



PART 4



Multiplayer Games

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



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.

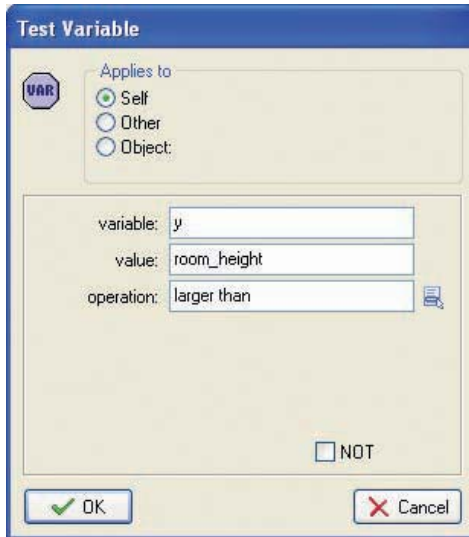


Figure 9-2. This action tests whether the island is below the visible edge of the room.

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.

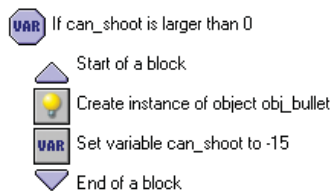




Figure 9-3. These are the actions required for shooting a bullet.

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+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+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`.

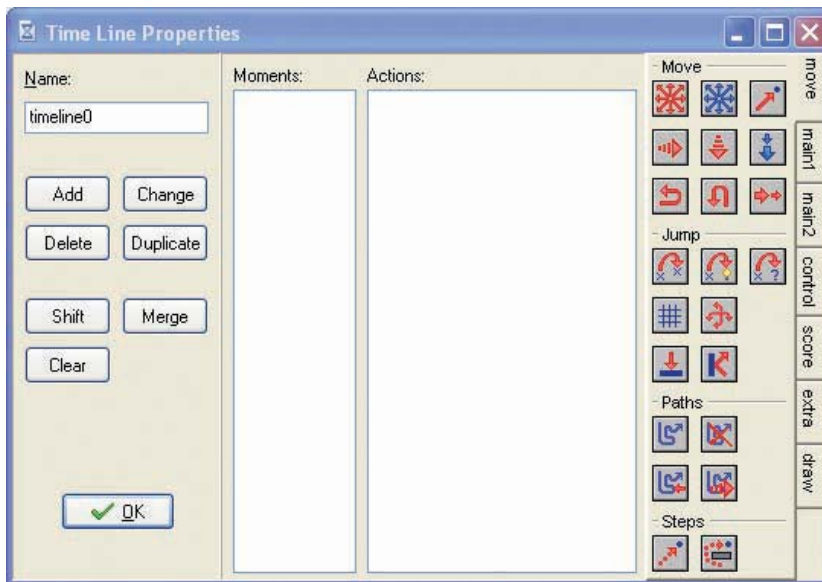





Figure 9-5. A time line has a list of moments and actions.

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.

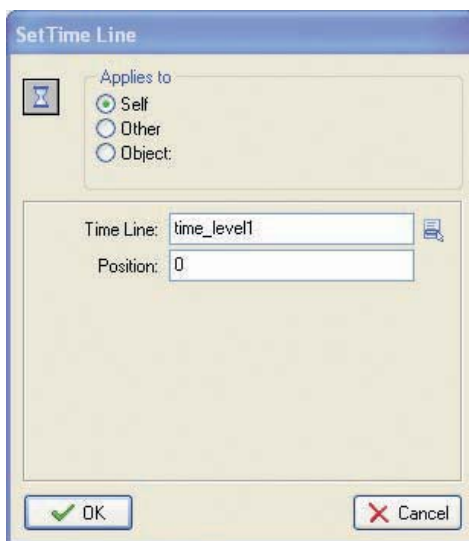


Figure 9-6. The **Set Time Line** action is used to start a time line.

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.

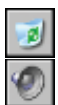
The two other enemy planes that we're going to create will shoot bullets. One will shoot bullets in a straight line, while the other will direct bullets toward the player's planes. We'll start by creating three types of bullets: one that moves downward and two others that move toward each of the player's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



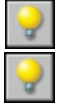
4. Include the **Create Moving** action and set **Object** to `obj_enemy_bullet`. Set **Speed** to `6` and **Direction** to `random(360)`, and enable the **Relative** option.



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!