Game Maker Camp

UNT Computer Science and Engineering
### Basic Game – “Catch the Clown” ........................................................ 6
- Overview .......................................................................................................................... 7
- The Game Idea .................................................................................................................. 7
- Creating a New Game ....................................................................................................... 8
- Adding Sprites ................................................................................................................... 9
- Adding Sounds .................................................................................................................. 11
- Objects, Events, and Actions .......................................................................................... 12
  - Adding Objects .............................................................................................................. 13
  - The Create Event .......................................................................................................... 15
  - The Collision Event ...................................................................................................... 17
  - The Mouse Event .......................................................................................................... 19
- Adding a Room .................................................................................................................. 21
- Saving and Testing ........................................................................................................... 22
- Polishing Your Game ....................................................................................................... 25
  - Backgrounds ................................................................................................................ 25
  - Alarm Events ............................................................................................................... 27
  - Control Object ............................................................................................................. 32
- Summary ........................................................................................................................... 36

### Maze Game – “Find the Flag” ........................................................... 37
- Overview ........................................................................................................................... 38
- A Simple Start ................................................................................................................... 38
- The Objects ....................................................................................................................... 39
- Creating Rooms ............................................................................................................... 43
- Collecting Diamonds ...................................................................................................... 44
- Making it a Bit Nicer ........................................................................................................ 47
  - Score .............................................................................................................................. 48
  - Sounds ............................................................................................................................ 49
  - Creating Rooms ............................................................................................................ 49
  - Monsters ......................................................................................................................... 49
- Summary ........................................................................................................................... 51
Shooter Game – “1945 Airplane Shooter” .................................................. 52
Overview .............................................................................................................. 53
Background Information .......................................................................................... 54
Variables .................................................................................................................. 54
Variable Actions ...................................................................................................... 55
Scrolling Backgrounds ............................................................................................ 56
Begin Game Steps .................................................................................................... 57
Scrolling Islands ....................................................................................................... 58
The Player Plane ....................................................................................................... 63
Enemies and Shooting .............................................................................................. 67
The Player Cannon .................................................................................................. 67
Enemy Planes .......................................................................................................... 71
Collisions .................................................................................................................. 71
Collision with the Player Bullet ................................................................................ 71
Collision with the Player .......................................................................................... 73
The Controller Object ............................................................................................... 75
Score, Lives, Health, and HUD .................................................................................. 76
Creating A HUD ....................................................................................................... 76
Checking Health and Lives ....................................................................................... 80
Adding More Enemies .............................................................................................. 83
Shooting Enemy ........................................................................................................ 83
Aiming Enemy .......................................................................................................... 86
Enemy from Behind .................................................................................................. 89
Spawning New Enemies ........................................................................................... 90
Finishing the Game ................................................................................................... 92

Platformer Game – “Princess Peach” ................................................................. 93
Overview .................................................................................................................. 94
The Basics ............................................................................................................... 94
Motion ....................................................................................................................... 95
Jumping ..................................................................................................................... 95
Basic Game

“Catch the Clown”
Overview

Even though GameMaker:Studio is easy to use, getting the hang of it can be a bit difficult at first, especially if you have had no prior experience with programming. This tutorial is meant for those that have some difficulty getting started and it will lead you step by step through the process of making your first game.

To make your first game you have to understand a number of the basic aspects of GameMaker:Studio, so please read this tutorial carefully and try to follow all the steps correctly. Once you have finished your first game, the second one is going to seem a lot easier!

The Game Idea

It is important that we first write a brief description of the game we are going to make, and because this is going to be our first game we better design something simple. It should keep the player interested for just a short time, so our game is going to be a little action game that we will name "Catch the Clown". Here is our description of the game:

Catch the Clown

"Catch the Clown" is a little action game. In this game a clown moves around in a playing field. The goal of the player is to catch the clown by clicking on him with the mouse. As the player progresses through the game the clown starts moving faster and it becomes more difficult to catch him. For each catch the score is raised and the goal is to get the highest possible score. Expected playing time is just a few minutes.
Clearly, a game like this will have limited appeal, but we have to start simple and later we can add some features to the game to make it more interesting.

The second step in creating a game is to write a more precise design document, and you are recommended to always do this before making your game, even if it is very simple rough draft or outline of your plans. Here is our design document for "Catch the Clown":

- **Game objects** - There will be just two game objects: the clown and the wall. The wall object has a square like image. The wall surrounding the playing area is made out of these objects. The wall object does nothing. It just sits there to stop the clown from moving out of the area. The clown object has the image of a clown face. It moves with a fixed speed. Whenever it hits a wall object it bounces. When the player clicks on the clown with the mouse the score is raised by 10 points. The clown jumps to a random place and the speed is increased with a small amount.

- **Sounds** - We will use two sounds in this game. A bounce sound that is used when the clown hits a wall, and a click sound that is used when the player manages to click with the mouse on the clown.

- **Controls** - The only control the player has is the mouse. Clicking with the left mouse button on the clown will catch it.

- **Game Play** - At the start of the game the score is set to 0. The room with the moving clown is shown. The game immediately begins. When the player presses the key the game ends.

- **Levels** - There is just one level. The difficulty of the game increases because the speed of the clown increases after each successful catch.

That should be good enough for the moment, and we can now start creating the game!

**Creating a New Game**

To start a New Game, Select New from the File Menu. Now would also be a good time to save your new game and give it a name. To do this, select Save from the File menu.

- Your camp counselors will instruct you on where to save your game.
Adding Sprites

As the game design document describes we will need two images for the two game objects. Such images are called sprites in GameMaker:Studio. There is a lot to know about sprites but, for the moment, simply think of them as little static images.

For making these images you can use any drawing program you like, for example the paint program that is part of any Windows system, but GameMaker:Studio also has a built-in sprite editor for this purpose. Creating nice-looking sprites is an art that requires a lot of practice, but fortunately there are large collections of images of all sorts available for free. By searching the web and you are bound to find images in large quantities.

For our game we have provided several options of each sprite for you to use. They can be found in the Game Assets folder.

- Ask your camp counselors for the location of this folder.

The clown:

The wall:

To add these sprites to the game we proceed as follows: From the Resources drop down menu at the top of the main GameMaker:Studio window, choose Create Sprite. The Sprite Properties form appears, like the one shown below:
Click on the Name field where currently is says sprite0 (the default name for the sprite). Rename it to "spr_clown".

Click on the Load Sprite button in this window. This opens a file browser.

Navigate to the Game Assets folder that your camp counselors specified and selected the image file clown.png. The sprite properties should now look like this:

You should also now make the sprite origin at its center. The origin is the point at which GameMaker:Studio will draw the sprite in the game room, and all you have to do here is press the button marked Center to place the draw origin at the center of the sprite. You should also change the collision mask of the sprite to be the full image and an ellipse as we will want it to bounce later and this mask shape will give the best results. This can be done by clicking the “modify mask” button. The collision mask is what GameMaker:Studio will use to base all the collisions between different instances off of and as such it enables you to give a complex sprite a simple collision shape, as you can see in this image:

Press the OK button to close the form.

Next we will add the wall object in the same way:

From the Resources menu, choose Create Sprite. Click on the Name field and rename it to "spr_wall".
• Click on the Load Sprite button and select the image file wall.png.
• Press the OK button to close the form.

There is no need to change the origin, nor the collision mask for this sprite as they are fine as they are.
As you might have noticed, the clown and wall sprite have now appeared in the list of resources at the left of the GameMaker:Studio window. Here you will always find all the sprites, sounds, objects, rooms, etc. that you have created in your game. Together we call them the assets (or resources) of the game. You can select an asset by clicking on its name, and then you can use the Edit Menu to change the asset, duplicate it, or delete it, while right-clicking on the asset name will show a similar menu with further options. This overview of assets will become crucial when you are creating more complicated games.

Adding Sounds

Now that we created the sprites we will need to create two sound effects. One must play when the clown hits a wall and the other must play when the clown is successfully caught with the mouse. We will use two wave (name.wav) files for this as wave files are excellent for short sound effects.

To create two sound assets:

• From the Resources menu, choose Create Sound, and the Sound Properties form appears. Click on the Name field and rename it to "snd_bounce".
• Click on the Load Sound button, navigate to the Game Assets folder, and select the sound file bounce.wav. All of the other settings in this window can be left on their default settings. The form should now look like this:
• Press the OK button to close the form.

• Create another sound asset and name it "snd_click".

• Click the Load Sound button and select the sound file click.wav.

• Click OK to close the form.

Within the sound properties form you can use the play button with the green triangle, to listen to the sound which is repeated constantly unless you press the red stop button. Again, notice that the two sounds are shown in the resource tree on the left of the main Gamemaker window.

