Game Maker Tutorial

A Scrolling Shooter
Written by Mark Overmars

Scrolling shooters are a very popular type of arcade action game, and are also rather easy to create with a package like *Game Maker*. In this tutorial we will make such a game and, in the process, learn a number of aspects of *Game Maker*, in particular the use of variables. In a scrolling shooter the player controls an object, for example a plane, spaceship, or car, which moves over a scrolling background. Obstacles appear on the background that must be avoided and enemies appear that must be shot. Often bonus objects appear that can be picked up for addition benefits. During the game the number, variety, and strength of the enemies increases making it harder and harder to survive.

In this tutorial we will create a scrolling shooter called *1945*, in which the player flies in a plane over a sea and enemy planes try to destroy the player. We will treat aspects like how to create the illusion of motion using a scrolling background, how to control the plane, how to make enemies and bullets, and how to deal with the score, with lives and with the damage of the plane. But first of all we delve into a very important aspect of *Game Maker* that extends the possibilities considerably: the use of variables.

Variables and properties

Before we will actually create the scrolling shooter game we need to delve a bit into an important concept of *Game Maker*: the use of variables. This simple concept will provide a very powerful mechanism to enable much more interesting game play. So what is a variable? It can best be seen as a property of an instance of an object. As you should know, there are certain properties we can indicate when we define an object. For example we can set whether it is visible or whether it is solid. There are also a number of actions that change certain properties. For example there is an action to change the position or the speed of the instance. Each instance has a number of such properties and there are also a number of global properties, like the score, that are not related to individual instances. All properties are stored in so-called variables that have a name. Here are some of the properties/variables that each instance has:

> `x` the x-coordinate of the instance
> `y` the y-coordinate of the instance
> `hspeed` the horizontal speed (in pixels per step)
> `vspeed` the vertical speed (in pixels per step)

direction the current direction of motion in degrees (0-360; 0 is horizontally to the right)

speed the current speed in this direction

visible whether the object is visible (1) or invisible (0)

solid whether the object is solid (1) or not solid (0)

And here are some global variables:

score the current value of the score

lives the current number of lives

mouse_x x-position of the mouse

mouse_y y-position of the mouse

room_speed current speed of the room (in steps per second)

room_caption caption shown in the window title

room_width width of the room in pixels

room_height height of the room in pixels

There are many, many more variables, both local for the instances, and global. All of them can be found in the *Game Maker* documentation. There are actions which manipulate the value of certain variables but, as we will see, you can also manipulate them directly. What is better, you can define your own variables and use these as well. For example, as we will see below, we want our spaceship to be able to shoot only once every five steps in the game. So our space ship needs a property to indicate whether it can shoot. We use a variable for this property that we call **can_shoot**. (A variable name can only consist of letters and the underscore symbol. Variable name are case-sensitive, so **Can_Shoot** is not the same variable as **can_shoot**.) In the creation event of the spaceship we set this variable to 1 (we always use 1 to indicate true). When the player wants to fire a shot we check the value of the variable to see whether this is allowed. Whenever a shot is fired we set the property to 0 (indicating that shooting is temporarily not allowed). Next we use an alarm event to set the property back to 1 after five steps. Below we will describe this in more detail. In a similar way we can use variables to indicate whether the spaceship has an active shield, whether it has some special weapon upgrade, etc.

There are two important actions to deal with variables directly, which can be found in the **Control** tab:

UAR **Set the value of a variable**

With this action you can change the value of a given variable. This can either be one of the built-in variables or your own variable. You specify the name of the variable and the new value. When you check the **Relative** box the given value is added to the current value of the variable. Please note that this can only be done if the variable already has a value assigned to it! Rather than providing a simple value for the variable you can also give an expression. For example, to double the score you could set the value of the variable **score** to value **2*score**.

If a variable has a value

With this action you can check what the value of a particular variable is. If the value of the variable is equal to the number given, the question returns true and the next action or block of actions is performed. If it is not equal, the next action or block of actions is not performed. You can also indicate that the check should be whether the variable value is smaller than the given value or larger than the given value. Actually, you are not restricted to using variables but can check any expression you like.

Below we will see a number of examples of the use of these actions.

There is one more thing you need to know about variables. As I indicated above there are local variables that "belong" to an instance and there are global variables. When you use your own variables these are always local variables that only exist for the instance in whose actions you use them. If you want to use your own global variables you have to prefix their name with the word `global` and a dot. So, for example, you can use a variable `global.bonus` to indicate the number of bonus points gathered by the player. Always be careful that you use variable names that do not yet exist and are also different from the names of sprites, sounds, etc. One way to achieve this is to always let your variables start with e.g. `var_`.

1945

Let us now look at the game we want to create. Before creating a game we need to write a design document. As the game *1945* that we are going to develop is rather complicated a full design document would typically consist of a couple of pages. To avoid making this tutorial too long we will just give a short description here. The game will look something like this:

1945 *design document*

Description
In this game you control a plane flying over a sea. You encounter an increasing number of enemy planes that try to destroy you. You should avoid these or shoot them. The goal is to stay alive as long as you can and to destroy as many enemy planes as you can.

Game objects
The background is formed by a scrolling sea with some islands. The player's plane flies over this sea. You can shoot bullets that destroy enemy planes. There are four types of enemy planes: a plane that you encounter and should be destroyed, a plane that fires bullets downwards, a plane that fires bullets towards the player's plane, and a fast enemy plane that comes from behind rather than from the front.

Sounds
There are some explosion sounds and there is some background music.

Controls
The player controls the game with the arrow keys. With the space key you fire a bullet. Only one bullet can be fired every five steps.

Game flow
The player immediately jumps into the game. The player has three lives. When all lives are gone a high-score table is shown. Pressing the <F1> (help) key will give a brief explanation. Pressing the <Esc> key will end the game.

Levels
There is just one level, but more and more enemy planes will arrive: first only the easy type but later the more difficult types.

The player controls the big yellow plane that flies upwards. All four types of enemy planes are shown in the image above. At the bottom the score, number of lives left, and the damage (in the form of the green bar) are shown.
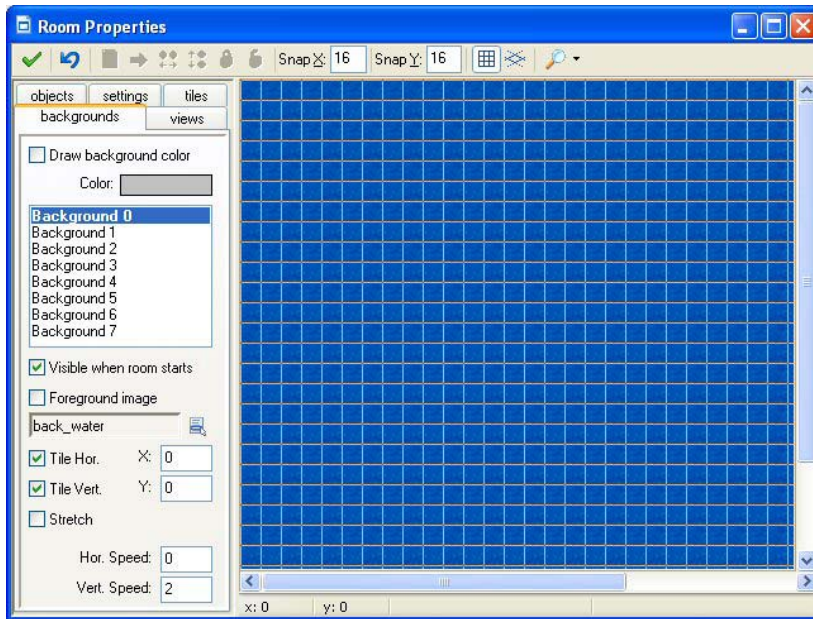
The illusion of motion

A scrolling shooter thanks its name to the fact that the game world scrolls over the screen, normally either from top to bottom or from right to left. This gives the illusion of motion. In the game *1945* the game world scrolls vertically. Even though the plane the player controls actually stands still on the screen you get the impression as if it flies over the scrolling background. You can control the position of the plane by moving it around on the screen. This gives the impression of the plane speeding up, when it moves forwards, or slowing down when it moves backwards. It is crucial that the plane does not move faster backwards than the background scrolls. This would give the illusion that the plane flies backwards, which is of course impossible.

