

Lecture 4 - A platform game

Written by Carl Gustafsson, heavily based on tutorial by Mark Overmars

Goal of the lecture

This lecture should give the reader knowledge about how to construct platform games. This includes tiled backgrounds, gravity, scrolling views and shooting.

This lecture was revised for round 2 of the Game Maker programming course at www.gameuniv.net. Changes to the original document are shown with **slightly greenish background**. If you read this document for the first time, just ignore those markings and read it as if nothing was marked.

This lecture was also revised for round 3 of the Game Maker programming course at www.gameuniv.net. Changes to the previous revision of the document are shown with **slightly blueish background**. People reading this document for the first time could ignore the different background colors. Most screenshots are revised, but the change is very little, so they are not marked.

Introduction

We are now going to create a platform game. In a platform game, the game world is viewed from the side. The player can walk around the world on platforms that are at different heights from the ground. It is possible to jump between platforms, to fall off them, and to climb ladders or ropes to other platforms. In the game world there are often objects to pick up and enemies to avoid. There could be triggers that open doors or trigger explosives. There could be keys that need to be picked up before certain doors open. In some platform games the entire level is not visible at once, it requires that the player moves so that the view of the world scrolls to reveal more parts of it.

All this, and more, of course, can be done with Game Maker. I hope we will be able to go through most of the above during this lecture.

As with the previous lecture, this one is based upon a tutorial by Mark Overmars. I am grateful that I can use those tutorials as a base for these lectures.

With platform games there are three important aspects that should be considered:

- Natural motion for the main character
- Interesting monsters and objects
- A good level design with increasing difficulty.

This looks a lot like the maze game from the previous lecture, except for the higher effort that is put in character motion.

Just as with the maze game, we split this game up in different versions and try to add more and more features to it between each version.

Platform 1

The first version of the platform game will contain only two different objects - the player and the blocks that build up the ground and platforms.

Create a sprite for the block. It should simply be a 16 x 16 pixels sprite that is completely black. The size of the sprite can be set in Game Maker's sprite editor (**Transform->Resize Canvas**). Remove the transparency from the sprite.

For the player sprite, let us just use a ball (**ball.gif**). Let the sprite be 32 x 32 pixels large.

Then create two objects. The block object should simply be made solid, no other changes need to be made to it. The player object (ball) on the other hand needs some behavior.

The motion of the player object is, as said before, very important. Motion is usually controlled using three of the cursor keys; up, left and right. The player should be able to move along the floor and the platforms

without falling through them or getting stuck in them. When moving outside a platform, gravity should have its course, and the player should fall (usually downwards). The same gravity should kick in when the player jumps. As soon as the player collides with a platform though, the falling should stop. It is usually good to put a constriction on the falling speed so that it does not exceed certain limits. That is because a computer looks at the time as a number of discreet intervals, and if the speed of an object is too high it might move so far in one time interval that it simply moves through an obstacle without "touching " it.

Left, right, left, right

We can start out with the left and right motion of the player. Firstly, should the player be able to move left and right only when standing on a platform, or when in the air too? Actually, if you look at it, most platform games let the player control horizontal motion while in the air too, even though this breaks the law of nature (for which there fortunately is no punishment). My suggestion is that we follow the flow. Let the player control the horizontal movement when in the air too. Actually, that is easier to implement and usually makes the game more playable.

Secondly, should the movement be acceleration/deceleration or should it be using constant speed? Both types are common in platform games. Let us go for the constant speed option.

Now, there are many ways to make an instance move. We could for example set a speed for it, just as in the maze game. However, for the purpose of learning, and because it probably is easier in platform games, we are going to make the instance move horizontally through directly changing its coordinates.

So, in the **keyboard <Left>** event, check if the position at coordinates (-4, 0) (Relative) is free, using the **If a position is collision free** event. If it is, have the instance "jump" to that position (**Jump to a given position**). Just to remove misunderstandings: When I write coordinates as **(-4, 0)** I mean that **x = -4** and **y = 0**. Do not interpret the comma character as a decimal point. It is very rare to refer to pixel coordinates with decimals, even though it theoretically is possible. Also, often when the coordinates are close to **0**, they should be **Relative** to something (like the position of the instance in question). So, for movement coordinates you should usually use the little **Relative** box.

*(Side note (may be confusing): If you uncheck the **Relative** box, you could obtain the same result if you enter **x - 4** for x coordinate and just **y** for y coordinate.)*

Do similar with the **keyboard <Right>** event. (But use **(4, 0)** for coordinates instead).

Up, up and away

The jump key (up) is not much harder. Basically one would only need to give the player a bit of vertical speed upwards. Well, the **start** of the jump is not much harder than that, so let us concentrate on that at first. Before we let the player jump we should check if the player is standing on some ground or on a platform. If not, the player should not be able to jump, right?

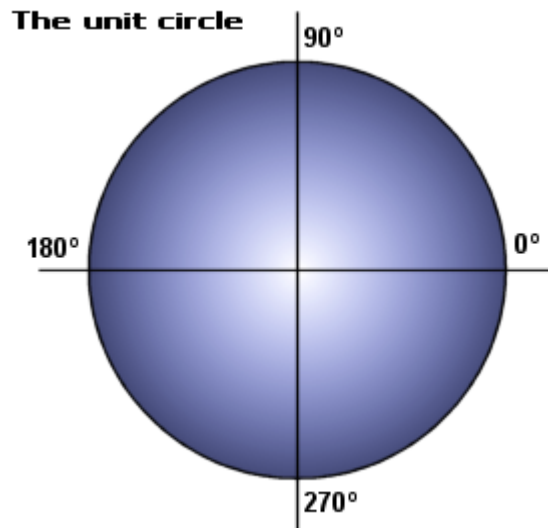
First, to the **<Up>** key, add the action **If there is a collision at a position**. As the coordinates for the position, enter **0** for **x** and **1** for **y**. Also check the **Relative** checkbox. The position that is checked is now one pixel below the player. Then, add a **Set the vertical speed** action. Since in Game Maker, and most other computer applications, the vertical axis (y) is pointing downwards along the screen, we need to give the vertical speed a negative value if we want the instance to move upwards. Enter **-10**. This should start the player moving upwards at a speed of 10 pixels per step, but only if the player is standing on something when pressing the **<Up>** key.

What goes up must come down

Now the jump has been started, but there is not yet anything that pulls the player back down to the ground, known as gravity. Gravity should kick in as soon as the player is not in contact with any type of ground. It should stop when ground contact has been reached again.

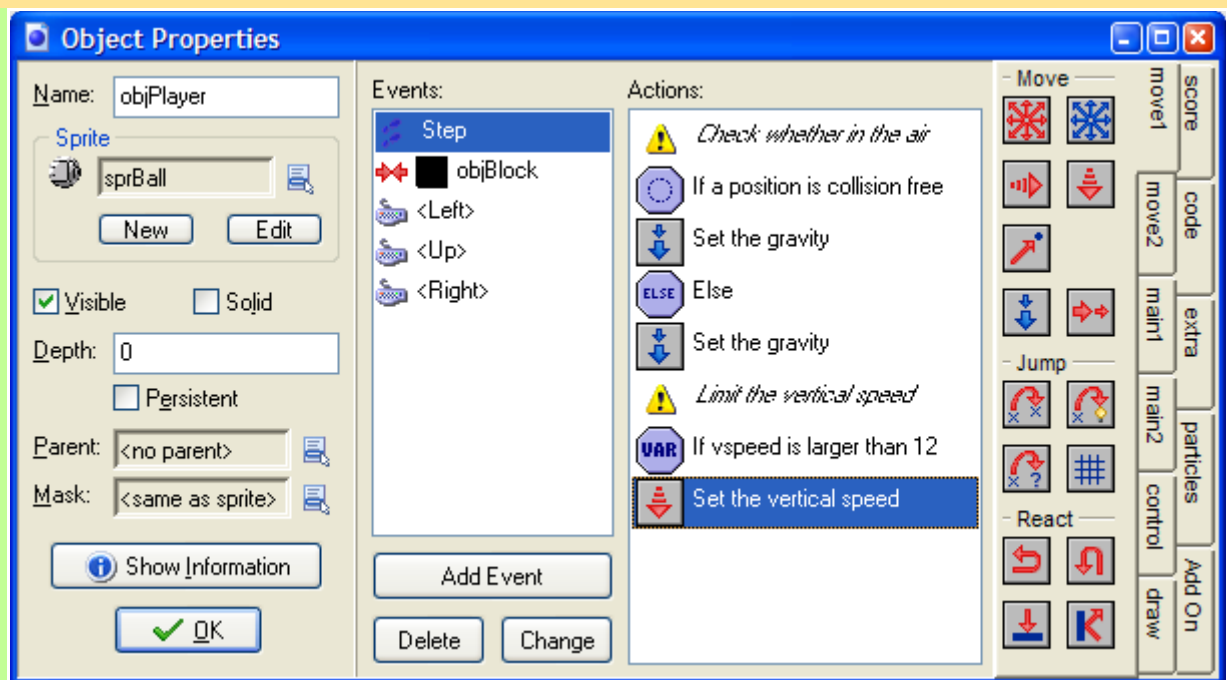
The ground check will have to be made all the time, so add it to the **Step** event.

If a position is collision free (0, 1), Relative, then **Set the gravity**. The **Direction** of the gravity should be downwards, which is **270** degrees (see image below). The value of the **gravity** could be set to **0.5**. Please experiment a bit with both the gravity value and the jumping speed (-10) until you are satisfied with the result. Low values give slow motion action, while high values make everything happen faster, as in old movies.