Objects, Events and Actions

Having created the sprites and sounds does not mean that anything is going to happen in your game. Sprites are only the images for game objects and we have not yet defined any game objects. Similarly, sounds will only play if we tell them to be played, and we can only tell them to be played from a game object too. So we need to create our two game objects next...

However, before we do this you will have to understand the basic way in which GameMaker:Studio operates. As we have indicated before, in a game we have a number of different game objects and during the running of the game one or more instances of these game objects will be present on the screen or in the game world. Note that there can be multiple instances of the same game object! So for example, in our "Catch the Clown" game there will be one wall object, but a large number of instances of that object surrounding the playing field, and there will be just one instance of the clown object.

Instances of game objects don’t do anything unless you tell them how to act, and you do this by indicating how the instances of the object must react to events that happen. There are many different events that can happen as your game progresses. The first (and often most important) event is when the instance is created. This event is called the Create Event, and more often than not some action is required here. For example we must tell the instance of the clown object that it should start moving in a particular direction.

Another important event happens when two instances collide with each other and is called a Collision Event. For example, when the instance of the clown collides with an instance of the wall, the clown must react and change its direction of motion.
Other events happen when the player presses a key on the keyboard or clicks with mouse on an instance etc... For the clown we will use a **Mouse Event** to make it react to a press of the mouse on it.

To indicate what must happen in any given event, you must specify actions. There are many useful actions for you to choose from, for example there in an action that sets the instance in motion in a particular direction, or there is an action to change the score, and there is an action to play sounds.

So defining a game object consists of a few aspects:

- We can give the object a sprite as an image
- We can set some properties
- We can indicate to which events instances of the object must react
- We can add actions that they must perform.

Please note the distinction between objects and instances of those objects. An object defines a particular game object with its behavior (that is, reaction to events). Of this object there can be one or more instances in the game and these instances will act according to the defined behavior. Stated differently, an object is an abstract thing - like in normal life, we can talk about a chair as an abstract object that you can sit on, but we can also talk about a particular chair, that is an instance of the chair object, which actually exists in our home.

### Adding Objects

So how does this all this come together for the game we are making? To start with, we will need two objects. Let us first create the very simple wall object, as this object needs no behavior at all since it will not react to any events in the game world.

To create the wall object you must follow these simple steps:

- From the drop down **Resources Menu** in the main GameMaker:Studio window choose **Create Object**. The Object Properties form appears, as is shown below:
Click on the **Name** field and rename the object to "obj_wall".

Click on the menu icon to the right of the Sprite field and in the list of available sprites select the "spr_wall" sprite.

Instances of the wall object must be **solid**, that is, no other instances in the game should be allowed to penetrate them. To do this click on the box next to the **Solid** property to enable it.

Press OK to close the form.

For the clown object we start in the same way:
• From the Resources menu, choose **Create Object**.

• Click on the Name field and rename the object to "obj_clown".
• Click on the icon to the right of the Sprite field and select the "spr_clown" sprite.

• Don’t close this window just yet.

• Note that we do not make the clown object solid! Now, for the clown there is a lot more that needs to be done as we have to specify its behavior.

**The Create Event**

The rest of the object form is taken up by two windows: one for adding **events**, and one for adding **actions** to those events. With the buttons below the events window you can add events, delete events or change events, of which there are a large number of different ones you can use, but you normally need just a few in your game.

Next to the events there is an empty list of actions that must be performed for the selected event (if any), and at the right of this list are a number of tabbed pages with little icons. These icons represent the different actions and in total there are close to 100 different actions you can choose from. If you hold your mouse above one of the icons a short description of the corresponding action is given in a tooltip. You can drag actions from the tabbed pages at the right to the action list to make them happen when the event occurs.

We are now going to define what should happen when an instance of the clown object is first created. In this case we want the clown to start moving in an arbitrary direction so proceed with the following steps:

• Press the **Add Event** button. The Event Selector, as shown below, will appear.
Click on the **Create** button. The create event is now added to the list of events, and it is automatically selected (with a blue highlight).

Next you need to include a **Move Fixed** action in the list of actions. To do this, open the **Move** tab and then press and hold the mouse on the action image named “move fixed” in the page at the right, drag it to the empty actions list, and release the mouse. An action form is shown asking for information about the action.

In the action window for the Move Fixed action you can indicate in which direction the instance should start moving. Select all eight directions (not the middle one which corresponds to no motion), and note that the selected directions change color when selected. When multiple directions are selected one is chosen at random when the event is run. Also on this window set the Speed to 4.
• Press OK to indicate that we are ready with this action.

You have now specified behavior that must be executed when an instance of the clown object is created, by adding the event, including an action, and setting the action properties. The object properties window for the clown object should now look like this:

- Do not close this window yet.

**The Collision Event**

The next event we will define is a **collision event** with the wall object. Here we will bounce the clown against the wall and we will play the bounce sound effect.
To do this, follow these steps:

- Press the **Add Event** button.

- In the Event Selector, click on the **Collision** button and select "obj_wall". The collision event is now added to the list of events.

- Include a **Bounce action** by dragging it from the tab at the right (from the "Move" tab, "Jump" section).

- Leave the default values for this action. We are not interested in precise bounces and we want to bounce against solid objects.

- Press **OK** to close the action form.

- Select the tab "Main1", and from it include the **Play Sound** action.

- Drag the Play Sound action below the Bounce action already present. In the action window, click on the icon to the right of the Sound property and from the list select "snd_bounce". Leave the Loop property as false as we want to play the sound only once.

- Press **OK** to close the action.

That concludes the collision event with the wall object, which now has two actions to be performed (in the given order) when the collision occurs. If you have made a mistake, you can right-click with the mouse on an action you added and, for example, choose Delete to remove the action (or press the “Delete” key on the keyboard). You can also choose Edit Values to change the properties of the action (Double-clicking on the action will do the same), and you can also drag them up and down to change the order in which they are executed within the event if you have them out of order.

The object properties window should now look like this:
Don’t close this window yet, there are still more actions to add.

The Mouse Event

Finally we need to define what to do when the user clicks with the left mouse button on the clown. We are going to add four actions here...

First we will add 10 points to the score. This is easy as GameMaker:Studio will automatically keep the score for you in a special variable. Next we will play the click sound. Then, after this, we will jump the clown to a random position, and we will set a new random direction of motion with a slightly increased speed. The last two actions are added to gradually increase the difficulty of the game.

The following steps explain how to create the mouse event that we require:

- Press the Add Event button. In the Event Selector click on the Mouse Event button and in the menu that appears select Left Pressed. This event happens when the user presses the left mouse button while the mouse cursor is on top of the instance and will only be triggered once for each press.

- From the tab labelled Score include the Set Score action.
• Set the new score to a value of 10 and also click on the box next to the property Relative to enable it. When Relative is enabled the value is added to the current score, otherwise the score would be replaced by the value.

• Click OK to close the set score window.

• From the tab Main include a Play Sound action and for the sound indicate "snd_click". Leave Loop as false.

• Click OK to close the play sound window.

• From the Move tab, include a Jump to Random action, which places the instance in a random, collision-free position. The parameters can be left unchanged for this action.

• Click ok to close the jump to random window.

• Finally we include a Move Fixed action.

• Just as before select all eight arrows (and not the center square), but this time for the speed indicate a value of 0.5 and enable the Relative property to add 0.5 to the current speed each time the clown is successfully captured or clicked.

• Click OK to close the move fixed window.
That is all the actions we need for the **Mouse Event** and the finished event list should look like this:

![Object Properties: obj_clown](image)

We are now finished with the clown object. We have included actions for the three events that are important, so press the OK button to close the form.

**Adding a Room**

Now that we have created the game objects there is one more thing to do. We need to create the room in which the game takes place.

For most games, designing effective rooms (the "levels" of the game) is a time-consuming task because here we must find the right balance of difficulty without discouraging the player. But for Catch the Clown the room is very simple: a walled area with one instance of the clown object inside it.

To start with, from the drop down Resources menu in the main GameMaker:Studio window, choose **Create Room**. This will create a room and open the **Room Properties** window. On the left you see some tabbed pages, and here you should select the tab labelled **Settings**.
Now follow these steps:

- In the Name field type in "rm_Main". This is the identifier for our new room.
- Set the size of the room to a width of 640 and a height of 480.
- Select the Objects tab. Enlarge the window somewhat such that you can see the complete room.
- At the top of the room editor, make sure the values for Snap X and Snap Y are both 32, as the size of our sprites is 32x32 pixels and this makes it easier to place the sprites at the correct locations.
- On the left you will see either the image of the clown object or the image of the wall object, which is the currently selected object. Place one instance of the clown object in the room by clicking with the mouse somewhere in the center of the grey area.
- Click on the icon with the menu symbol next to the field that reads "obj_clown". Here you can select which object to add, and you should now select "obj_wall".
Click on the different cells bordering the room to put instances there. To speed this up, you can press and hold the <Control> + <Shift> keys on the keyboard and drag the mouse with the mouse button pressed. You can remove instances using the right mouse button and selecting Delete from the subsequent pop-up menu, or by holding down the <Control> key while right clicking.

