on the CD.

---

**Note** The index of an array must be 0 or larger.

---

The arrays we've seen so far store a row of values. In some situations, we may need a complete matrix of values—for example, to represent the stones in a playing field. This is achieved very simply. Rather than using one index we use two indexes, separated by a comma. So you can write `aaa[5,8]`. To illustrate, let's create a multiplication table in which `mult[i,j]` is equal to `i*j`. The following piece of code achieves this:

```
{
  var i, j, mult;
  for (i=1; i<=10; i+=1)
  {
    for (j=1; j<=10; j+=1)
    {
      mult[i,j] = i*j;
    }
  }
}
```

In the next chapter we will see a number of such 2-dimensional arrays as we represent the playing field for a Tic-Tac-Toe game in this way.

## Dealing with Other Instances

As you know, you can apply actions to other instances. Sometimes within a script, you also want to apply some code to other instances or objects. As we saw in the bullet example earlier in the chapter, we can change a variable in another instance or object by preceding it with the index of the instance or object and a dot. To apply more complicated code to other instances, we can use the `with` statement. In a `with` statement, you specify the instance or object to which a block of code must be applied. Globally it looks like this:

```
with (<index>)
{
  <statement>;
  <statement>;
  ...
}
```

The index should either be the index of an instance, as obtained when calling the `instance_create()` function, or the index of an object (normally by giving its name), or the keyword `other` to apply the block of code to the other object in a collision event. Here is an example that destroys all balls in a game. `obj_ball` is the ball object.

```
{
  with (obj_ball)
  {
    instance_destroy();
  }
}
```

And here is another example in which we create a bullet with a speed and direction of motion:

```
{
  var bullet_instance;
  bullet_instance = instance_create(x,y,obj_bullet);
  with (bullet_instance)
  {
    speed = 12;
    direction = other.direction;
  }
}
```

Note that we use `other.direction` within the block of code. Here, `other` refers to the object for which the code was originally executed.

The `with` statement is very powerful, and understanding it will help you a lot in creating effective code.

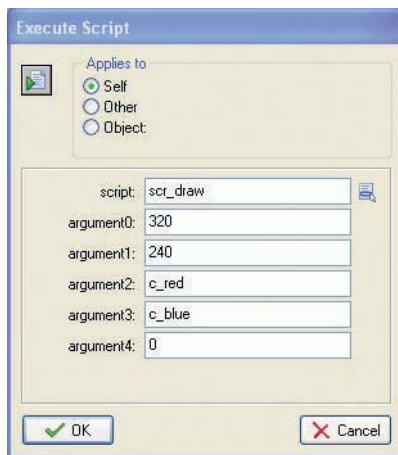
## Scripts As Functions

In the **Execute Script** action, you can indicate a number of arguments. The values of these arguments can then be used inside the script. In this way, you can create more generic scripts that can be used at multiple places. Inside the script you can obtain the values of the arguments using the variables `argument0`, `argument1`, `argument2`, and so on. (You can only obtain the values from these variables; you cannot change them.)

Next, we create a script that draws a square and a filled circle at a given position on the screen. We will use four arguments: two that indicate the position and two that indicate the color of the square and the filled circle. Here's the script:

```
{
  draw_set_color(argument2);
  draw_rectangle(argument0-50,argument1-50, argument0+50,argument1+50,false);
  draw_set_color(argument3);
  draw_circle(argument0,argument1,40,false);
}
```

Note that this script is largely the same as the one we showed you earlier, but this time we used arguments to set the color and determine the position for drawing. To use the script we include the **Execute Script** action in the **Draw** event of an object and indicate the value of the arguments, as in Figure 12-3.



**Figure 12-3.** This action executes a script using arguments.

You can now use this script to draw the shape at other positions and with other colors as well. You could also add the size of the shape as an additional argument to make the script even more generic.

Scripts that you create yourself can be called inside other scripts in exactly the same way as you call functions. You simply use the name of the script and add the arguments between parentheses with commas between them. Remember that even if the script has no arguments, you still need to include the parentheses but just leave them empty.

For example, the following script uses the `scr_draw` script to draw 100 shapes at random locations with random colors:

```
{
  repeat (100)
  {
    var color1;
    var color2;
    color1 = make_color_rgb(random(256),random(256),random(256));
    color2 = make_color_rgb(random(256),random(256),random(256));
    scr_draw(random(640),random(480),color1,color2);
  }
}
```

You can call this script in the **Draw** event of an object. If you run the program you will notice that the colors and positions change every step. This is of course precisely what we indicated. If you want the positions and colors to stay the same, you could have stored them in arrays. You can find this program in the file `Games/Chapter12/draw_wild.gm6` on the CD.

---

**Note** Scripts can actually call themselves. This is known as a recursive call. Be very careful with using recursive calls, however, because they can easily lead to errors in which the script continues to call itself forever.

---

We have seen that functions can return a value that can then be used in expressions. We can also let a script return a value so that we can use it in expressions. For this you use the `return` statement. Here is an example of a script that returns the squared value of its argument:

```
{
    return argument0*argument0;
}
```

Note that the execution of a script stops after a `return` statement. The rest of the script will no longer be executed. As an example, here is a script that checks whether all 10 entries of an array are equal to 0:

```
{
    var i;
    for (i=1; i<=10; i+=1)
    {
        if (aaa[i] != 0) return false;
    }
    return true;
}
```

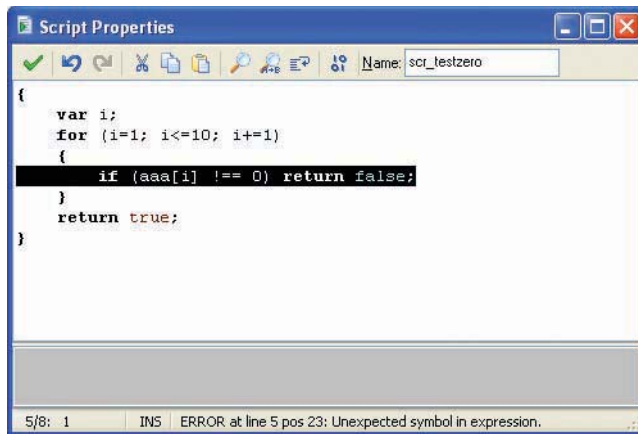
Whenever one of the entries is not equal to 0, the value `false` is returned and the execution is stopped. Only when all entries are 0 is the final `return true;` statement executed.

Rather than putting your whole code in one script, it is often a good idea to split it into several scripts and let these scripts call each other. That makes the code more readable and makes it possible to use the same piece of code multiple times.

## Debugging Programs

When creating scripts, you can easily make errors. You might misspell a function name or forget a closing bracket. When running the game, this can result in two types of errors.

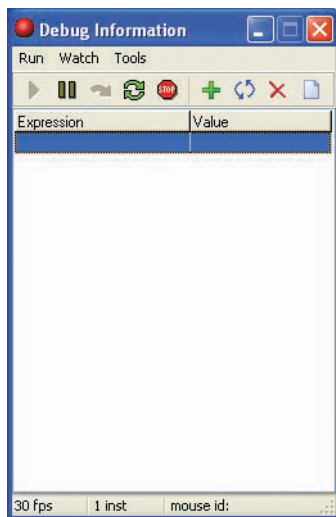
You might get an error during the loading of the game. This happens when you forget a bracket, for example. The error message indicates in which script and on which line the error occurs, and provides a brief explanation of the error. Carefully reading the error message will help you solve the problem. You can also check for such problems by clicking the 10/01 button in the toolbar of the script editor. A check is performed and when there is an error, this is reported. The editor highlights the line where the error occurs, as shown in Figure 12-4. We strongly recommend that you do this check after each change you make in the script.



**Figure 12-4.** After pressing the 10/01 key, an error is indicated in the script.

A second type of error can occur while a user is playing of the game—if you use a nonexistent variable, for example. Again an error message is displayed that should help you correct the error. Such errors are more difficult to find. They typically occur when you’ve misspelled a variable name or when you have used a variable name that was also the name of a resource. So you must be careful when writing code.

Game Maker contains a debugger. When you run the game using the feature **Run in Debug Mode** (located in the **Run** menu), an additional window will pop up, as shown in Figure 12-5.



**Figure 12-5.** When running the game in debug mode, this debug window is shown.

The debug window allows you to pause and continue the game or step through it. Also, you can check the value of different variables and even execute some code. For more details, see the Game Maker documentation.

## Congratulations

You have now mastered the basics of programming in the GML language. We hope you have realized that programming is not so difficult after all. Once you get the hang of it, programming in GML is actually a lot easier and faster than using the drag-and-drop actions that you used in previous chapters. Using scripts opens up a whole new set of possibilities in Game Maker. More than half of the documentation of Game Maker deals with functions and built-in variables that you can use to enhance your games in many ways. You will gain a lot more control over instances, rooms, views, sound, and graphics.

We did not describe the GML in full; it provides a number of additional statements and constructions you can take advantage of. But we examined the ones that you will need most. Once you are a bit more experienced with using the language, we recommend that you read the GML section in the documentation, which covers the language in full.

In the next chapter we'll create a game in which we only use scripts. The game will contain just one instance of a single object in which just three scripts are called. This may sound rather simple, but actually it is the first game in which the computer will be intelligent.

## CHAPTER 13



# Clever Computers: Playing Tic-Tac-Toe

In this chapter we are going to create a version of the game tic-tac-toe in which you are pitted against an intelligent computer opponent. This opponent must have a strategy that will regularly beat the player to keep it challenging, but the computer opponent must not be too strong; otherwise the player has no chance of winning, and will quickly become frustrated and give up. We will also show how the computer can adapt its play to the level of the player. The game will be almost completely written using the GML programming language, so make sure you read and understood Chapter 12 on GML before starting this chapter.

## Designing the Game: Tic-Tac-Toe

I am sure you'll know how to play *Tic-Tac-Toe*, but even so, it is good to describe it carefully before we start to aid you in making the game.

*The game of Tic-Tac-Toe is played on a 3×3 grid. The computer player uses red stones while the human player uses blue stones. The players take turns placing a stone of their color on an empty cell. When a player manages to create a horizontal, vertical, or diagonal row of three stones of his color, he wins the game. When all cells are filled and no row is created, the game ends in a draw.*

*The player uses the mouse to place the stones. The Esc key is used to end the game. The game consists of an arbitrary number of rounds. In each round the player who lost the previous round will start. The number of wins for each player and the number of draws are recorded, and displayed on the game interface for reference. Figure 13-1 shows the game in action.*

The game requires just a few ingredients: the playing field, the stones, and a mechanism to show the number of wins. The most complicated part will be how to determine the moves for the computer player. All the resources can be found in the [Resources/Chapter13](#) folder on the CD.



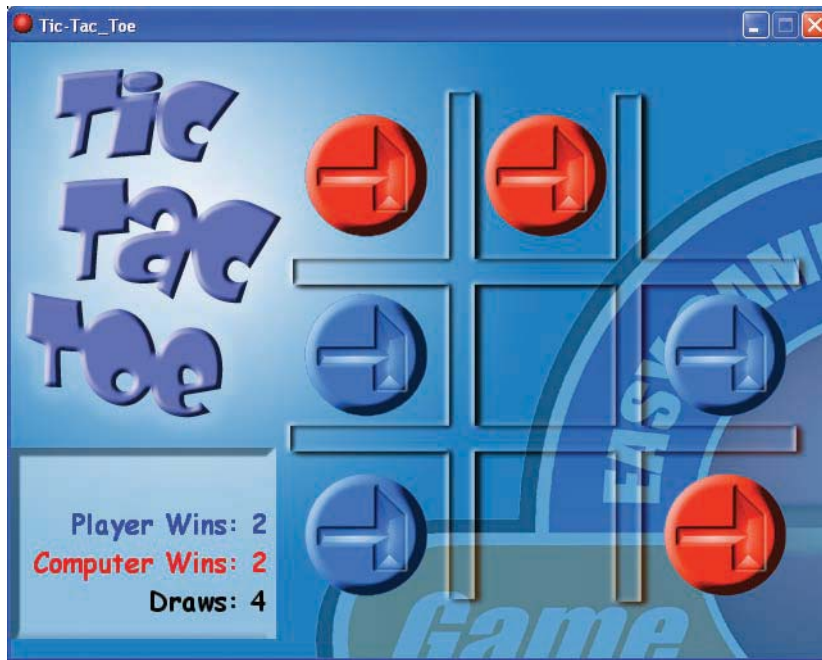


Figure 13-1. The Tic-Tac-Toe game looks like this.

## The Playing Field

We will first need two sprites for the stones, a background to contain the playing field, and some sound effects. As this is a game in which the player is supposed to think a lot, background music is not really appropriate, so we won't use it.

### Creating sprites, a background, and sound effects:

1. Start a new game.
2. Create a new sprite using the file `Stone1.gif` from the `Resources/Chapter13` folder on the CD.
3. Create another sprite using the file `Stone2.gif`.
4. Create a background using the file `Background.bmp`.
5. Finally create sound effects using the files `Place.wav`, `Win.wav`, `Lose.wav`, and `Draw.wav`.

We will also need a font for our game. We will use this to draw the score, that is, how many games are won by the player and the computer.

**Creating a font:**

1. Create a new font for the game. We named ours `fnt_score`. Select a nice font; for example, **Comic Sans MS**, give it a size of **16**, and select **Bold**.

The playing field is where all the action happens. We will create just one object in the game, which represents the playing field, and there will be just one instance of this object in the game. This object does not need a sprite, as the field is already drawn on the background.

**Creating the field object and the room:**

1. Create a new object. Give it the name `obj_field`. No sprite is required.
2. Create a new room. In the **backgrounds** tab, assign to it the background.
3. In the **settings** tab, give the room an appropriate caption.
4. In the **objects** tab, add one instance of the field object at an arbitrary place.

You might want to run the game just to test that the playing field is indeed there. Obviously, nothing can be done at this stage, as we are yet to specify the behavior for the field object. We will only use scripts for this.

Internally we represent the playing field with a variable `field` that will be a two-dimensional array. This variable represents the cells in the field, as shown in Figure 13-2. Each entry can have three values: `0` means that the cell is empty, `1` means that the human player placed a stone there, and `2` means that the computer player placed a stone there.

<code>field[0,0]</code>	<code>field[1,0]</code>	<code>field[2,0]</code>
<code>field[0,1]</code>	<code>field[1,1]</code>	<code>field[2,1]</code>
<code>field[0,2]</code>	<code>field[1,2]</code>	<code>field[2,2]</code>

**Figure 13-2.** The playing field is represented by a two-dimensional array called `field`.

Let's start by creating a script to initialize the field. Call this script `scr_field_init`. This script must set all the field entries to `0`. We will use two local variables for this, and then use a double loop to fill in the entries, as shown in Listing 13-1.

**Listing 13-1.** *The Script scr\_field\_init*

```

{
  var i,j;
  // clear the field
  for (i=0; i<=2; i+=1)
    for (j=0; j<=2; j+=1)
      field[i,j] = 0;
}

```

---

**Note** Notice the line that starts with `//`. This line is a comment, so it is not really part of the program. Comments are ignored by Game Maker, and exist purely to help you, or someone else, know what is going on in the code—would you remember what all your variables and loops do when coming back to a piece of code after six months?

---

The game must store the number of wins by the two players, as well as the number of draws. For this we will use three variables: `score_player`, `score_computer`, and `score_draw`. To initialize the game, we must initialize these variables to 0 and we must initialize the playing field, as shown in Listing 13-2.

**Listing 13-2.** *The Script scr\_game\_init*

```

{
  // initialize the score
  score_player = 0;
  score_computer = 0;
  score_draw = 0;
  // initialize the field
  scr_field_init();
}

```

As you can see, we call the first script (`scr_field_init`) from within this script. Scripts can be used as functions that can be called from other scripts—we will use this technique a lot in our game. The `scr_game_init` script is executed from the **Create** event of the field object.

#### Creating and executing the scripts:

1. Create the two scripts, `scr_field_init` and `scr_game_init`, as described earlier.
2. Reopen the properties form of the field object by double-clicking on it in the resource list.
3. Add a **Create** event. In it include the **Execute Script** action and indicate the script `scr_game_init`.



The next step is to make it possible for the player to place stones. When the player clicks the left mouse button on the screen, we must detect which cell the click occurs in. If the click is outside the playing field or on a cell that is already filled, there will be no resulting action.

To determine the cell that has been clicked, we consider the current position of the mouse, which is indicated by the global variables `mouse_x` and `mouse_y`. The cells each have a size of 140×140, so to get the correct cell index (0, 1, or 2), we divide the mouse position by 140 and then round it down to the nearest whole number using the `floor()` function. This would give the correct cell index if our playing field were in the top-left corner of the screen. Because the top-left corner of the field is at position (208,32), we must subtract this offset from the mouse position, as we want the position relative to the top-left corner of the playing field, not the top-left corner of the screen.

So, for example, say the human player clicks at `x = 350`. The sum we do is  $(350-208)/140 = 1.01$ . Rounded down, the result is 1, which tells us that the cursor has been clicked inside one of the middle columns of cells (remember that the array starts at 0, not 1).

If the results of the calculations for `x` and `y` are less than 0 or more than 2, we ignore the click. If they *are* in this range, we check whether the corresponding cell is empty. If the cell is empty, we change its value to 1 to place the stone and play the sound effect.

The script looks like the one shown in Listing 13-3.

**Listing 13-3.** *The Script `scr_field_click`*

```
{
  var i,j;
  // find the position that is clicked
  i = floor((mouse_x-208)/140);
  j = floor((mouse_y-32)/140);
  // check whether it exists and is empty
  if (i<0 || i>2 || j<0 || j>2) exit;
  if (field[i,j] != 0) exit;
  // set the stone
  field[i,j] = 1;
  sound_play(snd_place);
}
```

Note that we use a new statement here: `exit`. The `exit` statement ends the execution of the script. We need to call this script in the **Global left pressed** event. This event is called when the left mouse button is pressed anywhere on the screen (not necessarily in the field object).

#### Creating the mouse click script:

1. Create the script `scr_field_click`, as shown earlier.
2. Reopen the properties form of the field object by double-clicking on it in the resource list.
3. Add a **Mouse, Global mouse, Global left pressed** event. In it include the **Execute Script** action and indicate the script `scr_field_click`.



If you run the game now, you will notice that you do hear the sound effect when you click on a cell but that no stones appear. This makes sense, as we have not yet added any code to draw the stones. Rather than using stone objects, we will create a script that draws the stones. Actually, this script will draw everything that is required: the stones and the current score (remember that the field does not need to be drawn as it is on the background image).

The script (shown in Listing 13-4) consists of two parts. First, all cells that are nonempty are drawn. We use a double loop for this. Depending on the value of the field cell at that position, the red or blue stone sprite is drawn. Second, the score is drawn. For this we set the correct font and position, and for each line we set a different color. (Note that the function `string()` turns a number into a string.)