Now, after the gravity setting, squeeze in an **Else** action, and then set the gravity again, this time to **0** (the direction does not matter). Finally, limit the vertical speed of the player. This is done through checking the **vspeed** variable of the player and, if it is too high, set it to a constant value. Add an **If a variable has a value** action (from the **code** part of the control tab) and enter **vspeed** as the **variable**, **12** as the **value** and **larger than** as the **operation**. If the value of speed is too high, we simply set it to 12. Add a **Set the vertical speed** action and enter **12**.

Now the entire action list for the **Step** event should look something like this, with the addition of two comments, just to remember why things are done. Comments are very useful and should be added to lengthy action lists if you, two years from now, want to be able to remember anything about why the list looks the way it does. However, take care **not** to add comments immediately after an **if** action or an **else** action, since the comments are also considered as being actions and the execution flow will then be altered:



Player to control, coming in on runway 27

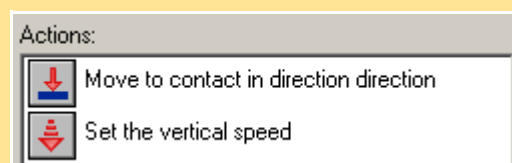
Now that the gravity works, we also have to make sure that the player **lands** in a correct way. In the event of collision with the **Block** object, the vertical speed should be set to 0. But not only that. Since everything is moving in discreet **steps**, the player is probably not exactly aligned with the ground when a collision is detected. Actually, when a collision occurs with a solid object, the player is automatically moved back to the previous position, and that might be a few pixels above the ground. So, the first thing we do is to move the player to a position next to the ground.

In the **Collision with objBlock** event, add a **Move to contact position** action. This action moves the instance in a specified direction, one pixel at a time, until it collides with something. It then stops just before

the collision is detected. As the **direction**, simply enter **direction**. This means that the direction that the player should be moved in is the direction that he is currently moving in. The **Maximum** value determines the maximum distance in which the instance can move checking for contact. It is useful because at times it might happen that this action checks for collisions in a direction where there are no other instances. It would then theoretically continue *ad infinitum*. Set the maximum to **12**, because that is the longest distance the player instance will move in a step (remember our speed limit above). In **against** it is possible to set whether the action should take into account only solid objects, or all kinds of objects. Leave it at **solid objects**.

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, otherwise the character might get "stuck" in a wall or a floor, or it might be able to move straight through it. In any case, the result is unwanted. We will try to look at this a bit closer later on. So far we only have a static image representing the player, so this problem does not occur yet.

There. Now we only have to set the vertical speed to 0 too. Do that. The action list should now look like this:



That is it. All behaviour for the player object is now specified.

Add a room to the game and design a little platform level using the blocks. Make sure to fill the bottom and sides of the room with blocks so that the player can not escape from the room by mistake. Later, we will perhaps add holes in the ground as a hazard for the player, but for now we should only test that the motion works as it should. Finally, add an instance of the player object (the ball) to the room. Then, save the game (unless you already did that) and test it.

We now have the very basic elements of a platform game - the player motion and the platforms. It might require a bit of testing and redesign before the platforms of the level are at a correct distance from each other. It should not be too easy, neither too hard to finish a level. Of course, the difficulty should increase in later levels.


Platform 2

A ball and some black platforms? Bah! We can do better than that. It is time to spice up the graphics a bit. We are now going to make the platforms look a lot better as well as changing the sprite for the player. The player should look a bit different depending on what is going on, so we need more than one image for the player.

Dressing the player

The first thing we do is to change the sprite for the player. We will use two sprites for this. One showing the player facing left, and one showing the player facing right. You can use any sprites you wish, preferably 32 x 32 pixels large, since that is what I am going to use. If you want to, you could use the **player_left.gif** and **player_right.gif** images in the resources file for this lecture. My suggestion for sprite names are **sprPlayerLeft** and **sprPlayerRight**. We need to change some properties of the sprites too. Remove **Precise collision checking** and set the **Bounding Box** to **Manual**. Make sure that the bounding box settings (left, right, top, bottom) are identical for the two player sprites. That is because, as I mentioned earlier, if an instance uses multiple sprites and they have different areas for collision detection, it is very probable that the instance might get caught in a wall or in the floor or the ceiling. The bounding box is the area of the sprite that is checked for collisions. If **Precise collision checking** is enabled, the single pixels of the image are also checked. This is often not needed. You could remove **Precise collision checking** from the block too, since that is already rectangle-shaped and no precise checking is needed.

Then, remove the ball sprite from the game and make the player object use one of the new sprites instead. It does not matter which one.

In order for the player instance to use the correct object at the right time, we have to instruct it. That is quite easy. In the **<Left>** key event, make it use the left facing sprite through adding a **Change the sprite** action ( (main1 tab). Select the facing left sprite. Let the **subimage** be left at **0**, and **speed** at **1**.

Do the same thing for the **<Right>** key event, but, naturally, choose the right facing sprite instead. You can test the game to see that it works as expected.

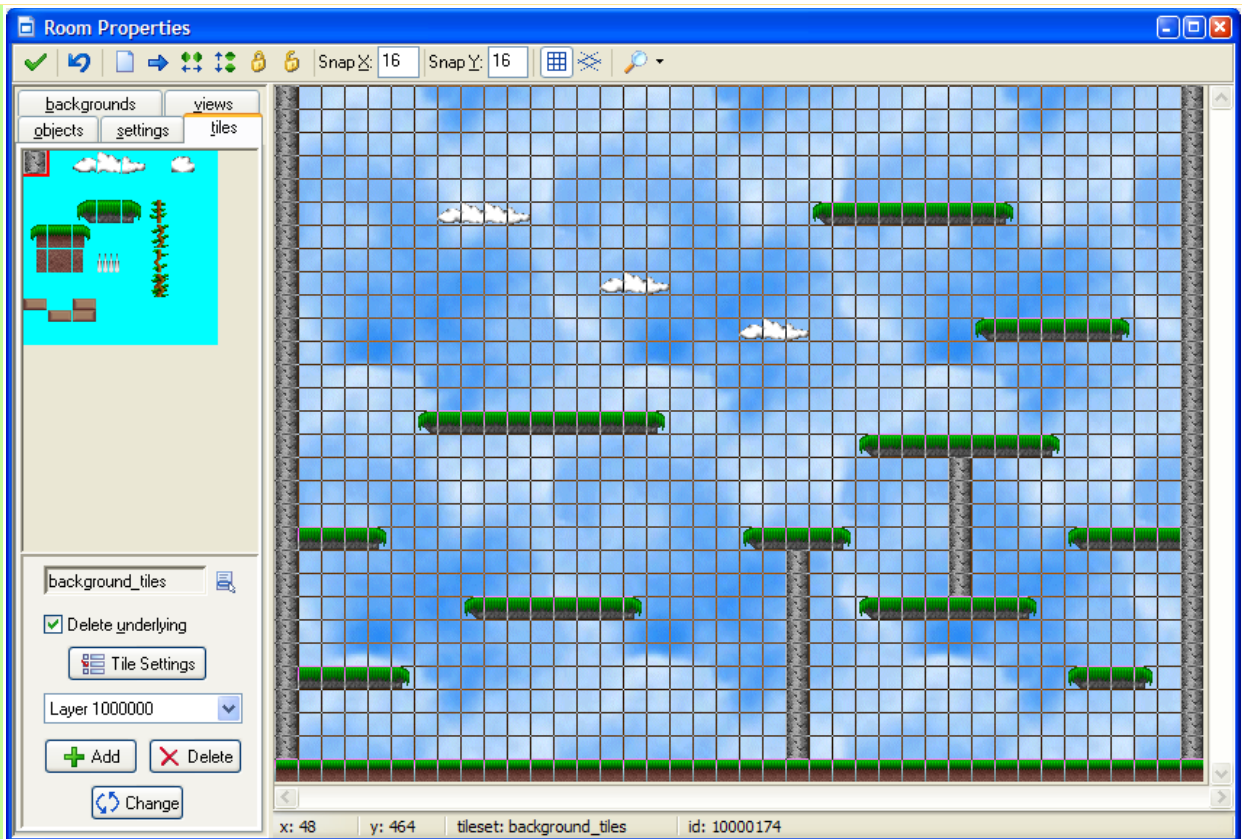
In more advanced games you will probably want to use animated sprites. In this case you also need a sprite for the character when it is not moving. Also you might want to add sprites for the character jumping, falling, shooting, etc. In this case you will have to change the sprite at various places in the events. In particular, in the **<no key>** event you probably want to set the sprite to the no moving one. Alternatively, you can draw the correct sprite in the drawing 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 I indicated before, better make sure that all sprites have the same bounding box and no precise collision checking.

A colorful background

Time to improve the looks of the background. There are many ways to make the background look better. Either you could draw a large image to use as background, or you could make lots of different background objects that look nice. The best approach, however, to this is to make use of the **tiles** feature of Game Maker. **Tiles** are bits and pieces of a certain image that are placed in different locations in the background. Tiled backgrounds are very common in computer games. That is because the memory usage is a lot more efficient this way than using a large image for the entire background. Memory usage is often a limitation on game complexity. Tiles do not have events or collision checking, they are simply drawn on the background. This makes them easier and faster to handle than objects.

