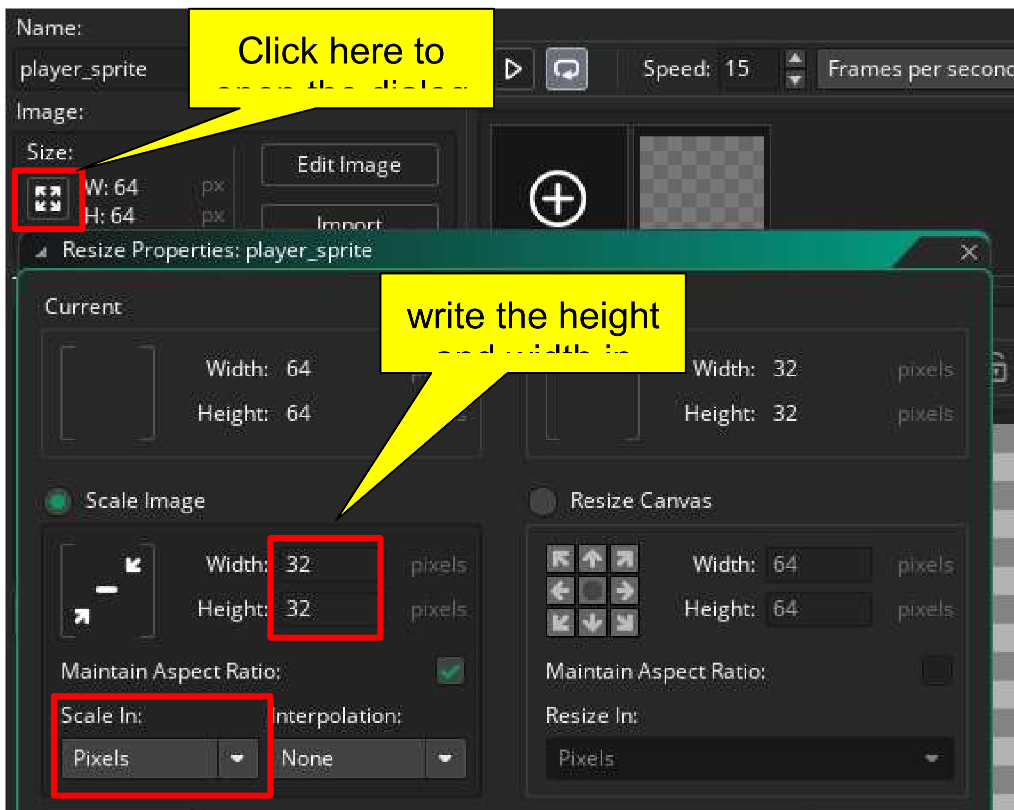
3.2 Platform game

Platform games have been very popular through the years. For the most part, they are linear games that only require skill in moving the character, shooting, jumping and so on along a 2-dimensional world. For this activity, we'll go directly to build a minimum playable level without decoration, which we'll be added later. The player will be able to move one character with just three keys, left, right and space, which will make the player character jump.

We'll start by creating two sprites made from a solid color rectangle of 32x32 pixels instead of an existing image. The first sprite will be used for the player, while the second will be the brick from which we'll build the floor and walls.