**Listing 13-4.** *The Script `scr_field_draw`*

```
{
  var i,j;
  // draw the correct sprites
  for (i=0; i<=2; i+=1)
    for (j=0; j<=2; j+=1)
      {
        if (field[i,j] == 1)
          draw_sprite(spr_stone1,0,208+140*i,32+140*j);
        if (field[i,j] == 2)
          draw_sprite(spr_stone2,0,208+140*i,32+140*j);
      }
  // draw the score
  draw_set_font(fnt_score);
  draw_set_halign(fa_right);
  draw_set_color(c_blue);
  draw_text(200,340,'Player Wins: ' + string(score_player));
  draw_set_color(c_red);
  draw_text(200,375,'Computer Wins: ' + string(score_computer));
  draw_set_color(c_black);
  draw_text(200,410,'Draws: ' + string(score_draw));
}
```

We must call this script in the **Draw** event of the field object.

#### Drawing the field:

1. Create the script `scr_field_draw` as described earlier.
2. Add the **Draw** event to the field object. In it include the **Execute Script** action and indicate the script `scr_field_draw`.



Now when you test the game, you should be able to place stones. The computer opponent is not yet doing anything, so only your own stones exist. In the next section we will create some simple opponent behavior. The current version of the game can be found in the file `Games/Chapter13/tic_tac_toe1.gm6` on the CD.

## Let the Computer Play

In this section, we are mainly going to concentrate on completing the first version of the game by adding the logic for a simple computer opponent. But first we need some scripts to test whether the player or the computer won the last game, or if it was a draw. We start with a script to check whether the player did win. There are eight different lines of three stones that can be filled to win: three horizontal ones, three vertical ones, and two diagonal ones. In the script (shown in Listing 13-5), we simply test all of these to see whether the cells contain the correct value. The function will return either the value `true` indicating that the player did win, or `false`, indicating that the player did not yet win. The value returned by the script can then be used later as a condition in other scripts. By this point you should know how to create a script, so we will just show the code from here on out. If you are confused at any point, remember that the game is available on the CD in `Games/Chapter13/tic_tac_toe2.gm6`, so feel free to open it up and have a look.

**Listing 13-5.** *The Script `scr_check_player_win`*

```
{
    if (field[0,0]==1 && field[0,1]==1 && field[0,2]==1) return true;
    if (field[1,0]==1 && field[1,1]==1 && field[1,2]==1) return true;
    if (field[2,0]==1 && field[2,1]==1 && field[2,2]==1) return true;
    if (field[0,0]==1 && field[1,0]==1 && field[2,0]==1) return true;
    if (field[0,1]==1 && field[1,1]==1 && field[2,1]==1) return true;
    if (field[0,2]==1 && field[1,2]==1 && field[2,2]==1) return true;
    if (field[0,0]==1 && field[1,1]==1 && field[2,2]==1) return true;
    if (field[0,2]==1 && field[1,1]==1 && field[2,0]==1) return true;
    return false;
}
```

---

**Note** To check whether two values are equal, you must use `==`, not `=`, as a single `=` is the assignment operator. Also remember that once a `return` statement is reached, the rest of the script is not executed.

---

Checking whether the computer wins is exactly the same, except that this time, we are testing whether the cells contain values of 2, not 1, as shown in Listing 13-6.

**Listing 13-6.** *The Script `scr_check_computer_win`*

```
{
    if (field[0,0]==2 && field[0,1]==2 && field[0,2]==2) return true;
    if (field[1,0]==2 && field[1,1]==2 && field[1,2]==2) return true;
    if (field[2,0]==2 && field[2,1]==2 && field[2,2]==2) return true;
    if (field[0,0]==2 && field[1,0]==2 && field[2,0]==2) return true;
    if (field[0,1]==2 && field[1,1]==2 && field[2,1]==2) return true;
    if (field[0,2]==2 && field[1,2]==2 && field[2,2]==2) return true;
}
```

```

    if (field[0,0]==2 && field[1,1]==2 && field[2,2]==2) return true;
    if (field[0,2]==2 && field[1,1]==2 && field[2,0]==2) return true;
    return false;
}

```

Checking for a draw is even simpler (see Listing 13-7)—we check all cells; if one is empty we return `false` as there is still a move possible. Only when all cells are filled do we return `true`.

**Listing 13-7.** *The Script `scr_check_draw`*

```

{
    var i,j;
    for (i=0; i<=2; i+=1)
        for (j=0; j<=2; j+=1)
            {
                if (field[i,j] == 0) return false;
            }
    return true;
}

```

To act on the outcome of these three possibilities, we will use another script, as shown in Listing 13-8. For each possible outcome, the correct score variable is increased; a sound is played; we redraw the screen to actually show the last move and the new score; wait for a second; show a message; and initialize the field again.

**Listing 13-8.** *The Script `scr_check_end`*

```

{
    // check whether the player did win
    if (scr_check_player_win())
    {
        score_player += 1;
        sound_play(snd_win);
        screen_redraw();
        sleep(1000);
        show_message('YOU WIN');
        scr_field_init();
    }
    // check whether the computer did win
    if (scr_check_computer_win())
    {
        score_computer += 1;
        sound_play(snd_lose);
        screen_redraw();
        sleep(1000);
        show_message('YOU LOSE');
        scr_field_init();
    }
}

```

```

// check whether there is a draw
if (scr_check_draw())
{
    score_draw += 1;
    sound_play(snd_draw);
    screen_redraw();
    sleep(1000);
    show_message("IT'S A DRAW");
    scr_field_init();
}
}

```

We must call this script after each move by either the player or the computer.

But we still need to give the computer the power to make a move. Let's create a very simple mechanism here; in the next section we'll create a much more intelligent opponent. Our simple mechanism makes a random move. We do this as follows. We select a random cell, test whether it is empty, and if so, place the stone there. If the selected cell is not empty, we repeat the search until we find one that is. Finding a random position works like this—we use the function `random(3)` to obtain a random real number below 3. Using the `floor()` function, we round this down, obtaining 0, 1, or 2. The script is shown in Listing 13-9.

**Listing 13-9.** *The Script `scr_find_move`*

```

{
    var i,j;
    while (true)
    {
        i = floor(random(3));
        j = floor(random(3));
        if (field[i,j] == 0)
        {
            field[i,j] = 2;
            exit;
        }
    }
}

```

This script makes use of a `while` loop to find a random free cell. `while(true)` can be dangerous, because as the expression is always true, the loop never exits by itself. In this case, however, we are exiting in our own code as soon as an empty cell is detected, so it is safe as long as an empty cell exists. If there are no empty cells, we already know it is a draw, so we need not worry about that case.

We are going to use this script in an updated version of the script `scr_field_click`, which we created earlier. When the player has made a valid move, there are three things we must do. First, we check whether the player won or whether there is a draw, in which case the field is initialized again, ready for the next game. Next, we let the computer make a move. Finally, we check whether the computer won or whether there is a draw.



We adapt the `scr_field_click` script by adding a few lines, as shown in Listing 13-10.

**Listing 13-10.** *The Adapted Script `scr_field_click`*

```
{
    var i,j;
    // find the position that is clicked
    i = floor((mouse_x-208)/140);
    j = floor((mouse_y-32)/140);
    // check whether it exists and is empty
    if (i<0 || i>2 || j<0 || j>2) exit;
    if (field[i,j] != 0) exit;
    // set the stone
    field[i,j] = 1;
    sound_play(snd_place);
    scr_check_end();
    // let the computer make a move
    scr_find_move();
    scr_check_end();
}
```

Once you have added the new scripts and made the change to `scr_field_click`, test the game, and you should find it is now fully operational. You can find it in the file `Games/Chapter13/tic_tac_toe2.gm6` on the CD.

The game as it stands is fine, but it is extremely easy to win as the computer plays random moves. In the next section, we will make the computer a bit more intelligent.

## A Clever Computer Opponent

To be able to make a clever computer opponent we must first be clever ourselves. How would you play the game? What would your strategy be? If you have played the game often, here is a strategy you might come up with:

- If there is a move available that will make you win, then play it.
- If there is no winning move available for you, but there is one available for the opponent, you'd better play that move to block them; otherwise you will lose.
- If neither is the case but the center cell is free, then play the center.
- If none of the above is true, play a random move.

This is not the best strategy possible but it is pretty good, and still leaves the player with a chance to win, so let's implement it. To give the game a bit more variation, we will program the computer opponent to only do the third step half the times it is presented. We are going to create four scripts, one for each of the four cases. The last one we already have—all we have to do is rename it from `scr_find_move` to `scr_find_random`. The other scripts still have to be constructed.

We will start with the script that tests for the existence of a winning move for the computer. If such a move exists, it is made, and `true` is returned. Otherwise `false` is returned. The script works as follows—we consider every empty cell. We place a stone there and test whether we won. If so, we return `true`. If not, we make the cell empty again and proceed with the next empty cell. The script is shown in Listing 13-11.

**Listing 13-11.** *The Script `scr_find_win`, Which Tries to Find a Winning Move*

```
{
  var i,j;
  for (i=0; i<=2; i+=1)
    for (j=0; j<=2; j+=1)
      if (field[i,j] == 0)
        {
          field[i,j] = 2;
          if scr_check_computer_win() return true;
          field[i,j] = 0;
        }
  return false;
}
```

The next script tries to find a potential winning move for the human player. If such a position exists, the computer places a stone there. It largely works the same, except that we are testing for a potential row of three human player stones, not three computer player stones. The cell is then given a value of `2` to place a computer stone there, to block the human player's winning move, as shown in Listing 13-12.

**Listing 13-12.** *The Script `scr_find_lose`, Which Tries to Block a Winning Move of the Player*

```
{
  var i,j;
  for (i=0; i<=2; i+=1)
    for (j=0; j<=2; j+=1)
      if (field[i,j] == 0)
        {
          field[i,j] = 1;
          if scr_check_player_win()
            { field[i,j] = 2; return true; }
          field[i,j] = 0;
        }
  return false;
}
```

Finally, we need the script that tries the center position (Listing 13-13.) It will only try it once out of every two times.

**Listing 13-13.** *The Script `scr_find_center`, Which Tries to Place a Stone in the Center*

```

{
    if (random(2) < 1 && field[1,1] == 0)
        { field[1,1] = 2; return true; }
    return false;
}

```

With all these scripts in place, we have to remake the script `scr_find_move` that determines the next move of the computer. This script calls the four scripts in order and, whenever one succeeds, it stops further processing because the opponent has made a move. This script appears in Listing 13-14.

**Listing 13-14.** *The New Script `scr_find_move`*

```

{
    if scr_find_win() exit;
    if scr_find_lose() exit;
    if scr_find_center() exit;
    scr_find_random();
}

```

That's the whole game. You can find this finished version in the file `Games/Chapter13/tic_tac_toe3.gm6` on the CD. It will be quite a bit harder to beat this opponent, and if you are not very good at the game you will most likely lose a few times! In particular, the game might be too difficult for young children. In the next section, we will see how we can automatically adapt the game to the level of the player.

## Adaptive Gameplay

When the player is good, we should confront him with a strong computer opponent. But when the player is a novice, the computer opponent should be weaker. Many games achieve this by letting the user manually select the level of the game (for example, easy, normal, or hard). But it can be more desirable when the game automatically adapts to the level of the player. For our game this can easily be achieved—we maintain the score of the player and the computer, so we know who is winning. When the player is winning a lot, we can make the computer play better, and when the player is losing a lot, we can make the computer play down its abilities.

As an indication of how good the player is, we use  $(\text{score\_player}+1) / (\text{score\_computer}+1)$ . The reason for adding 1 to both values is that we do not want to divide by 0, as this would cause an error. When this value is larger than 1, the player is better than the computer. When it is smaller than 1, the computer is better. In the `scr_find_move` script where we decide on the computer move, we will also compute this value, and based on the result, we decide which moves we check. When the value is larger than 1.2, we try all moves. When it is smaller than 0.5, we only do a random move. We add in the other two moves as the value increases. The updated version of `scr_find_move` is shown in Listing 13-15.

**Listing 13-15.** *Further Adapting the Script `scr_find_move`*

```
{
    var level;
    level = (score_player+1) / (score_computer+1);
    if (level > 0.5)
        { if scr_find_win() exit; }
    if (level > 0.8)
        { if scr_find_lose() exit; }
    if (level > 1.2)
        { if scr_find_center() exit; }
    scr_find_random();
}
```

You should now let a number of people play the game and see whether the game indeed adapts to their level of play. You might want to vary the numbers in the script to make the game easier or harder for the players.

## Congratulations

You have just created your first intelligent computer opponent. Also, you have created some adaptive gameplay, something that is very important for good games. You will find the last version of the game in the file `Games/Chapter13/tic_tac_toe4.gm6` on the CD.

You might want to extend the game even further. First, you can think a bit more about the strategy. A very good move in the game is a move where you create two winning positions for yourself in the next move. As the opponent can only play one of the two, you are then assured of winning the game in the next move. You might want to add this to the strategy. Another thing you could do is add a two-player mode in which two people can play against each other. This is relatively easy, as there is no need for an intelligent computer opponent anymore—here you only need to remember and indicate who is to play in each move.

The registered version of Game Maker makes available functions that make it possible to play such a game over a network with two computers, but that is something really advanced.

In this chapter you saw how useful GML code is—we put everything in scripts. In fact, a game like this would have been almost impossible to create without GML. And it wasn't really all that much work. Once you get accustomed to using GML code, you will probably start using actions much less.

Intelligent opponents make games more interesting. In this chapter we had just one intelligent opponent. In the next chapter, we will create a whole collection of intelligent enemies.



## CHAPTER 14



# Intelligent Behavior: Animating the Dead

**A**n interesting and challenging game must involve tricky opponents that require clever thinking or quick reactions to defeat. Up to now, the enemies in our games behaved rather stupidly—they were just following predefined rules or attacking randomly. In this chapter we will make the enemies a bit more intelligent, but care must be taken that we do not make them too clever—we still want the player to be able to win the game.

To facilitate this, we'll introduce some artificial intelligence (AI) techniques in this chapter—this term will probably conjure up images of powerful intelligent robot or cyborg opponents, the likes of which are seen in futuristic movies. This may sound like a complicated topic, but don't worry—the techniques we'll examine are actually rather easy to implement in Game Maker.

## Designing the Game: Pyramid Panic

The game we're going to create in this chapter is called *Pyramid Panic*. It is a maze game, like the *Koalabr8* game we created in Chapter 7, but the gameplay will be rather different. Here is the story:

*You play an explorer (see Figure 14-1) who has become trapped while investigating a large pyramid complex. All around lie the treasures of an ancient pharaoh, but pyramids are hazardous places and danger lurks around every corner. Deadly scorpions and beetles will block your progress and mummies will hunt you down. Only by keeping your wits about you can you hope to unravel the secrets of the great pyramid and escape as a rich man.*

*You control the explorer using the arrow keys. Many obstacles will block your path, keeping you from taking the treasures and eventually escaping to freedom. Beetles will only move vertically while scorpions only move horizontally. Mummies move in all directions. These enemies are clever and will react when they see you by trying to catch you and end your explorations. Some wall segments can be pushed, allowing you to reach other areas or hide from enemies. The pyramid also contains scarabs that you can use to make the mummies temporarily vulnerable—allowing you to hunt them for extra points.*

*Deep within the center of the pyramid lies its greatest treasure, the fabled sword of the sun god Ra. It is this great treasure that casts the unnatural light which reaches throughout the pyramid and allows you to see your way so clearly. It is precious beyond measure, but in taking it you will upset that delicate system and the pyramid will be plunged into eerie darkness. Only the small glow remaining in the sword will light your way now, and formerly simple puzzles will seem new and challenging. All is not lost, however, for the sword has a second function. When wielding the sword you will be able to press and hold the spacebar to temporarily reactivate its glow. The sword transmutes gold into pure light, lighting your way but reducing your score. When the sword is active, the mummies will flee as they do when a scarab is active, making your journey easier, but draining your wealth.*

The basic ingredients of the game are similar to the maze game in Chapter 7, but there is a big difference. We are including enemies that are clever and will react when they see you, which makes the game a lot more interesting to play. All the resources can be found in the [Resources/Chapter14](#) folder on the CD.

















**Figure 14-1.** Our intrepid explorer is chasing mummies.

## The Basic Framework

Let's start by creating the basic game framework. This will work in largely the same way as described earlier in Chapters 6 and 7, so we'll only briefly outline it here. If you want to immediately go to the next section, you can load the file [Games/Chapter14/pyramid1.gm6](#) from the CD.




**Creating the front-end:**

1. Open Game Maker and start a new game using the **File** menu.
2. Create sprites using the following files from the `Resources/Chapter14` folder on the CD: `Title.gif`, `Button_start.gif`, `Button_load.gif`, `Button_help.gif`, `Button_scores.gif`, and `Button_quit.gif`.
3. Create two backgrounds using the files `Background1.gif` and `Background2.gif`.
4. Create sounds using the files `Music.mp3` and `Click.wav`.
-  5. Create a title object using the title sprite. Add an **Other, Game Start** event and include an action to play the music (with **Loop** set to **true**).
-   
  
 6. Add a **Create** event. Include a **Set Score** action and set the score to 0. Also include a **Set Lives** action and set the number of lives to 3. Finally, include the **Score Caption** action and indicate that the score must not be shown.
-   
 7. Create a start button object using the start sprite. Add a **Left Pressed** mouse event and include an action to play the click sound followed by an action to move to the next room. Add **Key Press** events for the `<Space>` key and the `<Enter>` key, with the same actions.
-   
 8. Create a load button object using the load sprite. Add a **Left Pressed** mouse event and include an action to play the click sound followed by a **Load Game** action.
-   
 9. Create a help button object using the help sprite. Add a **Left Pressed** mouse event and include an action to play the click sound followed by a **Show Info** action.
-   
 10. Create a scores button object using the scores sprite. Add a **Left Pressed** mouse event and include an action to play the click sound followed by a **Show Highscore** action. Use the second background as an image for the high-score list and choose some nice font and colors.
-   
 11. Create a quit button object using the quit sprite. Add a **Left Pressed** mouse event and include an action to play the click sound followed by an **End Game** action. Also add a **Key Press, Others, <Escape>** event, with the same actions in it.
12. Create a front-end room using the first background, and place the title and five button objects in it.

Now follow the next instructions to create the completion screen. Refer to Chapter 6 for a more detailed explanation.



**Creating the completion screen:**

1. Create a sprite using the file `Congratulation.gif`.
2.  Create a new object using this sprite. Add a **Create** event and include a **Set Alarm** action to set **Alarm0** using `60` steps. Also include a **Set Score** action and set the score to `score * 2`. This gives the player an incentive to escape the pyramid alive.
3.  Add an **Alarm0** event and include the **Show Highscore** action. Use the same settings as before. Next, include a **Different Room** action to move to the front-end room.
4.  Create a completion room using the second background and place an instance of the congratulation object in it.