As an image for the tiles you could use the **tiles.gif** file from the resources. Load it into Game Maker as a background entity. Call it e.g. **bgrTiles**. If you look at the image, it looks like it is constructed from a lot of small square elements. Those are the **tiles**. They are all the same size and have a 1-pixel border between them. Check the checkbox **Use as tile set** in order to make this background available as a tile set in a room. You are then able to set the size for the tiles. The default settings (16x16) is good for this image, however, both the **horizontal sep** and the **vertical sep** need to be set to **1** in order for the tiles to be correct. This is because of the 1-pixel border around each tile in the image. If you make this adjustment, you will see that the tile grid fits perfectly in the image. Also, set the **bgrTiles** background image as **Transparent**.

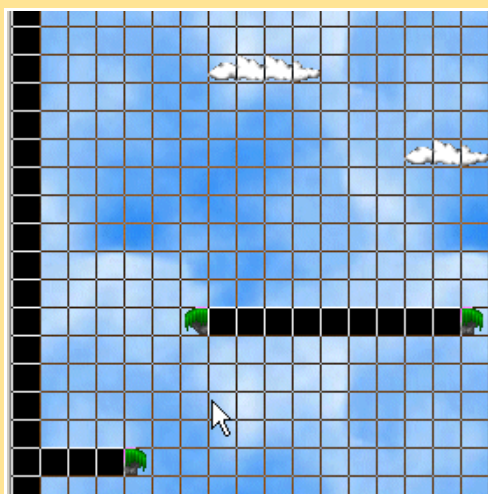
Now, create a new room (or re-use the previous room) and go to the **tiles** tab in the room window. If no tile set is selected, select the background **bgrTiles**. Now you have a set of tiles to use in the room. You can click in different locations in the tile window to select a tile. Then click in the room design window to place tiles. Use the tiles to draw some platforms, a ground and walls. It could perhaps look like this:



The right mouse button can be used to delete tiles if they are placed by mistake. You can use different **Layers** of tiles and set a specific **depth** for each layer. This can be used if you want any tiles to show up **in front** of the game objects. If used correctly this can give a three-dimensional feeling to the game. In order to place multiple tiles as you "draw" with the mouse, hold down the shift button.

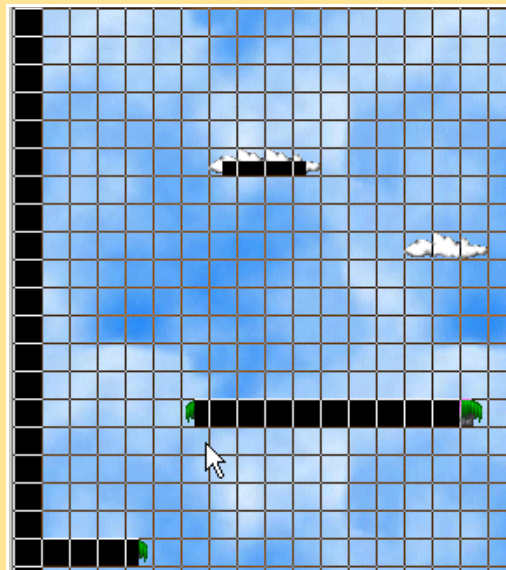
To make the background look even better, add the cloud image to it, just as in the first game we made. This is added as a "normal" background and will appear behind the tiles. (Add it to the background entities, select it in the **backgrounds** tab in the room and put a checkmark in the **Visible when room starts** checkbox (if not already there). Also remove the checkmark from **Draw background color**.)

Now, the tiles are not checked for collisions, so it is not possible to walk on them. For that, we add instances of the old black block object to the room again. Add the block in front of the tiles that should be "solid". Then, remove the **Visible** checkmark from the block object. That makes all the blocks invisible, but they still react to collisions. When playing the game it will then look like the player collides with the colorful graphics, since the blocks are not visible. In the image below I have begun adding blocks to the room.



However, for the clouds and for the ends of the grassy platforms we would not want the entire 16 x 16 pixels block to be solid. Rather we would want about half the square to be walkable. This is solved through adding half blocks to the game. Add two new sprites - **sprBlockHoriz** and **sprBlockVert**. These blocks should be completely black sprites that are 16 x 8 and 8 x 16 pixels large respectively. Add objects for them too, and in those objects, select the old block as the parent. This means that all collision events that check for the original block also will check for collision with these new blocks. Make them solid and invisible too.

Now you can add a bit finer detail to the room. You might need to set the **Snap X** and **Snap Y** to 8 though in order to be able to place the new blocks correctly. Change back the snap settings to 16 when you are done. Otherwise it is hard to see the objects in the room since the grid is too small. In the image below I have added the new blocks to parts of the room. You can see that in the cloud and at the end of the platforms the black blocks only fill half the grid square.



Don't forget to add the player, too, to the game. Then you can save it and try it out. Looks great, don't you think? :)

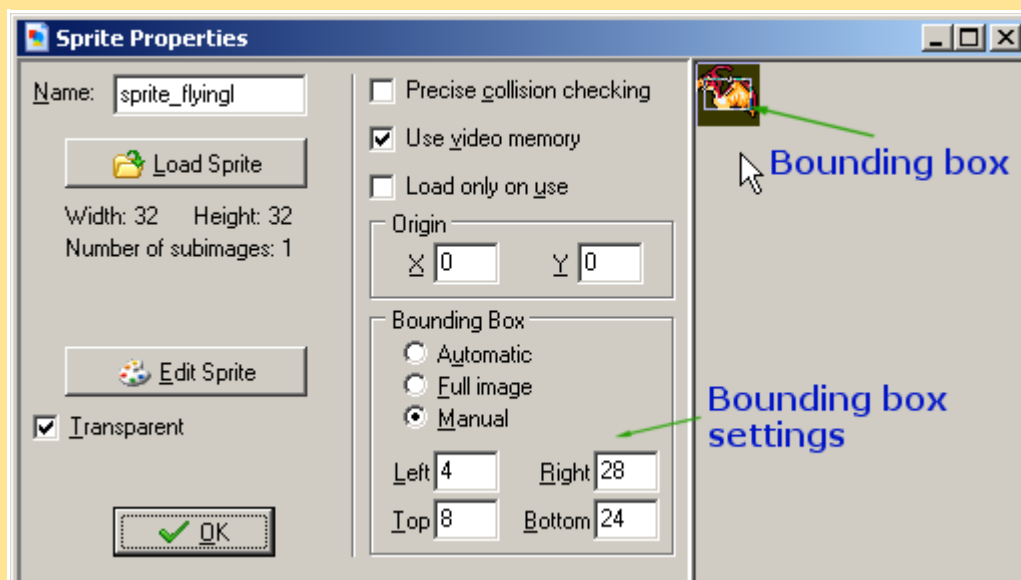
Platform 3

It is now time to add a few other actors to the game. As with the maze game, we need monsters, and we need some reason for the player to continue playing, like objects that add to the score, and objects that take the player to the next level.

Monsters

We will here add two monsters; one that flies and one that walks on the ground. The main difference between them is that the walking monster can be squashed by jumping on top of it. That should not be possible with the flying monster. Otherwise, colliding with the monsters will kill the player.

Start by adding sprites for the monsters. You can use the **monster_1_left**, **monster_1_right**, **monster_2_left** and **monster_2_right** images for this. A good thing is to place monster sprites in a sprite group, since there tend to be quite a few of them. The sprites should **not** have **Precise collision detection**, and the bounding box should be **Manual** and the same for both left and right sprites. Here is an example of how the bounding box could be for the flying monster:



So, the bounding box does not at all need to encapsulate all pixels of the sprite. It only determines which parts of the sprite that are tested for collision.

Also make two objects for the monsters, **objMonster1** and **objMonster2**. In the **Create** event we just set a speed and direction for them. Make them go to the right with a speed of 4 (walking monster) and 3 (flying monster). They then need a collision event with the square block object (The one that is the parent to the smaller blocks). In that event, just add the **Reverse horizontal direction** action. Finally, they need to be given the correct sprite depending on which direction they are moving in. That could be done through checking the horizontal speed in the **End Step** event. So, if the **hspeed** is larger than **0**, set the sprite to the right facing sprite, else set it to the left facing sprite. This is similar to the maze monsters.

Another thing that we should add in order to make it easier to handle the monsters is some kind of marker that stops the monsters from falling off a platform. Currently the monsters only change their direction when hitting a wall. It could however be possible that we want the monsters to turn around at a certain location where we can not add a wall (for example, because the player should be able to go through there). To do this, we add a new block sprite. Make it a fully blue square. It should also be invisible. Then make an object for it and call it **objMarker**. In the monsters objects, in the collision event with the marker, reverse the horizontal direction, just as with the collision with the wall objects.

Now you can use the blue square at the points in the game where the monsters should turn around without there being any wall or so. Other kinds of markers could be used in other ways, for example to make a monster shoot, jump, lay a bomb or something.

When the player hits a monster, the player should die. *Unless* it is the walking monster and it is hit from above. This is done in the player object. Start with the collision event with the flying monster. First, play a dying sound, like **ao.wav**. Then wait a second (sleep 1000 milliseconds) and restart the room. That is it. We have not introduced life count yet, otherwise the player would loose a life too.

The collision event with the walking monster is a bit trickier, because there is a need to kill the monster if it is hit from above. First, we need a sprite of a dead monster. Use **monster_1_flat.gif**. Then we need an object, **objMonster1Dead**, that uses that sprite. The dead monster object should just destroy itself after some time. So, in the **Create** event, set the **alarm0** to **10** steps. And, in the **alarm0** event, destroy the instance.'