So how do we create a scrolling background in *Game Maker*? There are actually two possibilities. The first and easiest possibility is to use a tiling background image that moves downwards through the room. The second, more complicated, way is to build a much larger room but only show part of the room using a so-called view. This view slowly moves upwards over the room. We will start with using a moving background. Later we will briefly investigate the second option.

As our game takes place over the see we need a background image that looks like a see viewed from above. We add the following small image as a background resource to the game and give it the name `back_water`:

Filling the background with it will give a nicely looking sea. To create a room with a moving background, add a room to the game in the familiar way. At the left click on the tab labeled **backgrounds**. We need to change three settings here. First of all, because we are going to fill the whole room with the background image we do not need to draw the background in a color, so uncheck the box labeled **Draw background color**. Second, in the middle click on the menu icon and select the `back_water` background image. The default setting is to tile the whole room with it so this is what we want. Finally, we need to make the background move. To this end, at the bottom, set the **Vert. speed** to 2. The form should now look as follows:

Run the game to verify that we indeed have a scrolling background that gives the illusion of motion.

To enhance the feeling of motion we are going to add a few islands to the see. An easy way would be to create a larger background image and add the island to this background. The disadvantage of this approach is that the islands will appear in a regular pattern, which the player soon notices. (You might have seen this in cartoons where there is a repeating scrolling background behind a running character.) So we choose a slightly more complicated approach and add the island as objects. So we first create three sprites with the following images (with transparent background, as is the default):

As we will never use them for collision checking you better uncheck the box **Precise collision checking**. For each of the islands we create an object. In the creation event we give the object a vertical speed that is the same as the scrolling speed of the background. In this way it looks as if the islands are part of the background because they stay at the same position with respect to the scrolling sea. To make sure that all other objects will stay above the islands we give the island objects a **Depth** of 10000. (You must run *Game Maker* in advanced mode to be able to do this!) Instances of objects are drawn in the order of the depth. The instances with highest depth are drawn first. Instances with lower depth are drawn on top of them. So by giving the islands a high depth they will always be drawn first and lie below the other objects.

One more thing needs to be done. When the island disappears below the bottom of the room we want to make it reappear at the top. To this end, in the **Step** event of the island

6

we will test whether the island disappeared below the screen and, if so, let it reappear at the top. Note that the variable **y** indicates the vertical position of the instance. (A value of 0 corresponds to the top of the room!) The variable **room_height** indicates the height of the room. So the island disappears below the bottom of the room when **y** is larger than **room_height**. So we can use the action to test variable values to see whether the island lies below the room:
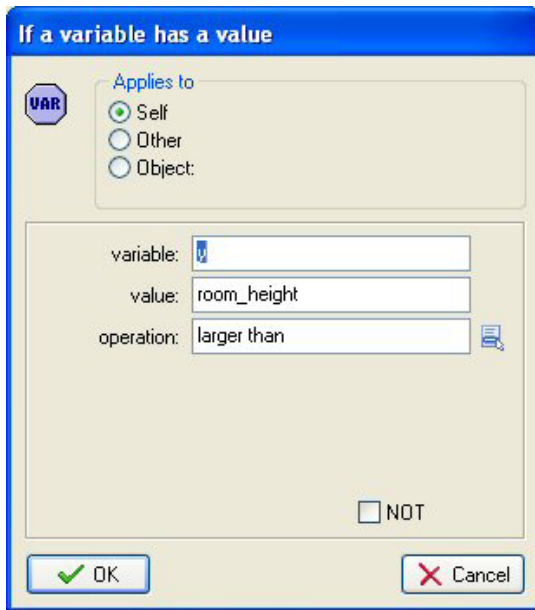
As you see, as value you can actually use another variable name. You can actually type full expressions here.

To move to the top of the room we use the action to jump to a position. But we like to jump to a random position above the room, not a particular one. By picking a random position there is less regularity in the places were the islands appear. So the player does not have the feeling that it is the same island reappearing. But how do we give a random value? For this there is a function **random()**. What is a function you might ask? A function computes a value (or performs an action) based on the value of certain arguments. The argument(s) are written between the brackets after the function name. Everywhere where you can type in values you can also use functions and variables (and expressions involving these). There are many functions in *Game Maker*. But for now we only need the function **random()**. In the jump action we use **random(room_width)** as the x-coordinate. This will create a random value between 0 and the width of the room, as we like. So the jump action will look as follows:

7
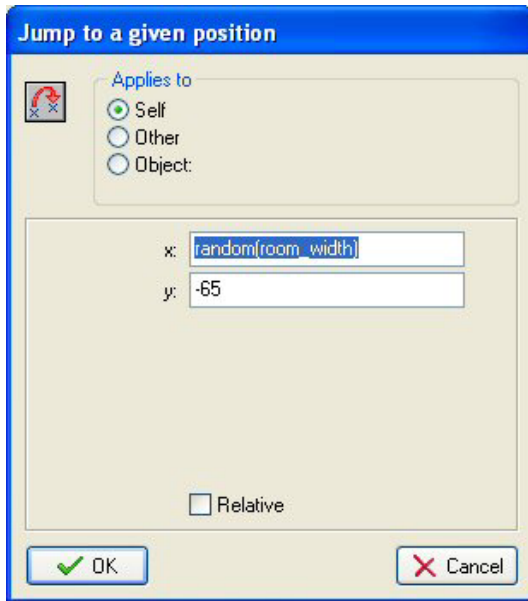
We use –65 for the y-position to make sure the island starts completely above the room. Because of its vertical speed it will move again in sight.

We have to this for all three islands objects. All that remains to be done is to place three islands at different heights in the room and we are done with our scrolling background. Even though the islands reappear at regular moments, because the position is different, the player will not really notice that they are the same. You could also have added some irregularity in the moment the islands appear by also setting a (negative) random value for the y-coordinate.

The main plane

Now that the scrolling background is ready it is time to make the plane that the player will control. This is actually rather easy. First of all we need a sprite for the plane. We will use a propeller plane. To give the propellers the illusion of rotating we use a sprite consisting of three images that are exactly the same except for the propellers:

When creating this sprite we do one important thing. We set the **X** and **Y** of the **Origin** to 32. This means that the origin of the sprite is the centre of the plane. So whenever we later set the plane at a position or test the position, we mean the position of the centre of the plane, not the top-left point, which is normally used as the origin. This is important to make sure that for example bullets come from the centre of the plane, not from the left side.

Next we add the object `obj_myplane`. As sprite we choose the plane sprite we just created. We give it a depth of –100 to make sure that it lies above bullets, etc. that we will create later. (Dealing with the depth property in an effective way is very important for many games. So you better understand how to use this.)

For the moment we only need to specify the motion of the plane. When the player does nothing, the plane will not move. (Remember, the background moves, not the plane.) If the player presses one of the four arrow keys the plane should move in the appropriate direction. Our main concern is to avoid that the plane moves outside the room. To this end we will control the motion ourselves, rather than give the plane a speed. This is done as follows.