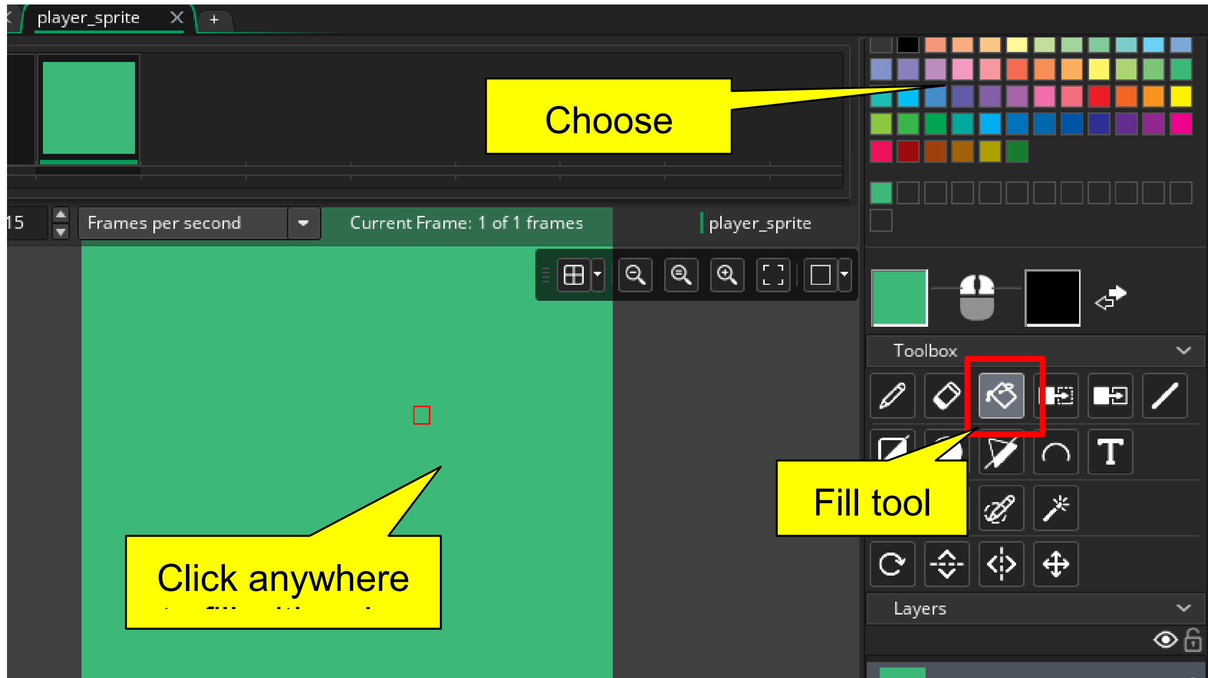
The dimension of the sprite was set in the previous activity by the imported image, but now we should set our dimensions first as there's no image to import. In the sprite properties there's a button to open a dialog which will be used to specify the height and width in pixels



The sprite has now the right size. Set the center of the sprite to the center of the image just as we did in the previous activity. To get a solid color sprite we need to edit the image and this is where the integrated image editor comes in handy. Click the "Edit Image" button in the

Co-funded by the Erasmus+ Programme of the European Union
The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

sprite properties dialog. This will open another tab with a set of conventional image editing tools. From it, we're just going to need to fill up the whole image with the chosen color.



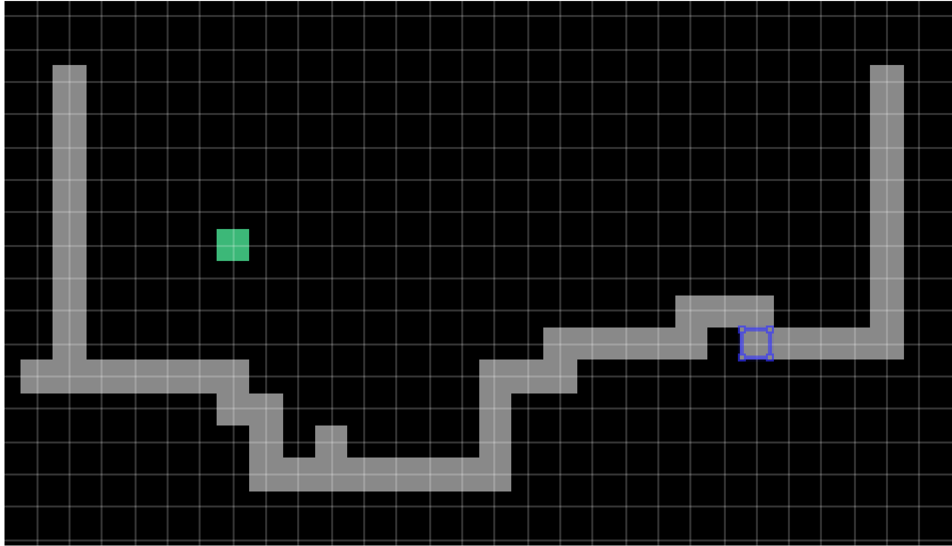
When you are done with this dialog, simply close it and the sprite will be updated with the changes. As both sprites are so similar, just duplicate the sprite created for the player, edit it, change name, enter the image editor and fill it with another color.

From these two sprites we'll create two objects, "player_object" and "brick_object", selecting the appropriate sprite for each one.