We also need to create the game information and change some of the Game Maker's default settings for the game.

**Changing the game settings:**

1. Double-click on **Game Information**, near the bottom of the resource list, and create a short help text based on the game's description earlier.
2. Double-click on **Global Game Settings** at the bottom of the resource list.
3. Select the **other** tab and disable the option **Let <Esc> end the game**, as we'll handle this ourselves.

The next step is to create a controller object for the game. This controller object will have a number of functions. First, it provides the option to return to the front-end screen. Second, it handles the ending of the game when all lives are lost. Finally, later in the chapter, it will display a panel at the bottom of the room with the number of lives left and the score.

**Creating the controller object:**

1.  Create a new object and name it `obj_controller`. It does not need a sprite. Add a **Key Press, Others, <Escape>** event, and include the **Different Room** action to move to the front-end room.
2.  Add the **Other, No More Lives** event. Within it, include a **Sleep** action, setting the number of milliseconds to `1000`. Next include a **Display Message** action with the message `"YOU LOST ALL YOUR LIVES!"`, or something similar.
3.  Still in the **No More Lives** event, include the **Show Highscore** action with the same settings as you used in the completion screen. Finally, include the **Different Room** action to move to the front-end room.

To test what we have so far, we will add a room between the front-end room and the completion room, to act as the game level.

**Creating a basic room:**

1. Right-click on the completion room in the resource list. From the pop-up menu, select **Insert Room**.
2. Select the **backgrounds** tab and in the middle select the second background.
3. Select the **settings** tab to give the room an appropriate caption and call the room `room_pyramid`.
4. Select the **objects** tab and place one instance of the controller object at the top left of the room.

That completes our basic game framework for Pyramid Panic, which can also be found in the file `Games/Chapter14/pyramid1.gm6` on the CD. Give it a little test to see that it works correctly. Note that the only way to stop the game is to press the Esc key—this returns you to the front-end room.

## Creating the Maze and the Explorer

Our next goal is to create the basic maze. Again, this is similar to what we did in Chapter 7, but this time we will use **Execute Code** actions in many places instead of actions. Let's start by creating two wall objects. We'll use two different ones to give some variation, but we'll ensure that they behave the same in the game by making the first wall the parent of the second wall.

**Creating the wall objects:**

1. Create two sprites using the files `Wall1.gif` and `Wall2.gif`. Make sure that both sprites are not transparent.
2. Create a new object for the first wall and name it `obj_wall1`. Give it the first wall sprite and enable the **Solid** option.
3. Create a new object for the second wall and name it `obj_wall2`. Give it the second wall sprite, enable the **Solid** option, and set **Parent** to the first wall object.

Next we need to create the hero of our game—the explorer. We use four different animated sprites for this. To make sure that the explorer stays correctly aligned with the cells, we'll disable precise collision checking and actually increase the bounding box for the sprites. As a result, Game Maker will act as if the sprites have a size of 32×32.

**Creating the explorer sprites:**

1. Create a sprite using the file `Explorer_left.gif`. This is an animated sprite. Give it the name `spr_explorer_left`.
2. Disable the option **Precise collision checking**, and select the **Full image** option found under **Bounding Box**. Since the sprite is 32×32, this will make the collision act on a 32×32 box.
3. In the same way, create the three other sprites using the files `Explorer_right.gif`, `Explorer_up.gif`, and `Explorer_down.gif`.

The explorer object must pick the correct sprite depending on its direction of motion. This time, rather than using actions we'll do all the motion using code. When the explorer reaches the area outside the room, he has escaped the pyramid and we can move to the completion screen.

#### Creating the explorer object:

1. Create a new object for the explorer and name it `obj_explorer`. Give it the downward moving explorer sprite.



2. Add a **Create** event, and in it, include the **Execute Code** action (**control** tab). Type in the following piece of code:

```
{
    image_speed = 0;
}
```

The `image_speed` variable we use here stores the speed with which the animation is played. When the explorer is standing still, this is set to 0.



3. Add a **Collision** event with the first wall object and include the **Execute Code** action within it. Here we must stop the motion and set the animation speed to 0. For this, we use the following piece of code:

```
{
    speed = 0;
    image_speed = 0;
}
```



4. Add a **Keyboard, <Left>** event and again include an **Execute Code** action using the following piece of code:

```
{
    if ( !place_snapped(32,32) ) exit;
    speed = 4;
    direction = 180;
    sprite_index = spr_explorer_left;
    image_speed = 0.25;
}
```

In the first line of this code we call the function `place_snapped()`, which tests whether the instance is aligned with the grid. If this is not the case (remember that the `!` sign means not), we exit the piece of code; otherwise we set the speed and direction. The variable `sprite_index` indicates the sprite that is used, and we set it to the left-facing explorer sprite. Finally, we set the animation speed to `0.25`. We use a small value to make sure that the animation looks more realistic and that it doesn't go too fast.

5. Add similar events for the **<Right>**, **<Up>**, and **<Down>** keys. The directions for these events should be 0, 90, and 270, and you should be sure to choose the correct sprite for each.



6. Add a **Keyboard, <no key>** event. If aligned with the grid we must stop the motion here. So include an **Execute Code** action with the following piece of code:

```
{
    if ( !place_snapped(32,32) ) exit;
    speed = 0;
    image_speed = 0;
}
```



7. Add the **Other, Outside Room** event and include the **Next Room** action.

For the time being, this concludes the description of the explorer object. At first, using code may seem harder than using actions, but when you get used to the way code is structured, you will find it faster, because you can more easily change it. It is now time to test the explorer's movement and his interaction with the walls.

#### Adapting the room:

1. Open `room_pyramid` by double-clicking on it in the resource list.
2. In the toolbar, set **Snap X** and **Snap Y** to 32, as our maze cells are 32×32.
3. Using the first wall object, create a maze in the room. (Be careful not to remove the controller object.) Make sure there is one exit to the outside. In some places, place an instance of the second wall object instead, to give the walls some variation.
4. Add one instance of the explorer in the room.

Now test the room to make sure the explorer's motion around the maze is working correctly. Once you reach the outside, you should be moved to the completion room. This version of the game can also be found in the file `Games/Chapter14/pyramid2.gm6` on the CD.

## Expanding Our Horizons

One of the main features of our game is exploration. In most of our previous games we have extended the game by adding extra levels, but for this game we'll do something a little different. Expanding on our use of views in Chapter 10, we'll create the whole pyramid as one huge level. This will provide our intrepid explorer with a worthy challenge, and give you plenty of space in which to devise cunning tricks and traps.







For this game we need a single view that shows the explorer and the pyramid. The view will scroll around the pyramid following the explorer. If any of this doesn't make sense, be sure to go back to Chapter 10 where there is a much more detailed explanation.

**Creating the view:**

1. Open `room_pyramid` and select the **settings** tab. Change the **Width** to `1920` and the **Height** to `2400`. This will be plenty big enough for our pyramid, but you can feel free to make it bigger later on.
2. Select the **views** tab. First enable the option **Enable the use of views**.
3. Select **View 0** in the list. Enable the option **Visible when room starts**.
4. Under **View in room** keep the default values: **X: 0, Y: 0, W: 640, H: 480**. Note that the view is shown in the room.
5. Under **Port on screen** also keep the default values: **X: 0, Y: 0, W: 640, H: 480**.
6. Under **Object Following** click the menu icon and select `obj_explorer`. Set **Hbor** to `300` and **Vbor** to `220`. This will keep the explorer nicely centered in the view.
7. Now you can extend your maze to fill the whole room. There's not much to do yet, but don't worry—we'll fix that soon.

It's probably best to try out your game now to check that the view works. As you move around the maze, the view should scroll so that the explorer is always in view. Our next step is to adapt the controller object to draw a status panel with information about the number of lives and the score.

**Extending the controller object:**

1. Create two sprites using the files `Panel.gif` and `Lives.gif`. Make sure that the panel sprite is not transparent.
2. Create a font that will be used for the score. We used a size 14 bold Arial font.
3. Reopen the properties form for the controller object. Set the **Depth** to `-100` to make sure it lies in front of all other objects.
4.  Add a **Draw** event. We need to make sure the panel is drawn in the correct position in the view. To this end, we first move the controller object to the correct position. As explained in Chapter 10, we can use the variables `view_xview[0]` and `view_yview[0]` for this. Include a **Jump to Position** action and set **X** to `view_xview[0]` and **Y** to `view_yview[0]+448`.
5.   Include the **Draw Sprite** action with **Sprite** set to the panel sprite, **X** set to `0`, **Y** set to `0`, and the **Relative** option enabled. Also include the **Draw Life Images** action, with **Image** set to the lives sprite, **X** set to `80`, **Y** set to `2`, and the **Relative** option enabled.
6.    Include a **Set Font** action and select the score font. Set **Align** to right. Also include a **Set Color** action and choose a nice yellow for the color. Finally, include the **Draw Score** action with **X** set to `585`, **Y** set to `5`, the **Caption** set to the empty string, and the **Relative** option enabled.

That's it for the view. Start up the game and check that the panel is drawn correctly. This version of the game can also be found in the file `Games/Chapter14/pyramid3.gm6` on the CD.

## Reactive Behavior

It is now time to start dealing with the topic of this chapter: behavior. We want our pyramid to be inhabited with creatures that our explorer needs to avoid. We'll create a number of different enemies; the first ones will be rather stupid, and will be controlled by some simple reactive behavior. Later, enemies will use rules that guide their behavior, and the cleverest enemies will have different states, depending on the current situation.

*Reactive behavior* is the simplest kind of behavior you can think of. The entity reacts to different events. Actually, this is what we have been doing throughout the book—we have created objects and indicated their reactions to events. You could say that Game Maker is completely based on the idea of reactive behavior. However, this is the first time we have studied the concept in depth, and it does warrant a good deal of discussion when trying to program better games.

Here we will create a beetle enemy that simply moves up and down. It will react to two different events: when it hits a block it will change its direction of motion, and when the explorer is in front of it, it will move toward the player and double its speed of motion. This gives the impression that the beetle notices the explorer and runs to try to catch him, making the game more interesting to play. We'll also create a scorpion that operates in the same fashion, but instead moves left and right.

Before we do this let's create a dummy object called `obj_enemy`. This object will be the parent of all the enemy objects; this makes it possible to define one collision event for the explorer that deals with all different enemies. We'll also use it to destroy the enemy—our explorer doesn't always go down without a fight! It is a good habit to divide your objects into subgroups and make them share a parent object for common behavior.

### Creating the basic enemy object:

1. Create a sound from the file `Die.wav` and call it `snd_die`.
2. Create an object and call it `obj_enemy`. It does not need a sprite.
3. Reopen the properties form for the explorer object, and add a **Collision** event with `obj_enemy`. Include the **Execute Code** action. Here we'll play the dying sound, reduce the number of lives, redraw the screen, sleep a while, and then set the explorer back to his start position and destroy the enemy:






```
{
    sound_play(snd_die);
    lives -= 1;
    screen_redraw();
    sleep(1000);
    x = xstart;
    y = ystart;
    move_snap(32,32);
    with(other)
    {
        instance_destroy();
    }
}
```

Note the use of the **with** statement. As explained in Chapter 11, with this statement we can execute code as if it were being run on another object. Later in the chapter we will also see how **with** can be used to act on whole groups of objects in the same way.

With the basic enemy defined we'll now create the beetle object.

#### Creating the beetle object:

1. Create two sprites using the files `Beetle_up.gif` and `Beetle_down.gif`, and give them the names `spr_beetle_up` and `spr_beetle_down`, respectively. In both cases, disable the option **Precise collision checking**, and select the **Full image** option found under **Bounding Box**.
2. Create an object and call it `obj_beetle`. Give it the `spr_beetle_down` sprite, and indicate `obj_enemy` as its **Parent**.
-  3. Add a **Create** event and include the **Move Fixed** action. Indicate a downward direction and a **Speed** of 2.
-  4. Add a **Collision** event with `obj_wall1` and include a **Reverse Vertical** action.
-  5. Add the **Step, End Step** event. Include an **Execute Code** action with the following code, which adapts the sprite to the direction of motion:

```
{
    if ( vspeed > 0 )
        sprite_index = spr_beetle_down
    else
        sprite_index = spr_beetle_up;
}
```

Now if you add some beetles to a room, you will notice that they nicely move up and down, and that the explorer dies when he touches one of them. But this is not very intelligent beetle behavior. We'll now add a piece of code that makes the beetle react to the presence of the explorer. If the explorer is on the same row of the maze as the beetle and in front of it, the beetle will speed up and run toward him.

#### Making the beetle object more intelligent:

-  1. Add the **Step, Step** event, then include an **Execute Code** action containing the following code:

```
{
    speed = 2;
    if ( x == obj_explorer.x )
    {
        if ( (direction == 90) && (y > obj_explorer.y) )
        {
            speed = 4;
        }
    }
}
```

```

    }
    else if ( (direction == 270) && (y < obj_explorer.y) )
    {
        speed = 4;
    }
}
}

```

You should now create a scorpion object that acts in exactly the same way as the beetles, only it moves horizontally, not vertically.

#### Creating the scorpion object:

1. Create two sprites using the files `Scorpion_left.gif` and `Scorpion_right.gif`, and give them the names `spr_scorpion_left` and `spr_scorpion_right`, respectively. In both cases, disable the option **Precise collision checking**, and select the **Full image** option found under **Bounding Box**.
2. Create an object and call it `obj_scorpion`. Give it the `spr_scorpion_right` sprite, and indicate `obj_enemy` as its **Parent**.
3. Define the behavior of the scorpion object using the same events and similar actions and code as for the beetle object.

You can stop and try your game out now if you like, but I think it's worth waiting until we've put some treasure in before we do that—after all, who wants to dodge all of those deadly monsters without some sort of reward!


## Time for Treasure!

Now it's time to add some treasure to the room for the explorer to discover. There will be two different treasure types: the first one will occur many times in the maze, and each piece of this treasure will give the player 10 points. The second treasure type is rarer but much more valuable when collected. You can place it at difficult locations in the room to give the player a real challenge to aim toward.


#### Creating the treasure objects:

1. Create two sprites using the files `Treasure1.gif` and `Treasure2.gif`. For each of them, disable the **Precise collision checking** option and select the **Full image** option found under **Bounding Box**.
2. Create a sound using the `Treasure.wav` file and call it `snd_treasure`.
3. Create an object for the first treasure and assign the first sprite to it. Give it a **Depth** of `10` to make sure that enemies appear over it.



-  4. Add a **Collision** event with the explorer object, and include an **Execute Code** action within it with the following code that increases the score, destroys the treasure, and plays the treasure sound:

```
{
    score += 10;
    instance_destroy();
    sound_play(snd_treasure);
}
```


-  5. Create an object for the second treasure, assign the second sprite to it, and give it a **Depth** of 10. Add a **Collision** event with the explorer, and include an **Execute Code** action containing the preceding code, except that this time the score increment must be 100 rather than 10.

You should now enhance your level to include some treasure for the explorer to discover and some enemies to guard it. Test your game and see how the beetles and scorpions add to the challenge.

## Movable Blocks

To create more interesting challenges, we'll introduce two special blocks that can be pushed by the player. One type of block can only be pushed horizontally, while the other type can only be pushed vertically. Only the explorer will be able to push the blocks. These will be magical blocks, which will return to their original position when the explorer stops pushing them. The original position of each block can easily be determined, as it is always available in the variables `xstart` and `ystart`. Then by checking the current position compared to its start position, we can determine which direction the block must move to get back to its original position. We will do this in the **End Step** event to make sure it happens after all other events have been processed.

### Creating the block objects:

1. Create a sprite using the file `Block_hor.gif`, and disable the **Transparent** option.
  2. Create a sound from the file `Block.wav` and call it `snd_block`.
  3. Create an object for the horizontal block and give it the above sprite. Enable the **Solid** option, and indicate the first wall object as the **Parent**, as the block should behave a lot like a wall.
-  4. Add the **Step, End Step** event. Include an **Execute Code** action and type in the following piece of code, which moves the block instance back toward its start position and plays the sound:

```
{
    if ( ( x < xstart ) && place_empty(x+2,y) )
    {
        x += 2 ;
        sound_play(snd_block);
    }
}
```

```

    }
    if ( (x > xstart) && place_empty(x-2,y) )
    {
        x -= 2;
        sound_play(snd_block);
    }
}

```



5. Reopen the properties form for the explorer object and add a **Collision** event with the horizontal block. Include the **Execute Code** action, and indicate the **Other** object at the top under **Applies To**, as we want to move the block. Now type the following code—in it we first check whether the explorer has moved horizontally. If he has, we check whether we can move the block the same amount.

```

{
    if ( obj_explorer.hspeed == 0 ) exit;
    if ( place_empty(x+obj_explorer.hspeed,y) )
    {
        x += obj_explorer.hspeed;
        sound_play(snd_block);
    }
}

```

6. Repeat the previous steps to create the vertical moving block. Create the sprite and the object and include similar pieces of code, this time moving vertically.

---

**Note** You might have wondered why we defined the collision event in the explorer object and not in the moving block object. The reason is that we want to replace the default collision behavior with the first wall. Otherwise, the explorer would stop moving.

---

Now let's test our blocks—add some horizontal and vertical moving blocks to your room. Make sure there is space next to the blocks so that they can be moved. With some careful level design, you can now create one-way doors to add to the maze's challenge, as well as other clever puzzles. The current version of the game can be found in the file [Games/Chapter14/pyramid4.gm6](#) on the CD. Be sure to come back soon, though—we're just about to introduce the big bad guys and you won't want to miss it!

## Rule-Based Behavior

To make our ultimate AI opponent, we are going to want some pretty complicated behavior. A convenient way of defining behavior is to use rules. Thinking in terms of rules helps you to clearly describe how entities must behave. A rule indicates that if certain conditions are met, certain actions must be taken. So a rule has the form

conditions ► actions

Although you may not have realized it at the time, we have already used a rule in the previous section. We indicated that if the x-coordinates of the beetle and the explorer are equal, the beetle must move fast toward the explorer:

x-coordinates equal and facing explorer ➤ move fast toward explorer

There even was a second, default rule that said that if the x-coordinates of the beetle and the explorer were not equal, or the beetle was facing the other way, then the beetle should move with a slower speed. Also, the collision with a wall can be formulated as a rule. So in total there were three rules:

x-coordinates equal and facing explorer ➤ move fast toward explorer

x-coordinates not equal or facing away ➤ move slowly

collision with wall ➤ reverse vertical direction away from the wall

The first two rules are exclusive, that is, they can never both be true. But the third can be true at the same time as one of the other two. In such a situation, it is important to decide what should happen. Should both actions be executed, or just one, and if so, which one? Does one rule have priority over another, or should a random one be picked? In our previous example the third rule should definitely have priority over the other rules, as the beetle should never be allowed to move into a wall.