Let us look at the motion for the left arrow key. Add the keyboard event for the <Left> key. We first need to check that we are not too far to the left. So we use the action to check whether the variable `x` is larger than 40, similar to the way we checked the y-coordinate of the islands above. If this check returns true we want to move the plane a small distance to the left relative to its current position. For this we use the "jump to a position" action to move relative –4 for x and 0 for y. Don't forget to check the box labeled **Relative**. For the right arrow we do a similar thing. We check whether `x` is smaller than `room_width`-40 and, if so, jump relative to an x of 4 and y of 0. Similar for the vertical motion, but this time we move only with a speed of –2 and 2. (Remember, we should never move faster backwards than the background is scrolling.) Also, don't forget that the positive y-direction is downwards. So to move upwards we must subtract 2 from the y-position. At the bottom of the screen we keep a larger margin of 120. This will be used later to place a panel with some game information.

Our plane can fly now. Place one instance of the plane in the room and run the game. You should get the illusion of flying above the sea. If you don't want to make it yourself you can find the game we made so far in the file `1945_1.gmk`.

Enemy and weapon

But what is a scrolling shooter if you cannot shoot and there are no enemies? We will now extend our game with some enemy planes and a canon on the plane that you can fire. You can find this version in the file `1945_2.gmk`.

Let us start with the canon. We need a bullet sprite. We put its origin in the centre as we did for the main plane. (You can also achieve this by clicking with the mouse at the appropriate spot in the image at the right of the sprite form.) To make it a bit spectacular we use a rather large bullet. Exaggerating things is often important in games.

We create an object with this sprite. We will give it the default depth 0 such that it will appear below the plane but above the islands. The object has rather simple behavior. In the creation event we give it a vertical speed of –8 to make it move upwards. To avoid that more and more bullets keep flying around we must destroy it once it leaves the room. This can easily be achieved. In the step event we test whether the variable **x** is smaller than –16. You should by now know how to do this. If so we destroy the object using the appropriate action. (You could also use the **Outside room** event that can be found in the other events.)

The bullet should be fired when the user presses the space key. As in most shooters the plane should keep on firing bullets as long as the key is pressed. But we don't want too many bullets at the same moment. This would make the game too easy. We only allow the player to fire two bullets every second, that is, one bullet every 15 steps. To achieve this we use a variable **can_shoot** as we mentioned before. In the creation event of the main plane we set this variable to 1, indicating that we can indeed shoot a bullet. In the space key event we test whether the variable **can_shoot** is equal to 1. If so we create a bullet just in front of the ship, relative at position (0,-16). Also we set the variable **can_shoot** to 0 indicating that we can no longer shoot and set alarm0 to 15. Remember that an alarm ticks down to 0 with one tick in each step. When the alarm reaches 0 the alarm event happens. In this event we set the variable **can_shoot** back to 1, indicating that we can shoot again. So the event for the space key will look something like this:
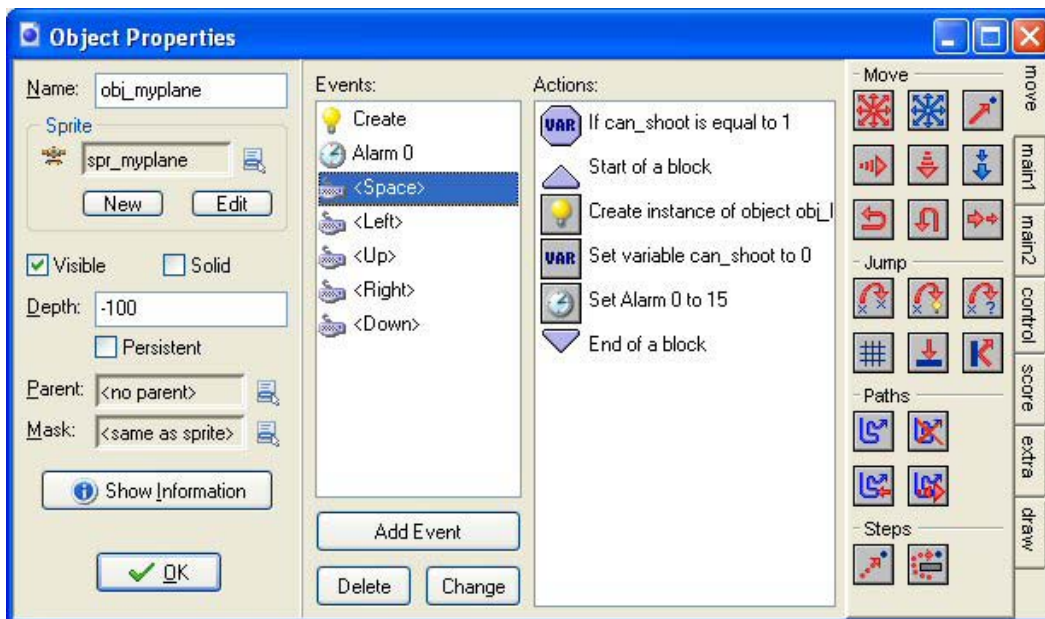

You can change the speed with which you can shoot by changing the value of the alarm clock. (In some games you can shoot faster if you are willing to constantly press the space key. This can be achieved by using a similar set of action in the space key pressed event but this time without the check for the variable.)

Let us now make the first enemy. It is a small plane that simply flies downwards. It does not shoot but when it hits the main plane the game is lost. Again we create a sprite for the enemy plane and an object. In the creation event we set the vertical speed to 4 to make the plane fly downwards. When the plane reaches the bottom of the room we make it reappear at a random place at the top of the room, in exactly the same way as we did for the islands. You should know how to do this by now.

We need to define two important collision events for the enemy plane: the collision event with the bullet, which should destroy the enemy plane, and the collision event with the main plane which should destroy the main plane and end the game.

We start with the collision event with the bullet. A number of actions are required here. But first of all we need a sound for a small explosion and a sprite that indicates the explosion. To create a sound, press **Add sound** and load some nice explosion sound. For the explosion we need a little sprite. Like for the enemy plane we set the origin to (16,16). We also create an explosion object and give it the explosion sprite. It does nothing except that once the complete explosion animation has played it must destroy itself. There is an event called **Animation end** for this under the other events.