When you are happy with the results, you should click on the green check at the top left of the window to close and save your room. Your room may look like this when you are done, or it could look different:
Saving and Testing

You might not have realized it but our game is now ready to test. The sprites and sounds have been added, the game objects have been designed and the first (and currently only!) room in which the game takes place has been created. Now it is time to save the game and to try it out.

Saving the games works as in almost any other Windows program - just click the Save icon and GameMaker:Studio will save your project.

Next we need to test the game. Testing is a crucial component of creating a game, and although you can test it yourself, you should also ask others to test it. Testing (or running the game in general) is simple! Choose the command Run Normally from the drop down Run menu in the main GameMaker:Studio window, or press the green Play button. Be patient, loading your game may take a few moments. The game will be loaded and, if you did not make any mistakes, the room will appear on the screen with the clown moving inside it.

Try testing it now and see whether the game behaves as expected. You should hear the correct sounds and the speed of the clown should increase every time you click on it. To end the game, close the game window.

Now it is time to fine tune the game. You should ask yourself for example the following questions: Is the initial speed correct? Is the increase in speed correct? Is the room size correct? Did we pick effective sprites and sounds for the game? If you are not happy, change these aspects in the game and test again. There should be several choices of sprites in the Game Maker Assets folder,
or you can find more online. Remember that you should also let somebody else test the game, because since you designed the game it might be easier for you than for other people. This becomes more important later on when your created games have more difficulty.

Once you are happy with your game you can create a stand-alone installer for the game. This is a version of the game that can run without the need for GameMaker:Studio. In the File Menu choose the command Create Application. In the screen that opens, choose where to save your new game, give your game a name and from the Save as Type dropdown menu select “Single Runtime Executable (*.exe)” and then click save. Once you have done this, you'll see the compile window at the bottom of GameMaker:Studio showing you the progress and when it reads "Finished!" you're all set to distribute your game! You can now take your game home with you and play it on your computer without first having to install GameMaker:Studio.

**Polishing Your Game**

Our first game is ready but it needs some finishing touches to make it a bit nicer. For example, some music, a score display, a nicer background... These are all things that will improve the game and make it a nicer experience for the player. To start with, let's add a background...

**Backgrounds**

The grey background of the room is rather boring, and so we are going to use a new type of resource, the background resource. To add one, go to the Resources Menu and choose Create Background. The Background Properties window will appear, and you should click on the "Name" field and rename it to "bck_main". Now, click on the Load Background button, navigate to the Game Assets folder and select the image file background.png.
Click OK to close the Background Properties window.

We need to assign this background image to the room, so double click on the game room (rm_main) in the resource tree to open it up, then click on the "backgrounds" tab. You then need to deselect the property Draw Background Color (as we do not need this) then click on the menu icon in the middle of the window and pick the bck_main in the popup menu. As you will see, in the room we have a nice background. Note the two properties Tile Hor. and Tile Vert. that are available on this tab. They indicate that the background must be tiled horizontally and vertically, that is, repeated to fill the whole room. For this to work correctly the background image must be made such that it nicely fits against itself without showing cracks or notable breaks in the pattern.

The room editor tab should now look like this:
Click the check mark in the top left corner to close the window.

Alarm Events

Even though the clown speeds up every click, it is still quite easy to click on it as the direction it moves in is always a straight line. To make it more difficult we will let the clown change its direction of motion from time to time. To accomplish this we are going to use an Alarm. Each instance can have multiple alarms (up to 12) and these alarms tick down every step (game tick) and at the moment they reach 0 the associated Alarm Event happens.

In the Create Event of the clown we will set the alarm, and in the alarm event we will change the direction of motion and set the alarm again. To do this, follow these steps:

1. Reopen the clown object by double clicking on it in the resource tree at the left of the window. The clown object will be located in the “Objects” folder.

2. Select the Create Event and from the Main2 tab add a Set Alarm action.

3. In a game, time is split into steps and the steps are defined by the room speed, which is the number of steps that the game has to complete per second. The default is 30, so the game completes 30 steps in every second of real time. So, the alarm is calculated in steps and we want it to run every two seconds, so there we should have a value of 60 (two times the current room speed). The alarm number should be left as 0, as it is the Alarm[0] Event that we are wanting to trigger.
Click OK to close the set alarm window.
Click on Add Event and choose the button Alarm, then from the pop-up menu select Alarm 0. In the event include the Move Fixed action (from the move tab), select all eight arrows and set the speed to 0 but, unlike previous uses of this action, enable the Relative property. In this way 0 is added to the speed, that is, it does not change, but the direction is allowed to change.

Click OK to close the Move Fixed window.

To set the alarm clock again, include another 'Set Alarm action, and set it to 60 steps for Alarm[0], exactly as before. This will cause a loop in the alarm and it will now run every 2 seconds of your game.

The Create and Alarm0 Events should now look like the following:
Click OK to close the object properties window.

Go to the File Menu and Save your game. Then Click the play button again.

Test your game again now and you'll find that it presents much more of a challenge than previously! Close the game window once you have tested your game.
We can also make the clown face the direction that it is moving in.

As you can see, the sprite is currently facing to the right, and in GameMaker:Studio, this represents 0°. So, what we want to do is add in an action to change the angle the sprite is drawn every time it changes direction.

For this, we need to go to **Main1** tab and select the **Transform Sprite** action:

In this action, you can scale the sprite, rotate the sprite and also mirror the sprite around the horizontal or vertical axis, but for now we just need to use the angle. In the space provided write the word “direction”. This is a special variable that GameMaker:Studio uses to get the angle at which an instance is moving, and placing it here will now make the sprite angle rotate to face the direction of movement.
Go ahead and place copies of this new action in the create event, alarm0 event, and obj_wall event on the clown object properties, after the actions that change its direction. These events will now look like this:
The final thing we are going to add to the game is a **controller object**. Many games have special objects with no sprites assigned to them that are used to control elements of the game that are not directly related to the gameplay. In this case, our controller will show the score and play some background music.

First let’s add some music to the resources for the game:

- From the Resources drop down menu, choose **Create Sound**, which will open a new **Sound Properties** window. Click on the Name field and rename it to "snd_music".
• Click on the Load Sound button, navigate to the Game Make Assets folder and select the sound file music.mp3.

• Press Open.

• Press the OK button to close the sound properties window.

Now create a new object by opening the Resources drop down menu and selecting **Create Object**. Give this object the name "obj_Control" and add a Create Event to it. In this create event, drag a play Sound action into it (located in the Main1 tab) and for a sound indicate "snd_music" and set **Loop** to **true** because we want the music to repeat itself forever as long as the game is running.

• Click OK to close the Play Sound window.

We want to display the player’s score with this object too, so you need to add a **Draw Event** to your object. Click Add Event, select the Draw button and choose draw. In this **Draw Event** you should add the draw score action from the score tab.

![Draw Event](image)

The position for drawing the score is defined by the x and y values that you input here, so set them both to 64, and leave the "Caption" as it is, since it is the score we are showing and not some other value. The draw score window should now look like this:
Click OK to close the draw score window.
The object properties window for obj_control should now look like this:
Click OK to close the object properties window.

The last thing we need to do is to add this object into the room, so double click the game room to open it again and then in the **Objects** tab, select "obj_Control" and place it anywhere in the room.
Since the object does not have a sprite, you will see that GameMaker:Studio shows you its position with a little blue ball marker with a question mark. This will not be shown in the game and is only there as a reminder to you that you have placed a spriteless instance in the room.

- Click the check mark to close the room properties window.

Run the game now and see how everything works together!

**Summary**

Your first game is now complete! It may not seem like much, but thanks to this simple game you have learned the basics of creating games with GameMaker:Studio... You now know about game assets and how to create them. You know how to create objects and the difference between them and instances. You know how to make a room, and add instances into it. These things are the building blocks on which you will construct better games in the future!
MAZE GAME

“Find the Flag”
Overview

Before starting creating a game we have to come up with an idea of what the game is going to be. This is the most important (and in some sense most difficult) step in designing a game. A good game is exciting, surprising and addictive. There should be clear goals for the player, and the user interface should be intuitive.

The next game we are going to make is a maze game. Each room consists of a maze. To escape the maze the player must collect all diamonds and then reach the exit. To do so the player must solve puzzles and monsters must be avoided. Many puzzles can be created: blocks must be pushed in holes; parts of the room can be blown away using bombs, etc. It is very important to not show all these things in the first room. Gradually new items and monsters should appear to keep the game interesting.

So the main object in the game is a person controlled by the player. There are walls (maybe different types to make the maze look more appealing). There are diamonds to collect. There are items that lie around that do something when picked up or touched by the player. One particular item will be the exit of the room. And there are monsters that move by themselves. But let us tackle these things one by one.