Creating rule-based behavior consists of defining a set of rules and indicating what should happen when multiple rules are valid. A rule consists of one or more conditions and the actions that must be followed in the case the conditions are true. Typical conditions that are often used in games are as follows:

- Is a position or direction collision-free?
- Is the enemy near to the player?
- Is the player in a particular direction relative to the enemy?
- Can the enemy see the player?
- Does the enemy or the player carry a particular item?

All these conditions can easily be tested in Game Maker as we'll see in a moment; however, first we need to create the mummy object that can use these conditions to drive its behavior.

#### Creating the mummy object:

1. Create four sprites using the files `Mummy_left.gif`, `Mummy_right.gif`, `Mummy_up.gif`, and `Mummy_down.gif`. Name them `spr_mummy_left`, `spr_mummy_right`, `spr_mummy_up`, and `spr_mummy_down`.
2. For each of the sprites, disable the **Precise collision checking** option, and select the **Full image** option under **Bounding Box**.

3. Create a new object for the mummy and name it `obj_mummy`. Give it the downward-moving mummy sprite and indicate `obj_enemy` as its **Parent**.
-  4. Add a **Create** event and in it include the **Move Fixed** action. Give it a downward direction and a **Speed** of 4.
-  5. Add the **Step, End Step** event. Here we'll set the correct sprite and animation speed. Include an **Execute Code** action and type in the following piece of code:

```
{
    image_speed = 1/3;
    if (hspeed < 0) sprite_index = spr_mummy_left;
    if (hspeed > 0) sprite_index = spr_mummy_right;
    if (vspeed < 0) sprite_index = spr_mummy_up;
    if (vspeed > 0) sprite_index = spr_mummy_down;
}
```

As it stands, the mummy will simply walk through the walls, but don't despair—we'll give it better behavior next.

## Walking Around

The first behavior we want to define for the mummy is simply walking around through the level. This is more difficult than it might seem. At each cell there are four possible directions the mummy can choose. Some might be blocked by walls, and are therefore forbidden. Once all of the forbidden choices are removed, we must pick one of the available ones. To ensure natural motion, we'll make sure that we never go back in the direction we came from, unless this is the only possible direction to take (when we go into a dead end). For the other directions, we'll pick a valid one randomly.

How do we turn this into rules? Let `ahead`, `left`, and `right` indicate whether these three positions are free. We have the following rules:

`not ahead` and `not left` and `not right` ► move back

`ahead` ► move straight ahead

`left` ► turn left

`right` ► turn right

The first rule is exclusive from the others, so if the first rule is valid, all others are not valid. The other three rules can all be valid at the same moment, and as we mentioned earlier, we should pick a random valid one.

The next step is to turn this into a GML script (see Listing 14-1). To determine the value of `ahead`, `left`, and `right`, we'll use the function `place_free`, and to make a random choice we'll use a similar method to Chapter 13. To make the script easier to understand, we have also split the code based on whether the mummy is moving horizontally or vertically.

Listing 14-1. The Script *scr\_behavior\_walk*

```

{
  var ahead, left, right;
  if ( !place_snapped(32,32) ) exit;
  if (vspeed == 0)
  {
    ahead = place_free(x+hspeed,y);
    left = place_free(x,y+4);
    right = place_free(x,y-4);
    if (!ahead && !left && !right) {direction += 180; exit;}
    while (true) // forever
    {
      if (ahead && random(3)<1) {exit;}
      if (left && random(3)<1) {direction = 270; exit;}
      if (right && random(3)<1) {direction = 90; exit;}
    }
  }
  else
  {
    ahead = place_free(x,y+vspeed);
    left = place_free(x+4,y);
    right = place_free(x-4,y);
    if (!ahead && !left && !right) {vspeed = -vspeed; exit;}
    while (true) // forever
    {
      if (ahead && random(3)<1) {exit;}
      if (left && random(3)<1) {direction = 0; exit;}
      if (right && random(3)<1) {direction = 180; exit;}
    }
  }
}


```

Let's analyze this script in detail. We use three local variables to store values that specify whether the three possible positions are collision free. We first test whether the mummy is aligned with the grid. If not, we exit the script, as nothing needs to be done.

Next we distinguish between a horizontal motion (*vspeed == 0*) and a vertical motion. We'll discuss just the horizontal motion here, as the vertical motion is treated in a similar fashion. We first determine the values of the variables *ahead*, *left*, and *right*. Next, we check whether the first rule is valid. If so, we reverse the direction and exit the code. To pick one of the other rules randomly, we use a loop that continues forever until a rule is picked. To pick a rule with a one-in-three chance we check each time whether *random(3)<1*. This happens on average one out of three times. So in each execution of the loop we try each rule with a one-in-three chance and, if it is valid, we execute the rule and exit the code. We can even change the chances with which rules are picked. For example, we could give a motion straight ahead a bigger chance by, for example, checking whether *random(2)<1*. It is essential that we know that at least one rule is valid; otherwise, the loop would run forever—in our earlier example we know this won't be the case.

Note that this time we use a script rather than an **Execute Code** action. This is because we want to use this as a function in later code. For the moment we'll use this script in the **Step** event of the mummy object.

#### Giving the mummy object behavior:

1. Create the script `scr_behavior_walk`, as indicated earlier.
2. Reopen the properties form of the mummy object.
-  3. Add the **Step, Step** event, then include an **Execute Script** action and indicate the script `scr_behavior_walk`.

Place mummies in some of the rooms and test their behavior; the mummies should traverse the maze successfully.

## Moving Toward the Explorer

Our second behavior tries to move the mummy toward the explorer. To this end we must first determine the direction the player is in relation to the mummy. We'll store this in a local variable called `dir`. This direction between two points can be calculated using the function `point_direction()`, which gives a value between 0 and 360 degrees. To turn it into one of four choices, we can divide it by 90, round it to the nearest integer number, and make our decision based on the resulting value. This will always be 0, 1, 2, or 3. The value 0 means to the right, 1 to the top, 2 to the left, and 3 to the bottom. We can now give the mummy the following rules to follow:

`dir == 0` and can move right ► move right

`dir == 1` and can move up ► move up

`dir == 2` and can move left ► move left

`dir == 3` and can move down ► move down

These rules are exclusive. Whether we can actually move in a particular direction depends on whether that direction is collision free in the first place. Also, we want to avoid the mummy reversing its direction. If all rules fail, we call the previous behavior. This can all be achieved using the script in Listing 14-2.

**Listing 14-2.** *The Script `scr_behavior_towards`*

```
{
  if ( !place_snapped(32,32) ) exit;
  // find out in which direction the explorer is located
  var dir;
  dir = point_direction(x,y,obj_explorer.x,obj_explorer.y);
  dir = round(dir/90);
  if (dir == 4) dir = 0;
  // the four rules that move the mummy in the explorer's direction
  if (dir == 0 && direction != 180 && place_free(x+4,y))
    { direction = 0; exit; }
```

```

    if (dir == 1 && direction != 270 && place_free(x,y-4))
        { direction = 90; exit; }
    if (dir == 2 && direction != 0 && place_free(x-4,y))
        { direction = 180; exit; }
    if (dir == 3 && direction != 90 && place_free(x,y+4))
        { direction = 270; exit; }
    // otherwise do the normal walking behavior
    scr_behavior_walk();
}

```

Try replacing the behavior of the mummy with this new behavior. You will notice that the mummy moves toward you rather rapidly, and it is almost impossible to finish the game—we made the mummy a bit too clever! To make things fairer for the explorer, we should not always use our intelligent behavior. In particular, we should not do it at all if the explorer is far away. This can be achieved using the following behavior. We first compute the distance between the mummy and the player. Then we check whether the mummy can see the player. For this we check whether the line between the centers of the two instances does not collide with a wall. We now use the following rules:

- if player can be seen ► move toward player
- if distance is larger than 200 ► walk around
- otherwise ► with one-in-two chance move toward the player, otherwise walk around

These rules are in order of priority, so we'll execute the first one that is valid. This is achieved using the script in Listing 14-3.

**Listing 14-3.** *The Script `scr_behavior_total`*

```

{
    if ( !place_snapped(32,32) ) exit;
    // Calculate distance and visibility
    var dist,vis;
    dist = point_distance(x,y,obj_explorer.x,obj_explorer.y);
    vis = !collision_line(x+16,y+16,obj_explorer.x+16,obj_explorer.y+16,
                        obj_wall1,false,false);

    // Execute the rules in order
    if (vis) { scr_behavior_towards(); exit; }
    if (dist>200) { scr_behavior_walk(); exit; }
    if (random(2)<1) { scr_behavior_towards(); exit; }
    scr_behavior_walk();
}

```

To use the new script, make the following changes.

**Improving the mummy object behavior:**

1. Create the scripts `scr_behavior_towards` and `scr_behavior_total`, shown earlier.
2. Reopen the properties form of the mummy object.
3. Select the normal **Step** event. Select the **Execute Script** action and change the script to `scr_behavior_total`.

Our mummy now displays some interesting behavior. Try putting some mummies in the pyramid room in interesting configurations, to test out everything we have so far. You can also try to change the behavior—for example, by changing the maximum distance away that the mummy can be before it stops reacting to the explorer, or the chance that the mummy responds when in range. You will find the current version of the game in the file `Games/Chapter14/pyramid5.gm6` on the CD. You might find the game with the mummies is a bit hard at this point, but don't worry too much as we'll modify them next to provide a more realistic challenge.

## Dealing with States

We have so far created some interesting behavior for our pyramid denizens using rules. However, think about real life; people don't just behave according to one set of rules—they adapt their behavior based on the current situation. We could add this type of thing in Game Maker using rules, but that would make the collection of rules huge and hard to keep consistent.

Instead, it is easier to introduce states. For each situation, a character will choose an appropriate state. By creating a state for each situation, we can keep the number of rules for each state small but still create complicated and realistic behavior. When something happens, a character can change state, which can result in different behavior. To start, we'll give our mummy three states: wandering around, searching for the explorer, and chasing the explorer.

State changes can occur when particular events happen, or they can occur according to rules. For example, the wandering mummy changes into a searching mummy when it gets close to the explorer. This one in turn becomes a chasing mummy when it can see the explorer. Such a system, with a finite number of states and events that cause state changes, is called a *finite-state machine*. It is common to draw such a finite-state machine as a diagram in which nodes correspond to the states and arrows correspond to state changes. For our mummy, such a diagram would look like Figure 14-2.

For each of these states, we can then define behavior. In the wandering state, the mummy just walks around. In the searching state, the mummy moves toward the explorer from time to time. Finally, in the chasing state the mummy always moves toward the explorer. We have all of the basic behaviors for this already. We'll use the states to control how they are used.

Using states is very simple in Game Maker. We'll just use a different object for each state. We'll keep our mummy object, but we'll remove the behavior from it, and use it as a parent for the other mummy objects. Remember that the mummy object still has the enemy object as its parent, so our new objects will inherit behavior from both. Three objects are then created for the three different states. In the step event of these objects, we call the correct behavior script.



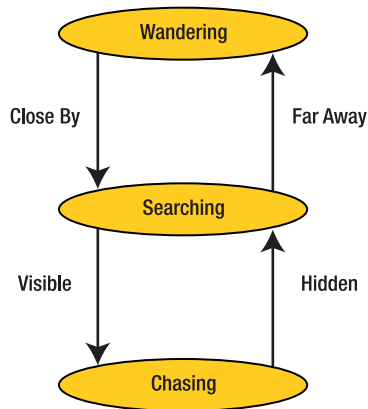




Figure 14-2. The finite-state machine consists of three states.

The only remaining thing to do is to indicate the state changes. We use the **Begin Step** event for this. This event is executed at the beginning of each step, which is a good time to test whether the state should change. If we do need to change state, we'll use the `instance_change()` function to transform the object into the appropriate type. As well as changing the behavior of the mummies in each state, we'll change the speed. This will make the mummies a little easier to avoid, and after all, mummies just wandering about the pyramid are hardly going to move at the same speed as the ones who are chasing our explorer!

#### Creating a different mummy object:

1. Reopen the properties form of the mummy object.
2. Select the normal **Step** event and click the **Delete** button to remove it.
3. Create a new object and call it `obj_mummy_wander`, give it the downward-looking mummy sprite, and indicate `obj_mummy` as its **Parent**.
-  4. Add the **Step, Step** event. Include an **Execute Script** action and indicate the script `scr_behavior_walk`.
-  5. Add the **Step, Begin Step** event. Here we want to tell the mummy to change into a searching mummy when we get close to the explorer (the searching mummy must still be created). Because the searching mummy goes at a slightly faster speed, we also need to check that our position is snapped to a grid at that new speed (in this case 2). Without this check, the mummies can get out of alignment with the grid and move through walls. Include an **Execute Code** action and type the following code:

```

{
    speed = 1; // wandering mummies go slowly
    if ( point_distance(x,y,obj_explorer.x,obj_explorer.y) < 200 )
    {
        if (!place_snapped(2,2) ) exit;
        instance_change(obj_mummy_search,false);
    }
}

```

6. Create a new object and call it `obj_mummy_search`, give it the mummy sprite, and indicate `obj_mummy` as its **Parent**.



7. Add the **Step, Step** event. Include a **Test Chance** action and indicate 2 for the number of sides. Include an **Execute Script** action and indicate the script `scr_behavior_walk`.



8. Next Include an **Else** action and another **Execute Script** action, and indicate the script `scr_behavior_towards`. This gives us our search behavior as described earlier.



9. Add the **Step, Begin Step** event. Here we want to change back into a wandering mummy when we get too far away from the explorer or into a chasing mummy when we see the explorer. Again we need to check that we are snapped to the grid at the new speed (although we can ignore this for the wandering mummy as all positions are snapped to 1 pixel). Include an **Execute Code** action and type the following code:

```
{
    speed = 2;
    if ( !collision_line(x+16,y+16,obj_explorer.x+16,obj_explorer.y+16,
                        obj_wall1,false,false) )
    {
        if ( !place_snapped(4,4) ) exit;
        instance_change(obj_mummy_chase,false);
    }
    if ( point_distance(x,y,obj_explorer.x,obj_explorer.y) > 200 )
    {
        instance_change(obj_mummy_wander,false);
    }
}
```

10. Create a new object and call it `obj_mummy_chase`, give it the mummy sprite, and indicate `obj_mummy` as its **Parent**.



11. Add the **Step, Step** event. Include an **Execute Script** action and indicate the script `scr_behavior_towards`.



12. Add the **Step, Begin Step** event. Here we want to change back into a searching mummy when we no longer see the explorer. Note that we don't need to check whether the place is snapped here because any place that is snapped to 4 is also snapped to 2. Include an **Execute Code** action and type the following code:

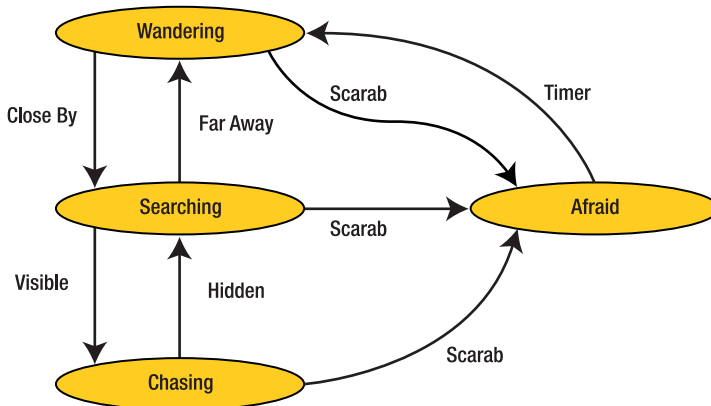
```
{
    speed = 4; // chasing mummies really shift
    if ( collision_line(x+16,y+16,obj_explorer.x+16,obj_explorer.y+16,
                      obj_wall1,false,false) )
        instance_change(obj_mummy_search,false);
}
```

13. Delete the script `scr_behavior_total`, as we no longer need it.
14. In the pyramid room, replace all the mummy instances with wandering mummy instances.

Now test the game again. The new mummies should be more realistic, and carefully sneaking past them should be a fun challenge. You can find this version of the game in the file `Games/Chapter14/pyramid6.gm6` on the CD.

## Scarabs

If you add a lot of mummies to your game, it is still pretty difficult to escape the pyramid alive. To give the explorer a better chance against the mummies, we're going to introduce a scarab object. When the explorer uses a scarab, the mummies become temporarily vulnerable and the explorer can destroy them. To this end we need an additional state for the mummy—the afraid state. When the mummy is afraid, it will no longer move toward the explorer but will instead simply wander around. If the player catches a mummy, the mummy will be destroyed and the player's score will increase. Our new finite-state machine will now look like Figure 14-3.



**Figure 14-3.** The extended finite-state diagram contains four states.

To implement this, we add a new mummy object for the afraid state. The afraid mummy will have the same behavior as the wandering mummy, but its state changes are handled differently.

### Creating the afraid mummy object:

1. Create a new object and call it `obj_mummy_afraid`, give it the mummy sprite, and indicate `obj_mummy` as its **Parent**.
2. Add the **Step, Step** event. Include an **Execute Script** action and indicate the script `scr_behavior_walk`.

Next we must create the scarab object.

**Creating the scarab object:**

1. Create a sprite using the file `Scarab.gif`.
2. Create a new object, call it `obj_scarab`, give it the scarab sprite, and give it a **Depth** of 10. It does not need any behavior.

Next we'll extend the explorer object to deal with scarabs. As well as collecting scarabs, the explorer needs to be able to use them. For this we'll create a **Key Press, <Shift>** event where we turn all mummies into afraid mummies, and we'll set up some alarms to turn the mummies back when the scarab is used up.

**Extending the explorer object:**

1. We need some sounds for when we activate the scarab power, so create `snd_power_start` with the file `PowerStart.wav` and `snd_power_end` with the file `PowerEnd.wav`.
2. Reopen the properties form for the explorer object. Select the **Create** event and change the code by setting the variable `scarab_count` to 0. We'll use `scarab_count` to check how many scarabs the explorer is carrying and whether he can use one to chase the mummies away. The new code should look like this:

```
{
    image_speed = 0;
    scarab_count = 0;
}
```



3. Add a **Collision** event with `obj_scarab`. Include an **Execute Code** action and add the following code, which plays a sound, increases the score, and destroys the scarab object. It also adds 1 to the explorer's scarab count.

```
{
    sound_play(snd_treasure);
    score += 50;
    scarab_count += 1;
    with (other) instance_destroy();
}
```



4. Add a **Key Pressed, <Shift>** event. Include an **Execute Code** event with the following code, which sets **Alarm0** to 300 (10 seconds) and turns all mummies into afraid mummies. We also set **Alarm1** so that we can play the end sound slightly before the timer ends so the explorer has some warning. But before this is done, the code checks to see that the explorer has collected a scarab, and if he has, it reduces `scarab_count` by 1 to indicate that the scarab has been used.

```
{
    if ( scarab_count > 0 )
    {
        with (obj_mummy) instance_change(obj_mummy_afraid,false);
        alarm[0] = 300; // scarab timer
        alarm[1] = 240; // end of scarab sound timer
    }
}
```

```

        scarab_count -= 1;
        sound_play(snd_power_start)
    }
}

```

-  **5.** Add the **Alarm0** event. Include an **Execute Code** action with the following code, which turns all surviving mummies back into wandering mummies:

```

{
    with (obj_mummy) instance_change(obj_mummy_wander,false);
}


```

-  **6.** Add the **Alarm1** event. Include a **Play Sound** action with `snd_power_end` to play the end of power sound.