When the sound and the explosion object are ready we can fill in the collision event for the enemy plane with the bullet. The following actions are required. First of all we play the explosion sound. Next we destroy the bullet. For this use the action to destroy an instance but, at the top, indicate that it should apply to the **Other** instance, which is the bullet in this case. Next we create the explosion object, relative at position (0,0), that is, at the same place as the enemy plane. We do not destroy the enemy plane! Instead we move it to a random position above the room such that it looks as if a new enemy plane is coming. Finally, we set the score relative to 5. We need relative here because we want to add 5 points to the score; we do not want to set the score to 5. So the event should look something like:
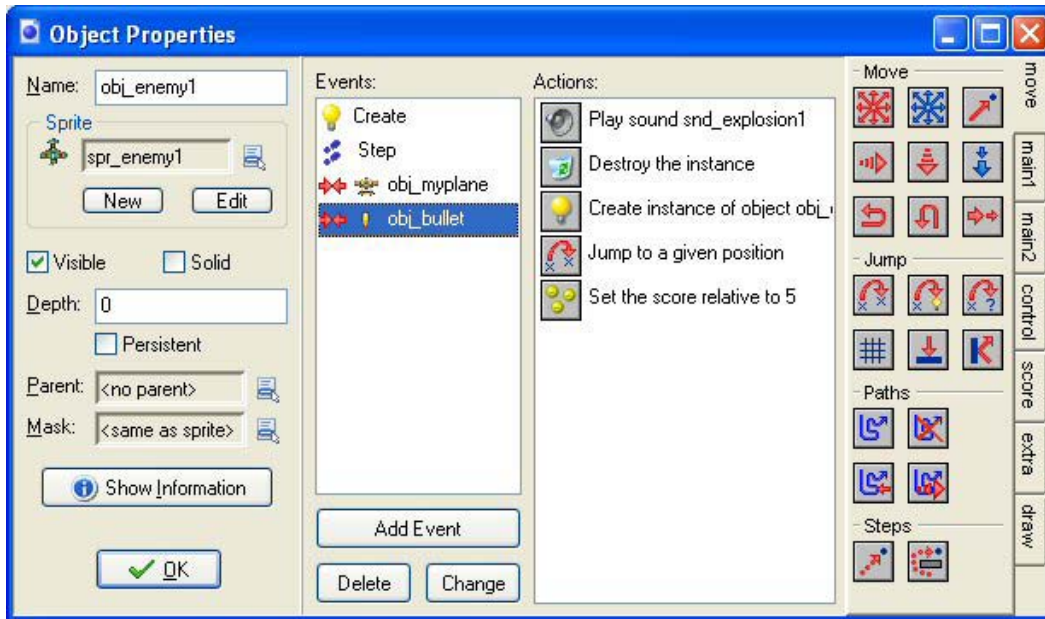
Next we need to handle the collision with the main plane. Again we will need an explosion sprite (a bit bigger this time) and an exploding sound (a bit louder this time). We add the sound to the game and we make a sprite for the explosion. Again we create an explosion object for it, but this object is a bit more elaborate than the other explosion object because it will also handle the ending of the game. In its **Animation end** event we do a few things. We first destroy the instance to make it invisible. Next we sleep a while to make sure that the exploding sound is finished. The next action shows the high-score list, such that the player can type his or her name when reaching a high score. This goes all automatic in *Game Maker*. You can though indicate what the appearance should be of the high-score table. You can have a nice background image, choose the font, etc. You should experiment a bit with this. Finally we restart the game. So the event looks like this:
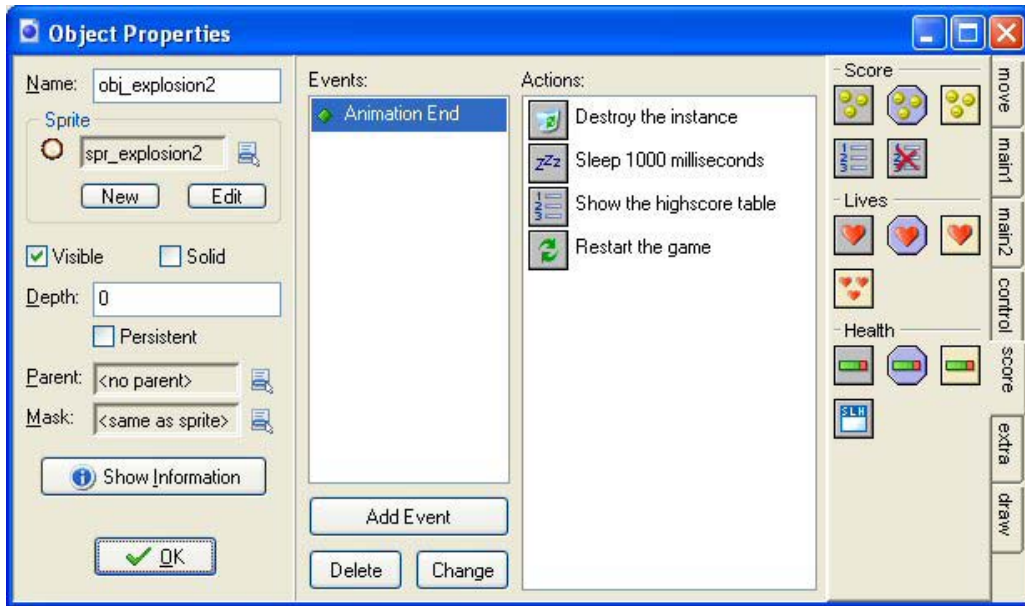
In the collision event of the enemy plane and the main plane, we change the main plane, that is, the other instance, into an explosion. Also we play the exploding sound and we destroy the enemy plane.

What remains is to place an enemy plane in the room but we will do this slightly different. A good game becomes more difficult all the time. So we like to start with just one enemy plane and get more of them over time. To this end we create one more object, which we will call **controller_enemy**. It will control the creation of enemy planes. We make the object invisible during the game by unchecking the box labeled **Visible**. A sprite is not required for it. In its creation event we create an enemy place at a random location just above the room. This will be the first enemy. We also set the alarm clock to 200. In the event for this alarm clock we create another enemy plane and set the alarm clock again, but this time to 500. That is all. The effect is that at the beginning of the game there is one enemy plane. After 200 steps, that is, about seven seconds, a second enemy plane appears. After about 15 seconds, a third plane appears, etc. (The reason that we let the second plane appears faster than the other ones is because the game with just one enemy plane is too boring.) Place one instance of the controller object in the room and we are done.

This finishes the second version of *1945*. We have a playable game with an enemy.

Scores, lives, and damage

It is a bit unsatisfactory that the game ends whenever you are hit. To make the game a bit more interesting we will let enemies create damage. Only when the plane gets too much damage will it be destroyed. Also we will introduce multiple lives and create a nice information panel that shows this information, together with the score. Fortunately, this will all be very easy because *Game Maker* has built-in mechanisms to deal with score,

13

lives, and health (being the opposite of damage). The new game can be found in the file `1945_3.gmk`.

To create all this we make a new object called the **controller_life**. It does not need a sprite as we will control the drawing itself using the drawing event. As you know, normally in each step the sprite that is set for the instance is drawn at the correct location in the room. But when you put actions in the drawing event this is no longer the case. Now these actions determine what is being drawn. There is a whole collection of actions available just for drawing. Most of them can be found in the **Draw** tab. But you can also use other actions here. The drawing actions only make sense in the drawing event. At other places they are basically ignored.

As a start we create one big sprite that functions as information panel. It looks like this:


It will show the score, the damage (in the black area at the left), and the number of planes left, that is the number of lives. In the drawing event of the **controller_life** we draw this information panel sprite at the correct position using the action to draw a sprite. We fill the parameters in as follows:
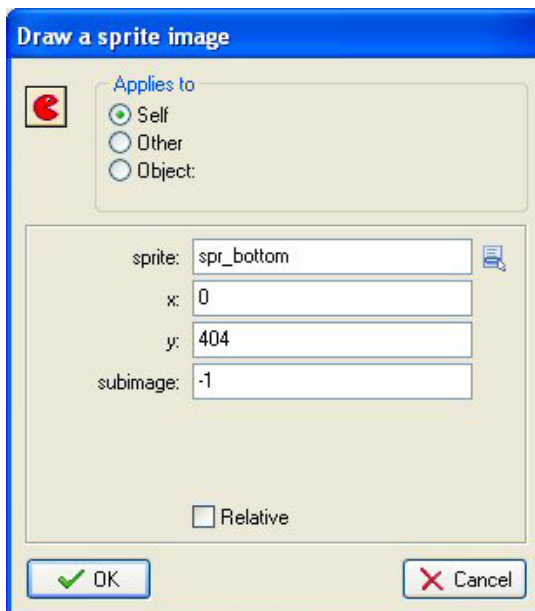

This will place the correct sprite at the bottom of the screen. (Using –1 for the subimage means that the current subimage is drawn. As there is just one subimage in the sprite we don't really care about this, but if a sprite consists of multiple subimages you can indicate

here the subimage you want to see.) To make sure that the bottom panel lies above everything else we give the `controller_life` object a depth of −10000.