Back to the player object and its collision event with the walking monster. Let us make a kill-monster-rule. *If the player is above the monster, or perhaps 8 pixels below the top of the monster, and the player is moving downwards, the monster should die. Otherwise the player dies.* We thus need to check the vertical speed of the player as well as its vertical location compared to the vertical location of the monster. That requires some calculations. If we should try to solve that with a lot of **If a variable has a value** actions it would be quite hard to get an overview. Instead, we are going to use a new action: **If an expression is true**. Add such an action to the collision event with the walking monster.

In the **expression** parameter we will now enter a few **variables** and **operators**. If you know C/C++ programming or similar you will know what the expression means, but if you have never written any program code before it might look very strange. I will therefore try to break it apart and explain the parts as good as I can.

The player instance should be **above** the monster, or, at least above eight pixels below the top of the monster. Perhaps this image could explain what I am getting lost in:



Thus, the player's y coordinate should be less than the monster's y coordinate plus 8 pixels. This is written like this:

y < other.y + 8

The other condition is that the player should be moving downwards. That means that the vertical speed should be larger than 0. That is written like this:

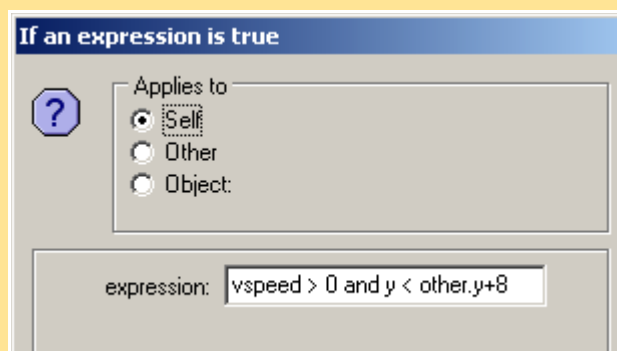
vspeed > 0

Now we need to combine those expressions to a single expression. Since **both** of them need to be true in order for the monster to die, we combine when using the **and** operator. Game Maker Language is so flexible that you can write the **and** operator either as "and" or "&&". Use whichever you like. I started programming with Pascal, which uses the "and" word written out, so I will use it here. I find it easier to read. :)

So, the entire expression should now look like this:

vspeed > 0 and y < other.y + 8

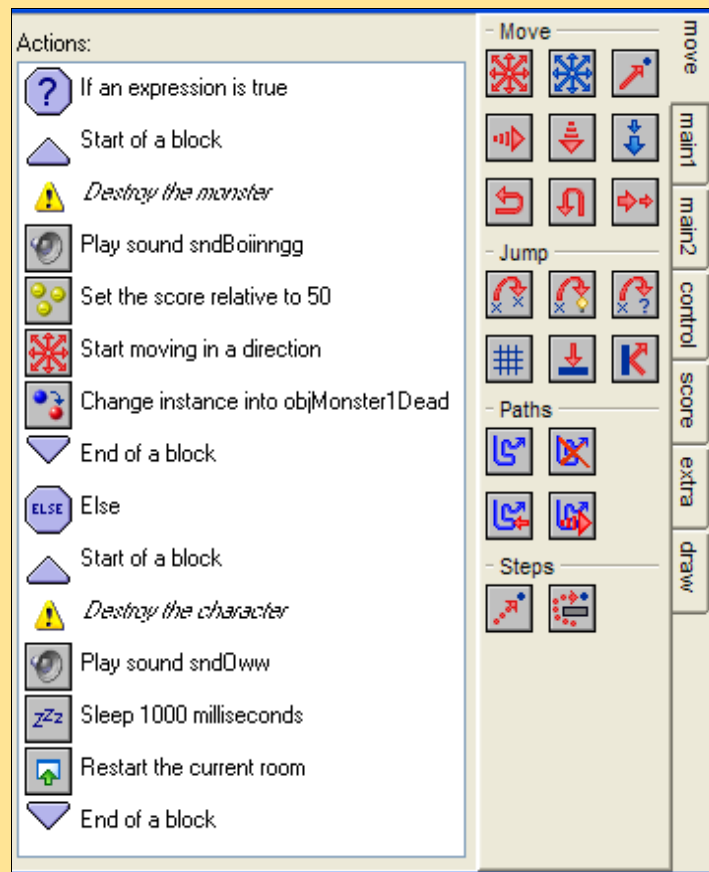
The **other** keyword is a special instance reference. It represents the instance with which this instance is currently colliding. That is, one of the walking monster instances in this case. So, enter the above string as the **expression**. The **Applies to** should still be **self**. It should look like this:



So, if that is true, we should kill the monster. Add a **Start of a block** action. Then play a sound (for example the **boiinngg.wav** sound. You may of course add any other sound of your liking.

Then the player should have some score for killing the monster. Increase the score by **50** points. The monster is always moving, and thus the dead monster object will also have a speed when the monster changes into it. This is not what we want. Therefore we need to stop the monster instance when it dies. Do this with a **Start moving in a direction** action. In that action, just select the middle button, with the square, in order to stop the instance. Also, make it apply to the **other** instance by checking that option. Finally, the monster should change into the **objMonster1Dead** object. That is done with the **Change the instance** action. Select the dead monster object, and choose **yes** in **perform events**. That means that the **Create** and **Destroy** events of the dead monster and monster objects respectively are executed. The last thing in the **Change the instance** action is the **Applies to** setting. Set it to **other**. That means that whatever this action does applies to the **other** object.

Great. Now we have taken care of killing the monster. End the block and add an **Else** action. Then start another block. This block is executed if the player does **not** manage to kill the monster. Here we could play a little death sound, for example **au.wav**. Then **Sleep** for **1000** milliseconds (ms) and then restart the room. Finally, end that block too. The entire action sequence should now look something like this:



Here, there are a few comments added, just to make it easier to remember what is happening.

Pits

We could add some deadly pits to the game. Deadly pits are often holes in the ground that contain pointy spikes or hot lava or something else that is dangerous. In this game we can implement pits through adding another square block, 16 x 16 pixels. Make the block red. Call the object **objDeath**. Put it in some pits that you make in the room. The object should be invisible, and you could add the spiky-looking things from the tiles image to the same place as the death object, just so that it looks dangerous enough.

In the player object, in its collision event with the death object, add the same actions as in the collision event with the flying monster - play sound, sleep 1000 ms and restart the room.

Valuable objects

In most platform games the player can increase the score through collecting certain objects in the game. We should add something like that to this game. It could be for example mushrooms. (Is it not Mario who collects mushrooms?? :)).


As a sprite for the mushrooms, use the **mushrooms.gif** file. This file contains 10 images of different mushrooms. In order to give the player a bit of variation, the mushroom that is placed in the game will be randomly picked from these images. Add a mushroom object and let it use the mushroom sprite. In the **Create** event of the mushroom object we pick an image at random. That is done through adding the **Set a variable** action. The variable that we should set is **image_index**. That variable determines which image of

the sprite that is shown. The **value** of the variable should thus be a random number between **0** (first image) and **9** (tenth image). In Game Maker there is a function called **random** that is good at this. Random takes one argument and will return a random value that is between **0** and the argument that is provided. So, if we write **random(10)**, it will return values between **0** and **10**. Well. That was not **completely** correct. In fact, it will return values from **0** to **9.999999999999999999**. Fortunately, when the **image_index** is set to a real value like that (with decimals), Game Maker interprets this to the closest image and it works well. So, as the **value** in the **Set a variable** action, enter **random(10)**. Now we also have to **stop** the animation. If we do not, Game Maker will loop through all the images over and over. In order to stop the animation, set the variable **image_speed** to **0** in a similar manner as we just did with the **image_index** variable.

Place some mushrooms in the room. When you place them in the room design window they all look the same, but when playing the game later on, they have a random mushroom image.

Now we should let the player have some score when colliding with the mushroom. So, in the player object, in the collision event with the mushroom object, do the following: Play a Get-mushroom-sound (for example **Ploup.wav**), increase the score by **10** and destroy the mushroom instance (**Destroy the instance**, applies to **Other**).

Reaching the next level

We need a mechanism to move the player to the next level once this level is completed. It could be completed for example when all the mushrooms are collected. This level ending could be made similar to the one in the maze game. Make an **objLevelExit** object and use for example the image **levelexit.gif** as sprite. Add it to "the end" of the room. Initially, it should be hidden until all the mushrooms are taken. It should then appear again. So, in the **Create** event, simply move it out of the way. Make it **jump** to the coordinates **(-100, -100)** or something. Then, in the **Step** event, check if there are any mushrooms left (similar to the maze game). If not, make it **Jump to start position** (there is an action called that, looks like this: ).

In the player object, in the collision event with the level exit object, play a happy sound, like **harp.wav** and sleep 1000 ms. Then check if there exists a next room. If it does, go there, else, show the highscore and restart the game. This is very similar to the maze game so I will not go into further detail here.

That is that for the third game version.