A Simple Start

As a first start we forget about the diamonds. We will create a game in which the player simply must reach the exit. There are three crucial ingredients in the game: the player, the wall, and the exit. We will need a sprite for each of them and to make an object for each of the sprites.
The Objects
Let us first create the objects. For each of the three objects we use a simple 32x32 sprite:

Create these three sprites as you have done in the previous game and name them spr_person, spr_wall, and spr_goal. These sprites are located in the GameMaker Assets folder as directed by your camp counselors.
Remember to set the origin of all the sprites at the center. Also modify the mask of the player to be a rectangle. The wall and goal’s masks can also be set to rectangles.

Next we create three objects. Let us first make the wall object. We will give it the spr_wall sprite as image, name it obj_wall and make it solid by checking the box labeled Solid. This will make it impossible for other objects, in particular the person, to penetrate the wall. The wall object does not do anything else. So no events need to be defined for it.

Secondly, we need to create the person that is controlled by the player. Some more work is required here. It must react to input from the user and it should not collide with a wall. We will use the arrow keys for movement. (This is natural, so it is easy for the player to understand.)
There are different ways in which we can make a person move. The easiest way is to move the player one cell in the indicated direction each time the player pushes the arrow key. A second way, which we will use, is that the person moves in a direction as long as the key is pressed. Another approach is to keep the player moving until another key is pressed (like in PacMan).

We need actions for all four arrow keys. The actions are rather trivial. They simply set the right direction of motion. (As speed we use 4.) To stop when the player releases the key we use the keyboard event for <no key>. Here we stop the motion. There is one complication though. We really want to keep the person aligned with the cells of the grid that forms the maze. Otherwise motion becomes rather difficult. E.g. you would have to stop at exactly the right position to move into a corridor. This can be achieved as follows. In the control tab there is an action to test whether the object instance is aligned with a grid. Only if this is the case the next action is executed. We add it to each arrow key event and set the parameters to 32 because that is the grid size in our maze:
Clearly we also need to stop the motion when we hit a wall. So in the collision event for the person with the wall we put an action that stops the motion. There is one thing you have to be careful about here. If your person’s sprite does not completely fill the cell, which is normally the case, it might happen that your character is not aligned with a grid cell when it collides with the wall. In this case the person will get stuck because it won’t react to the keys (because it is not aligned with the grid) but it can also not move further (because the wall is there). The solution is to either make the sprite larger, or to switch off precise collision checking and as bounding box indicate the full image. This is why we set the mask to be a rectangle rather than an ellipse as we did in the first game.
Finally, let us create the goal object. This is the object the player has to reach. It is a non-solid object. We decided to give it a picture of a finish flag. This makes it clear for the player that they have to go here. When the person collides with it we need to go to the next room. So we put this action in this collision event (it can be found in the tab main1) (restart game is on the main2 tab) (else is located in the Control tab). This has one drawback. It causes an error when the player has finished the last room. So we have to do some more work. We first check whether there is a further room. If so we move there. Otherwise we restart the game. So the event will look as follows:

![Object Properties: obj_goal](image)

Obviously, in the full game we better do something more when the player finishes the last level, like showing some nice image, or giving him a position in the list of best players. We will consider this later.

**Creating rooms**

That was all we had to do in the actions. Now let us create some rooms. Create one or two rooms just like you did in the first game but this time make them look like a maze (Resources menu, Create Room). In each room place the goal object at the destination and place the person object at the starting position. Name your rooms: room0, room1, room2, etc. so that they follow each other in the order you want. In the ‘object to add’ box, select your wall object and create your maze by clicking in the editing field on the right. Add just one object of your player character and just one goal object per room. Create as many rooms as you want with whatever maze pattern you want. Just make sure to name them room0, room1, room2, etc.
And that is all. The first game is ready. Play a bit with it. Try changing the speed of the person in its creation event, create some more levels, change the images, etc. Once you are satisfied in the basic operation of your game, we can move on to adding more details and challenges to the game.

## Collecting Diamonds

Having a simple maze game is nice, but the goal of our game was to collect diamonds. The diamonds itself are easy. But how do we make sure the player cannot exit the room when not all diamonds are collected? To accomplish this we add a door object. The door object will behave like a wall as long as there are still diamonds left, and will disappear when all diamonds have gone.

Beside the wall, goal, and person object we need two more objects, with corresponding sprites: the diamond and the door. Create these sprites as you have done previously. Name them obj_door and obj_diamond. Use the sprites located in the GameMaker Assets folder. Set the origin of both to be the center of the sprite and leave them using a rectangle mask.
Also create the corresponding objects for each sprite. Next, add a sound for when the player collects the diamonds, and for when they reach the door.

The diamond is an extremely simple object. The only action it needs is that it is destroyed when the person collides with it. So in the collision event we put an action to delete it.
The door object will be placed at a crucial place to block the passage to the goal. It will be solid (to block the person from passing it). In the collision event of the person with the door we must stop the motion.

In the step event of the door we check whether the number of diamonds is 0 and, if so, destroys itself. There is an action for this. We will also play some sound such that the player will hear that the door is opened. So the step event looks as follows:
Now you have two more items that you can add to your game, the diamond and the door. Try adding these to your rooms, or make new rooms that use these two items.

Making it a bit nicer

Now that the basics of the game are in place, let us make it a bit nicer.

The walls look pretty ugly. So let us instead make three wall objects, one for the corner, one for the vertical walls, and one for the horizontal walls. Create sprites for each type of wall. Give them the right sprites and make them solid and centered. Now with a bit of adaptation of the rooms it looks a lot nicer. Giving the rooms a background image also helps. After all, who wants to play a game when all the walls are the same?
To avoid having to specify collision events of the person with all these different walls (and later similar for monsters), we use an important technique in *Game Maker*. We make the main wall object the parent of the other wall objects (corner, horizontal, and vertical). This means that the wall objects behave as special variants of the corner wall object. So they have exactly the same behavior (unless we specify different behavior for them) Also, for other instances, they are the same. So we only have to specify collisions with the main wall and not for each type of wall. This will automatically be used for the other wall objects.

**Score**

Let us give the player a score such that he can measure his progress. This is rather trivial. For each diamond destroyed we give 5 points. So in the destroy event of the diamond we added 5 points to the score. For finishing a level we will give the player 40 points so we add 40 points (relative) to the score in the collision event for the goal with the person. Finishing the game will give the player 100 points.

Because of all of these changes, the goal object does become a bit more complicated. By adding all of these changes, your goal object should now look like this:

It adds something to the score, plays a sound, and then either goes to the next room or, if this is the last room, shows a message (You have Won!), and finally it restarts the game.
To display the score, we will create a controller object. It does not need a sprite. This object will be placed in all rooms. It does some global control of what is happening. For the moment we just use it to display the score. In its drawing event we set the color and then use the action to draw the score.

**Sounds**
A game without sounds is pretty boring. So we need some sounds. First of all we need background music. For this we use some nice midi file. We start this piece of music in the start_controller, looping it forever. Any event can be accompanied by a sound if you choose to create it that way. Try adding sound to more of the actions in your game. Perhaps a sound when the player runs into a wall, or a running sound when the player is moving. Just use your imagination on where to have more sounds.

**Creating rooms**
Now we can create some rooms with diamonds. Note that the first maze room without diamonds we can simply leave in. This is good because it first introduces the player to the notion of moving to the flag, before it has to deal with collecting diamonds. By giving a suggestive name to the second room in the settings menu of the room create screen, the player will understand what to do. Try giving your rooms different names.

**Monsters**
The game as it stands now starts looking nice, but is still completely trivial, and hence boring to play. So we need to add some action in the form of monsters.

We will create two different monsters: one that moves left and right, and one that moves up and down. Adding a monster is actually very simple. It is an object that starts moving and changes its direction whenever it hits a wall. When the person hits a monster, it is killed, that is, the level is restarted and the player loses a life.
Let us now create the monster that moves left and right. We use a simple sprite for it and next create an object with the corresponding sprite. In its creation event it decides to go either left or right. Also, to make life a bit harder, we set the speed slightly higher. When a collision occurs it reverses its horizontal direction.
We also need to add an event to the person object so that it is “killed” if it collides with the monster and then it will restart the level. You could also add a score penalty in this action so that the player loses points for dying.

The second monster works exactly the same way but this time we start moving either up or down and, when we hit a wall, we reverse the vertical direction.

When the person collides with a monster, we have to make some awful sound, decrease the number of lives by one, and then restart the room. (Note that this order is crucial. Once we restart the room, the further actions are no longer executed.) The controller object, in the "no more lives" event, shows the high-score list, and restarts the game.