In its creation event, the `controller_life` object sets the score to 0, the number of lives to 3, and the health to 100. There are actions for this in the **Score** tab. To draw the score we use the appropriate action in the **Score** tab. (We first set drawing color to be yellow.) In the action to draw the score, we fill in the parameters as follows (no caption because the caption is already on the background):
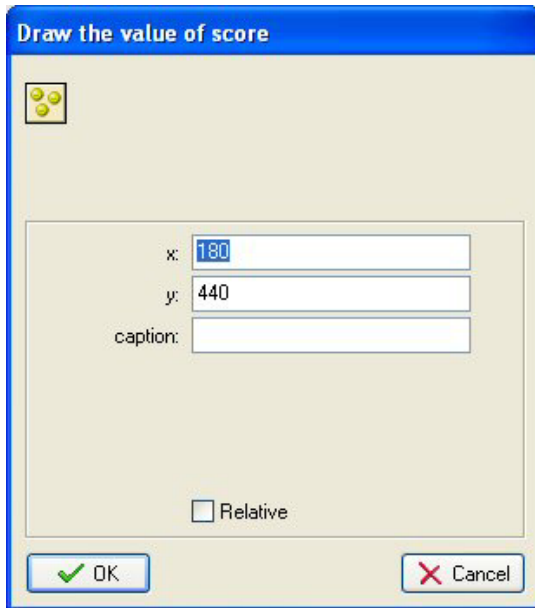

For drawing the lives we use a different mechanism. Rather than just drawing the number we will draw a number of little images of the plane. For this we will use a small sprite depicting the plane. There is an action for this in the **Score** tab.

15
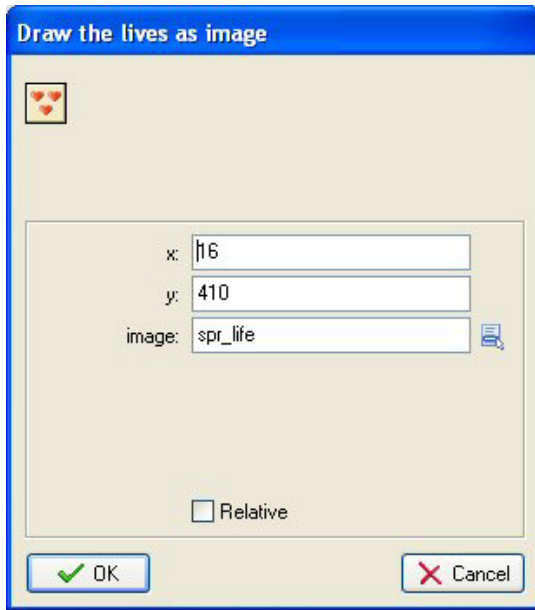
As we do now show the score and number of lives ourselves we no longer want them displayed in the window caption. There is an action in the **Score** tab to indicate what must be shown in the window caption. Put this in the creation event of the object and indicate that nothing should be shown.

Also for drawing the health there is a special action. The health is displayed in the form of a health bar. You can indicate the position, size, and color scheme used. We fill in the parameters as follows:
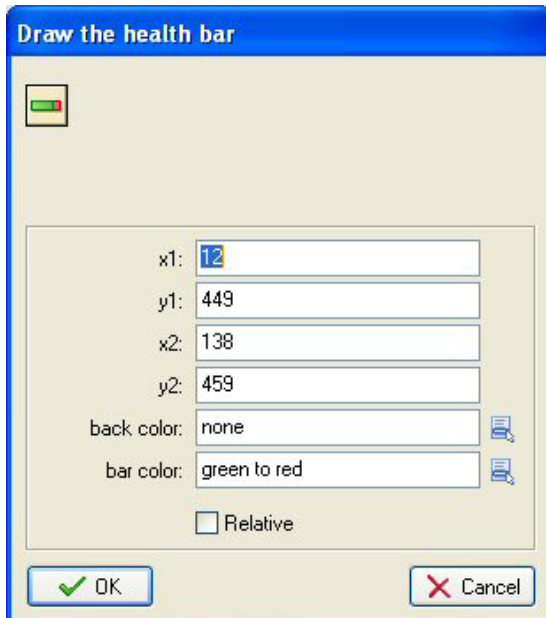

The total drawing event looks something like this:
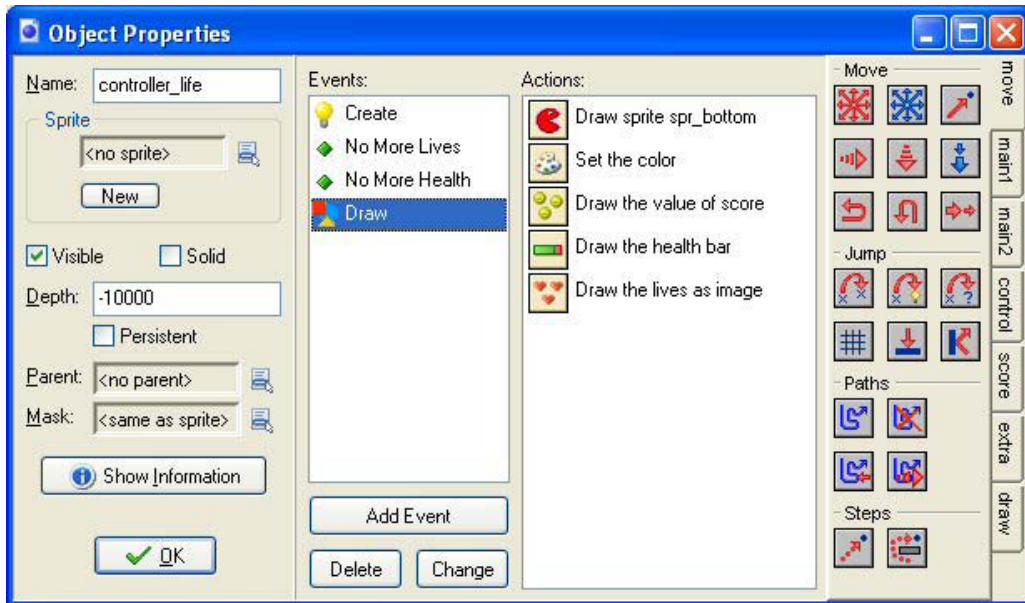
But we still need to actually check for the health and lives. First of all we have to make some changes in the collision event of the enemy plane and the main plane. It should no longer destroy the main plane, but only destroy itself (turning it into an explosion) and decrease the health, that is, set the health relative to -30 (so it can withstand 3 hits). The **controller_life** object will check when the health becomes smaller than 0. There is an event for this under the other events. In this event we blow up the main plane by turning it into a big explosion. We also reset the health and play the correct sound. So the event looks like this:
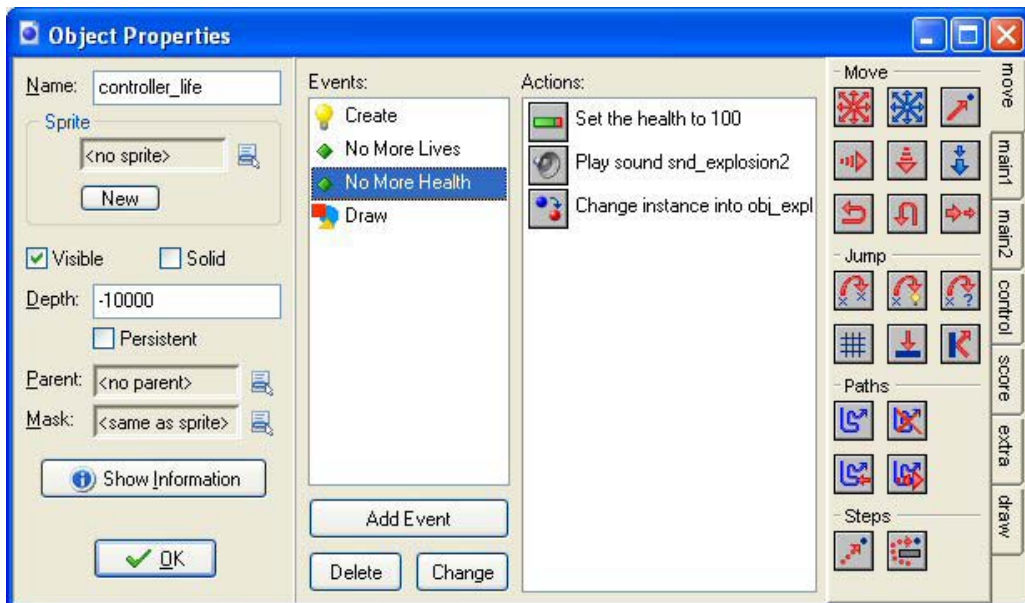
The big explosion object, in its **Animation End** event destroys itself. Sleeps for a while, to let the sound finish, creates a new main plane at the current position (that is, relative at position (0,0)) and reduces the number of lives, that is, set it relative to –1.