Platform 4

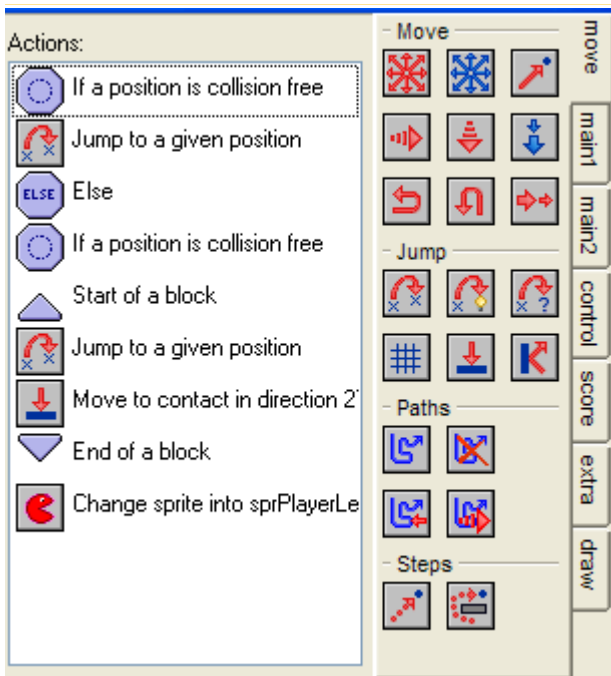
In the fourth version of the game we take a closer look at the player motions. We will enable the player to climb up ladders or vines and to walk up ramps. We will also make use of Game Maker's scrolling feature - the **views**.

Ramps

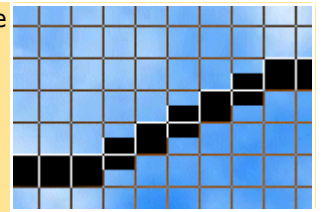
It would be nice if we could make sloping ramps that the player could walk up along. The ramps could be constructed from the horizontal blocks that are 16 x 8 pixels large. (We used those for the clouds earlier). Walking down a ramp is made automatically because of the gravity feature that is already in use. Walking upwards is however not yet possible. To remedy that we change the **<Left>** and **<Right>** keyboard events for the player a bit.

Start with the **<Left>** keyboard event. Instead of just checking the position to the left of the player, we check the position that is **eight pixels above and left of** the player too. If that position is free, we move there. Start out by simply adding an **Else** action to the action sequence that already exists in that event. This means that if the above check (if the position four pixels to the left is free) is not resolved to **true**, then the action sequence following the **Else** action will be carried out.

After the **Else** action, we check **If a position is collision free** and enter **(-4, -8)** in that action. Also check **Relative**. Then you need to start a new block. In that block, jump to the location we just checked **(-4, -8)**, **Relative** and then add a **Move to contact position** action. In that, enter **270** as **Direction** (straight down) and **8** as **Maximum**. If we would have a larger maximum, we would not *fall* down from platforms, but instead be instantly moved down to the bottom (or at least as many pixels as the **maximum** value). The action sequence should then look like the image to the left. Note that the **sprite changing** action is moved to the end of the list.



There. Now it should be possible to build some ramps, like this (right):



Add a few such ramps to the game room and try that they work.

Do the same thing with the **<Right>** key, but with **positive x coordinates**.

Ladders

Now we will add the ladders feature to the game. It should be possible to climb ladders or vines or whatever graphical representation you like.

Create a new sprite and make it 16 x 16 pixels and completely black. Along the middle of the sprite, draw a 4 pixels wide rectangle that runs through the entire sprite. You should now have a horizontal rectangle. Make the sprite **Transparent**



and remove **Precise collision checking**.

Make an object for the ladder. The object should use the ladder sprite above and it should **not** be **solid** and **not visible**. As soon as the character is in contact with a ladder object, its behaviour should change. It should not be affected by gravity anymore and it should be able to climb up and down. Also, its sprite should change into a climbing sprite.

First, remove the gravity from the **Step** event if the player is in contact with the ladder. Open up the player object and view its **Step** event. That event turns on gravity if there is no solid instance below the player. At the end of the action sequence here, add the following:

- Check if there is an object at position **(0, 0), Relative** that is of the type **objLadder**
- If so, start a new block
- Set the gravity to **0**
- Set the vertical speed (**vspeed**) to **0**
- Change the sprite into a climbing sprite, which you have to add first. You could use the **player_up.gif** image. Don't forget to set the bounding box to the same as the other player sprites.
- End the block.

Now the character should not fall off a ladder, and the sprite should be correct. Time to enable the climbing too.

To the **<Up>** keyboard event of the player, do the following in the *beginning* of the action sequence:

- Check if there is an object at position **(0, 0), Relative** of the type **objLadder**
- If so, Start a new block
- If a position is collision free **(0, -3), Relative**
- Jump to position **(0, -3), Relative**
- End the block.

The **<Down>** key has not been defined at all yet, but it should contain the actions listed above too, but with a positive **y** coordinate instead.

It should now be possible to climb the ladders too. Add some ladders to the room and hide them behind vine tiles or something.



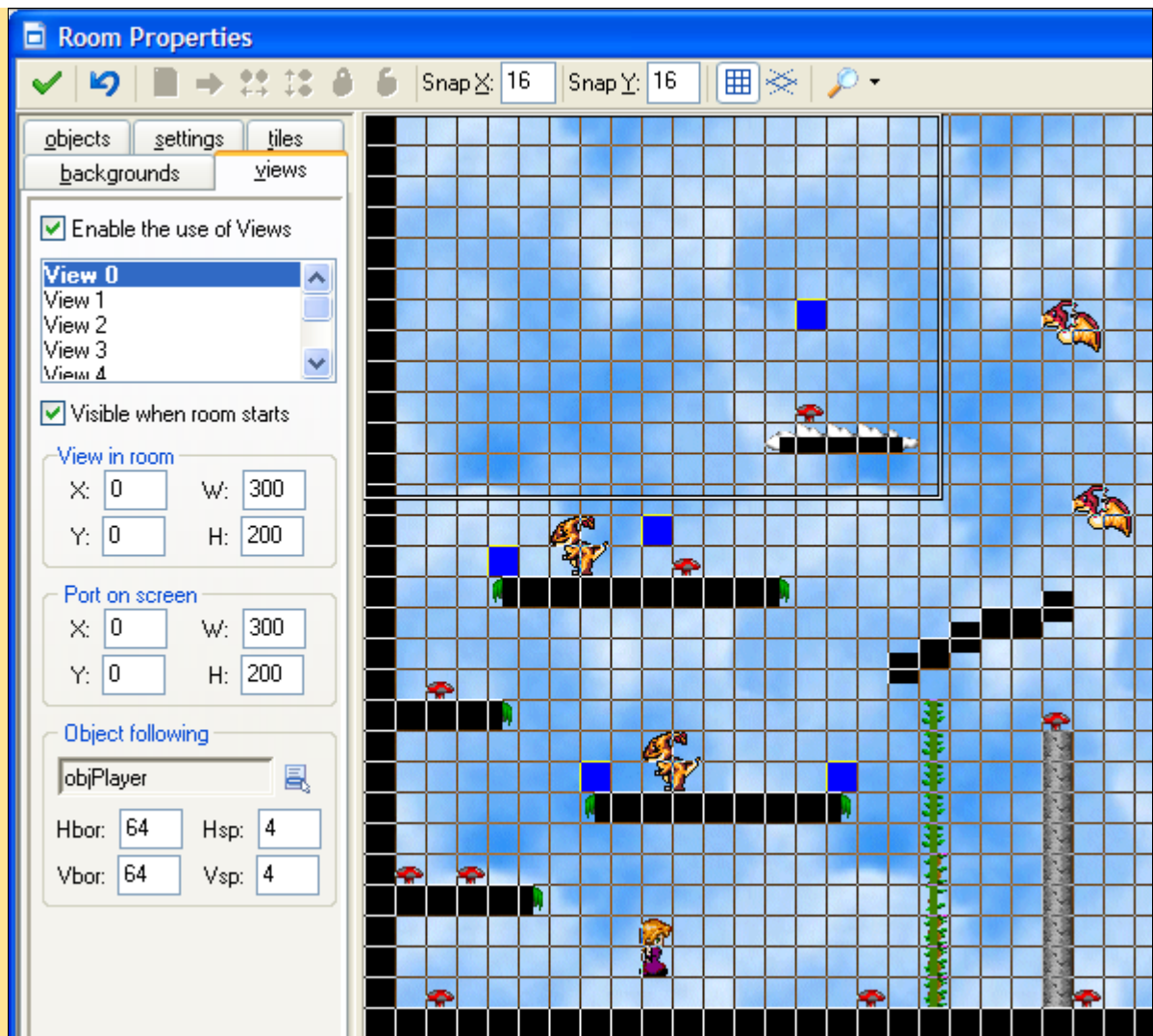
Room with a view

Up to now you have displayed the entire room in the game at the same time. We are now going to introduce a scrolling feature of Game Maker. It is called **Views**. Basically, you can specify an area of the room that is viewed on the screen and then scrolled around so that different parts of the room are visible at different times. Viewing only a part of the room makes the game more challenging to play, since it is harder to plan ahead if you can not see what is waiting. It also opens the possibility to hide valuable objects at hard-to-reach places.

Open the room editor. I hope you have a little level designed there. :) Click on the **views** tab. Here is where the settings for the views are changed. It is actually possible to have more than one view of a room at the same time. That is good for two-player games where the players use the same computer.