Summary
You should now have the basic outline of your maze game complete. There is much more that you can add to this game. It all depends on how much time and effort you are willing to put into your game. A few suggestions on things that you could try adding are:

- Bombs which explode part of a wall so that the player can pass through them.
- Blocks which can be pushed when the player runs into them.
- Holes which the player can fall into if they aren’t careful.
- Bonuses which give your player some extra benefit. i.e. shooting an enemy
- Frightened monsters that run from the player.
- Points for killing monsters.
- Etc.

You could also change the sprites and images that were used to make your game look better, or to fit into some kind of theme.
Shooter Game

“1945 – Airplane Shooter”
Overview

Scrolling shooters are a very popular type of arcade action game, and are also rather easy to create with a package like GameMaker:Studio. In this tutorial we will make such a game and, in the process, you will learn a number of features of GameMaker:Studio, 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 or destroyed. Often bonus objects appear that can be picked up for additional 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 the 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 explore the use of a very important aspect of GameMaker that extends the possibilities considerably: the use of variables.
Background Information:

Variables

Before we will actually create the scrolling shooter game we need to delve a bit into an important concept of GameMaker:Studio; 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 from our first two games, 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 x or y position of the instance or the speed of the instance. All these things are governed by variables and the values we assign to them give the instance its properties.

Each instance has a number of such variables and there are also a number of global variables, like score, that are not related to individual instances, but rather relate to the whole game and are accessible to all instances, such that changing their values affects everything in the game which uses them.

Variables are given names to make them easy to understand and ensure that you don’t forget what properties they control.

Here is a list of some of the instance variables that we will use in this game:

- x: the x coordinate of the instance
- y: the y coordinate of the instance
- hspeed: horizontal speed (in pixels per step)
- vspeed: vertical speed (in pixels per step)
- direction: current direction of motion in degrees, 0 being right, and going anticlockwise to 360 being right again
- speed: the current speed in this direction
- visible: whether the object is visible (1, or true) or invisible (0, or false)
- solid: whether the object is solid (1, or true) or not 0, or false

And here are some global variables:

- score: the current value of the score
- lives: the current number of lives
- health: the current health of the player (defaults to 100)
- mouse_x: the x coordinate of the mouse
- mouse_y: the y coordinate of the mouse
- room_speed: current speed of the room in steps per second
• 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 to the game. All of them can be found in the GameMaker:Studio documentation (or you can also see a complete list by going to the Scripts menu and selecting Show built-in variables). There are actions which rely on the value of certain variables but, as we will see, you can also manipulate these variables directly through other actions. What is better, you can define your own variables and use these as well.

For example, in our game we want the plane to be able to shoot only once every five steps in the game (a step is a game tick, ie: one complete cycle of events). So our plane needs a property to indicate whether it can shoot. We will use a variable for this property that we will call can_shoot.

In the Create Event of the plane we set this variable to true and when the player wants to fire a shot we check the value of the variable to see whether this is allowed or not. Whenever a shot is fired we set the property to false.

Next we use an Alarm Event to set the property back to true after five steps.

In a similar way we can use variables to indicate whether the plane has an active shield, whether it has some special weapon upgrade, etc. The next pages will describe this in more detail.

NOTE: a variable name can only consist of letters and the underscore symbol and are case-sensitive, so Can_Shoot is not the same variable as can_shoot.

**Variable Actions**

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

**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 (false), 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. Later, 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 indicated above there are instance variables that "belong" to an instance and there are global variables. When you use your own variables these are always instance 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... As having two variables with the same name will cause serious errors in your game that may be difficult to detect later.

**Scrolling Backgrounds**

A scrolling shooter takes its name from the fact that the game world scrolls across the screen, normally either from top to bottom or from right to left. This gives the illusion of motion, and in the game 1945 the game world scrolls vertically (i.e.: top to bottom). Even though the plane the player controls actually stands still on the screen you get the impression that it is moving as it flies over the scrolling background.
You can control the position of the plane by moving it around on the screen, which will give 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 as this would give the illusion that the plane flies backwards, which is of course impossible.

So how do we create a scrolling background in GameMake:Studio? 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 it by using a view. This view slowly moves upwards over the room. However, for our needs at the moment, we will start with using a moving background and then later we will briefly investigate the second option.

**Begin game steps**

Now that we have covered a few of the details of what is needed to build this game, we can begin creating it.

As our game takes place over the sea we need a background image that looks like the sea viewed from above, so we add the following small image as a background asset to the game and give it the name "bck_water":

![Background Image](image)

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 usual way and then left click on the tab labelled **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 as a color, so uncheck the box labelled **Draw background color**. Second, click on the **Menu** icon and select the bck_water background image. The default setting is to tile the whole room with it so this is what we want and needs no changes. Finally, we need to make the background move. To do this, at the bottom, set the **Vert. speed** to 2. The backgrounds tab should now look as follows:
Set your room size to 640x480 in the settings menu of the room screen.

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

**Scrolling Islands**

To enhance the feeling of motion we are going to add a few islands to the sea. An easy way would be to create a larger background image and add the island to this background which we would then scroll as shown before. The disadvantage of this approach is that the islands will appear in a regular pattern, which the player will soon notice (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**.

First, create three sprites with the following images (you can do this easily by dragging each of the images into the GameMaker:Studio window, then selecting "Sprite" from the options window that opens):
As we will never use them for collision checking (they are going to be a purely graphic effect), it's best to ensure precise collision checking is unchecked. This will improve the speed the game runs at as checking for precise collisions is expensive (i.e.: Uses more processing power) and should only be used when absolutely essential. For each of the islands we create an object and in the **Create Event** we give the object a vertical speed that is the same as the scrolling speed of the background which we have set to 2. 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. Instances of objects are drawn in the order of their 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 (and on top of any backgrounds, which have the highest depth of all). Your objects should look like this:
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 do this, in the Step event of the island 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 where 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 GameMaker:Studio. 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:

We use -65 for the y-position to make sure the island starts completely above the room and because of its vertical speed it will move again into sight of the player. We have to do 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. Your island objects should now look like this:
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.

Place instances of each island in your room and then play again to see your islands scrolling.

The Player 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 which is going to be an old military-style propeller plane. To give the propellers the illusion of rotating we use a sprite consisting of three sub-images that are exactly the same except for the propellers (this image can be found as myplane_strip3 in the assets folder).

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 center of the plane. So whenever we later set the plane at a position or test the position, we mean the position of the center of the plane, not the top-left point, which is the default position of the origin. This is important to make sure that, for example, bullets come from the center of the plane, not from the left side.
Next we add the object "obj_myplane". As its sprite we choose the plane sprite we just created, and we give it a depth of –100 to make sure that it lies above bullets, islands, etc. Remember, 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 should 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 moving outside of the room on either side or the top and bottom. To accomplish this, 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 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).

The complete event and action list will look like this for the player plane:
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. The action events should now look like this:
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.

**Enemies and Shooting**

Now, what is a scrolling shooter if you cannot shoot and there are no enemies? Not a lot really! So, we will need to extend our game with some enemy planes and formations as well as a canon on the player plane that you can fire to shoot them down with...

**The Player Canon**

Let us start with the player canon. We first need a bullet sprite:

To make it a bit more spectacular and obvious, we use a rather large bullet sprite (exaggerating things is often important in games) and we put its origin in the center of the sprite as we did for the main plane.
We then create an object and assign it this sprite. We will give it the default depth 0 such that it will appear below the plane but above the islands. This object has rather simple behavior - in the **Create Event** we give it a vertical speed of –8 to make it move upwards.

To avoid the buildup of more and more bullets as the player shoots, we must destroy it once it leaves the room which is something that easily be achieved. In the **Step Event** we test whether the variable \( y \) 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, as this would make the game too easy.

We are only going to permit 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 true, indicating that we can indeed shoot a bullet.

In the space key event we test whether the variable can_shoot is equal to true. If so, we create a bullet just in front of the plane, relative at position (0,-16). Also we set the variable can_shoot to false indicating that we can no longer shoot and set the Alarm[0] Event to 15.
Remember that an alarm ticks down to 0, with one tick passing in each step. When the alarm reaches 0, the **Alarm Event** is triggered. In this event we set the variable **can Shoot** back to **true**, indicating that we can shoot again.

You can change the speed with which you can shoot by changing the value of the alarm (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 actions in the space key pressed event but this time without the check for the variable).
**Enemy Planes**

Let us now make the first enemy. It will be a small plane that simply flies downwards towards the bottom of the screen. It does not shoot but when it hits the main plane the game will be over. Again we create a sprite for the enemy plane and a new object, then assign the sprite to that 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! If you need help, look back at your island object or go back in this manual and look again.

**Collisions**

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. In the next steps, we will create these collision events.

**Collision with the Player Bullet**

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 the Add Sound icon in the main GameMaker:Studio IDE and load the explosion file from the GameMaker Assets folder.

For the explosion we need another sprite.