Finally, we need to check whether we did run out of lives. Fortunately, there is again an event for this. For the `controller_life` object, in this event we show the high-score list and restart the game. This finished the third version of the game. You can now play it and it already looks pretty good. But the game soon becomes boring. So we have to enhance it a bit more and add some variation.

More enemies

In this section we are going to add three new types of enemy planes to our game. One will fire bullets straight down. The second plane will fire bullets towards the main plane. The third type will not fire bullets but comes from the bottom of the room and is more difficult to avoid or shoot. We will make them appear at later stages during the game. The new version can be found in the file `1945_4.gmk`.

To create the first new type of enemy plane we must make a new sprite for it, similar to the sprite for the first enemy but with a different color. Secondly we need a new object for it. Because the object will behave almost the same as the first enemy plane we make a copy of this object (right-click on it and choose **Duplicate**). Double-click on the new duplicate object to change it. Give it a new name and set the correct sprite. Because this is a more special plane the player should get a higher score when shooting it. So in the collision event with the bullet we change the score to 10.

For the enemy plane to shoot, we need a bullet sprite and a bullet object. This object, in its creation event, gets a vertical speed downwards. In its step event we again take care that the object is destroyed when it moves out of room at the bottom. In the collision event of this bullet with the main plane we set the health relative to -5, destroy the bullet, and play a sound.

We now must make the enemy plane shoot bullets from time to time. We do this in the step event of the plane. We use the action to throw a dice and as parameter we put 30. This means that the next action is performed on average once every 30 steps. In this next action we create the enemy bullet.

Finally we must make sure that at some stage the second enemy plane starts appearing. For this we use the `controller_enemy` object again. In the creation event we set alarm 1 to a value of 1000. In this alarm event we create the second enemy plane and set the alarm 1 again to 500 to create another plane a bit later. So the first plane of this type will appear after about 30 seconds and an extra one appears about every 15 seconds.

For our next type of enemy plane we again need a new sprite and we need a new bullet sprite. We make a copy of the second enemy plane object and, as above, in the collision

event with the normal bullet we change the score to increase it to 20. We also create a second enemy bullet object. The enemy plane creates this new type of bullet in its step event. Again we use the dice action but now we only create a bullet once every 80 steps, because our new type of bullet is a lot harder to avoid.

The new type of bullet works as follows. In its creation event we use the action to move towards a position. But what position should we use? Well, we want to shoot towards the position of the main plane. So we want to know the x and y position of this object instance. This can be achieved easily in *Game Maker*. To get the value of a variable in another instance we precede the variable name with the name of the object. So we use **obj_myplane.x** to indicate the value of the x coordinate of the plane. When there are multiple instances of this object we get the value of the variable in the first instance. When there is no instance of the object we will get an error message. This can be a problem for our game because when the plane is destroyed there temporarily is no plane. So we better first check whether the plane is there. There is an action that can count the number of instances of a particular object. We use this to check whether the main plane is available and, if so, direct the bullet towards the plane. Otherwise the bullet will go downwards. So the creation event will look as follows:
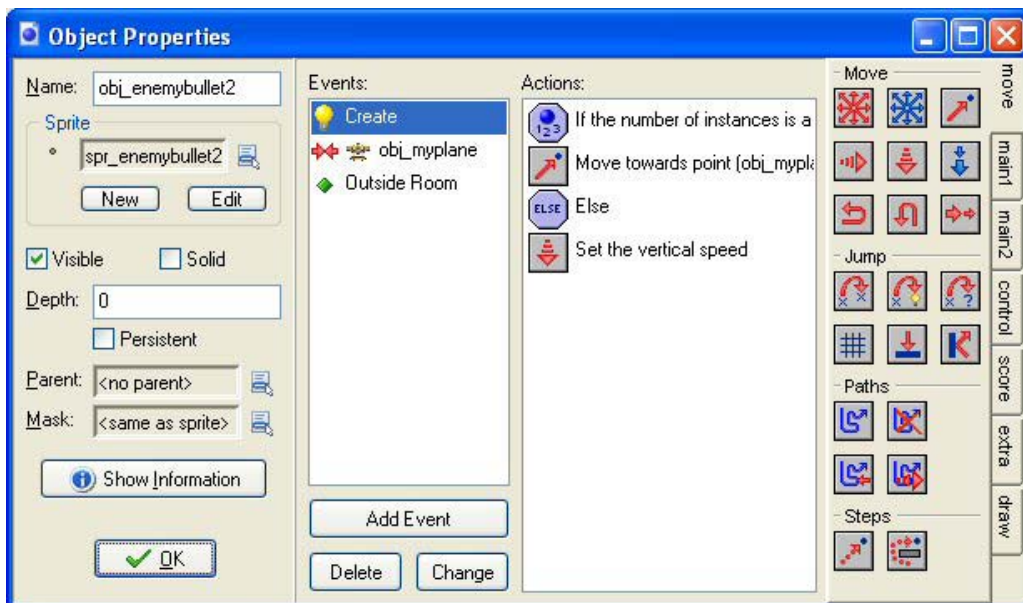
The action to move towards a point now has the following parameters:
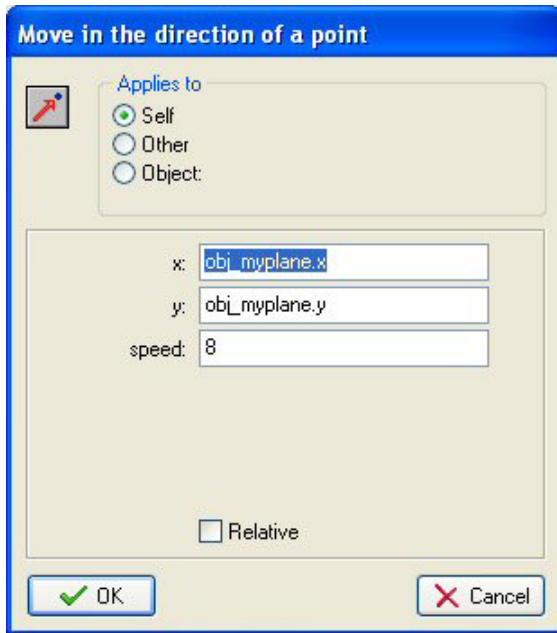
One other change is required. As the bullet can fly in any direction it is a bit more difficult to test in the step event whether the bullet lies outside the room. But actually, there is a special event for this: the **Outside** event. In this event we simply put an action to destroy the object.

Finally we have to handle the creation of the new type of enemy planes. As indicated before we use the `controller_enemy` object for this. In the creation event we set alarm 2 to 2000. In this alarm event we create the new type of enemy plane and set the alarm 2 again to 1000 to create another one some time later.

This finishes the adding of our third type of plane. It remains to add the planes that come from below. This is done in exactly the same way as the first enemy plane, except that the plane starts below the room and moves upwards rather than downwards. The `controller_enemy` object again creates them, this time using alarm 3. You should by now understand how to do this, but you can also look in the game.

This finishes our fourth version of *1945*. It has now become a playable game that gets harder all the time. There actually is some fun in playing it and trying to achieve the highest score.

Finishing the game

We have now created a scrolling shooter with different enemy planes. It has become a playable game that gets harder all the time. There actually is some fun in playing it and trying to achieve the highest score. But to turn it into a real game a couple of finishing touches are still required. We need some background music, a loading image, a better icon, etc. The final version of the game can be found in the file `1945_5.gmk`. I also put a

delay in planes to reappear when shot, making it more interesting for the player to try and shoot planes, rather than simply avoid them. Finally there is a special fire bonus when the player reaches a score of 400 and another one when he reaches 1000 points.