Put a checkmark in the checkbox **Enable the use of views**. Select the first view (**view 0**). Now check the box **Visible when room starts**. The **View in room** settings determine where in the room the upper left corner of the view is. Leave **X** and **Y** at **0**. They will later change automatically as the view follows the player around in the room. The **W** and **H** set the width and height of the view. The default size of the entire room is 640 x 480 pixels, so, set these two values to 300 and 200 to view a bit less than a quarter of the room at any one time. The **Port on screen** determine where on the screen the view is placed and how large it is. It is also possible to set the size of the view here, which may be used to scale the contents of the view. Since we are only going to use one view, we can leave **X** and **Y** at **0**. Set the **W** and **H** of the **Port on screen** to **300** and **200**, just as in **View in room**. If you want to, you could change these in order to "zoom" the game to your own liking. However, note that zooming might slow the game down on certain computers. To try it out, you can experiment with setting these values to **600** and **400**. The game will then be zoomed in 2x when playing.

Now, jump down to the bottom setting, which is **Object following**. This is set to decide which object the view should follow around in the room. Select the player object here. The **HBor** and **VBor** settings tell Game Maker how close to the border of the view the followed object can be before the view scrolls in that direction. Set them both to **64**. Finally, the **HSp** and **VSp** determine the maximum speed of the scrolling. Set them both to **4**. If these had been set to **-1**, the view would scroll to the correct location instantly.



Test the game now. Only a part of the room should be visible, and it should scroll along as the player moves. That was not too hard, was it? :) However, the game itself might be a little too hard to play like this, but we have at least tried the feature now.

Falling platforms

We should now try to add a platform type that is destroyed once the player has touched it. This adds a new hazard to the game, especially when used over deadly pits. :)

Add the **falling_block.gif** image as a sprite, **sprFallingBlock**. Actually, we will not make it fall, but rather disappear. Now, in the sprite editor for that sprite, we are going to make a little disappearing animation. Select in the menu: **Animation->Disappear**. In the pop-up window that appears, enter **5** in order to make the block disappear in five frames. Check the **Show preview** box to see the animation. It might go a little fast. Set the **Speed** (lower left corner) to **3** or something like that. Now the block animation looks OK. Note that this speed setting will not have any impact whatsoever in the game later. We will have to set the animation speed in another way.

Make a new object, **objFallingBlock**, and add the new sprite to it. Make the object solid. Also, set as parent the **block** object that is the parent of all other blocks. The default for an animating sprite is to start animating as soon as the instance is created. This is, however, not what we want. We want it to only display the first image until the player moves on top of it. It should then slowly disintegrate into nothing. So, in the **Create** event, set the variable **image_index** to **0**. That sets the first image of the sprite to be displayed all the time. We should also set the animation speed here in order to stop the animation. Set the variable **image_speed** to **0**. This stops the animation of the sprite.

To make the block disappear when the player walks on top of it, add the **Step** event. In that event, first check if there is an instance of the type **objPlayer** above it. We test the position 2 pixels above the block. So, add an **If there is an object at position** action. As **x**, enter **0**. As **y**, enter **-2**. Then choose the object to be checked for as **objPlayer**. The **Relative** box should be **checked**. After that, we check that the disappearing animation has not already started. If it is already started, there is no need to start it again... So, check if the variable **image_speed** is equal to **0**. If both these tests are true, we simply set the variable **image_speed** to **0.2**. That means that the sprite will animate with a speed that is 0.2 times the room speed. The default room speed is 30, so the animation speed would be 6 frames per second. This gives the

player about 1 second per falling block before the block is destroyed completely.

The block should now start to disappear.

But! What happens at the end of the animation? The block instantly appears again and starts disappearing over again. This continues without ending. We want it to stop when the animation has played through once. Fortunately there is an event that can help us here - the **Animation End** event. It can be found in the **Other** group of events. In that event, just **destroy the instance**.

There! Now we can add some disappearing blocks to the game in tricky situations.



(As you can see in the image above, I already went through part 6 of this lecture (Score Panel) before making up the "falling blocks"... :))

Platform 5

Bang, bang, Lucky Luke!

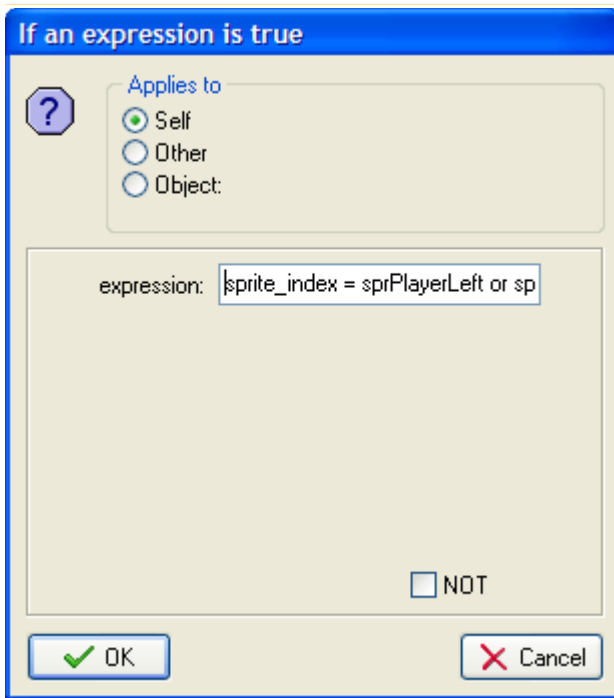
We are now, for the first time in this course, going to introduce shooting in a game. The player should be able to, after having picked up some ammunition, shoot at the enemies in the game.

First, we need to add a few sprites to the game - the **bullet.gif** and the **ammunition.gif** images. Then, add objects for them, too, and make them use the sprites we just added.

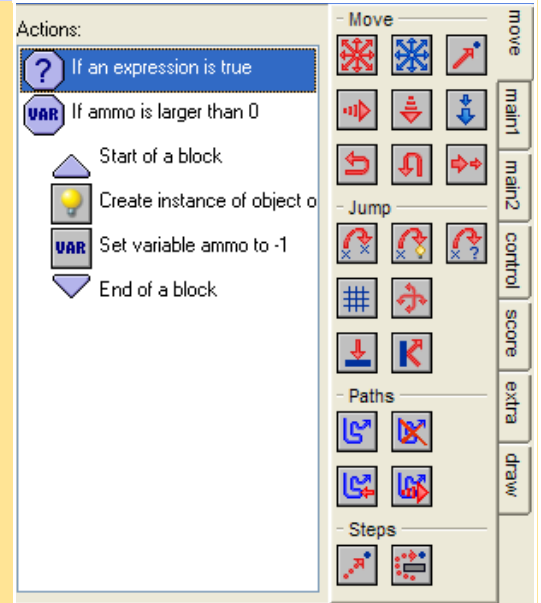
Then, we take a look at how to handle the ammunition ("ammo" for short). In order to keep a count on the bullets left we need a variable. The basics about variables will be covered more in detail in the next lecture, so if you are unsure about them, just try to follow the instructions. In the **Create** event of the player object, **Set the value of a variable**. Enter **ammo** as **variable** and **0** as **value**. Now we have created a new variable called **ammo** and given it the value **0**.

The ammo object created above should not do anything at all. It should just sit there, waiting to be picked up. However, in the player object, in its collision event with the ammo object, make it add **10** to the **ammo** variable (**Set the value of a variable, ammo, 10, Relative**) and then **Destroy** the **Other** instance. Now the player can pick up ammo that you have placed in strategic locations in the room.

Time to shoot. For shooting, the **<Space>** key is common, so add an event for it to the player object. Make sure to choose the **Key Press** event type, and not the **Keyboard** event type. Otherwise, there will be a bullet fired as long as **<Space>** is being held down. That is not what we want.



The player should only shoot when facing left or right, not when climbing up or down on a ladder. Therefore we need to check which way the player is currently facing. In this case, that is easiest done by checking which sprite the player is currently showing. So, to the **<Space> press** event, add an **If an expression is true**. In this action we can write an entire "expression" that is checked to decide what action to take. As **expression**, enter the following: **sprite_index = sprPlayerLeft or sprite_index = sprPlayerRight**. This means that if the sprite that is displayed is **sprPlayerLeft** or **sprPlayerRight** Game Maker will execute the next action.



There, a check must be made to see if there are any bullets left. **If a variable has a value, variable: ammo, value: 0, operation: larger than**. Then, start a new block. In the new block, **Create an instance** of the bullet object at location **(0, 0), Relative**. That will create a new bullet at the same location as the player. Finally, the **ammo** variable should be decreased by **1 (Set the value of a variable, ammo, -1, Relative)**. End the block.

Now the bullets are created correctly, but they need to move too :). So, in the bullet object, in its **Create** event, the speed is set, but first a check must be made to see in which direction it should move. Add an **If a variable has a value** action. We should now check which sprite the player is currently using. If the left player sprite is in use, the bullet should move to the left, if the right player sprite is in use, it should move to the right. Otherwise, there should not be any bullet (e.g. if the player is climbing). So, as **variable**, enter **objPlayer.sprite_index**. That is, if your player object is called **objPlayer**. Otherwise, use the name you have for the player object. The **value** that should be checked for is the name of the left facing sprite for the player. If you have followed the name convention I use, it could be called **sprPlayerLeft** or something similar. Enter the name of your left facing player sprite in the **value** field. The **operation** should be **Equal to**. Now, if the player is facing left, the next action is executed.

The next action should thus be to set the speed and direction of the bullet to **Left** and speed **12**. Thereafter you add an **Else** action. If the first check is not true (the sprite of the player is not sprPlayerLeft), the bullet should move to the right. Do that here.