- Remember to center the origin of this sprite.
Like for the enemy plane we set the origin to its center (16,16). We then create an explosion object and assign it the explosion sprite. It does nothing except that once the complete explosion animation has played it will destroy itself. For that you should add an event called **Animation End**, which can be found under the **Other Events** and there add a **Destroy Instance** action.

![Object Properties: ob_explosion1](image)

When the sound has been added to the game and the explosion object has been created, we can now add 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 in the collision, which is the bullet in this case.

- Next we create the explosion object, **Relative** to the instance position at (0,0), that is, at the same place as the enemy plane. *We do not destroy the enemy plane!*

- Move the plane 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.
The event should now look like this:

Collision with the Player

Next we need to handle the collision with the main plane. Again we will need an explosion sprite:

And an exploding sound (bit louder this time).

Again we create a new object for this larger explosion, 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 two things:

- We inform the player that he has died using the **Display a Message** action from the **Main2** tab
- Finally, we restart the game.
In the Collision Event of the enemy plane and the player plane, we use the Change Instance action to change the main plane, that is, the other instance, into an explosion.

Also we should play the new explosion sound and destroy the enemy plane. That's it for the basic collisions for now.
The Controller Object

What remains is to place enemy planes in the room but we will do this slightly differently...

A good game becomes more difficult all the time, so we would like to start with just one enemy plane and get more of them over time. To do this, we create one more object, which we will call "obj_Enemy_Control", and it will control the creation of enemy planes. We make the object invisible during the game by unchecking the box labelled Visible, and a sprite is not required for it as it draws nothing.
In its **Create Event** we create an enemy plane at a random location just above the room (random(room_width), -65), and this will be the first enemy in the game. We also set the **Alarm[0]** **Event** to 200 steps. In the event for this alarm we then create another enemy plane and reset the alarm again, but this time to 500. And that's all! The effect is that at the beginning of the game there is one enemy plane, then after 200 steps (about seven seconds), a second enemy plane appears. From now on, after about 15 seconds, a third plane appears and this continues (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). The **alarm[0]** event will look the same as the create event but the time will be 500.

Place **one** instance of the controller object in the room and we are done for now.

Save and test your game now and you will see that we have a playable scrolling shooter game!

**Score, Lives, Health, and HUD**

It is a bit unsatisfactory that the game ends whenever you are hit so, to make the game a bit more interesting we will let enemies do damage to the player rather than destroy it completely with one hit. 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 GameMaker:Studio has built-in mechanisms to deal with score, lives, and health (being the opposite of damage).

To create all this we make a new object called the "**obj_Life_Control**". It does not need a sprite either as we will control the drawing itself using the drawing event.

**Creating a H.U.D.**

As a start we create one big sprite that functions as information panel (or H.U.D. - Heads Up Display).

Create a new sprite and load in the "bottom.png" sprite from the GameMaker Assets folder. Leave the origin at 0,0.

![Score and Damage Panel](bottom.png)

This will show the score, the damage (in the black area at the left), and the number of planes left (i.e.: the number of lives). In the drawing event of the "**obj_Life_Control**" 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 sub-image means that the current sub-image is drawn. As there is just one sub-image in the sprite we don't really care about this, but if a sprite consists of multiple sub-images you can indicate here the sub-image you want to see. To make sure that the bottom panel lies above everything else we give the "obj_Life_Control" object a depth of –10000.

In its creation event, the "obj_Life_Control" object should set the score to 0, the number of lives to 3, and the health to 100. There are actions for all these functions in the Score tab.

To draw the score we use the appropriate action in the Score tab (we should first set the 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.

Create this sprite just as you have in the past and call it spr_life. There is an action for this in the **Score** tab.
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:

![Drawing Event Diagram]

**Checking Health and Lives**

But we still need to actually check for the health and lives values. 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 (the enemy plane) turning it into an explosion and decreasing the health global variable, that is, set the health relative to -30 (so it can withstand 3 hits). The enemy object should now look like this:

![Object Properties: obj_enemy1]
The "obj_Life_Control" 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 and we also reset the health and play the correct sound. So the event looks like this:
In the big player explosion object, in its Animation End, it should destroy itself, then create a new player 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. The player explosion animation end event should now look like this:
Finally, we need to check whether we did run out of lives. Fortunately, there is again an event for this in the **Other** events list. For the "obj_Life_Control" object, in this event we display a message and restart the game (you know how to do this!). The obj_life_control object now looks like this:

![Object Properties: obj_life_control](image)

This finishes another, better, 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.

### Adding More Enemies

In this section we are going to add new types of enemy planes to our game. One will fire bullets straight down, the second plane will fire bullets towards the main plane and 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.

### Shooting Enemy

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.

![Sprite](image)

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 open its object properties ready to change. Give it a new name...
and set the correct sprite. Now, because this is a 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 (relative).

To make the enemy plane shoot, we need a bullet sprite and a bullet object.

This object, in its Create Event, gets a vertical speed downwards of 10 or 12, and in its step event we again take care that the object is destroyed when it moves out of room at the bottom (just like we did for the player bullet at the top). In the collision event of this bullet with the player 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 (control tab) 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 "obj_Emplyy_Control" object again. In the creation event we set Alarm[1] to a value of 1000. In the Alarm[1] Event we create the second enemy plane and set 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. If you have done everything correctly, your obj_enemy2 and obj_enemybullet should now look like this:
Aiming Enemy

For our next type of enemy plane we again need a new sprite:

![Aiming Enemy Image]

We also need a new bullet sprite so that the player can see the difference in a normal bullet and a homing bullet:
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 Create 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’s instance.

This can be achieved easily in GameMaker:Studio. 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. Use this same method to define the y-coordinate. This time you will use `obj_myplane.y`. Give this bullet a speed of 8 to make it a little more difficult to avoid. Note, that with this method if there are multiple instances of the object being used in the room, we get the value of any one of those instances. Since there is only one player plane, this is not a problem for our game but it is something you would want to know when you create your own game.
Be careful with this however, as 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 player 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 we will change the creation event to look as follows:
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 Room Event**. In this event we simply put an action to destroy the object.

**Enemy From Behind**

It remains to add the planes that come from below. We can use yet another new sprite for this:

![Enemy From Behind](image)

Now, to make this enemy work, you just need to duplicate the first *down* scrolling plane, only now make the plane start at `room_height + 16` (which will make it start "beneath" the bottom of the room) (this will be done in the `obj_enemy_control` just as the other enemy planes were done). In the plane’s object, on the create event you will give it a *negative* vertical speed of -4 so that it moves up from the bottom and, the step event should check to see if the y position is less than -16 before re-starting it again.
Spawning New Enemies

Finally we have to handle the creation of the new type of enemy planes. As indicated before we use the "objEnemy_Control" object for this. In the **Create Event** we set **Alarm[2]** to 2000. In this alarm event we create the new type of basic shooting enemy plane and set the alarm 2 again to 1000 to create another one sometime later.

The "objEnemy_Control" object will also create the more advanced shooting enemy and the enemy that attacks from behind in much the same way, using the **Alarm[3]** and **Alarm[4]** event. You should by now understand how to do this, so go ahead and do it! The obj_enemy_control will now look like this with all of the enemies added.
The last thing that remains to be done is to create the collision events of the enemy bullets with the player plane. This will look as follows for both types of enemy bullet.
This new incarnation of your 1945 scrolling shooter 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, which a highly 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.

You can 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 could be a special fire bonus when the player reaches a score of 500 and another one when he reaches 1000 points.

Whatever you do, 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 randomly.

- You could create extra types of enemy planes, for example, planes that fly from the side or fire more powerful missiles.

- How about adding some boats in the water that shoot at you? And a bomb for the plane that can be dropped on them as you can't shoot them...

The possibilities are endless, get creative and make this game your own.
Platformer Game

“Princess Peach”

(Advanced)
Overview
Platform games are very common, in particular on handheld devices. In a platform game you look at the scene from the side. The player normally controls a character that walks around in the world. This world consists of platforms. The player can walk on these platforms, jump or drop from one platform to the other, use ladders or ropes to get to different places, etc. On the platforms there are objects to collect, enemies to avoid or kill (often either by shooting them or by jumping on top of them), switches that can be pressed to open passages, etc. Also the player normally requires skill to jump over dangerous areas. In some platform games you see the whole level at once, but in most you see only a part around the character. In such a case, finding your way around becomes an additional challenge.

Creating a good platform game is not trivial, also not with Game Maker. There are three important aspects to take into account:

- Creating natural motion for the character.
- Creating enough variation in monsters, background, etc.
- Carefully designing the levels such that they are fun to play and get increasingly difficult.