Next we define a collision event between the explorer and the scared mummy. Here we play a sound, increase the score, and destroy the mummy instance. As this event overrides the collision event with the basic enemy object, the explorer is not killed in this case.

#### Handling the collision between the explorer and the scared mummy:

1. Create a sound with the name `snd_scared` using the file `Scared.wav`.
2. Reopen the properties form for the explorer object.

-  **3.** Add a **Collision** event with `obj_mummy_afraid`. Insert an **Execute Code** action and add the following code, which plays a sound, increases the score, and destroys the mummy:

```

{
    sound_play(snd_scared);
    score += 100;
    with (other) instance_destroy();
}

```

The explorer can now collect and use scarabs, but there is no indication of how many he has collected. In order to make this clearer to the player, let's extend our display at the bottom of the screen to include the explorer's scarab count.

#### Extending the score display:

1. Reopen the properties form for `obj_controller`.
2. At the end of the **Draw** event add a **Draw Sprite** action with **Sprite** set to `spr_scarab`, **X** set to 290, **Y** set to 1, and the **Relative** option enabled.
3. Add a **Draw Variable** action and choose `obj_explorer.scarab_count` as the variable. Set **X** to 340 and **Y** to 5, and enable the **Relative** option.





In order to help the player we'll visually indicate whether or not the mummy is afraid. We can achieve this by changing the color of the sprite. We'll actually take this a step further and also use different colors for the searching and chasing mummy. This will make it clear to the player when he must be careful.

---

**Note** Changing the sprite color is only possible in the registered version of Game Maker. If you do not have a registered copy, you must solve this differently—for example, by using a different sprite or by indicating the status on the panel at the bottom.


---

#### Changing the color of the different mummy objects:

-  1. Reopen the properties form for `obj_mummy_afraid`, and select the **Step** event. Include the **Color Sprite** action from the **main1** panel. Indicate a blue color.
-  2. Reopen the properties form for `obj_mummy_wander`, and select the **Step** event. Include the **Color Sprite** action and indicate a white color.
-  3. Reopen the properties form for `obj_mummy_search`, and select the **Step** event. Include the **Color Sprite** action and indicate an orange color.
-  4. Reopen the properties form for `obj_mummy_chase`, and select the **Step** event. Include the **Color Sprite** action and indicate a red color.

Before we finish this section, we've got one more trick up our sleeves. With plenty of mummies in the game, it's still pretty hard, so to help the player out let's add some health potions to give the player extra lives.

#### Creating the potion object:

1. Create a sprite `spr_potion` with the `Potion.gif` file.
2. Now create an `obj_potion` object and assign the `spr_potion` sprite. Give it a **Depth** of 10 so that enemies move on top of it.
3. Reopen the properties form for the explorer object.
-  4. Add a **Collision** event with the `obj_potion` object and insert the **Execute Code** action. In the code window type the following code:

```
{
    lives += 1;
    with (other) instance_destroy();
}
```

Add some scarabs, potions, and some more mummies and test the game thoroughly. If it seems too hard, you can add extra scarabs or potions. If it is too easy, adding some more mummies will fix that. Now that the explorer has a way to fight back, you should begin to find that the game is much more fun to play and that balancing it is much easier. You will find the current version of the game in the file `Games/Chapter14/pyramid7.gm6` on the CD.

## Let There Be Light

As the final touch for our game, we'll be adding a second objective. As well as escaping alive, our explorer will be searching for the pyramid's biggest treasure, the Sword of Ra. The Sword of Ra will be worth a staggering 5,000 points, but it won't be all positive. Drawing the sword will switch off all of the lights in the pyramid. The explorer will be able to use the dim light that the sword still produces to light his way, and it will also scare away the mummies, but every second it is activated will drain his score and reduce his eventual wealth.

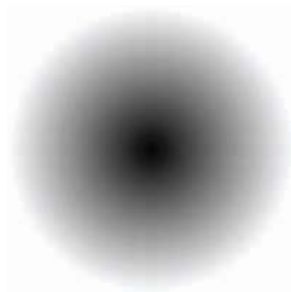
---

**Note** This section requires a registered version of Game Maker because it uses some features that are not available in the free version.

---

To implement this, we need a couple of ingredients. First, we need a way to show only part of the world. We'll use a global variable called `global.swordon` to indicate whether or not the sword is activated. This will tell us whether our light should light only close to the explorer or further out. We'll also extend the controller object to control the light. In its **Create** event, we'll set `global.swordon` to false to indicate that the sword is not activated. When the player presses or releases the spacebar, we switch the value from false to true and back. Finally, in the draw event of the controller we must hide the part of the world that cannot be seen. To make sure that the panel at the bottom of the screen is unaffected, we'll do this draw step before drawing the panel.

To hide part of the room, we use a special form of blending. We add the image in Figure 14-4 as a background, and then subtract this image from the room image. Normally images we draw replace the background image. But we can change the blend mode to get different effects. Don't forget to set the blend mode back to normal afterward.



**Figure 14-4.** We subtract this image from the room image.

As this image is black (0) in the center, nothing is subtracted there, but more is subtracted as you go toward the outside of the image, until the image is completely white on the outside. At that point everything is subtracted, leaving the area outside the image completely black (we simply fill this area with black rectangles as nothing can be seen there). You can see the result in Figure 14-5, which is pretty impressive and results in a creepier feeling.



**Figure 14-5.** Subtracting the image from the room gives a creepy feeling.

The script in Listing 14-4 achieves this. As arguments, the script gets the location of the light. Depending on the value of the variable `global.swordon`, it determines the size of the lit area. Next, it draws black rectangles about this area. Finally, it sets the blend mode, draws the light image in the correct size, and sets the blend mode back to normal.

**Listing 14-4.** The Script `scr_light`

```
{
    // Only draw if the explorer has found the sword
    if ( !obj_explorer.has_sword ) exit;
    // First determine the size of the lit area
    var x1,y1,x2,y2,ww;
    if ( global.swordon ) ww = 800 else ww = 300;
    x1 = argument0-ww/2;
    x2 = argument0+ww/2;
    y1 = argument1-ww/2;
    y2 = argument1+ww/2;
    // Hide things that are far away by drawing black rectangles
    draw_set_color(c_black);
    draw_rectangle(0,0,x1,room_height,false);
    draw_rectangle(x2,0,room_width,room_height,false);
    draw_rectangle(0,0,room_width,y1,false);
    draw_rectangle(0,y2,room_width,room_height,false);
    // Now hide nearby stuff by subtracting the light image
```



```

draw_set_blend_mode(bm_subtract);
draw_background_stretched(back_light,x1,y1,ww,ww);
draw_set_blend_mode(bm_normal);
}

```

The script must be called at the beginning of the **Draw** event of the controller object.

#### Extending the controller object:

1. Create a background resource from the file `Light.bmp` and name it `back_light`.
2. Create the script `scr_light` as described earlier.
3. Reopen the properties form for the controller object.



4. Add the **Create** event and include the **Set Variable** action, with **Variable** set to `global.swordon` and **Value** to false.



5. Select the **Draw** event. At the start of the list of actions include the **Execute Script** action. As **Script** select `scr_light`. As **Argument0** indicate `obj_explorer.x+16`, and as **Argument1** indicate `obj_explorer.y+16`. This will center the light on the explorer.

Next we need to create the sword object.

#### Creating the sword object:

1. Create a sprite from the file `Sword.gif`.
2. Create an object using this sprite and call it `obj_sword`. Set the **Depth** to 10. No events or actions are required.

Now we need to edit the explorer object to handle collision with the sword, and to allow the player to use the sword's power when it is needed.

#### Extending the explorer object:

1. Reopen the properties form for the explorer object. In the **Create** event, open the **Execute Code** action and add the line `has_sword = false;`. This makes sure that the explorer starts out without the sword.



2. Now add a **Collision** event with `obj_sword`. Include the **Execute Code** action and type the following code:

```

{
    score += 5000;
    sound_play(snd_treasure);
    show_message("As you draw the mighty Sword of Ra the lights go out!" +
                "##Frighten the mummies, but don't lose your treasures. ");
    has_sword = true;
    with (other) instance_destroy();
}

```

This is very similar to the **Collision** events with the other types of treasure, but we also set the variable `has_sword` to true and show a message. Showing a message like this helps to indicate to the player that something really special has happened and makes the player feel a real sense of achievement.



3. Add the **Keyboard, <Space>** event, and include the **Execute Code** action. Add the following code:

```
{
  if ( has_sword && (score > 0) )
  {
    global.swordon = true;
    with (obj_mummy) instance_change(obj_mummy_afraid,false);
    score -= 10;
  }
}
```

This checks that the explorer has found the sword, and also that he still has score to feed it. If both of those things are true, it changes all of the mummies into scared mummies and subtracts 10 from the score. Because this event happens every step that the spacebar is held, this subtracts 300 points per second. You'll soon spend all 5000 points the sword gave you if you're not careful!



4. Add the **Key Release, <Space>** event. Include the **Execute Code** action and type the following code:

```
{
  if ( has_sword )
  {
    global.swordon = false;
    if ( alarm[0] <= 0 )
    {
      with (obj_mummy_afraid) instance_change(obj_mummy_wander,false);
    }
  }
}
```

This sets `global.swordon` back to false and turns all of the mummies back to wandering mummies. Before it does this, though, it double-checks that **Alarm0** is not running. This ensures that it does not turn the mummies back if a scarab is currently in use.

And there you go. Put one instance of the sword object at a special place in the pyramid room and test that it works. But before we finally wrap this game up, there's one more change to make to the mummies. Now that the lights are out, it would be a bit unfair if the mummies still detected the explorer from the same distance. To change that, let's check the `obj_explorer.has_sword` variable; if it is true we'll make the distance smaller to reflect the small amount of light.

**Adapting the behavior of the mummy object:**

1. Reopen the properties form for the wandering mummy, select the **Begin Step** event, and double-click on the **Execute Code** action. Now change the code as follows:

```
{
  var maxdist;
  if ( !obj_explorer.has_sword )
    maxdist = 200
  else
    maxdist = 75;
  speed = 1; // wandering mummies go slowly
  if ( point_distance(x,y,obj_explorer.x,obj_explorer.y) < maxdist )
  {
    if ( !place_snapped(2,2) ) exit;
    instance_change(obj_mummy_search,false);
  }
}
```

2. Reopen the properties form for the searching mummy, select the **Begin Step** event, and double-click on the **Execute Code** action. Change this code as follows:

```
{
  var maxdist;
  if ( !obj_explorer.has_sword )
    maxdist = 200
  else
    maxdist = 75;
  speed = 2;
  if ( !collision_line(x+16,y+16,obj_explorer.x+16,obj_explorer.y+16,
    obj_wall1,false,false) )
  {
    if ( !place_snapped(4,4) ) exit;
    instance_change(obj_mummy_chase,false);
  }
  if ( point_distance(x,y,obj_explorer.x,obj_explorer.y) > maxdist )
  {
    instance_change(obj_mummy_wander,false);
  }
}
```

That finishes the changes we must make. You will find the final version of the game in the file [Games/Chapter14/pyramid8.gm6](#) on the CD. Now you should take some time to complete your masterpiece. Be sure to get some friends to test out the game for you to ensure that it isn't too hard, and have fun—there are a lot of great objects to play with here, so be sure to go wild!

## Looking to the Future

This is the last game project in the book, but it's just the beginning of your game design career. Taking this game as a springboard, you could explore lots of directions. Maybe you want to add a whole set of different pyramids, each with different treasures and devious new traps. Or perhaps it's time for our explorer to break free of claustrophobic corridors and step out in a new direction—plunging deep in the rainforests of the Amazon or perhaps fighting winds across the icy wastes of Antarctica. You might even feel like adding some totally new game mechanics to the game—perhaps the explorer has to take to the skies in his trusty biplane, navigating through deadly desert sandstorms and avoiding terrible swarms of man-eating locusts on the way to his next target.

The fun doesn't stop with this game. Using the game design skills you have picked up so far in this book along with the advice in Chapter 15, you are in the perfect situation to start designing your own games. Imagine yourself an unlikely dustball hero, create yourself an unstoppable army of weasels, travel into deep space to play ping-pong with stars—the only limit is your imagination. Go forth and design—you have nothing to lose but your boredom!



## CHAPTER 15



# Final Words

**S**o that's it—you've made it to the end. You've completed all the challenges this book has to offer and, we hope, learned a lot about creating games in the process. You've battled demons, flown spacecrafts, dodged crates, juggled starfish, rescued koalas, commanded tanks, survived dogfights, hidden from mummies, and still had time for a game of tic-tac-toe along the way! We hope you've enjoyed this journey. Just because this book has come to an end, it doesn't mean that your gaming projects have to. So before we say a final farewell, we'll quickly mention some other directions you could explore if you want to continue enhancing your game development skills.

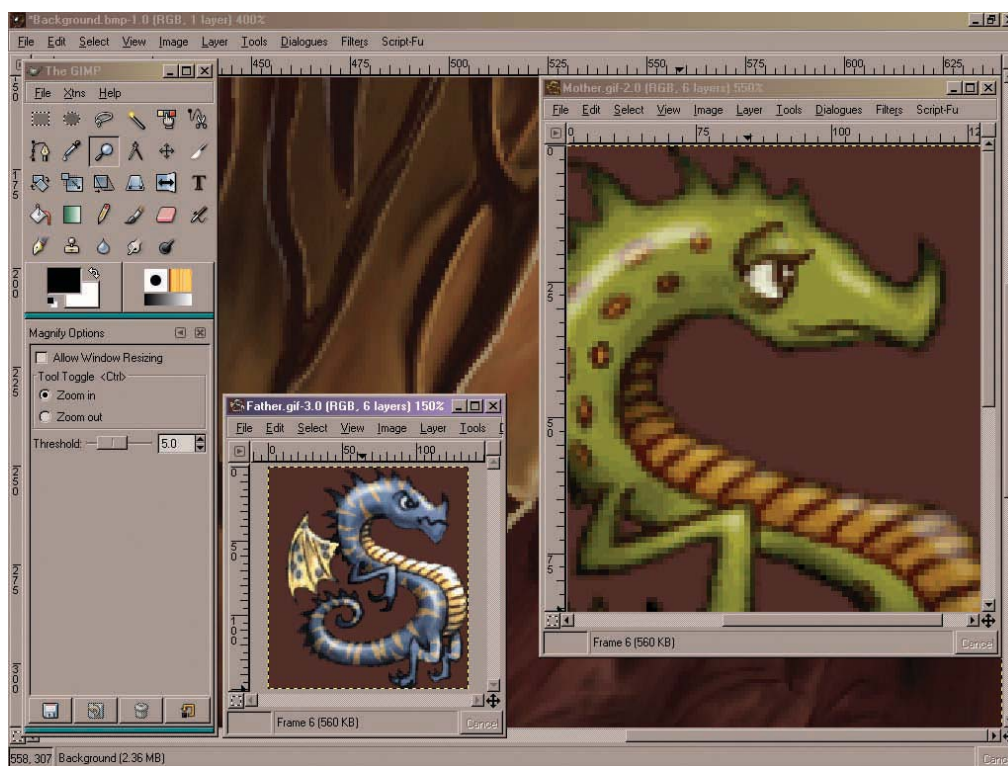
## Creating Resources

As we said at the start, you're free to use the graphics, music, and sound effects provided with this book for your own Game Maker projects. However, those of you with artistic or musical talents will eventually want to try making your own resources for your games. In this section, we'll briefly mention some of the tools available for doing this. In each case we've chosen to focus on free tools, as you'll lose nothing from giving them a try before looking elsewhere.

### Artwork: The GIMP

The GNU Image Manipulation Program—GIMP for short—is a professional-quality 2D art package, which is free to download and use on your PC (see Figure 15-1). It has many more features than the Image Editor in Game Maker and is comparable to Adobe Photoshop—the main 2D art package used in the games industry (and in this book). It's not the easiest package to learn how to use, but if you're serious about creating game artwork, then the GIMP will allow you to stretch your artistic talents while learning some professional techniques. Visit the GIMP website for more information and to download the program to your machine:

<http://gimp-win.sourceforge.net>.



**Figure 15-1.** This is a screenshot from the GIMP.

While Photoshop and the GIMP are great packages, they can be a bit too much for beginners to handle, particularly when trying to use them to create sprites. HUMANBALANCE Co.'s GraphicsGale and Cosmigo's Pro Motion are cheap alternatives that are easier to use and designed specifically for creating sprites. You can download trial versions of these packages from the following websites:

- [www.humanbalance.net/gale/us](http://www.humanbalance.net/gale/us) (GraphicsGale)
- [www.cosmigo.com/promotion/index.php](http://www.cosmigo.com/promotion/index.php) (Pro Motion)

## Music: Anvil Studio

Music can add a lot of atmosphere to games as well as being a great deal of fun to create. Willow Software's Anvil Studio (see Figure 15-2) is a free beginner's package, which includes loads of features as well as helpful tips and tutorials to get you on your way. Packages like Anvil are MIDI based, and allow you to compose MIDI music files that are small in size and easy to include in your Game Maker games. MIDI music is the computer equivalent of writing down music on paper and getting the computer to perform the piece of music each time it needs to

be played. This means that the way it sounds depends on the computer's sound card, so the music will often sound different on different computers. You can get around this by recording a performance of a MIDI song on your computer as a WAV file and including this in your game. The music will then sound exactly the same on every computer, but the WAV file will be much larger in size than the original MIDI. You can download Anvil Studio from the following website: [www.anvilstudio.com](http://www.anvilstudio.com).