We can now create a level by placing "brick_objects" in room0 (the default room). Place one brick_object next to another forming a floor bounded by two vertical walls also made from "brick_objects". To speed up this process, you can get help from the Alt key.

The floor doesn't have to be totally plain. In fact, obstacles, holes and steps are required for a funny game. It is also the right time to add our player_object to the room. Place it anywhere over the floor. The final look should be something like this:

Co-funded by the Erasmus+ Programme of the European Union
The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



The next steps are to add behaviour to the player object, allowing it to move, fall and jump. and also to collide with the wall and floor in a way it can not penetrate it. Therefore, we need to add an interaction between the player object and any brick object. For this activity we'll take a different approach from the previous activity. The main part of the behaviour will be set up in the *step* event, whereas in the spaceship game, there were many events with simple actions and the game logic was split amongst them. Here, the player object is going to periodically evaluate the situation and take actions relaying mostly on the step event. But before the game starts, the player object has to be initialized with some variables that hold values for:

- Horizontal and vertical speed
- Gravity
- walk speed

These are set up in the create event for the player object, and the code is

```
hspeed = 0 ;            // horizontal speed
vspeed = 0;             // vertical speed
gravity = 0.1           // gravity constant
wlk_speed = 4;    // walk speed
```

These four variables will control the movement of the player. Of the four, three are special built-in variables that belong to every object and have a special or a predefined meaning. By setting values to built-in variables, the game engine will change the object state depending on the value of the variable. Here, for example, vspeed and hspeed are set to 0 and this means the object is not moving, but gravity (which also is a built-in variable) is set to some positive value, whose effect is to accelerate the object downwards. If you run the game now, you'll see how the object falls freely until it disappears from the room. The last variable just

Co-funded by the Erasmus+ Programme of the European Union
The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

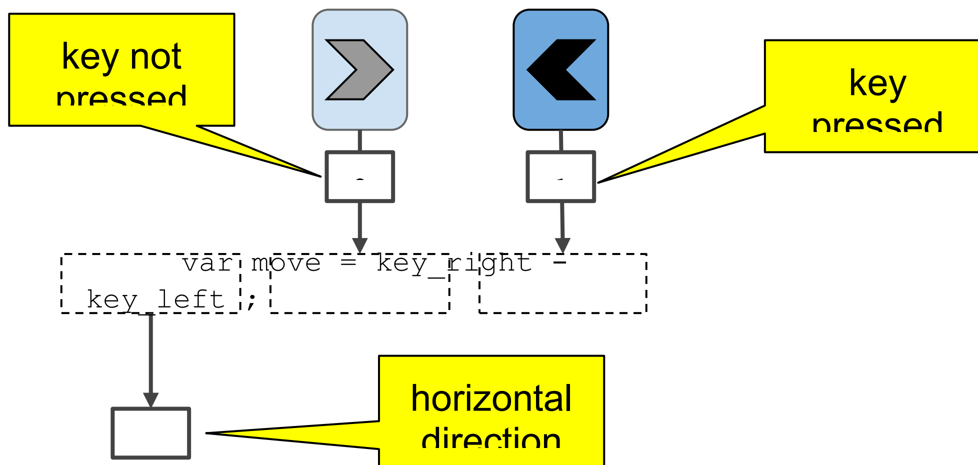
holds the speed at which the character will move in the game. It's not a built-in variable but just a "programmer's" variable.

Let's move on to the step event where there are many thing to check and do. The first is to check the keypresses in the keyboard to apply changes in the movement, speed or position of the player.

```
key_left = keyboard_check(vk_left);
key_right = keyboard_check(vk_right);
key_jump = keyboard_check_pressed(vk_space);
```

The function keyboard_check() returns true or false on the specified key, depending on the actual status of the key. The above lines will read the keyboard for the three important keys in our game: left, right and space. We're not interested in any other key.