The Basics
In each platform game there are two basic objects: the character that is controlled by the player, and a block object that is used for the floors (platforms) the player can walk on. The same block is often used for the walls that the player cannot pass. We need two sprites: one for the character and one for the block. For the character we use a simple ball. For the block we use a black square. We create two objects. The block object is simply a solid object that has no events or actions. It simply sits there. The character object is a lot more complicated.
Motion

The crucial aspect we treat in this first section is how to define the motion of the character. The problem is that the character must walk on top of the floors. It must not intersect the floor. If the character jumps or falls off a platform it must land correctly on the next platform. There are a number of different ways in which the character can walk, jump, and fall. Different platform games use different modes. Normally we just use three keys to control the motion. The left arrow key should move the character to the left, the right arrow key should move it to the right, and the up key or the space key makes it jump.

Let us first consider the left and right motion. The first choice to make it whether the player can only change its direction of motion while on a platform or also in the air while jumping or falling. Even though the second option is not natural (it is rather difficult to start moving left while you are falling down) we decide to go for the first option, that is, we allow horizontal motion wherever the character is. This tends to lead to nicer game play and is actually also easier to implement.

The second choice is whether the motion has constant speed or whether it accelerates when you keep the key pressed. For simplicity reasons we opt for the first choice. Allowing for acceleration though normally gives nicer game play: the player must for example start a run at a distance to jump over a wide hole.

As you should know there are different ways to let a character move. We can set a speed of motion or we can simply move the character directly. In platform games it is normally the easiest to let the vertical motion be done automatically (as we will see below) but to do the horizontal motion ourselves. This is rather easy. In the keyboard event for the left arrow key we check whether the position at relative position (-4,0) is free. If so we let the character jump to that position. We treat the right arrow key in a similar way. See the enclosed example game.

Jumping

Next we need the vertical motion. This is more difficult. To let the character fall down we can use gravity. But it should stop moving when we hit the floor. Also, you normally want a maximal falling speed, otherwise the character will move too fast. (This is both not very pleasing but it can also cause problems in the implementation. E.g. the character might fall through a floor if it moves too fast.) To solve this problem, in the step event of the character we check whether the position just below the character is collision free. If so, the character is in the air and we set the gravity to a positive value. Otherwise we set it to 0. We also check the variable vspeed which indicates the vertical speed. If it is larger to 12 we set it back to 12. In this way we limit the vertical speed to 12. So the event looks something like this:
Next we have to land correctly on the floor. This is more difficult than it might seem. It will happen when the character collides with the block object. In this collision event we should set the vertical motion to 0. But this might leave the character hanging a bit in the air above the ground. (The reason is that the character is placed back to its previous position before the collision.) To this end we want to move the character to the exact point where the collision occurs. Fortunately there is action for this in *Game Maker*:

**Move to Contact.** With this action you can move the instance in a given direction until a contact position with an object is reached. If there already is a collision at the current position the instance is not moved. Otherwise, the instance is placed just before a collision occurs. You can specify the direction but also a maximal distance to move. You can also indicate whether to consider solid object only or all objects.

We use this action. As direction we indicate the variable direction which is the current direction of motion of the instance. As a maximal distance we specify 12 (although this is not really necessary here):
So the total collision event with the block looks as follows:

You could argue that we should only do this when we hit a floor below us. But actually we also want to move to the contact position if we hit a floor from below or if we hit a wall from the side. There is one important thing here that is often a cause for problems: We assume that the character at its previous position is indeed collision free. You would expect this but this is not always the case. A mistake that is often made is that when the character has an animated image, also the collision mask changes in every step. This could mean that the new image at the previous location still causes a collision. So you better make sure that the character has one collision mask.
Finally we have to let the character jump when the up arrow key is pressed. But this must only happen when the character is currently on the floor. So we first test whether the position below the character creates a collision and, if so, set the vertical speed e.g. to -10. You might have to play a bit with the value of 10 for the vertical speed and the value of 0.5 for the gravity to get the motion you want.

Now the basis for the platform game is ready. Design a level with some floors and walls, constructed from instances of the block object. Place an instance of the character in the room and you are done.

**Better Graphics**

The basic platform game we created in the previous section works but it looks rather bad. There are two aspects we want to change: the way the player looks, and the way the background looks.

**The Character Images**

Let's start with the character graphics. We will use two different (non-animated) sprites: one for the character facing to the left and one for the character facing to the right. The easiest now is to place in the event for the left arrow key an action to change the sprite to the one facing left. Similar, in the right arrow key you switch to the one with the character facing right. It is very important that you switch off precise collision checking for the two sprites. There are a number of reasons for this. First of all, it avoid that the sprite get stuck halfway down the edge of the platform. Secondly, when the sprite is changed from left facing to right facing they should use the same collision mask otherwise the character might get stuck. The same is even more important when using animated sprites. For the same reason you better make sure that the bounding boxes of the sprites are the same. To this end, modify the mask of the sprite and use manual bounding boxes for this. So when adding the sprites the mask properties should be something like this:
In more advanced games you will probably want to use animated sprites. In this case you also need a sprite for the character when it is not moving. Also you might want to add sprites for the character jumping, falling, shooting, etc. In this case you will have to change the sprite at various places in the events. In particular, in the no key Keyboard event you probably want to set the sprite to the not moving one. Alternatively, you can draw the correct sprite in the Draw event based on the situation. For example, you can check whether xprevious<x to find out whether the character has moved to the right. As indicated before, better make sure that all sprites have the same bounding box and no precise collision checking.

The Platforms and Walls
Secondly we want to improve the background and the platforms. Here we use a standard technique. Rather than using objects for all the different wall and floor elements, we use so-called tiles. Tiles are pieces of background images that are drawn at particular places in the room. They do not have associated events nor do they create collision. The good part is that they are fast and use little memory. So you can create large rooms without the need for large images.

To add tiles to your rooms you first need a background image that contains the tiles. Tiles in a background image preferably have a fixed size and have a little (1-pixel) border between them such that they can easily be separated. A small tile set is provided in the Resources folder. We add it as a transparent background resource named back_tiles. When adding it to the game, in the background properties indicate that it should be used as a tile set and fill in the correct tile size and separation, as follows:
Now, when creating a room, you can click on the **tiles** tab page. You can select the tile set (that is, the appropriate background resource). Now you can draw tiles by clicking on the appropriate tile and next placing them in the room, like you would do for objects. The right mouse button deletes tiles. Use your imagination to create challenging rooms. (Note that you can place tiles on different depth layers by adding layers. For example, you can make a layer of tiles that lie in front of the moving characters. We will not use them here but they are great for giving a better 3D effect.)

There is a problem left though. As indicated above, tiles are just nice graphics. They do not generate events or collisions. So the character would fall straight through them. To avoid this we still need the block objects we had before. We place the block objects at the appropriate places on top of the walls and platforms you did create with the tiles on the background.
Now by making the block objects invisible you will not see the black blocks but the beautiful tiles. But the block objects are actually there, so the character cannot pass through the walls and will land on the platforms.

There might be one problem here. The 16x16 block objects will be too large to cover the background nicely. So we want to make a few other block objects of size 16x8 and 8x16. Again we make them solid. To avoid having to specify collision events with these as well, we use the parent mechanism. This is a very powerful mechanism that you should learn to use. If an object A is a parent of object B, B behaves as a special case of A. It inherits all the behavior of A (unless you overwrite this with other behavior). Also, collisions with B are treated the same as collisions with A. So for the smaller blocks we set the parent to the bigger block. In this way they will be treated the same as the bigger block.

**Threats and Treats**

Just jumping around from platform to platform is rather boring. You definitely need some more challenges and goals. In this section we treat a number of these.

**Monsters**

Let us first add some monsters. We will make two monsters, one that moves left and right on a platform and the other that flies left and right in the sky. Jumping on top of it can squash the first one; the second one should be avoided at all times.

Let's start with the monster that moves on the platforms. We need two sprites for it, one with the monster facing left and the other with the monster facing right. Again, better don't use
precise collision checking for the same reasons as indicated above and pick some relevant bounding box. Now we create the monster object. In the creation event we let it move to the right with a particular speed. Whenever it hits a wall it reverses its horizontal speed. To set the correct sprite for the monster we use the End Step event. This event happens just before the instances are drawn. In it we set the correct sprite based on the value of the variable hspeed that indicates the horizontal speed. If it is smaller than 0 we let the monster face left and otherwise we let it face right. (If you have the Pro Edition of Game Maker you can also use the action to mirror the image. In that case you need only one sprite.)

To avoid monsters from falling off platforms, we introduce another object, which we call a marker. This marker will be an invisible blue block. Whenever a monster touches it, it reverses its direction of motion. Having invisible markers is a good general trick to let instances perform certain actions at particular places in your room. Besides changing direction you could use markers to shoot, to lay bombs, etc.