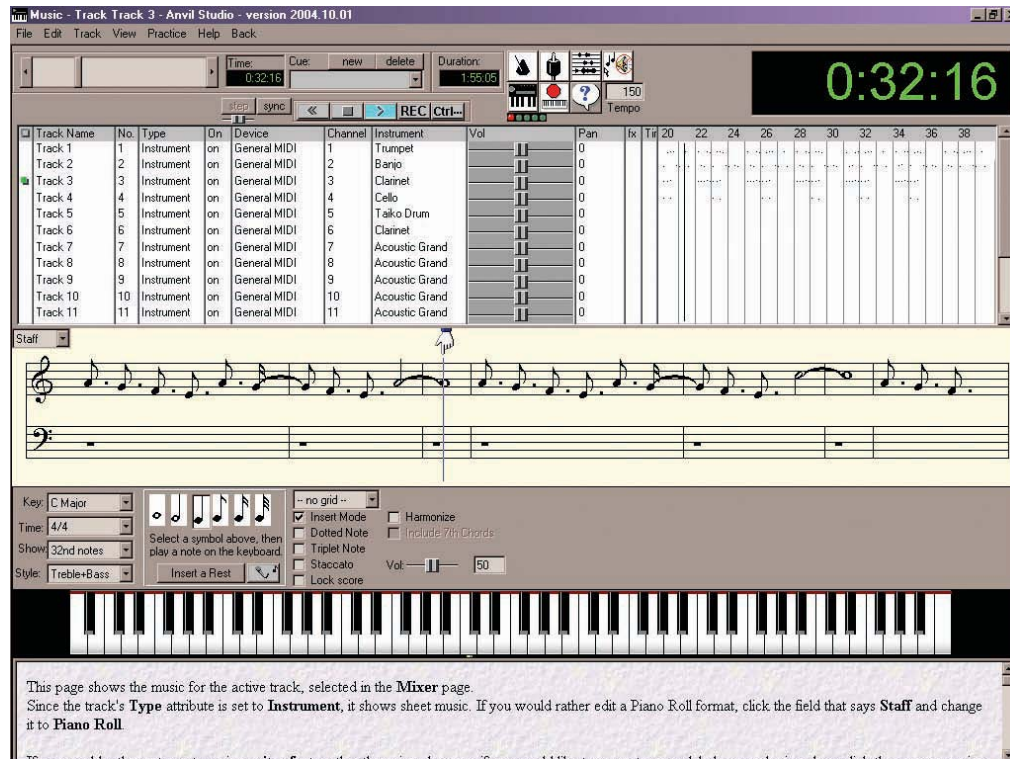


Figure 15-2. This is a screenshot from Anvil Studio.

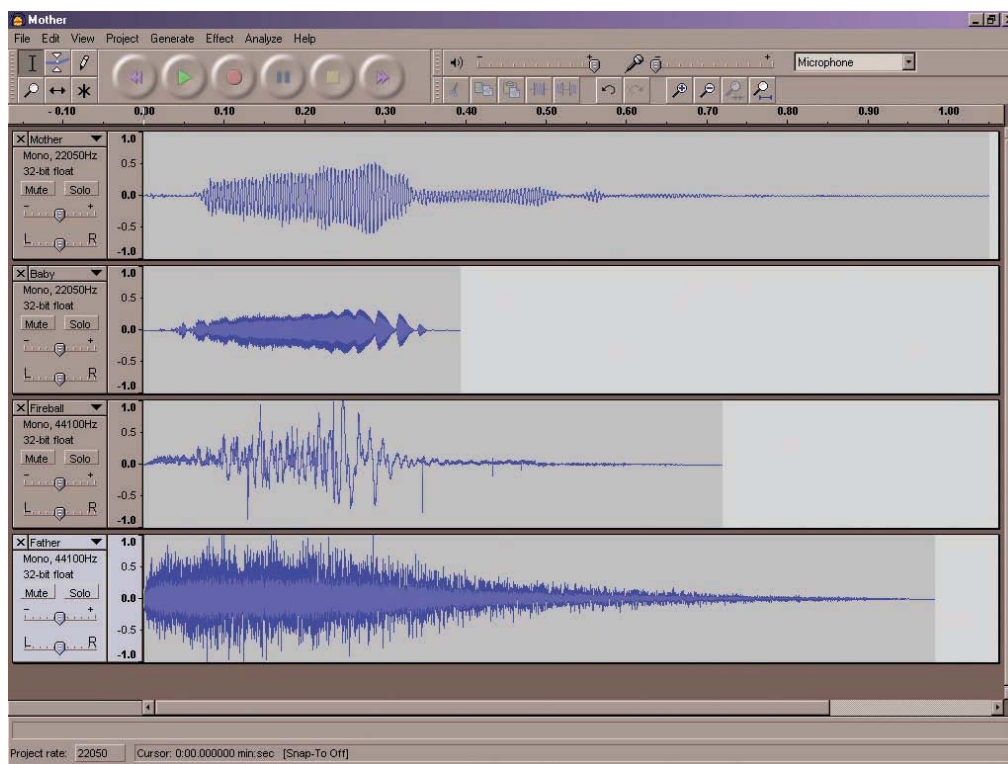
Although Anvil is a fine place to start with making music, MIDI Tracker (from RF1 Systems) is a little simpler to use and not expensive to buy online. In fact, most of the MIDI files from the games in this book were originally composed on MIDI Tracker. Steinberg's Cubase SE is an entry-level music production package in the same range as those used in professional game development. This was the package used to create the MP3 music files included in this book. However, packages like this require special kinds of sound cards in order to work properly, so don't rush out and buy one without reading up on them first.

- [www.rf1.net/software/mt](http://www.rf1.net/software/mt) (MIDI Tracker)
- [www.steinberg.de](http://www.steinberg.de) (Cubase SE)



## Sound Effects: Audacity

Audacity (see Figure 15-3) is an excellent open source program for recording and editing sound effects. With a reasonable sound card and a microphone, it should provide all you need for creating sound effects for your games. It's easier to create sound effects if your sound card is full-duplex—that is, it can both record and play sounds at the same time. Most modern PC sound cards can do this, although some laptop sound cards don't. You can download Audacity from <http://audacity.sourceforge.net>.



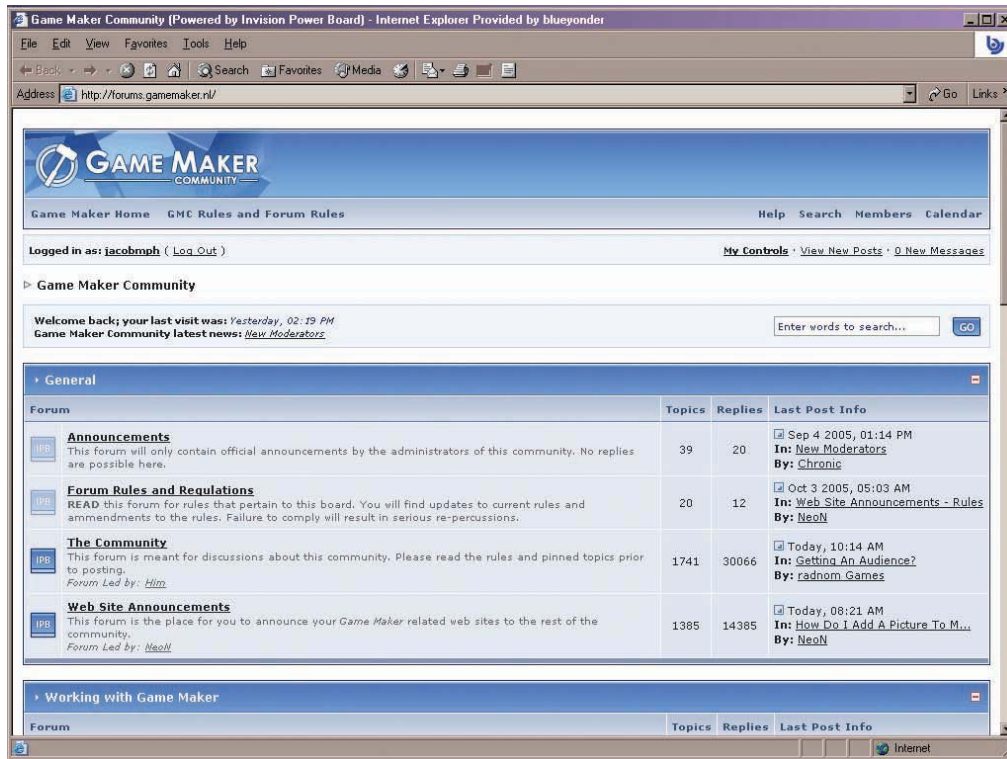
**Figure 15-3.** This is a screenshot from Audacity.

If you feel the need to use something more professional, then Sound Forge Audio Studio is the package used to create the sound effects in this book. This is an entry-level version of the Sound Forge packages that are used to create sound effects in professional games. You can find more information at [www.sonymediasoftware.com](http://www.sonymediasoftware.com) (the website of Sony Media Software, the publishers of Sound Forge).

## The Game Maker Community

Whatever you go on to do with Game Maker, we hope that you will join the online Game Maker community (see Figure 15-4) and share your ideas, knowledge, and expertise with other

users. The Game Maker forum has over 20,000 worldwide members and is a great place to get help or advice at any time of the day or night. You'll even find advice on the tools described in the previous section, as well as a whole host of others.



**Figure 15-4.** You'll want to join the Game Maker community.

Point your web browser to <http://forums.gamemaker.nl> and you'll be in contact with Game Maker users from all around the world in no time at all. From time to time you may even see us contributing to the discussions.

---

**Caution** Remember that it's not a good idea to give out personal information (like your full name, school/college, or home address) on any kind of Internet forum.

---

## Note to Teachers

Game Maker is already used in a growing number of clubs and schools worldwide as an engaging activity for students from ages 7 to 70. It is certainly ideal for students from 14 up, and you can find out more about some of the projects happening worldwide through the teacher's forum at [www.gamelearning.net](http://www.gamelearning.net).

## Good Luck

Twenty years ago every self-respecting computer enthusiast owned a collection of books about programming games for the simple computers of the day. These books usually made the reader type in pages of code, which generally contained a generous helping of typing errors and bugs to work around. If you were lucky, then hours of painstaking labor might eventually be rewarded with the chance to control the letter O as it was chased around the screen by a couple of nasty-looking As. Nonetheless, it was books just like these that inspired us to write this one, because they were responsible for starting us down a path that led to the game-related careers that we have today.

These days it's tools like Game Maker that will provide the first step on the ladder for games industry professionals of the future. It's exciting to think that some of you may go on to work in the games industry and help to define the shape of game development over the decades to come. It's difficult to predict how far games will have evolved by then, but you can be sure that they will look quite different from the 2D games that you've worked on in this book. Nevertheless, despite the technological advancements that the future will bring, it's unlikely that their game mechanics will be such a far cry from the ones you've explored here. We wish you the best of luck and hope that you'll remember your first game development projects as fondly as we remember our own.

Jacob and Mark, 2006





# Bibliography

The design chapters in this book are founded on over two decades of amateur and professional experience developing computer games. Nonetheless, the following texts have helped to add structure and conviction to many of the ideas and concepts discussed therein:

Crawford, Chris. *The Art of Computer Game Design*. Out of print: available online at [www.vancouver.wsu.edu/fac/peabody/game-book/Coverpage.html](http://www.vancouver.wsu.edu/fac/peabody/game-book/Coverpage.html), 1982.

Crawford, Chris. *On Game Design*. Indianapolis: New Riders Publishing, 2003.

Loftus, Geoffrey and Loftus, Elizabeth. *Mind at Play: The Psychology of Video Games*. New York: Basic Books, 1983.

Malone, Thomas and Lepper, Mark. "Making Learning Fun: A Taxonomy of Intrinsic Motivations for Learning." In *Aptitude, Learning and Instruction: III. Conative and affective process analyses*. Hillsdale, NJ: Erlbaum, 1987, p. 223–253.

Rollings, Andrew and Morris, Dave. *Game Architecture and Design*. Scottsdale, AZ: Coriolis Group, 2000.

Rollings, Andrew and Adams, Ernest. *On Game Design*. Indianapolis: New Riders Publishing, 2003.

Salen, Katie and Zimmerman, Eric. *Rules of Play, Game Design Fundamentals*. Cambridge, MA: MIT Press, 2004.

Wilson, Phil. *Hogs of War: Supplementary Game Design (original concept by Ade Carless)*. Unpublished design document, 1999.





# Index

&& (and) operator, GML, 232  
// (comments), 248  
{ } (curly brackets), GML, 228  
= (equal) symbol, GML, 228  
+ (increment) operator, GML, 229, 236  
! (not) operator, GML, 232  
|| (or) operator, GML, 232  
\_ (underscore) symbol  
  filenames, 12  
  GML, 229

## A

action games, 87  
actions, 14–18  
  Destroy Instance, 176, 178  
  Else, 70–71  
  End Block. *See* blocks  
  Execute Code. *See* Execute Code action  
  Execute Script. *See* Execute Script action  
  Set Score, 28–29, 32  
    Super Rainbow Reef, 113  
    Galactic Mail, 57–59, 61  
  Set Time Line, 182–183  
  Set Variable, 172  
    boss plane object (*Wingman Sam*), 186–187  
    enemy bullet objects (*Wingman Sam*), 185  
    flying plane objects (*Wingman Sam*), 177,  
      179–180  
  Sleep, *Galactic Mail* flying rocket object, 59  
  Start Block. *See* blocks  
  Test Instance Count, *Galactic Mail* flying rocket  
    object, 59  
  Test Variable, 172  
    boss plane object (*Wingman Sam*), 187  
    enemy plane objects (*Wingman Sam*), 178  
    flying plane objects (*Wingman Sam*), 175,  
      179  
    looping island objects (*Wingman Sam*), 173  
    time lines, 182–183  
  Wrap Screen. *See* Wrap Screen action  
adaptive gameplay, *Tic-Tac-Toe* computer  
  opponent, 256–257  
Advanced mode (Game Maker), 10, 42–43  
adventure games, 88  
alarms  
  *Super Rainbow Reef*  
    stationary Pop object, 124–125  
    congratulation object, 106–107

## Tank War

  parent rocket object, 202  
  pickup object, 200  
  shell objects, 198  
  wall object, 198  
  *Wingman Sam* boss plane, 186–188  
angles, *Super Rainbow Reef* Pop object, 110–111  
animated characters. *See* Lazarus, Lazarus  
  character  
Anvil Studio, 292–293  
arms races, 152  
arrays (GML), 237–239  
artificial stupidity, computer-controlled  
  opponents, 220  
assignment statements (GML), 228  
asteroid object (*Galactic Mail*), 47–48  
  adding visual variety, 62  
  creating, 47–48  
Audacity, 294  
Audio Studio, 294  
audio. *See* sound

## B

baby dragon object (*Evil Clutches*), 31–32  
backgrounds  
  images (*Evil Clutches*), 33–34  
  Lazarus, 82  
  music  
    *Evil Clutches*, 34–35  
    *Wingman Sam*, 188  
  scrolling (*Wingman Sam*), 173  
  *Tank War* arena, 192–193  
  *Tic-Tac-Toe*, 246  
balancing (multiplayer games)  
  beginnings, 214–218  
  choice, 218–219  
  computer-controlled opponents, 220  
beetle object (*Pyramid Panic*)  
  creating, 268  
  making intelligent, 268–269  
Begin Step event, koala object (*Koalabr8*), 134  
behavior  
  reactive behavior (*Pyramid Panic*), 267–269  
  rule-based behavior (*Pyramid Panic*), 271–275  
Bigleg object (*Super Rainbow Reef*), 113–116  
  moving Bigleg, 116  
  parent/child objects, 115–117  
  small Bigleg, 115  
  small moving Bigleg, 116

- block objects
    - Pyramid Panic, 270–271
    - Super Rainbow Reef*, 120–123
  - blocks, 59–60
    - End Block action
      - Galactic Mail*, 60
      - Lazarus*, 70–71
    - Start Block action
      - Galactic Mail*, 59
      - Lazarus*, 69, 71
  - boss object (*Evil Clutches*)
    - background music, 34–35
    - randomly creating baby dragons, 32
    - summoning demons, 30–31
  - boss plane (*Wingman Sam*)
    - adding hit counter, 186–187
    - creating plane object, 186
    - displaying hit counter, 187
    - making box more challenging, 187–188
  - bouncing bombs (*Tank War*)
    - creating, 203–204
    - shooting, 204–205
  - brackets (GML), 233
  - brainstorming features, 150–151
  - bullet object (*Wingman Sam*), 176
    - enemy bullets, 184–185
  - button objects (*Super Rainbow Reef*), 104
- C**
- calling functions, 231
  - challenges, 88–92
    - difficulty, 88–89
    - goals, 89–90
    - interactive challenges, 87
    - rewards, 90–91
    - subgoals, 92
  - Change Instance action
    - Galactic Mail*, 54
    - Lazarus*, 68, 70–71, 74, 76–77
  - Change Sprite action, 52–53
  - characters, balancing beginnings (multiplayer games), 214–218
  - cheats
    - Lazarus*, 82
    - Super Rainbow Reef*, 126
  - Check Collision action, 69, 79
  - Check Empty action, 69–72
  - Check Grid action, 132, 134
  - choice, 93–94
    - balancing, 218–219
  - code
    - executing, 225–228
      - Execute Code actions. *See* Execute Code actions
      - Execute Script actions. *See* Execute Script actions
    - scripts as functions, 240–242
    - writing. *See* GML
  - collisions/Collision event
    - Evil Clutches*
      - baby dragon, 31
      - demon object, 28–29
    - Galactic Mail*, flying rocket object, 54–55
    - Koalabr8*
      - exit objects, 136
      - saw objects, 139
      - sound effects, 148
    - Lazarus* (falling boxes), 74, 80
      - sound effects, 81
    - parent objects, 117
    - Pyramid Panic*
      - basic enemy object, 267–268
      - explorer colliding with scarab, 281
      - explorer with block objects, 271
      - explorer with treasure objects, 270
      - extending explorer object to handle collision with sword, 286–287
      - handling collision between explorer and scared mummy, 282
      - Potion object, 283
      - rule-based behavior, 272
      - wall object, 264
    - Super Rainbow Reef*
      - Bigleg object, 113
      - Pop and Katch objects, 111
    - Tank War*
      - bouncing bomb objects, 203–204
      - pickup object, 200
      - rocket objects, 202
      - shells, 198
      - tank objects, 201, 204
      - tanks, 193–194
    - Wingman Sam*, enemy planes, 178
  - color
    - changing mummy color (*Pyramid Panic*), 283
    - random color generation, 235
  - commands
    - File menu
      - Advanced Mode, 10, 42
      - Exit, 8
      - New, 42
      - Open, 7
    - Resources menu
      - Create Script, 225–226
      - Create Object, 14, 18
      - Create Room, 20
      - Create Sprite, 11, 13
    - Run menu, Run in Debug Mode, 243
  - comments, 248
  - comparison operators, 232

competition (multiplayer games), 211–214  
 combining with cooperation, 213  
 dependent, 212  
 independent, 211  
 completion screens  
     *Koalabr8*, 130  
     *Pyramid Panic*, 262  
     *Super Rainbow Reef*, 106–107  
 computer opponents  
     balancing, 220  
     *Tic-Tac-Toe*  
         adaptive gameplay, 256–257  
         simple logic, 251–254  
         simple strategy, 254–256  
 conditional actions, 58–59  
 conditional statements, 232–234  
 congratulation object, *Super Rainbow Reef*  
     completion screen, 106  
 control, 93–94  
 controller objects, 75  
     *Koalabr8*, 135  
         adding events to handle rescued koalas,  
             136–137  
         creating, 135  
         editing to fix dead koala bug, 138–139  
     *Lazarus*, 75–77  
     *Pyramid Panic*  
         creating, 262–263  
         extending, 266, 286  
     *Super Rainbow Reef*  
         adding no more lives event, 118–119  
         creating, 113  
         showing remaining lives, 119  
     *Tank War*, 193  
         creating, 193  
         drawing scores, 208–209  
         recording scores, 195  
     *Wingman Sam*  
         creating, 179  
         playing background music, 188  
 cooperative multiplayer games, 213–214  
     combining with competition, 213  
     dependent, 213  
     independent, 213  
     *Wingman Sam*. *See* *Wingman Sam* game  
 Cosmigo Pro Motion, 292  
 Create event, 25  
 Create Instance action, 57  
 Create Object command (Resources menu), 14, 18  
 Create Random action, 77  
 Create Room command (Resources menu), 20  
 Create Script command (Resources menu),  
     225–226  
 Create Sprite command (Resources menu), 11, 13  
 Cubase SE, 293  
 curly brackets {}, GML, 228  
 cyclic relationships, 218–219