The next step is to make some calculations for the speed, taking into account the keys pressed. Often, programming consists in doing some mathematical trick or simplifications that may be difficult to grasp at first glance. Here the trick to get a horizontal speed from the keys pressed is to subtract the `key_left` value from the `key_right`. This yields a 1, 0 or -1 depending on the value of keypresses, which directly sets the direction of the movement (1: right, 0 no movement and -1 moving left). The direction is held in a variable called "`move`"



Next, we apply that value to set a horizontal speed based on the walk_speed value, which here acts as a factor for the speed.

```
hspeed = move * wlk_speed;
```

At this moment, a new speed has been set. Run the game and see that the keyboard left and right keys control the player movement. The next step is to check if it is going collide with anything. If so, we'll set its horizontal speed to zero. In the first example in this tutorial we used "collision events" but here we need a finer control of the collision. Therefore we will

Co-funded by the Erasmus+ Programme of the European Union
The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

ask the game engine if there's some collision either side or below the object separately. For the horizontal collision, we are going to see if there's some brick object next to the player character in the direction we are moving. We get a response by calling a special function whose specification is as follows

```
place_meeting(x_pos, y_pos , other_object)
```

Where "x_pos" and "y_pos" are the position where we need to check for collision and other_object is the class of objects we might collide with. Given that the variable move holds the movement direction of the object, a possible solution is

```
if (place_meeting(x + move, y , brick_object)) {
      hspeed = 0;
}
```

Actually we are not asking if there is a collision where the player is but just a bit left or right (depending on the value of "move"), so we avoid the collision just before it happens. What we do then is to set the horizontal speed to zero. Again, x and y are a pair of built-in variables that we can check anytime. These hold the position of the player object. place_meeting will take into account the sprite dimensions to verify if any pixel of both objects is overlapping to return true.

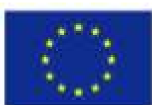
For the vertical movement, we want to check again if there's brick just below the player:

```
if (place_meeting(x , y + 1, brick_object)) {
     gravity=0;
     vspeed=0;
}
```

Notice that we need to ensure that the gravity is also zero, otherwise the game engine will increase the vertical speed on its own. But this solution introduces one small problem. Once the gravity is set to zero due to a collision, it will never change again. What we do is to set it again, and always to its original value just before this code, so if there's no collision it will remain active. Modify the above code slightly:

```
gravity=0.1;
if (place_meeting(x , y + 1, brick_object)) {
     gravity=0;
     vspeed=0;
}
```

Notice that we need to see if there's a collision before it happens, therefore we add 1 to the vertical position to check. That is, one pixel below the character



Co-funded by the Erasmus+ Programme of the European Union
The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

The jump is achieved by speeding the object vertically in the "up " direction, which is really a negative vertical speed. So the simplest solution may be

```
if ( key_jump) vspeed = -4;
```

The gravity will gradually pull the player down. However, the code above would result in a flawed game control, as the player could continously make the character move up by holding the space key always pressed. The jump action requires the player character to be on a brick, stepping on the floor. So we add a condition to the above code

```
if ( (key_jump) &&  (place_meeting(x,y + 1, brick_object))) {
      vspeed = -4;
}
```

The game is already fun to try, and the player can completely move the character through the room.

As for the decoration, there are many options to get a nice looking game. But, probably, the main appealing visual effect are animations. Here the character should be changed from the ugly greenish square to a nice looking character, whose side movements should be animated to resemble a natural walk movement, as shown in the sprites section of this tutorial.

What we need is to draw a set of characters in different positions. However for this tutorial, we'll be using again the free resources from opengameart.org, which will save us some time.. And the file which we'll practise with is located in the following url