When the character hits a monster, the character should die. But actually, as in most platform games we like to make it possible for the character to jump on top of the monster and squash it. So in the collision event of the character with the monster we must check whether we hit the monster from above to squash it. To find out we perform the following test:

\[ \text{vspeed} > 0 \&\& \text{y} < \text{other.y} + 8 \]

It is true if vspeed is larger than 0, so the character moves downwards, and the character is close to the top of the monster so it is indeed hitting it from above. In this case the monster must be destroyed. (In the example we turn the monster into a flat dead monster, which destroys itself after a while. This gives a nicer graphical effect.) In this simple platform game, dying for the character corresponds to restarting the level, which can be achieved by some simple actions.
The flying monster is even easier. We proceed in exactly the same way. Only, in the collision event of the character with the flying monster, no test needs to be performed because you cannot squash a flying object.

You might want to add some more monsters, e.g. with different speeds, to make things harder. You can also make a monster or rock that falls down or moves up and down. Just use your own imagination.

**Pits**

Most platform games require careful timing of jumps to avoid falling into pits. Falling into a pit normally kills the character. To this end, we add a new object, called death. This object is a red block that again is not visible. You can place it at the bottom of the pit. (In the tiled room you can put some spikes there.) In the collision event of the character with the death object it should play a sound, wait a while, and restart the room. You can also make pits that go down infinitely. In this case you want to add similar actions in the **Outside Room** event (in the other events) of the character, maybe including a test whether \( y > \text{room\_height} \) to make sure the character fell down, rather than jumped up outside of the playing field.

**Collecting points**

Most platform games have some mechanism in which the player can collect points. Normally you have to pick up certain objects or catch certain things. In our example the player can collect mushrooms. So we make a mushroom object. To give a bit of variation, the mushroom sprite contains 10 different mushrooms sub-images. The mushroom object picks one at random upon creation using the **Change Sprite** action:
We set the sub-image to random(10). random(10) is a function call. It will return a random number below the argument given (so below 10 in our case). We set the speed to 0 to stop cycling through the sub-images. In the collision event of the character with the mushroom object we play a sound, destroy the other object (that is, the mushroom) and add 10 to the score.

In some platform games, collecting things has a more important function than just raising your score. For example, you might get an extra life when you collect enough objects. Also there might be objects that restore your health (assuming monsters don't kill you but simply weaken you), make you move faster, jump higher, etc. These can easily be added.

Next level
Of course there should be a way to finish a level, such that the player can move on to the next level. To this end, we create an obj_levelexitobject. When the character gets there you are moved to the next level. In the example this is done rather simple. We add a test action to see whether the next room exists. If this test is true we move to the next room. Otherwise the high score list is shown and the game is restarted.

You might choose to make the level exit only appear when for example all mushrooms have been collected. To this end, in the creation event of the obj_levelexitobject, move it to a position – 100,-100 (so off the screen). Now in the step event of the object we check whether the number of mushroom objects is equal to 0 (there is an action for this) and, if so, move the object back to its starting position (again there is an action for this). All very simple.
More Motions

Our current platform game has just some limited motion possibilities. The character can move left and right, and it can jump. To make things more interesting, let us add some possibilities.

Ramps

It is nice if the player can walk up sloping ramps (down goes automatically because of the falling). To this end, we have to replace the code in the left arrow key event. We put there the following:

Rather than just testing whether the position to the left is collision free we also test whether a position 8 pixels higher is collision free. If so we move the character there and use the landing action to move it down to the contact position. So the event will look as follows:

![Object Properties: obj_character](image)

The right arrow key is handled in a similar way.

Ladders

People always want ladders in platform game along which the character can move from one platform to the other. This requires a bit of work. A ladder will be represented by a thin vertical block that is invisible (the real ladder or vine or whatever that is used for climbing is drawn again using tiles) and not solid. When the character is not in contact with a ladder, motion should be as before. But when it is in contact with the ladder things must go different. First of all, the character should not fall down. So in the step event we have to make a change to this effect adding some actions that set the vertical speed and gravity to 0 when in contact with a ladder. Also we set the sprite to the climbing sprite in that case.
The second thing that needs to change is the event for the up key. When the character is at a ladder, the up arrow key should move it up, rather than jump. Again we need a few additional actions for this. We test whether the character is in contact with a ladder and, if so, move it up a bit. We use similar actions for the down key.

**Using a View**

Up to now we always showed the entire room. For many platform games this is not what you want. Instead you want to see only a part of the room, around the character you are controlling. This makes the game more challenging because the player must try to detect his way through the platforms. You can also hide prizes at difficult to reach places in the room.

Fortunately this is extremely simple to achieve in *Game Maker*. When designing the room, click on the **views** tab. Click on the checkbox **Enable the use of Views** to start using views. Select the first view and check the box **Visible when room starts** to make sure this view can be seen. Give it a width of 300 and a height of 200 (or something else that you like). (As we are going to let the view follow the character there is no need to specify the left and top position of the view in the room. Also, because we use just one view, we don’t have to specify the x and y position of the port on the screen.) At the bottom we can indicate which object to follow. Here we choose the character. The view will now automatically move to keep the character in focus. We don’t want the character to get too close to the border. To this end we set the **Hbor** and **Vbor** values to 64. There will now always be a 64 pixel area visible around the character. Finally, to get a smooth view motion we set the maximal view speed to 4. (This also gives a very nice effect at the start because the character comes slowly into view.) So the settings will look as follows:
Having the view is nice but it makes the window in which things happen rather small. To avoid this, in the **Global Game Settings** we indicate a fixed scale of 200 percent. Clearly you can play with these values to get the effect you want.

**Some Further Touches**

**Shooting Monsters**

The next step is to enable the player to shoot monsters. To make things a bit more interesting, the player first needs to find some ammunition to be able to shoot. To this end we introduce a variable we call `ammo` that indicates how much ammunition the player has. In the **Create** event of the character we set this to 0 using the action to set a variable. The ammunition object has a simple sprite and does nothing. It just waits to be picked up by the player. When the character collides with the ammunition object we add 10 to the variable `ammo` (set it relative to 10) and destroy the ammunition instance. Next we need a bullet object. When the player presses the <Space> key an instance of this object must create, assuming there is ammunition, and the value of the variable `ammo` is decreased by 1. But there is one important issue. We like the bullet to shoot in the direction the character is facing. To this end we check the value of variable `sprite_index`. This variable contains the index of the sprite for the character object. Based on it we create a bullet with the correct direction of motion. When we are climbing no bullet is created. (Shooting while climbing is not possible.) So the space event looks as follows:

![Object Properties: obj_character](image)

It remains to destroy the bullet when it hits a wall or when it goes outside the room and to kill the monster when the bullet hits it.
A Score Panel
A player now has a score and ammunition. We are also going to give it some lives. Hitting a monster or falling in a pit will cost a life. There is an easy mechanism for lives in Game Maker. We are going to create a special obj_controller object. It does not need a sprite. In its Create event it sets the number of lives to 3. Whenever the player dies we decrease the number of lives. In the No More Lives event for the controller we show the high score table and restart the game.

But it would also be nice if we can see the number of lives, the score, ammunition, etc. To this end we are going to make a little panel with this information. We are going to draw this in the Draw event of the controller object. There is though a problem here. Where should we draw it? We cannot draw it at a fixed place in the room because the view changes and we want the panel always in view. Fortunately we can ask for the position of the view. This is indicated by the two variables view_xview and view_yview that indicate the left position and top position of the view respectively. So we can draw the panel with the information relative to this position. Here is what the draw event of the controller object looks like:

![Object Properties: obj_controller](image)

Note that we also draw an image when the player can shoot.
What next?

The sections above have explained some of the basics of making platform games. Now it is your turn. You will have to use these techniques and some more ideas of yourself to create a real nice platform game. Remember that the most crucial part of platform games is formed by the levels. Start making levels one by one. Play them until you are happy with them. Every so often, introduce some new game play aspect. Here are some additional ideas that you can use:

- Different monsters, e.g. bouncing balls and monsters that shoot
- Keys that you need to find in order to open doors
- Mines that you can place somewhere and that go off when a monster (or yourself) steps on them
- Water to swim in (this will completely change the motions; no gravity anymore, or a mild upwards gravity until you reach the surface, limited time before you run out of air, air bubbles to grab, etc.)
- Walls and floors you can destroy, e.g. by shooting them or jumping on them with force
- Trampolines that make you jump higher
- Platforms that appear and disappear
- One-way streets
- Moving platforms (this is not easy!)
On Your Own

If you have completed the other tutorials in this manual, then at this point you should have a good understanding of how to create a game using Game Maker Studio.

Using the skills that you have developed in the other sections of this manual, it is now time for you to build your own game. Maybe you want to build off of one of the examples and change it up to make your own, or maybe you want to start from scratch. Refer to the other sections in this manual to add any features that you want in your game.

Maybe you want to try to make your own Mario game using ideas from the platformer. Or perhaps you want to make a PacMan type of game with what you learned in the maze game. Your game can be whatever you want it to be, just remember to have fun!