**D**  
 damage, *Tank War*  
     damage mechanism, 196–197  
     health bars, 197  
 darkness (*Pyramid Panic*), 284  
     adding behavior of mummy object, 288  
     creating sword object (Sword of Ra), 286  
 dead koala object (*Koalabr8*), 138  
 debugging programs, 242–243  
 delayed actions, stationary Pop object (*Super  
     Rainbow Reef*), 124–125  
 demon object (*Evil Clutches*), 27–29  
     summoning demons, 30–31  
 dependent competition, 212  
 dependent cooperation, 213  
 depth, 47  
     *Galactic Mail*  
         asteroid object, 47–48  
         explosion object, 56  
         flying rocket object, 53  
         landed rocket object, 51  
         special moon object, 50  
         title object, 60  
 design, 9  
     challenges, 88–92  
         difficulty, 88–89  
         goals, 89–90  
         rewards, 90–91  
         subgoals, 92  
     emergence, 153, 161  
         springs (*Koalabr8*), 161  
     features, 149–154  
         arms races, 152  
         brainstorming, 150–151  
         emergence, 153  
         equivalent features, 151  
         *Koalabr8*, 160–161  
         one-trick ponies, 152  
     *Galactic Mail*, 41–42  
     game mechanics, 86–87  
     genres, 87–88  
     good game design, 85–86  
     interactivity, 92–95  
         audio feedback, 95  
         choices and control, 93–94  
         interactive challenges, 87  
         unfair punishment, 94  
     *Koalabr8*, 127–128  
     *Lazarus*, 65  
     levels, 154–159  
         difficulty curves, 158–159  
         dividing levels, 162–163  
         learning curves, 156–159  
         training levels, 161–162



- multiplayer games
    - balancing beginnings, 214–218
    - balancing choice, 218–219
    - balancing computer-controlled opponents, 220
    - competition, 211–214
    - cooperation, 213–214
    - providing opportunities to save, 160
    - Pyramid Panic*, 259–260
    - Super Rainbow Reef*, 101–103
    - Tank War*, 191–192
      - balancing choice, 218–219
      - balancing computer opponents, 220
      - balancing tank differences, 215–218
    - Tic-Tac-Toe*, 245–246
    - Wingman Sam*, 169–170
  - Destroy Instance action, 28
    - Galactic Mail*, 57
    - Lazarus*, 74
    - Wingman Sam*
      - bullet object, 176
      - enemy planes, 178
  - detonator object (*Koalabr8*), 144–145
  - difficulty, 88–89. *See also* challenges
  - difficulty curves, level design, 158–159
  - disabling Esc key (*Super Rainbow Reef*), 124
  - Display Messages
    - Lazarus*, 69, 80
    - Pyramid Panic*, 262
  - Draw actions, 137
  - Draw event, 119
    - showing remaining lives (*Super Rainbow Reef*), 119
    - tanks (*Tank War*), 193–194, 197
  - Draw Life Images action (*Koalabr8*), 137
  - Draw Sprite action, 119
  - drawing, *Tic-Tac-Toe*
    - field, 250
    - stones, 250
  - drawing functions (GML), 231
- E**
- Else action, 70–71
  - emergence, 153, 161
    - Koalabr8*, 161
  - End Block action
    - Galactic Mail*, 60
    - Lazarus*, 70–71
  - End Steps, 51
  - equivalent characters (multiplayers), 214
  - equivalent features, 151
  - Esc key, disabling (*Super Rainbow Reef*), 124
  - events, 14–19. *See also individual event names*
    - inheritance, 117
  - Evil Clutches* game, 9–10
    - adding challenges/rewards, 88–92
    - adding interactivity, 92–95
  - baby dragon object, 31–32
  - background
    - image, 33–34
    - music, 34–35
  - boss object, 13–17
    - background music, 34–35
    - randomly creating baby dragons, 32
    - summoning demons, 30–31
  - demon object, 27–29
  - designing, 9–10
  - dragon object, 18–19
  - fireball object, 24–27
  - rooms, 20–22
  - running, 23
  - sound effects, 35–36
  - sprites, 10–13
  - summoning demons, 30
  - Evil Squares* game, 86
  - Execute Code actions, 225
    - Pyramid Panic*, 263
      - basic enemy, 267–268
      - beetle object, 268
      - block objects, 270–271
      - different mummy object, 279
      - explorer object, 264–265, 281–282, 286–287
      - explorer/mummy collision, 282
      - mummy object, 273
      - potion object, 283
      - treasure objects, 270
  - Execute Script actions, 225–228
    - scripts as functions, 240–242
    - Pyramid Panic*
      - controller object, 286
      - different mummy object, 278–279
      - mummy object, 277
  - exit objects (*Koalabr8*), 136
  - explorer object (*Pyramid Panic*)
    - creating object, 264–265
    - creating sprites, 263
    - extending, 281–282, 286–287
    - handling collision between explorer and scared mummy, 282
  - explosions
    - Galactic Mail*, 56–57
    - Tank War*, 195–196
    - Wingman Sam*, 178
- F**
- falling boxes (*Lazarus*). *See Lazarus* game, falling boxes
  - false keyword, 171
  - features, 149–154
    - arms races, 152
    - brainstorming, 150–151
    - emergence, 153
    - equivalent features, 151

- Koalabr8*, 160–161
- one-trick ponies, 152
- field object (*Tic-Tac-Toe*), 247–248
  - creating, 247
  - initializing, 247–248
- File menu commands
  - Advanced Mode, 10, 42
  - Exit, 8
  - New, 42
  - Open, 7
- files
  - filenames, 12, 14
  - opening, 7
- finish object (*Galactic Mail*), 61
- finish room (*Galactic Mail*), 62
- fireball object (*Evil Clutches*), 25–27
- flying planes (*Wingman Sam*), 174–176. *See also* *Wingman Sam* game
  - adding damage variable, 179–180
  - creating information panel, 180–181
  - displaying scores, 181
- flying rocket object (*Galactic Mail*)
  - adding collision events, 54–55
  - creating, 53–54
  - editing, 57–60
- fonts, 247
- for loops, 236
- forward slashes (*//*), comments, 248
- frameworks
  - Koalabr8*, 129–131
    - completion screen, 130
    - front-end, 129–130
    - game settings, 130
  - Super Rainbow Reef*, 103–107
    - completion screen, 106–107
    - front-end, 103–106
- friction, *Tank War*, 193
- full screen option, 23
- functions, 230–232
  - executing scripts as functions, 240–242

**G**

- Galactic Mail* game
  - adding Game Information, 62–63
  - asteroid object, 47–48
    - adding visual variety, 62
    - creating, 47–48
  - design, 41–42
  - explosion object, 56–57
  - finish object, 61
  - finish room, 62
  - flying rocket object
    - adding collision events, 54–55
    - creating, 53–54
    - editing, 57–60

- landed rocket object
  - creating, 51–52
    - including a Key Press event, 55–56
    - including change sprite action, 52–53
    - including keyboard events, 53
  - levels, 58–60
  - moon object, 45–47
    - adding create event, 46–47
    - adding visual variety, 62
    - creating, 45–46
    - including wrap action, 47
  - rooms, creating/populating with moons and asteroids, 48–49
  - running, 49–50
  - scoring
    - Set Score action, 57–58, 61
    - Show Highscore action, 56
  - sound, 45
  - special moon object, 50
  - sprites, 43–44
  - title screen, 60–61
- Galactic Squares* game, 86
- Game Information, 62–63
  - Galactic Mail*, 62–63
  - Super Rainbow Reef*, 106
- Game Maker, 3–8
  - Advanced mode, 42–43
  - installing, 3–5
  - interface, 6, 11
  - registering, 3, 5
  - system requirements, 4
  - website, 5, 8
- Game Maker Language. *See* GML
- Game Maker online community, 294–295
- game mechanics, 86–87
- game over message (*Lazarus*), 79–80
- gamelearning.net teacher's forum, 295
- games
  - designing, 9
  - Evil Clutches*. *See* *Evil Clutches* game
  - Evil Squares*, 86
  - frameworks. *See* frameworks
  - Galactic Mail*. *See* *Galactic Mail* game
  - Galactic Squares*, 86
  - genres, 87–88
  - Koalabr8*. *See* *Koalabr8* game
  - Lazarus Squares*, 86
  - Lazarus*. *See* *Lazarus* game
  - Pyramid Panic*. *See* *Pyramid Panic* game
  - running, 6–8, 22–23
  - saving, 22
  - Super Rainbow Reef*. *See* *Super Rainbow Reef* game
  - Tank War*. *See* *Tank War* game
  - Tic-Tac-Toe*. *See* *Tic-Tac-Toe* game
  - Wingman Sam*. *See* *Wingman Sam* game

genres, 87–88  
 GIMP (GNU Image Manipulation Program), 291–292  
 Global Game Settings  
   *Koalabr8*, 130  
   *Pyramid Panic*, 262  
   *Wingman Sam*, 188–189  
 global variables, 171, 230. *See also* variables  
 GML (Game Maker Language), 225  
   applying code to other instances (with statement), 239–240  
   arrays, 237–239  
   assignment statements, 228  
   conditional statements, 232–234  
   creating scripts, 225–226  
   debugging, 242–243  
   executing scripts, 225–228  
     scripts as functions, 240–242  
   function, 230–232  
   Hello World program, 226–228  
   keywords, 230  
   loops, 234–237  
   opening and closing brackets, 233  
   operators, 228  
     comparison operators, 232  
   random color generation, 235  
   return statement, 242  
   variables, 228–230  
     incrementing, 236  
 GNU Image Manipulation Program (GIMP), 291–292  
 goals, 89–90. *See also* challenges  
 good game design, 85–86  
 GraphicsGale, 292  
 gravity  
   dead koala object (*Koalabr8*), 138  
   Pop object (*Super Rainbow Reef*), 110  
 grids, 132–134  
 guns (*Wingman Sam*)  
   bullet objects, 176  
   enemy bullets, 184–185  
   key release events, 177  
   shooting keyboard events, 177

**H**

hazards (*Koalabr8*)  
   dead koala object, 137–138  
   detonators, 144–145  
   fixing dead koala bug, 138–139  
   locks and switches, 143–144  
   rocks, 145–147  
   saws, 139, 146  
   TNT, 138  
 health bars (*Tank War*), 197  
 Hello World program, 226–228

help, 8  
   adding Game Information, 62–63  
 hiding  
   objects, 142  
   tiles, 142  
 hit counters (*Wingman Sam* boss plane object), 186–187  
 Hogs of War, 153–154  
 HUMANBALANCE Co. GraphicsGale, 292

**I**

images  
   backgrounds (*Evil Clutches*), 33–34  
   GNU Image Manipulation Program (GIMP), 291–292  
 incrementing variables, 236  
 independent competition, 211–212  
 independent cooperation, 213  
 inheriting events, 117  
 initialize statement, loops, 236  
 installing Game Maker, 3–5  
 instance functions (GML), 231–232  
 instances, 24  
   applying code to other instances, 239–240  
 interactivity, 92–95  
   audio feedback, 95  
   choices and control, 93–94  
   interactive challenges, 87  
   unfair punishment, 94  
 interface (Game Maker), 6, 11  
 invisible blocks (*Super Rainbow Reef*), 121  
 invisible walls (*Koalabr8*), 142  
 island objects (*Wingman Sam*), 173–174

**J**

Jump Position action  
   *Galactic Mail*, 51  
   *Lazarus*, 68, 72, 74

**K**

Katch object (*Super Rainbow Reef*), 108–112  
 Key Press event  
   *Evil Clutches* dragon/fireball objects, 26–27  
   *Galactic Mail* landed rocket object, 55–56  
   *Lazarus* character, 69–71  
 key release event, 177  
 Keyboard actions (*Koalabr8*), 133–134  
 keyboard events  
   *Galactic Mail*, 53  
   *Tank War*, 194  
   *Wingman Sam*  
     movement events, 175–176  
     shooting events, 177  
 keywords  
   GML, 230  
   true/false, use with variables, 171

*Koalabr8* game, 127–128  
 controller object, 135  
   adding events to handle rescued koalas, 136–137  
   creating, 135  
   editing to fix dead koala bug, 138–139  
 design, 127–128  
 exit objects, 136  
 features, 160–161  
 framework, 129–131  
   completion screen, 130  
   front-end, 129–130  
   game settings, 130  
 hazards  
   dead koalas, 138  
   detonators, 144–145  
   fixing dead koala bug, 138–139  
   locks and switches, 143–144  
   rocks, 145–147  
   saws, 139, 146  
   TNT, 138  
 koala object  
   adding Begin Step event, 134  
   creating koala object, 132–134  
   creating koala sprite, 131–132  
   dead koalas, 138  
 levels  
   dividing levels, 162–163  
   training levels, 161–162  
 locks and switches, 143–144  
 restart button, 147  
 rooms  
   adding tiles, 141–142  
   test room, 135–136  
 saw objects, 139  
   creating, 139–140  
   making saws turn for rocks, 146  
 sound effects, 148  
 springs, 161, 163  
 tiles, 140–142  
   adding tiles to rooms, 141–142  
   creating tile set, 140–141  
   making walls invisible, 142  
 wall object  
   creating, 131  
   making walls invisible, 142

**L**

landed rocket object (*Galactic Mail*)  
 creating, 51–52  
 including a Key Press event, 55–56  
 including change sprite action, 52–53  
 including keyboard events, 53

*Lazarus* game, 65–66  
 background, 82  
 completing game, 79–80  
 design, 65–66

detecting for Lazarus being trapped/freed, 78–79  
 falling boxes, 73–78  
   creating box sprite and box objects, 74  
   creating controller object, 76–77  
   creating falling box objects, 74–75  
   creating next box object, 75  
   testing, 78  
 Game Information, 82  
 Lazarus character, 66–72  
   adding key events, 69–71  
   adding Step event, 71–72  
   creating Lazarus object, 68  
   creating sprite, 67–68  
   creating squished Lazarus object, 68–69  
   Else action, 70–71  
 levels  
   adding cheats, 82  
   creating, 82  
   reaching new levels, 79–80  
   starter object, 80  
   sound effects, 81  
   test room, 72–73, 78  
   wall object, 72–73

*Lazarus Squares* game, 86  
 learning curves, level design, 156–157, 159  
 levels  
   adaptive gameplay (*Tic-Tac-Toe* computer opponent), 256–257  
   design, 154–159  
   difficulty curves, 158–159  
   learning curves, 156–157, 159  
   dividing levels, 162–163  
   *Galactic Mail*, 58–60  
   *Koalabr8*  
     dividing levels, 162–163  
     training levels, 161–162  
   *Lazarus*. *See Lazarus* game, levels  
   *Super Rainbow Reef*, 125–126  
   training levels, 161–162  
   views. *See* views

levels of difficulty, 88–89. *See also* challenges  
 light (*Pyramid Panic*), changing room to darkness, 284  
   adding behavior of mummy object, 288  
   creating sword object (Sword of Ra), 286  
   extending explorer object, 287  
 lives (*Super Rainbow Reef*), 117–119  
 local variables, 230  
 locks (*Koalabr8*), 143–144  
 logic (*Tic-Tac-Toe* computer opponent), 251–254  
 loops  
   GML, 234–237  
   *Wingman Sam* island objects, 173–174

**M**

## mazes

*Pyramid Panic*

- adapting room, 265
- creating wall objects, 263

*Koalabr8*. See *Koalabr8*

## mechanics, 86–87

## MIDI Tracker, 293

## moments, time lines, 182–183

moon object (*Galactic Mail*), 45–47

- adding create event, 46–47
- adding visual variety, 62
- creating, 45–46
  - special moon object, 50
- including wrap action, 47

motion (*Wingman Sam*)

- looping island objects, 173–174
- scrolling background, 173

movable block objects (*Pyramid Panic*), 270–271

## Move Fixed action, 15–16, 19–20

*Evil Clutches*, 31*Lazarus*, 74

- setting speed of 0, 134

Move Free action (*Galactic Mail* moon object), 46–47movement keyboard events (*Wingman Sam* planes), 175–176

## multiplayer games

- balancing computer-controlled opponents, 220
- competition and cooperation, 211–214
- Tank War*. See *Tank War* game
- Wingman Sam*. See *Wingman Sam* game

mummies (*Pyramid Panic*)

- adapting mummy behavior to darkness, 288
- changing color, 283
- creating, 272–273
  - afraid mummy, 280
  - different mummy, 279
- giving mummy object behavior, 273–274
- handling collision between explorer and scared mummy, 282
- moving mummy toward explorer, 275–277