https://opengameart.org/content/various-walkcycle-8-characters

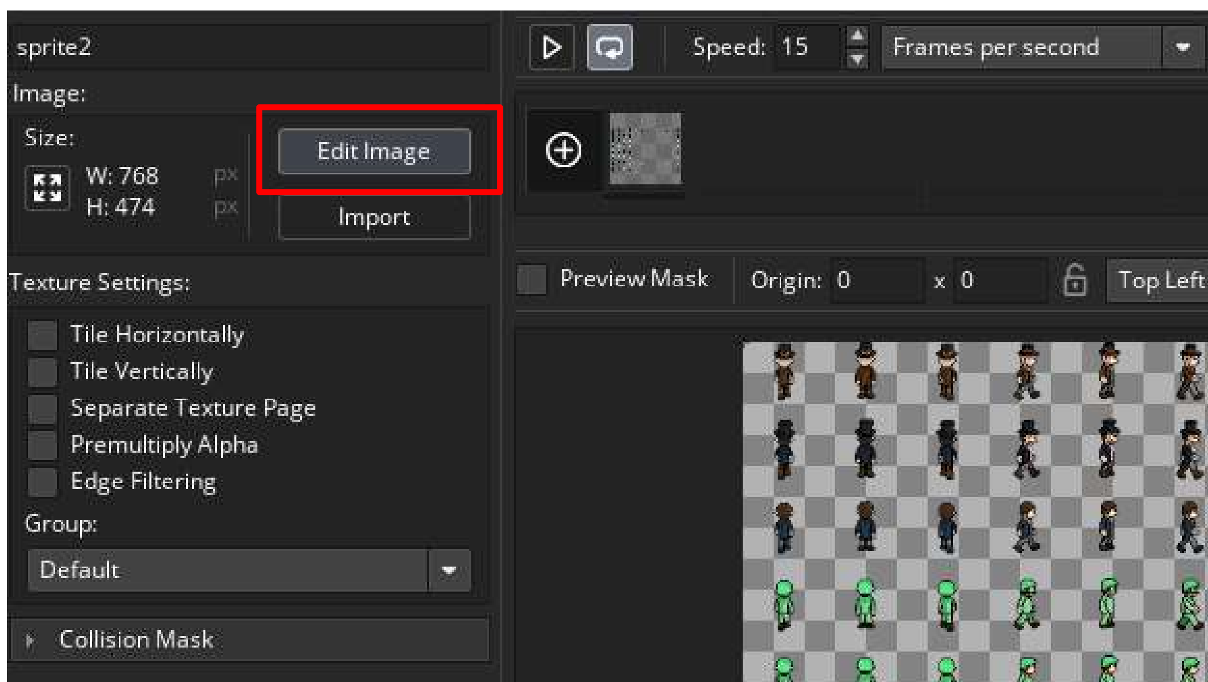
If you open the image and behold it, you'll see several different pictures of characters, each one in different positions or cells placed in a row and column layout. The following image is part of that, which is much larger

Co-funded by the Erasmus+ Programme of the European Union
The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

For this tutorial we'll be using the first character, which resembles a Lord. The good thing about this image is that we'll have to go through all the main usual steps to get an animated sprite:

1. Split the image
2. Select the desired images and discard the rest
3. Verify the correct order of the subimages
4. Make a duplicate of the sprite and reverse the images to get the animation in the opposite direction

First of all, we need to create a new sprite called "walk_right_sprite" from an existing image and later we will split the big image into subimages. Follow the known procedure to create a sprite from an existing image, then enter the edit window using the "edit" button .



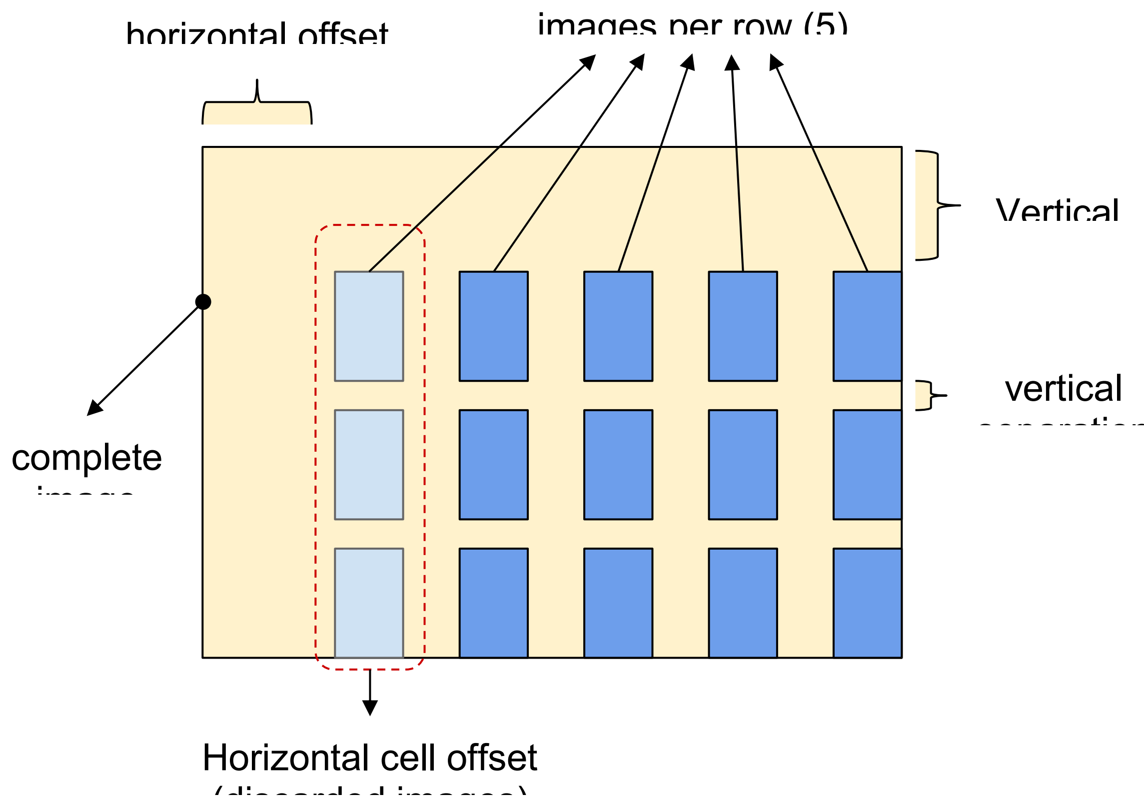
Editing the image will take us to the built-in image editor we used before to paint the squares. Here new menus appear in the menu bar. Locate the "Image" menu and click in the "Convert to Frames" menu item. What we face now is a special dialog made to adjust how the image is splitted, into how many sub-images, depending on how rows and columns and with some offset. It's very likely that the right result is got after several trial and error attempts. To easy this process, the tools always shows a grid which reveals how the image will be split, allowing the user to instantaneously view the effect of his choices, and to understand the questions on the left side of the dialog which have the following meaning