Oh, and please find a good sound for the bullets and add it to the **Create** event. :)

Now the bullet moves, but it still does not have any impact on any of the other game objects. So, add collision events to the bullet object. If it collides with the black **objBlock** object, it should simply be destroyed. If it collides with any of the monsters, the sound **boiinngg** should play, **50** points should be **added** to the **score**, the **other** instance should be **destroyed**, and the **bullet** instance should be **destroyed**. (This requires two **Destroy the instance** actions - one for the **other** instance (the monster) and one for **self** (the bullet)).

Actually, that was all that we were going to add to the fifth version of the game. Make sure there is some ammo in the room and try it out.

Platform 6

To the sixth iteration of the platform game we are going to add...

A Score Panel

Lets see. We now have score and ammo. We should also have a lives count. How should the player be told the values of these variables? Through a Score Panel.

The first thing we need to do is to add a **font** to the game. The font is used to draw score and ammo in the score panel. Add a new font entity and call it **fontArial**. Select the **Arial** font in the font selection and set the size to **10**. Click on **Normal** to set the character range to include all normal letters and digits (ASCII values 32 to 127).

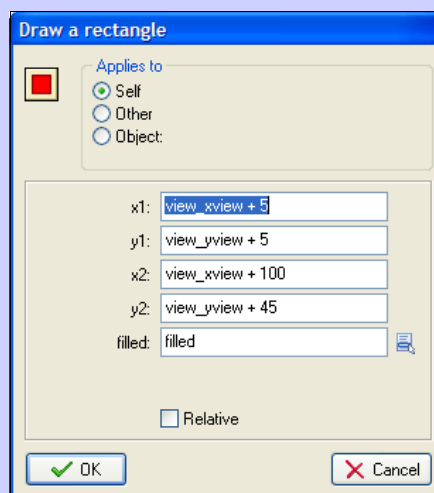
We need an object to control all this - a Life Controller (**objLifeController**). In the **Game Start** event of the life controller, set the number of lives to **3**. Also, set the **Window Caption** to not show anything at all here. (Same as with the maze game).

This is the first time (if I remember correctly) that we are going to use the **Draw** event. The **Draw** event is used when there is a need to manually draw things in the game view. Up to now we have relied on the automatic drawing routines of Game Maker that draws the sprite that is assigned to an object. In the **Draw** event it is possible to draw any sprite you like, but you can also draw shapes, like lines, rectangles and circles, and you can draw text. For the Score Panel we want a rectangular background with the score, lives and ammo drawn on top of it. Most of the actions we use here are found on the **Draw** tab or on the **Score** tab.

First, add a **Set the color** action. This action sets the color that will be used in the succeeding draw actions. The color set here will be used for the rectangle that is the Score Panel background. Try to find a nice colors for it, for example (231, 230, 180) for the inside of the rectangle. The outline of it will have another color which we will define later.


Now that the color is set, we can **Draw a rectangle**. The coordinates of the rectangle action are as follows: (x1, y1) - upper left corner, (x2, y2) - lower right corner. There is a little problem here though. If we simply enter some coordinates, like x1: 5, y1: 5, x2: 45, y2: 100, the rectangle will surely be drawn at that location, **but**, it will stay in the same place in the room. You can try if you like. Do not forget to add the Life Controller instance to the room though (add it to every room). It does not need any sprite since we draw everything in it manually. Now, run the game and walk around a bit. The Score Panel does not follow around when scrolling. Instead, it just sits in the same place in the room and disappears out of view when the player moves away from it.

To fix that, we need to add the current coordinates of the **view** to it. Those coordinates can be read from the variables **view_xview** and **view_yview**. So, the **Draw a rectangle** action should have the following properties: **x1: view_xview + 5, y1: view_yview + 5, x2: view_xview + 100, y2: view_yview + 45**. Set **Filled** to **filled** in order to draw a filled rectangle. Otherwise an outline will be drawn, which is what we will do next.



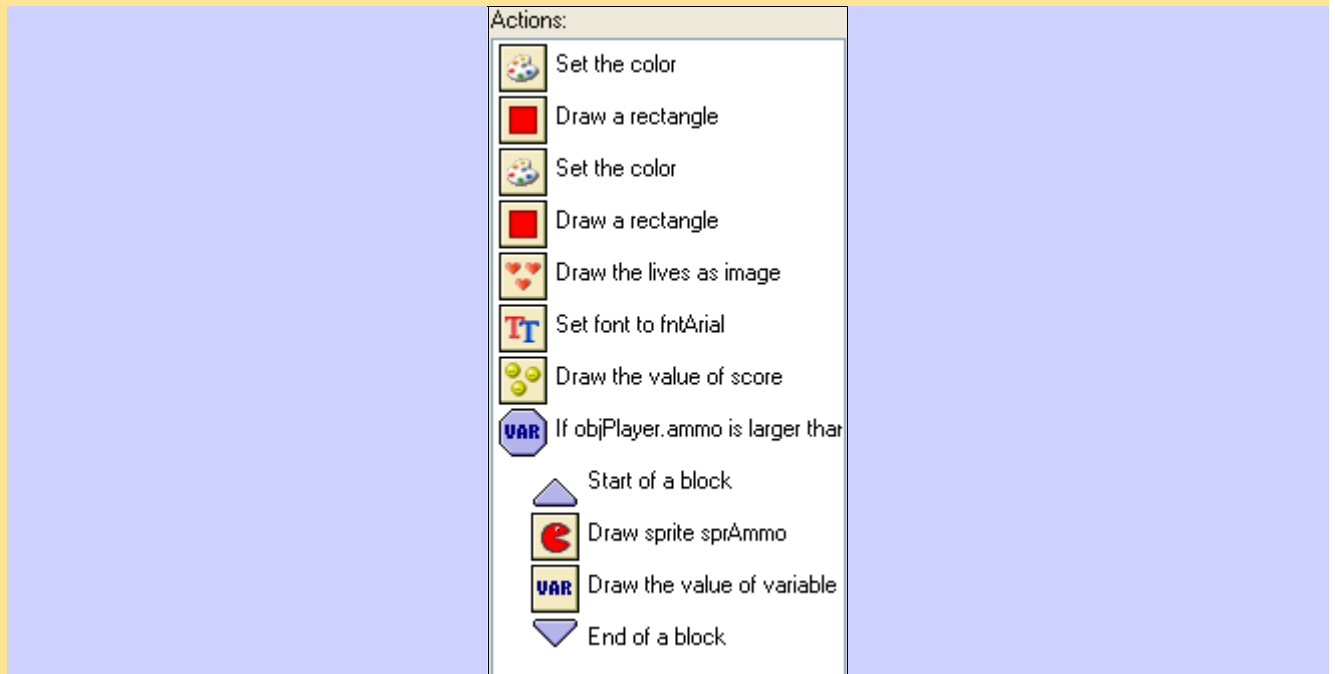
We also draw an outline around the rectangle in a black color, so set the color to (0, 0, 0) for black, and draw an rectangle again, using exactly the same values as above, but choose **outline** in the **Filled** field instead.

Now, if you try out the game again, the Score Panel should nicely follow the view as it scrolls around. Looks good, does it not? :)

Now we should draw the lives. For this, add a new sprite, **sprLife**, and use the image **life.gif** (or similar). Add the action **Draw the lives as image** () to the **DRAW** event of the **objLifeController** that we are currently editing. As coordinates, enter (**view_xview + 10, view_yview + 10**) since the lives, too, have to follow the view around. As **sprite**, choose the life sprite you just added.

Now we choose a font to draw the score. We have only added one font to the game, but we should select it here. Add a **Set the font** action and set the font to **fntArial**. Leave the **align** parameter to **left**.

Time to **Draw the value of score**. As coordinates, enter (**view_xview + 10, view_yview + 30**). Let the caption be **Score:**. Then we should draw a weapons icon and the amount of ammo available. But we should only draw it if there **is** any ammo. So, add an **If a variable has a value** action. Enter **objPlayer.ammo** as the **variable** and **0** as the **value**. The **operation** should be **Larger than**. Then start a new block. In that block, **Draw a sprite image**. The sprite that should be drawn is the ammo sprite. It should be drawn at the coordinates (**view_xview + 70, view_yview + 5**). **Set subimage** could to **0**. After that, **Draw the value of a variable** (from the **control** tab). The variable should be **objPlayer.ammo**, and the coordinates should be (**view_xview + 70, view_yview + 10**). Finally, end the block.



The Score Panel is now ready. Run the game and test that it works. If you do not like the color and font of the text, you can change that, too, so that it better fits the color and style of the other parts of the Score Panel.

One more thing we add to the Life Controller. Add a **No More Lives** event. In that event, first display the highscore table and then restart the game.

The player can now see the score, the ammo, and the number of lives left. There is however nowhere that the number of lives is decreased. We will have to add that. So, in the player object, check all collision events with the monsters and the death object. Decrease the number of lives in each of those events.

It would be good to be able to increase the lives too. Add the **heart.gif** as a sprite and create an object for it. Now, when the player collects the heart, the number of lives should increase by one, and the heart should be destroyed. Also, play a sound, e.g. the same sound as the one that is played when a level is finished (harp sound).