You can use this game as a basis to extend further. Here are some ideas of what you could add. It is rather common in scrolling shooter games to find power-ups that give the player extra fire power, repair the damage, give extra points, give an extra plane, etc. Such power-ups can appear when the player shoots a plane or they might simply be encountered during the flying. Also you could create extra types of enemy planes, for example, planes that fly from the side or fire more powerful missiles. I leave it to you to work on these and make your own scrolling shooter. In the folder 1945_sprites you can find a large collection of sprites, created by Ari Feldman, that are particularly suited for this game. You can use them to enhance your game.

Using time lines

In this section and the next we will discuss two further features of *Game Maker* that are very useful when creating more complicated scrolling shooters: time lines and views.

In the game we created so far, the appearing of enemies was controlled by the `controller_enemy` object which from time to time added an enemy plane to the game. Also, enemy planes where never destroyed. They just reappeared. The result was a game that got increasingly more difficult because the number of planes got larger and larger. Even though this gives reasonable game play it has a number of problems. First of all, the planes appear at random positions that the game designer cannot control. As a result planes might end up on top of each other or very close to the boundary which is not very appealing. Also, there is little surprise in the game. Enemies just come at you in random ways and there is little else the player can do than just shoot all the time and avoid them.

To get more interesting game play the game designer should have more control over which planes appear where at what moment. This allows you to create enemy planes that come in formations, to create certain challenges (like a massive group of simple enemies that should primarily be avoided or a few dangerous enemies that better be shot) and variation. Also it will enable the player to learn how to play the game because he might remember useful things from previous plays. All good games control the enemies rather precisely for the reasons described above. Simply put, it results in more interesting game play.

So we want to better control the appearance of certain enemies. To this end we will use the time line resources. But first of all we have to make some small changes to the game. For all enemy planes we no longer make them reappear when the got shot or leave the screen. We simply destroy them. So once a plane is gone it is gone forever. The time line will control the creation of the enemies. (You can find the game in the file `1945_6.gmk`. It is not a full game. Only the start is shown. You have to extend it a lot to turn it into an interesting game.)

A time line resource globally works as follows. In it you specify a number of moments of time (measured in steps of the game) and for each such moment you can indicate actions that must be executed at that particular moment. Because we will use the time line for the enemy planes we will only use actions to create enemy planes but in general you can use any actions you like.

So let us create our time line. Choose **Add time line** from the **Add** menu. A form will appear that looks partially similar to the object form. At the left you can indicate the name of the time line and there are buttons to add, delete and manipulate time moments. In the middle you see the list of defined moments. Next to this list there is the list of actions for the currently selected moment and at the right there are the tabbed pages from which you can choose the actions.

At moment 0 we want to create an enemy plane. So click on the button **Add**. As moment indicate the value 0. The moment is added to the list. Now you can drag action to the action list. We need just one action to create an enemy plane at the appropriate place. As it takes about 100 steps for an enemy plane to traverse the room, we add a second moment at step 100. Here we create two enemy planes next to each other. In this way we continue. We add moments and add creation actions for enemy planes. By creating the planes at different heights above the room we can create nice formations. So after adding a number of moments the form will look as follows:
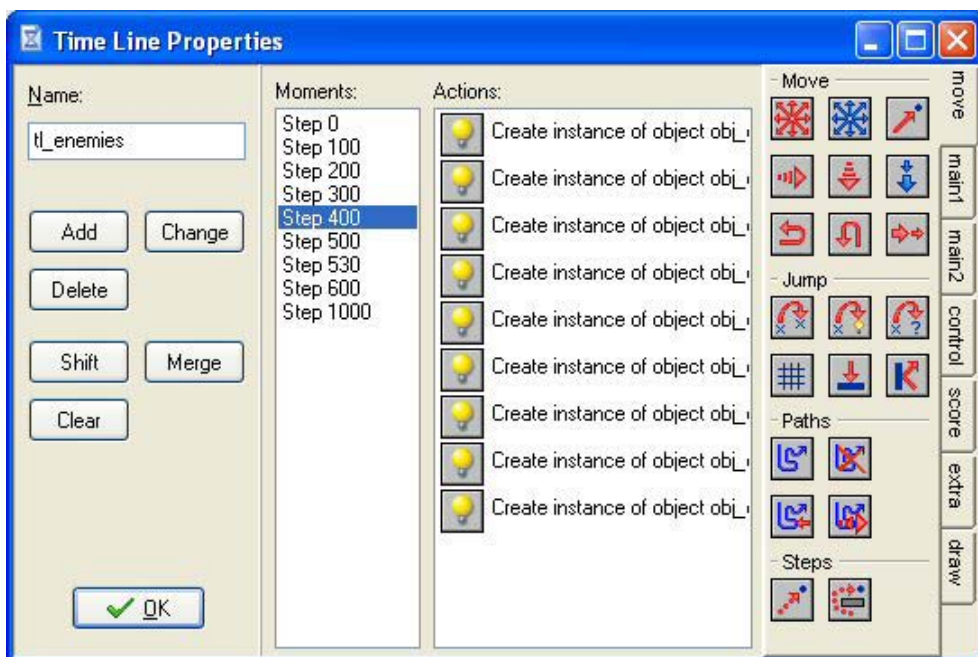
At the indicated moment 400 we create a row of 9 enemy planes, two of which are shooting bullets.

You should by now get the idea. Just keep on adding moments with more and more dangerous planes in interesting formations and maybe put the moments closer together. It will take a bit of work to create a complete sequence of moments that make up a nice level of the game. In a full game you will use different time lines for different levels. At the end of the sequence you need to add a moment that ends the level or the game. Here you should probably display a message that you finished the level or, better, have some nice ending sequence, like an aircraft carrier appearing on which your plane lands.

We are not done yet. A time line does not automatically execute the actions! You must assign a time line to an object and then the actions of the time line will be executed for that object. There are two special actions for this; one to set the time line for an object and a second one to set the position in the time line. We use the **controller_enemy** object again. It only needs one action in its creation event that sets the appropriate time line.
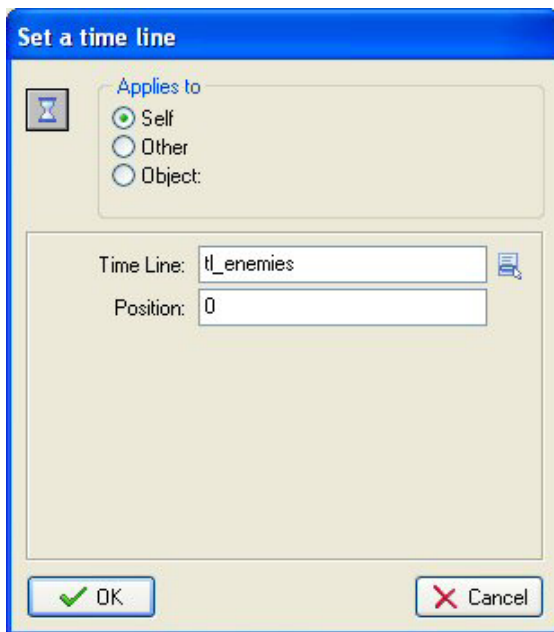
This is all. As you will notice, using time lines is a bit more work because you need to specify all enemy planes that appear but it gives a lot of flexibility and makes the game a lot more interesting.

There are many things you can do with time lines. Here we used them to control the global flow of the game but you can also use them to control the behavior of game objects over time. There are also many tricks you can use. For example, to loop a time line, at the end add a moment in which you set the position of the timeline back to 0. And to halt a time line for a while, add a moment in which you test some condition and, if so, set the time line position relative to -1, that is, in the next step the same moment is happening and the condition is tested again. So the time line won't progress until the condition is false. Instances can also change time lines based on certain events, etc. There

23

are even variables that control the speed of a time line. So time lines form a powerful resource.