- Number of images: This is the total number of sub-images that the sprite contains.

The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

- Images per row: How many images each row has. Bear in mind that these two first questions also answer the "How many images for each column"
- Image width and image height: The width and height, in pixels, of each sub-image.
- Horizontal and vertical cell offset: Sometimes, the firsts rows or columns of the image doesn't contain any sub-image. We may specify an offset to discard any of these
- Horizontal and vertical pixel offset: Same as above, but this time the discarded sections are measured in pixels.
- Horizontal and vertical separation: There can be some separation between adjacent cells. This measurement is made in pixels.

The following figure helps to grasp the meaning of the questions above.



For the image used in this example, the needed values are:

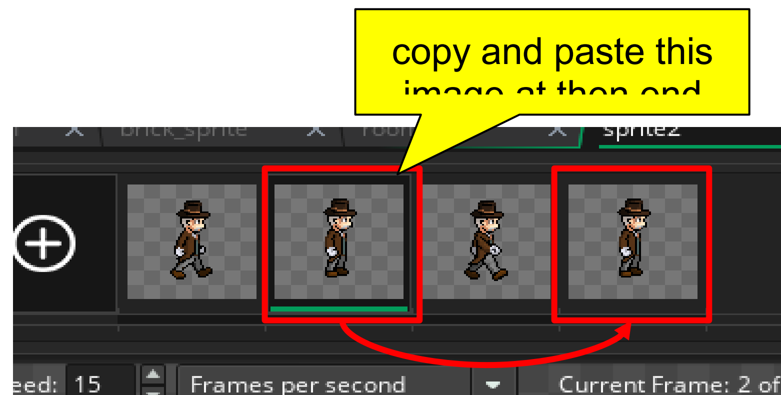
- Number of Frames : 96
- Frames per Row : 12
- Frame Width: 64
- Frame Height: 48

Co-funded by the Erasmus+ Programme of the European Union
The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

- Vertical Separation: 13

All other values are set to 0.

Once you click the "Convert" button, the sprite is transformed and all the sub-images are assigned to it. Most of them are not of our interest. Hence, select every discardable image from the list. Once all of the unwanted sub-images are selected, we delete them leaving just the following three ones (notice that one of the following sub-images has been copypasted and it's repeated. It is placed at the end)