If everything is done correctly, you should now have a pretty decent platform game. Of course it would have been nicer with some animated player and monster graphics, but that is easily added, if you can just find good graphics. :)

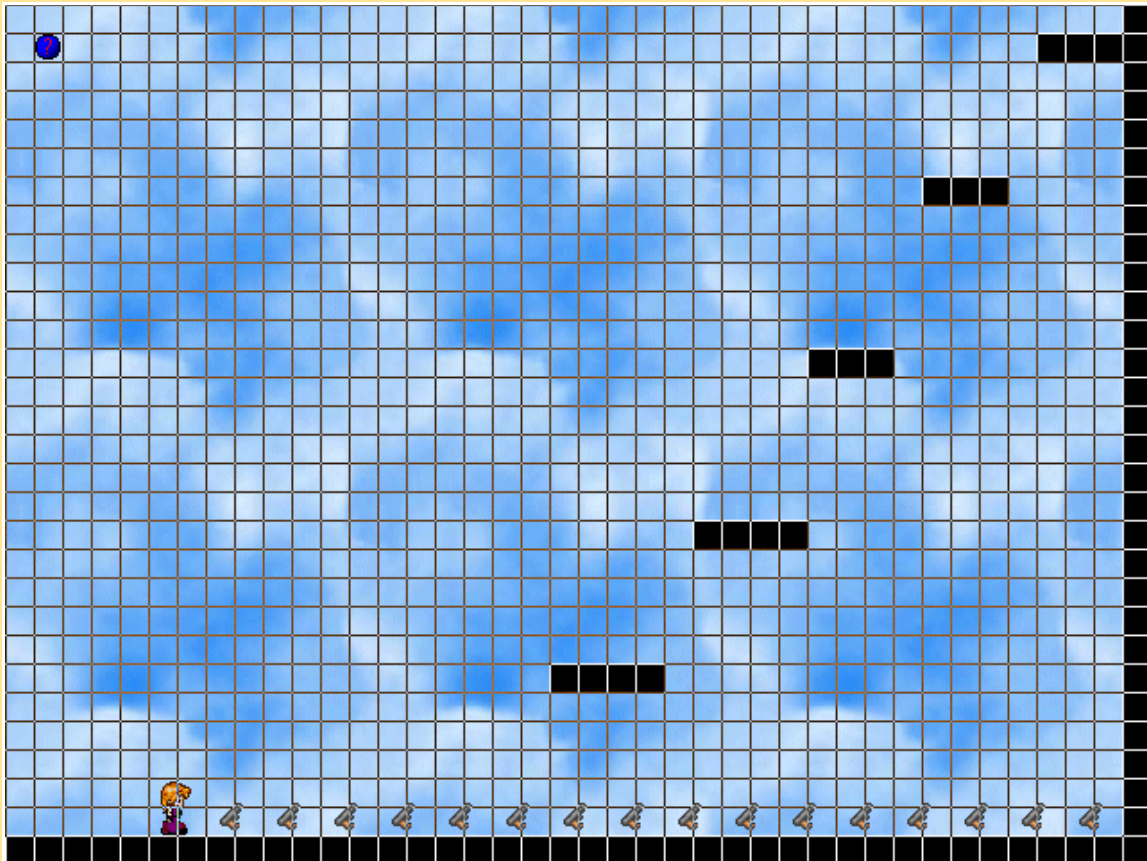
There are however a few bugs in the game, and I figured we should use the debugger to see what is happening.

Debugging

Since there has been requests about some debugger training, I thought I should include it here. In the game there are two (perhaps more? :)) common bugs that both have to do with instances disappearing out of the room. Of course, clever level design can make them never appear. Nevertheless we shall try to make them occur.

Create a new room and drag it to be the first room. Only add a floor, a wall along the right side of the room and some platforms so that the player can move to the top right corner of the room. In order to view the

platforms in the game, you have to add some tiles on top of them, just as usual. Also add lots of ammo, the player and the Life Controller. Finally, set the **Views** setting to the same as the first room settings. Something like this:



Now, make this room the first room and start the game in debug mode (F6 or the red play arrow). The game will now start as usual, but with an extra debug window.

Collect the ammo. Then move to the lower left corner of the room and fire some bullets. The bullets will disappear from the room. **But do they disappear from Game Maker?** If this had been a shooting game mainly, you would shoot lots of bullets, and a few of them might go away outside the room. What happens to those bullets? Let us see how many bullet instances that Game Maker is currently handling. To do this, we add a **watch**. Click the green plus sign in the debug window. A pop-up appears asking for an expression. Here you can add some expression that you want to view at run-time. Enter **instance_number(objBullet)** here (Change "bullet" to your name of the bullet object.) and press **Enter**. The watch is added to the list and you can now see how many bullet instances that Game Maker is currently handling. They should only increase as you fire more bullets. They never decrease. This means that Game Maker gets more and more work to do and, depending on the system power, will eventually degrade in performance so that the game runs slower and slower. This is just because Game Maker maintains all the bullet instances and, for each step, calculates a new position for them and checks for collision with all the objects for which they have collision events defined.

Now, try to shoot at the wall to the right. The instance count for the bullet objects increases momentarily, then decreases again as the bullets hit the wall and are destroyed. So, if they hit a wall or something, it works as it should.

In the image below I have also added the variable **objPlayer.ammo** to the watches list. As you can see, it is the same as displayed in the Score Panel. So it seems to work as it should. You can add any expression you like to this watches list. It can be very useful when you are trying to determine what is wrong.



We must fix this bullet problem. It is easily done. Stop the game and open the bullet object window. Add an **Outside Room** event. (Of the **Other** type). In this event, simply **Destroy the instance**. Now, run the debugger again and test the same thing. Does it work better? The bullet instances should now decrease as the bullets fly out of the room. Good.

This is a very common error - forgetting to take care of the instances that move out of the room. This can for example happen to enemies too. Or to the player, as we will see now.

Start the debugger again and jump up the platforms to the right. Finally, jump up through the "ceiling" and up/right out of the room. The player disappears out of view and nothing else happens. **Where** has the player gone?

Add a new watch and enter **objPlayer.y**. You can also add **objPlayer.x** to see what happens to the x coordinate. You should now see that the player is falling... and falling... and falling... eternally (Reminds me of the "gliding" accident in Wheel Of Time). We have to fix this too, otherwise the game would "lock up" in an impossible to solve mode when the player falls out of view.

Stop the game and open the player object. Add the **Outside Room** event. Now, we can not just destroy the player for being out of view. This is because at times, the player might **jump** through the ceiling, but land back on a platform, still in the game. If we just destroyed the instance in this event, the player would die from jumping up out of the room. The player should not die until falling **below** the bottom of the room. So, in this event, add an **If a variable has a value** action. The variable to check is the **y** variable. The **operation** to perform is **larger than** (since **y** is increased downwards). But what value should we enter? The default room height is **480** pixels, but perhaps we want differently sized rooms and how could we then know when the player is below the room? Fortunately there is a variable that can be checked for this. It is called **room_height**. So, simply enter **room_height** in the **value** property.

Now, after that check, start a block and do the usual death thing. Play the death sound, sleep 1000 ms, decrease the lives and restart the current room. Then end the block again.

Try the game and see that the player is now killed if she falls below the room. Great! Now we can also add bottomless pits to the game. <Sadistic smile here>

Another annoying thing is that the Score Panel sometimes is drawn behind other objects (like the falling blocks and such). This can be fixed through changing the **depth** of the object. In the Life Controller object, set its **depth** to **-10** or something similar. The **depth** setting determines which objects are in front of other objects. The lower the **depth**, the more in front something is. Basically, all instances of the objects are drawn from highest depth to lowest depth, meaning the instances with a higher depth get drawn first, and the other ones on top of the first ones.

That is that for the fourth lecture.

Good luck on the assignments!

Carl

Assignment 4 - Add features to the Platform game

Due date: Thursday, 7 July 2005, 08:00 AM

Maximum grade: 100

In this assignment you will make some additions to the platform game that we developed in lecture 4. The following features should be added to the game:

- A starting screen with game name (use your imagination here) and a starting/help instruction. (Similar to the previous games).
- An info screen (also like in the previous games).
- A new type of "monster" - a bouncing ball. This monster should bounce around against the walls and the platforms.
- Walls that can be destroyed by shooting at them five times.
- Trampolines that make the player jump higher when colliding with them. Needed in order to reach high platforms that otherwise would be unreachable.
- Doors that open when a key is collected.
- Add 5 new levels to the game. Make them increasingly difficult. Part of the grading will be based upon the level design.

I hope that is enough for you to do. ;)

As usual, compress the gm6 file with some zip program and upload it here.

Good luck!

Carl

Bonus assignment 4 (optional) - More features to Platform game

Due date: Thursday, 7 July 2005, 08:00 AM

Maximum grade: 100

Yes! Do not stop at the mandatory assignment. Go on to do this too. 😊

Though, know that you are **not required** to make these optional assignments. They are mostly there just to keep you going.

Here is what you could add to the platform game:

- Mines that can be picked up just like the "normal" ammo. They are then laid with e.g. the "M" key. When laid, they stay in place until hit by a monster. It will then explode, killing the monster.
- Of course... add a walking monster that is tough and can only be destroyed by a mine explosion.
- Add water. In water, the gravity is slightly upwards. Also, the movement is completely different. The player can move in all directions, but quite slow. When under water, there is an oxygene level that slowly drops, and when no oxygen is left, the player dies. (Note that this might be pretty hard to implement).
- Moving platforms. This is also quite tricky to get right. Add both platforms that move horizontally and that move vertically.
- Dangerous spikes that appear and disappear from the floor at some locations.

Good luck!

Carl

Read part of the GM manual

Due date: Thursday, 7 July 2005, 08:00 AM

Read the following sections of the Game Maker 6.1 manual:

- MORE ACTIONS
- MORE ABOUT ROOMS
- FONTS
- PATHS
- TIME LINES

(Nothing to submit here either...)

Carl