A room with a view

Up to now we fooled the player in the sense that we were not really flying through a game world but the background was just scrolling. This has a number of disadvantageous. In particular, the background cannot change easily (e.g. it is difficult to halfway fly over land. Also, even though you can use time lines, it is more difficult to have different actions appearing nicely spread out over time.

In this final section we will briefly indicate a different way of creating a scrolling shooter. In this approach there is a big room through which the plane really moves. We use a so-called view such that the player sees only part of the room at any time. To achieve this, a number of changes to the game are required. We will briefly discuss them here.

Let us first create the basic room. We give the room a width of 640 as before but now we give it a height of 4800. The background image we no longer give a speed. We put islands at various places in the room. The island objects no longer need a speed and we can also remove the step event. They are now fully static objects. The plane we place at the bottom of the room and we give the plane, in its creation event, a vertical speed of -2 to make sure that it flies with some speed if the user does nothing. Finally we define the view. To this end, click on the **views** tab in the room form. Indicate that we want to enable views. Define the first view and make it visible at the start. As a top position, indicate 4320, that is, the view will be at the bottom of the room. So it looks as follows:
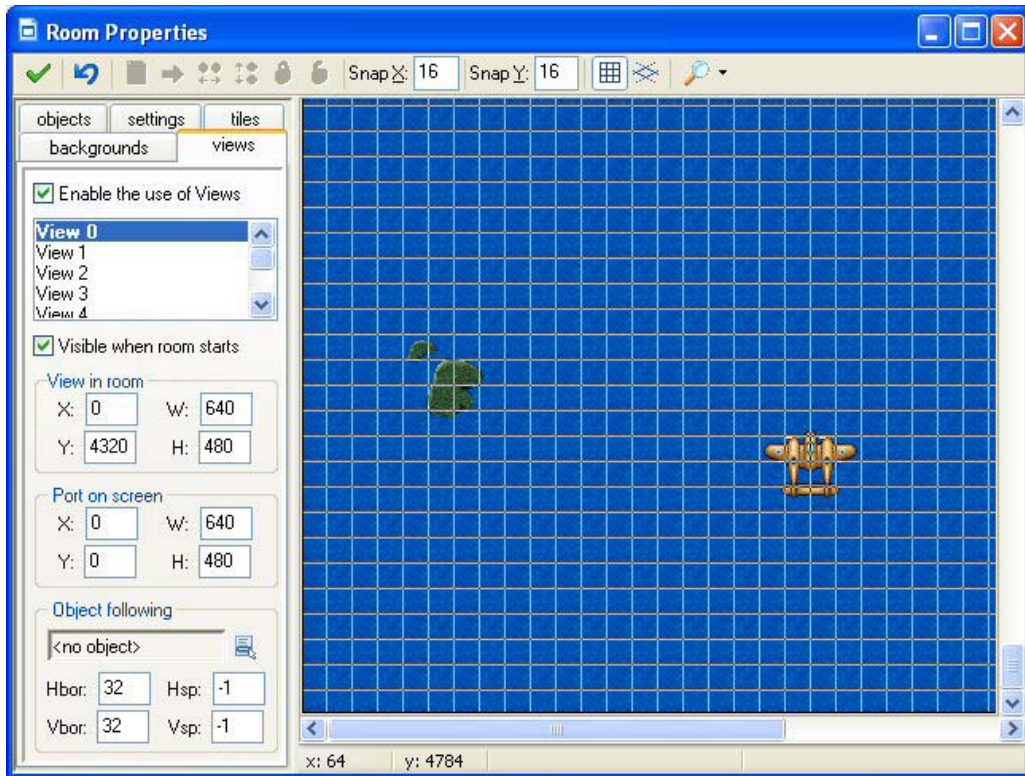
We have to make sure that view moves with constant speed. To this end we will (mis)use the step event of the life controller object. In it we add the action to set a variable and set variable `view_yview` relative to -2. Variable `view_yview` indicates the top position of the first view in the room. (When there are multiple views you should use `view_yview[0]`.) So in each step the position of the view moves 2 pixels upwards creating the scrolling. There are many other variables related to views but we won't need them here. We should also check whether `view_yview` is larger than 0. If it becomes 0 the view has reached the top and we should stop the game.

Note that, for the main plane we wanted to avoid that the player can move the plane out of view. To this end we checked whether the position of the plane was not too high up or too high down. We now have to use the variable `view_yview` in these comparisons to make sure that the plane stays within view.

We have a similar problem with the position of the information panel, and the health bar, etc. We did place them at the bottom of the room, but now we must draw them at the correct place relative to `view_yview` to make sure they stay in sight.

For the bullet that the plane shoots we must increase the speed a bit to compensate for the moving view. Also we should destroy it when it moves out of the view.

It remains to handle the enemy planes. We no longer use a time line or controller object. We will place all enemy planes already in the room, but we will not let them move until they become visible. So we do not set their initial speed and in the step event we first

25

check whether they are in the view. If so we let them move. Also, we will only let them shoot if they are in the view. For example for the third enemy plane the step event looks as follows:
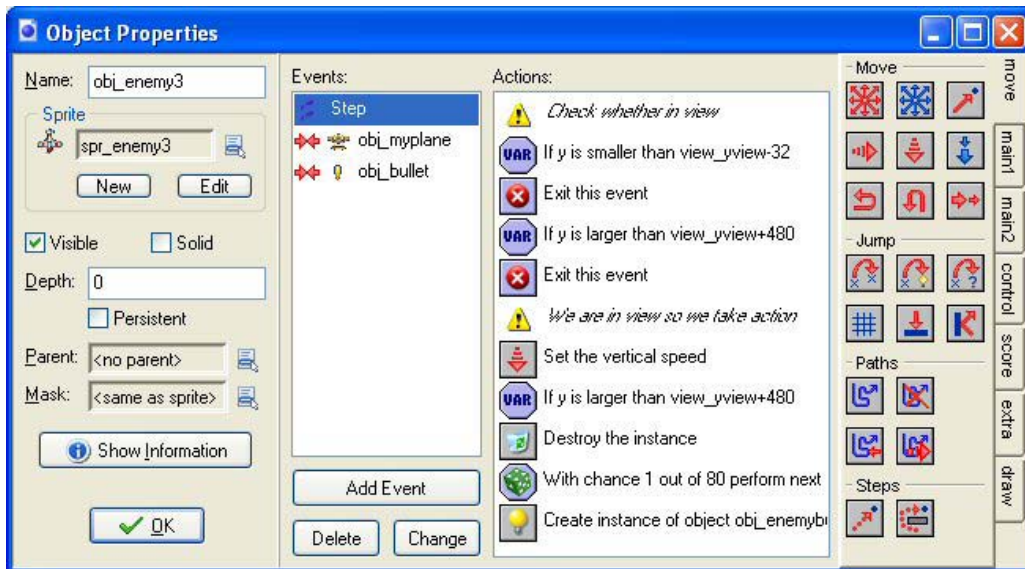
Note that we should use a smaller vertical speed than before. Both for the enemy planes and for the bullets we should also change the conditions when they must be destroyed.

The most complicated one is enemy plane 4. It must come from behind. The trick is as follows. We again place it in the room from the start but we make it invisible. Also, as long as it is invisible we do not let it react to collisions with bullets or the main plane. When it lies below the view we make it visible and let it move (with larger speed). See the example game on how this is achieved.

Now it is time to build the room. We place sets of different enemy planes at strategic places in the level. Again it is easy now to create formations and nice challenges. The game can be found in the file 1945_7.gmk. As you can see, using views you can create very interesting scrolling shooters with relatively little work. Clearly the game need a lot more work. It is in some sense still too easy to finish and it is definitely too short. Also, some nice boss at the end would be great. Just use it as a first step and build from there.