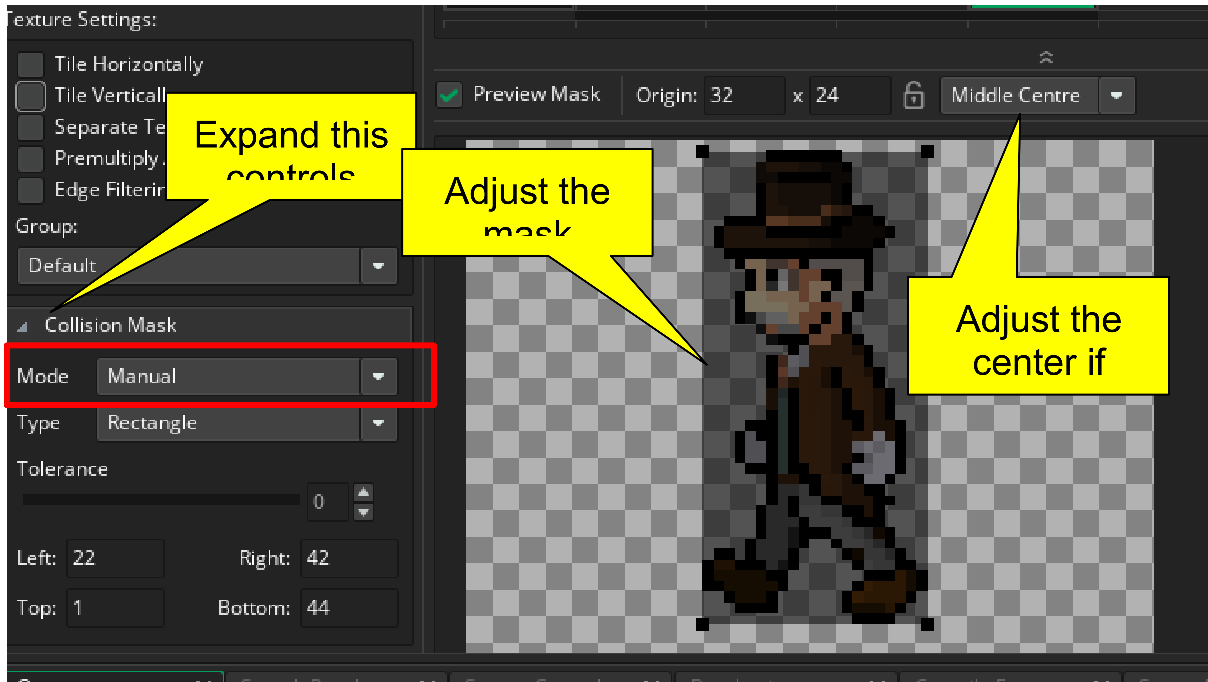


If everything is ok, then we may already click the "animate" button (the one with a triangle in the sprite dialog) and see the animation. Ideally, the work is all done on this sprite but we must have another sprite with the same animation in the opposite direction.

We can do this by duplicating the "walk_right_sprite" thus creating a new sprite which we'll name "walk_left_sprite". Again, we go to "Edit" the image and this time we find the "Mirror" option under the "Image" menu. Inside it, select the "All frames" Submenu. All images have been mirrored (changed from left to right) and the animation now produces a "walk_left" effect.

Before closing the dialogs, we still need to tune one important thing. The character doesn't expand to all the image background in the sprite, and the game engine might produce unrealistic collisions. What we need to do is to set a well adjusted collision mask. To do so, expand the "Collision Mask" controls and set the collision mask type to manual. Then move the points to shrink the collision box around his trunk and head, leaving arms and part of the hat and feet out of the collision box. Check that the center is "Middle Centre" as well.

Co-funded by the Erasmus+ Programme of the European Union
The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



At this moment, we can replace the ugly greenish rectangle sprite by any of these new two in the player_object sprite property and test the game. To run into problems (and solve them) set the collision mask of the object to that of the sprite you have set a shrunk collision mask.