## music

## background

- Evil Clutches*, 34–35
- Wingman Sam*, 188

## creating, 292–293

## naming resources, 12, 14

## New command (File menu), 42

**O**

## Object Properties form, 14–19, 32

## objects, 13–20

- actions, 14–18
- adding to rooms, 21–22
- assigning sprites, 14

## creating, 14, 18

- afraid mummy (*Pyramid Panic*), 280
- asteroid (*Galactic Mail*), 47–48
- baby dragon (*Evil Clutches*), 31–32
- basic enemy (*Pyramid Panic*), 267–268
- beetle (*Pyramid Panic*), 268
- Bigleg (*Super Rainbow Reef*), 113–116
- blocks (*Pyramid Panic*), 270–271
- boss (*Evil Clutches*), 30–35
- boss plane (*Wingman Sam*), 186
- bouncing bombs (*Tank War*), 203–204
- boxes (*Lazarus*), 74–75
- bullets (*Wingman Sam*), 176, 184–185
- buttons (*Super Rainbow Reef*), 104
- congratulation object (*Super Rainbow Reef*), 106
- controller objects. See controller objects
- dead koala (*Koalabr8*), 138
- demon (*Evil Clutches*), 27–28
- detonator (*Koalabr8*), 144–145
- different mummy (*Pyramid Panic*), 279
- enemy bullets (*Wingman Sam*), 184–185
- enemy planes (*Wingman Sam*), 178, 184
- enemy planes that shoot (*Wingman Sam*), 185
- exit objects (*Koalabr8* maze), 136
- explorer (*Pyramid Panic*), 264–265
- explosion object (*Galactic Mail*), 56–57
- explosions (*Tank War*), 195–196
- field object (*Tic-Tac-Toe*), 247
- finish object (*Galactic Mail*), 61
- fireball (*Evil Clutches*), 25
- flying rocket object (*Galactic Mail*), 53–54
- Katch (*Super Rainbow Reef*), 108–109
- koalas (*Koalabr8*), 132–134
- landed rocket (*Galactic Mail*), 51–52
- Lazarus* (*Lazarus*), 68
- locks (*Koalabr8*), 143–144
- looping islands (*Wingman Sam*), 173–174
- moon (*Galactic Mail*), 45–46
- mummy (*Pyramid Panic*), 272–273
- pickup (*Tank War*), 200
- planes (*Wingman Sam*), 175
- Pop (*Super Rainbow Reef*), 109–111
- potion (*Pyramid Panic*), 283
- restart button (*Koalabr8*), 147
- rockets (*Tank War*), 202
- rocks (*Koalabr8*), 145
- saws (*Koalabr8*), 139
- scarab (*Pyramid Panic*), 280–281
- scorpion (*Pyramid Panic*), 269
- shells (*Tank War*), 198
- special moon (*Galactic Mail*), 50
- squished *Lazarus* (*Lazarus*), 68–69
- switches (*Koalabr8*), 143–144
- sword (*Pyramid Panic*), 286

- tanks (*Tank War*), 193–194
  - title (*Galactic Mail*), 60–61
  - title (*Super Rainbow Reef*), 104
  - TNT object (*Koalabr8*), 138
  - treasure (*Pyramid Panic*), 269–270
  - wall (*Koalabr8*), 131
  - wall (*Lazarus*), 72–73
  - wall (*Super Rainbow Reef*), 108
  - controller objects. *See* controller objects
  - depth (*Galactic Mail*), 47
    - asteroid, 47–48
    - explosion, 56
    - flying rocket, 53
    - landed rocket, 51
    - special moon, 50
    - title, 60
  - events, 14–19
  - hiding, 142
  - instances, 24
  - parents, 115–117
  - one-trick ponies, 152
  - online Game Maker community, 294–295
  - opening files, 7
  - operators (GML), 228–229, 232
  - opponents. *See* competition (multiplayer games); computer opponents
  - Outside room event (*Evil Clutches* baby dragon), 31
- P**
- parent objects, 115–117
    - rockets (*Tank War*), 202
    - shells (*Tank War*), 198
    - tanks (*Tank War*)
      - adding damage mechanism, 196–197
      - creating, 193–194
      - displaying secondary weapons, 201–202
      - drawing health bars, 197
      - editing to support shields, 204–205
      - recording pickups, 201
  - parentheses (), GML, 232
  - pickups (*Tank War*), 200–201
  - planes (*Wingman Sam*)
    - boss plane
      - adding hit counter, 186–187
      - creating plane object, 186
      - displaying hit counter, 187
      - making box more challenging, 187–188
    - enemy planes
      - bullets, 184–185
      - creating planes, 178, 184
      - creating planes that shoot, 185
      - explosions, 178
    - flying planes, 174–176
      - adding key release events, 177
      - adding movement keyboard events, 175–176
      - adding shooting keyboard events, 177
      - creating plane objects, 175
      - damage variable, 179–180
      - displaying scores, 181
      - information panel, 180–181
  - Play Sound action, 34, 58
  - Pop object (*Super Rainbow Reef*), 108–112, 124–125
  - potion object (*Pyramid Panic*), 283
  - Pro Motion, 292
  - programming. *See* GML
  - properties, variables, 170–172
  - Pyramid Panic* game, 259
    - basic enemy object, 267–268
    - beetle object
      - creating, 268
      - making intelligent, 268–269
    - block objects, 270–271
    - changing room to darkness, 287–288
    - completion screen, 262
    - controller object
      - creating, 262–263
      - extending, 266–286
    - design, 259–260
    - explorer
      - creating object, 264–265
      - creating sprites, 263
      - extending object, 281–282, 286–287
      - handling collision between explorer and scared mummy, 282
    - front-end, 261
    - game settings, 262
    - maze
      - adapting basic room, 265
      - adapting room, 265
      - creating wall objects, 263
    - mummy objects
      - adapting behavior to darkness, 288
      - changing color, 283
      - creating, 272–273
      - creating afraid mummy, 280
      - creating different mummy, 279
      - giving mummy behavior, 273–274
      - moving mummy toward explorer, 275
    - potion object, creating, 283
    - rooms
      - adapting basic room for maze, 265
      - adapting room for maze, 265
      - basic room, 263
      - completion room, 262
      - front-end room, 261
    - rule-based behavior, 271–272
      - creating mummy object, 272–273
      - making mummies walk, 273–274
      - moving mummy toward explorer, 275
    - scarab object, 280–281
    - scoring
      - completion screen setup, 262
      - extending score display, 282

- front-end setup, 261
    - Sword of Ra, 286–287
  - scorpion object, 269
  - states, 279
  - sword object, 286
  - treasure objects, 269–270
  - view, 266
- R**
- reactive behavior (*Pyramid Panic*), 267–269
  - registering Game Maker, 3, 5
  - Relative option, Set Variable action, 172
  - Relative property
    - Collision event, 28–29
    - Key Press event, 26–27
  - repeat loops, 234–235
  - Resource menu commands
    - Create Script, 225–226
    - Create Object, 14, 18
    - Create Room, 20
    - Create Sprite, 11, 13
  - resources
    - creating
      - images, 291–292
      - music, 292–293
      - sound effects, 294
    - filenames, 12, 14
    - rooms, 20–22. *See also* rooms
    - sprites. *See* sprites
    - time lines, 182–183
  - Resources menu commands
    - Create Object, 14, 18
    - Create Room, 20
    - Create Sprite, 11, 13
  - restart buttons, 147
  - Restart Game action, 29
  - Restart Room action, 69
  - return statement, 242
  - Reverse Vertical action, 17
  - rewards, 90–91. *See also* challenges
  - RF1 Systems MIDI Tracker, 293
  - rock object (*Koalabr8*), 145–147
  - rocket objects (*Galactic Mail*)
    - flying rocket, 53–60
    - landed rocket, 51–56
  - role-playing games (RPGs), 88
  - Room Properties form, 20–21
  - rooms, 20–22
    - adding objects, 21–22
    - backgrounds. *See* backgrounds
    - creating, 20–21
    - Galactic Mail*, 48–49
    - finish room, 62
    - title room, 61
    - Koalabr8*
      - adding tiles, 141–142
      - test room, 135–136
    - Lazarus*
      - Restart Room action, 69
      - test room, 72–73, 78
    - leaving instances outside rooms, warning message, 72
    - Pyramid Panic*
      - adapting basic room for maze, 265
      - adapting room for maze, 265
      - basic room, 263
      - completion room, 262
      - front-end room, 261
    - Speed, 51
    - Super Rainbow Reef*
      - completion room, 107
      - front-end room, 105
      - test room, 112–114
    - Tank War*, 193
      - views, 205–209
    - test rooms
      - Koalabr8*, 135–136
      - Lazarus*, 72–73, 78
      - Super Rainbow Reef*, 112–114
      - Tic-Tac-Toe*, 247
    - rotating sprites, 52
    - RPGs (role-playing games), 88
    - rule-based behavior (*Pyramid Panic*), 271–272
      - creating mummy object, 272–273
      - giving mummy object behavior, 273–274
      - moving mummy toward explorer, 275
    - Run in Debug Mode command (Run menu), 243
    - running games, 6–8, 22–23, 49–50
- S**
- saving, 22
    - providing opportunities to save, 160
    - Super Rainbow Reef*, 123–124
  - saw objects (*Koalabr8*), 139, 146
  - scarab object (*Pyramid Panic*), 280–281
  - scoring
    - Galactic Mail*
      - Set Score action, 57–59, 61
      - Show Highscore action, 56, 61
    - Pyramid Panic*
      - completion screen setup, 262
      - extending score display, 282
      - front-end setup, 261
      - Sword of Ra, 286–287
    - Set Score action, 28–29, 32, 57–61
    - Show Highscore action, 29, 56, 61
    - Tank War*
      - drawing scores, 208–209
      - recording scores in controller object, 195
    - Tic-Tac-Toe*, 248
    - Wingman Sam*, displaying scores/high-score table, 181
  - scorpion object (*Pyramid Panic*), 269

- scripts. *See also* GML
  - conditional statements, 232–234
  - debugging, 242–243
  - executing, 227–228
    - scripts as functions, 240–242
  - Pyramid Panic*
    - changing room to darkness, 286
    - making mummies walk, 274
    - moving mummies toward explorers, 275–276
  - return statement, 242
  - Tic-Tac-Toe*
    - adaptive gameplay, 256–257
    - computer opponent simple logic, 251–254
    - computer opponent simple strategy, 254–256
    - creating/executing, 248–249
    - drawing field, 250
    - drawing stones, 250
    - field click, 248–249
    - initializing field object, 247–248
    - mouse click, 249
    - writing, 225–226. *See also* GML
  - scrolling background (*Wingman Sam*), 173
  - Scrolling Shooter game style, 173
  - Set Score action, 28–29, 32
    - Super Rainbow Reef*, 113
    - Galactic Mail*, 57–61
  - Set Time Line action, 182–183
  - Set Variable action, 172
    - Wingman Sam*
      - boss plane object, 186–187
      - enemy bullet objects, 185
      - flying plane objects, 177–180
  - shell objects (*Tank War*)
    - creating parent shells, 198
    - creating players' shells, 199
    - firing shells, 199
  - shields (*Tank War*), 204–205
  - shooting (*Wingman Sam*)
    - bullet objects, 176
      - enemy bullets, 184–185
    - enemy plane objects, 185
    - key release events, 177
    - keyboard events, 177
  - Show Highscore action, 29, 56, 61
  - Simple mode, 11
  - simulator games, 87
  - Sleep action
    - Galactic Mail*, 59
    - Lazarus*, 80
  - small Bigleg object (*Super Rainbow Reef*), 115
  - Smooth Edges property, 66
  - solid blocks (*Super Rainbow Reef*), 120–121
  - Solid option (*Lazarus*), 72–74
  - sound
    - audio feedback, 95
    - explosions (*Wingman Sam*), 178
    - Galactic Mail*, 45
    - Lazarus*, 81
    - music
      - background music, 34–35
      - creating, 292–293
    - Play Sound action, 34, 58
    - Pyramid Panic*, 261
    - Super Rainbow Reef*, 123
  - sound effects
    - creating, 294
    - Evil Clutches*, 35–36
    - Koalabr8*, 148
    - Tic-Tac-Toe*, 246
  - Sound Forge Audio Studio, 294
  - Sound Properties form, 35
  - special moon object (*Galactic Mail*), 50
  - speed, 51
  - springs (*Koalabr8*), 161, 163
  - Sprite Editor, 44–45
  - Sprite Properties form, 11–12, 44
  - sprites, 10–13
    - assigning to objects, 14
    - change sprite action (*Galactic Mail*), 52–53
    - creating, 11–13
      - Galactic Mail*, 43–44
      - Koalabr8*, 131–132
      - Lazarus* character, 67–68, 74
      - Tic-Tac-Toe*, 246
    - Draw Sprite action, 119
    - Pyramid Panic* explorer, 263
    - rotating, 52
    - Smooth Edges property, 66
    - Transparent option, 12–13
    - Wingman Sam*
      - boss hit counter, 187
      - boss plane, 186
      - bullets, 176
      - enemy bullets, 184
      - enemy planes, 178, 184
      - enemy planes that shoot, 185
      - flying planes, 175
      - information panel, 180
      - island objects, 173
  - starfish game. *See Super Rainbow Reef* game
  - Start Block action
    - Galactic Mail*, 59
    - Lazarus*, 69, 71
  - statements (GML), 228
    - assignment statements, 228
    - conditional statements, 232–234
  - states (*Pyramid Panic* mummy object), 278–279
  - Steinberg Cubase SE, 293
  - Steps, 51
    - landed rocket object (*Galactic Mail*), 51
    - Lazarus* character, 71–72
  - strategy (*Tic-Tac-Toe* computer opponent), 254–256
  - strategy games, 87



- subgoals, 92
- Super Rainbow Reef* game, 101–103
  - Bigleg object, 113–116
    - moving Bigleg, 116
    - small Bigleg, 115
    - small moving Bigleg, 116
  - blocks, 120–123
    - normal blocks, 120
    - solid blocks, 120–121
    - special blocks, 121–123
  - design, 101–103
  - disabling Esc key, 124
  - framework, 103–107
    - completion screen, 106–107
    - front-end, 103–106
  - Game Information, 106
  - levels, 125–126
  - lives, 117–119
  - parent/child objects, 115–117
  - Pop and Katch objects, 108–112
    - stationary Pop, 124–125
  - possible split-screen multiplayer version, 211–212
  - rooms
    - completion room, 107
    - front-end room, 105
    - test room, 112–114
  - saving games, 123–124
  - sound effects, 123
  - wall object, 108
- switches (*Koalabr8*), 143–144
- sword object (*Pyramid Panic*), 286

**T**

- Tank War* game, 191
  - arena, 192–193
  - background sound, 193
  - controller object, 193
    - creating, 193
    - drawing scores, 208–209
    - recording scores, 195
  - damage mechanism, 196–197
  - design, 191–192
    - balancing choice, 218–219
    - balancing computer opponents, 220
    - balancing tank differences, 215–218
  - explosions
    - large explosion objects, 195–196
    - small explosion objects, 196
  - health bars, 197
  - pickups
    - creating pickup object, 200
    - recording, 201
  - room, 193. *See also* *Tank War*, views
  - scoring
    - drawing scores, 208–209
    - recording scores in controller object, 195

- secondary weapons, 199
  - bouncing bombs, 203–205
  - displaying, 201–202
  - rockets, 202–203
- shells
  - creating parent shell object, 198
  - creating players' shell objects, 199
  - firing, 199
- shields, 204–205
- tank objects, 193–194
- views, 205–209
- walls
  - creating, 192–193
  - editing wall object to make it reappear, 198
- teacher's forum at gamelearning.net, 295
- Test Instance Count action
  - flying rocket object (*Galactic Mail*), 59
  - Lazarus*, 76
- test rooms
  - Koalabr8*, 135–136
  - Lazarus*, 72–73, 78
  - Super Rainbow Reef*, 112–114
- Test Variable action, 172
  - Wingman Sam*
    - boss plane object, 187
    - enemy plane objects, 178
    - flying plane objects, 175, 179
    - looping island objects, 173
- Tic-Tac-Toe* game
  - background, 246
  - computer opponent
    - adaptive gameplay, 256–257
    - simple logic, 251–254
    - simple strategy, 254–256
  - design, 245–246
  - drawing stones, 250
  - field object, 247–248
    - creating, 247
    - drawing, 250
    - initializing, 247–248
  - field\_click script, 248–249
  - font, 247
  - mouse click script, 249
  - room, 247
  - scoring, 248
  - sound effects, 246
- tiles, *Koalabr8*, 140–142
  - adding to rooms, 141–142
  - creating tile set, 140–141
  - hiding, 142
  - making walls invisible, 142
- time lines, 182–183
- title object, *Super Rainbow Reef*
  - adding create event, 118
  - creating, 104
- title screen, *Galactic Mail*, 60–61
- TNT object (*Koalabr8*), 138

Transparent option, sprites, 12–13  
 treasure objects (*Pyramid Panic*), 269–270  
 true keyword, use with variables, 171

## U

underscore (\_) symbol  
     filenames, 12  
     GML, 229  
 unfair punishment, 94  
 user interface (Game Maker), 6, 11

## V

var keyword (GML), 230  
 variables, 63, 170–172  
     Collision event (*Tank War*), 194  
     incrementing (GML), 236  
     keyboard events (*Tank War*), 194  
     properties, 170–172  
     Set Variable action, 172  
         boss plane object (*Wingman Sam*), 186–187  
         enemy bullet objects (*Wingman Sam*), 185  
         flying plane objects (*Wingman Sam*), 177, 179–180  
     Test Variable action, 172  
         boss plane object (*Wingman Sam*), 187  
         enemy plane objects (*Wingman Sam*), 178  
         flying plane objects (*Wingman Sam*), 175, 179  
         looping island objects (*Wingman Sam*), 173  
     using in scripts, 228–230  
 views  
     *Pyramid Panic*, 266  
     *Tank War*, 205–209  
 Visible option, walls (*Koalabr8*), 142

## W

wall object  
 walls  
     *Koalabr8*  
         creating, 131  
         making walls invisible, 142  
     *Lazarus*, 72–73  
     *Pyramid Panic*, creating, 263  
     *Tank War* arena  
         creating, 192–193  
         editing wall object to make it reappear, 198  
 weapons  
     *Tank War*  
         bouncing bombs, 203–205  
         displaying secondary weapons, 201–202  
         rockets, 202–203  
         shells, 198–199  
     *Wingman Sam*, 176–179  
 website (Game Maker), 5, 8  
 weighting choices (multiplayer games), 218  
 while loops, 235–236  
 Willow Software Anvil Studio, 292–293

*Wingman Sam* game, 169–170  
     background music, 188  
     boss plane  
         adding hit counter, 186–187  
         creating plane object, 186  
         displaying hit counter, 187  
         making box more challenging, 187–188  
     bullet object, 176  
         enemy bullets, 184–185  
     controller object  
         creating, 179  
         playing background music, 188  
     design, 169–170  
     displaying scores/high-score table, 181  
     editing global game settings, 188–189  
     enemy planes  
         creating bullets, 184–185  
         creating planes, 178, 184  
         creating planes that shoot, 185  
         explosions, 178  
     flying planes, 174–176  
         adding key release events, 177  
         adding movement keyboard events, 175–176  
         adding shooting keyboard events, 177  
         creating plane objects, 175  
         damage variable, 179–180  
         information panel, 180–181  
     independent cooperation, 213  
     motion  
         looping island objects, 173–174  
         scrolling background, 173  
     single player version, 179  
     time lines, 182–183  
     with statement  
         GML, 239–240  
         *Pyramid Panic* basic enemy object, 268  
 Wrap Screen action, 45, 47  
 writing scripts, 225–226. *See also* GML

## Z

*Zelda: Wind Waker*, 85