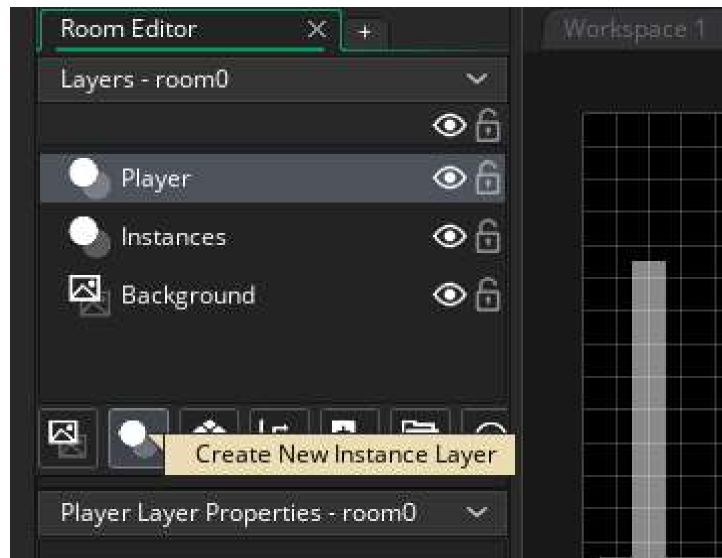
You may notice some issues when testing th

- Part of it is (or it might be) drawn behind the walls.
- It is continuously animated, even if it is not moving, and too fast.
- It doesn't flips sides when changing direction.

To properly draw every object in the room, the game engine needs to know what's above what, in the room. For this to be known, objects are assigned or deployed in "layers", and the layers are sorted from top to bottom. The game engine will draw every object on one layer before starting drawing any object in the next layer. The trick here is then to create a new layer, and put the player object in this new layer which is drawn above the default layer which holds the brick objects.

We create a new "Instance Layer" and name it "Player". This is done in the left side panel which is name "Room Editor". Once it's done, we select the previous default "Instances" Layer, delete the player object from it, select again the Player Layer and place the object in the same position but in this new layer.

The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.



As for the animation, we're not going to get a perfect animation in every situation (jumping, moving sides, stopped, etc), but we're going through the necessary steps to get the know-how on this. Every change in this steps discovers some idea that can be included in the next steps. At any moment, the game can be tested to better understand the changes introduced. This is what we're going to do:

1. Reduce the animation speed.
2. Stop the animation if the player is not moving
3. Stop the animation if jumping
4. Change the sprite shown depending on the direction of the movement.

To reduce the animation speed, we only need to set a value to the "image_speed" built-in variable, which holds a multiplier of the default speed for animations. Therefore, if we set it to 0.5, we'll be reducing the animation speed to half. And this is just the right value there. This speed is going to be applied always (even if no animation is occuring), so we could just add the following line at the beginning:

```
image_speed = 0;
```

To stop the animation if the player is not moving we might set the "image_speed" variable to 0, but then we are not sure what sub-image would be shown, so our method will be to set one precise subimage within the actual sprite. This is achieved by setting the "image_index" built-in variable. In our case, to the first sub-image, which has the value of 0. Basically we need to check if hspeed is 0, this will tell us we're standing still. Therefore the code to apply this ideas is:

```
if (hspeed == 0 ) image_index = 0;
```

Co-funded by the Erasmus+ Programme of the European Union
The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.

When jumping, the character may be moving either side, but animation should not occur (though it is actually funny if it does walk while in the air). The action is just the same as above, but the condition is another, so may combine both conditions into one action.

```
if ( (hspeed == 0 )  ||
      ! (place_meeting(x,y + 1, brick_object)))
      image_index = 0;
```

The code above is read: "if the horizontal speed is zero (therefore standing quiet) or it is not on a brick (but "in the air"), then don't animate (by always showing the first sub-image).

Finally, as for the direction, we have to change the sprite (as if we had selected another from the object properties) and match the direction of the movement. The sprite shown by an object can be changed by setting the built-in variable "sprite_index" to the name of the sprite. To know which sprite to show, check the value of hspeed against 0. Positive values means moving to the right, otherwise to the left. Summarizing, the following code would be added at the end of the step event in the player object.

```
image_speed =0.5;
if ((hspeed == 0 ) || !(place_meeting(x,y+1, brick_object) ))
    image_index=0;
if (hspeed > 0) sprite_index = walk_right_sprite;
if (hspeed < 0) sprite_index = walk_left_sprite;
```

Mind that we do no do a "if else" statement, because we want to leave the sprite unchanged if we are standing still. If you test the game and you find differences between the collisions and position of the two directions, please verify that the the "middle center" is selected as the origin of the sprites and the two collision masks are approximately the same in both sprites.

Co-funded by the Erasmus+ Programme of the European Union
The European Commission support for the production of this publication does not constitute an endorsement of the contents which reflects the views only of the authors, and the Commission cannot be held responsible for any use which may be made of the information contained therein.