

Lecture 3 - A maze game

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

Goal of the lecture

After this lecture, the reader should know about maze games. The reader should be able to create simple grid-based games, moving from room to room, creating moving monsters and using the highscore table.

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

In the third lecture we are going to learn the basics about creating a maze type of game. If there are any other "veterans" out there who owned a Commodore 64 in the 1980's, you will know what I mean when I mention **Boulder Dash**. Boulder Dash was a typical maze game. The player ran around in caves, digging dirt, collecting diamonds, pushing boulders and watching out for monsters. I spent *many* hours playing Boulder Dash, just to see what the next level would be like and what new surprises the game developers would throw at the unexpectant player. Ah, the memories.

The game we will build in this lecture reminds me a lot of Boulder Dash. This lecture, as well as the previous one, is built upon a tutorial by Mark Overmars. Thank you, Mark, for giving me permission to use your tutorials as a base for the lectures!

Game idea

In the beginning of the game creation process is the game idea. The game idea springs from some kind of brainstorming, or perhaps just an occasional idea that strikes you when you are in the shower. So, quickly bring out a piece of waterproof paper and jot down a small note describing the game idea.

The idea behind this game, which we simply call **Maze** is that the player should move around in a maze-like room trying to find the way to a certain spot in the room called the **goal**. On the way to the goal the player encounters numerous puzzles and dangers that must be solved or passed. This could be for example diamonds that need to be collected, dangerous holes in the ground, blocking walls or deadly monsters.

A good game is exciting, surprising and addictive. The player should want to stay with the game "just a little longer" in order to see what will happen next and try to solve the next puzzle.

So, the objects that will make up this game are:

- A person, controlled by the player
- Walls of different sorts
- Diamonds to collect
- Bonus items, like coins, rings, etc that have special meaning
- An exit object that, when touched, leads to the next room
- Moving monsters.

This looks like a lot of stuff, but we will try to take it all in good order. A good thing to do is to store different versions of the game as the development goes on. In this way it is always possible to go back to a previous version if something should go wrong. It is also fun to go back just to see all the new spiffy features that are added to the latest version compared to the old versions.

Maze 1

First version of the maze game. Create a new Game Maker file and save it as **maze_1.gmd**. All images and sounds needed for the game can be found in the [resources zip file](#) for this lecture. In the first version of the game we start out with just the main character ("person"), the walls and the exit object.

Add three new sprites and load the following images to them:

Sprite name	File name	Image
sprWall	wall.gif	
sprGoal	goal.gif	
sprPerson	person.gif	

The wall

Now that we have the sprites it is time to create the objects that use the sprites. First, create the wall object, call it **objWall**. Make it use the wall sprite and make the object **solid**. Thus, no other objects will be able to pass through it. Oh, and remove the **Transparent** check mark in the **sprWall** sprite, otherwise it does not look good. No part of the sprite should be transparent, since it is completely rectangular and all pixels of it should have a color.

The person

Next, create the person object, **objPerson**. The person should here be represented by the teddy bear sprite (**sprPerson**). The person object should react to player input from the keyboard, and it should not collide with a wall. It is customary to use the cursor keys for movement in games like this.

Now it is time to discuss movement. How should the person object move? There are a few different alternatives. Some of those are:

- Jump one complete "step" (grid unit) each time a cursor key is pressed. Used in older role-playing games.
- Move smoothly in the direction of the pressed cursor key for as long as the key is held down, then stop at the next grid point. This is most like **Boulder Dash**, and also used in more modern role-playing games and maze games.
- Keep the player moving in one direction until another key is pressed, or the player hits a wall. Used in e.g. Pacman.

We will here choose the second alternative and try to move the person as long as a key is being held down. The person should then stop at the next grid intersection. Why do I keep talking about this grid and stuff? That is because the game world will be built from discrete blocks that all are 32 pixels wide and 32 pixels high. This construct is made because of a number of reasons:

- It is easier to design the levels if all walls are straight and all blocks are placed in a "grid".
- It is easier to store such levels on files, useful for level design applications and run-time reading of level files.
- If the blocks are placed in an arbitrary location, not related to any grid, there is a higher possibility that the player gets "stuck" somewhere in the level.
- The "AI" for the monsters is much easier to implement if all objects are placed according to a grid.

So, we will do just that. All objects in the game will be placed according to this imaginary grid, and there is a finite number of "locations" in a level.

Lets set up the events and actions for the person object. Add a **keyboard** action for the **<Left>** key. Before the actual moving starts, a check must be made to see that the person is standing on a grid point. If the person is in between two grid points, it should just continue moving in the same direction.

So, we have to test if the person is aligned with the grid. There is an action for that in the **control** tab, **If instance is aligned with grid**. Add that action to the **<Left> keyboard** event. In the window with properties for the action that appears, enter **32** for both **snap hor.** and **snap vert.**. This means that the grid that the person object is tested against is 32 x 32 pixels per "cell". These control actions work like this: If the test that the action performs resolves to "true", the following action in the list will be executed. If the test resolves to false, the next action in the list will be ignored, and the action sequence will go on with the second next action. That is useful in this case. We want that if the person object is aligned with the grid when the cursor key is being pressed down, the person object should change direction according to the cursor key. If it is NOT aligned, nothing should happen.

So, after the **If instance is aligned with grid** action, add a **Start moving in a direction** action. Set the direction to **left** (click on the left arrow) and set the **speed** to **4**. We must be careful to choose a speed that is a factor of the grid size. $32 / 4$ equals 8, and so 4 is a good choice. Other values could have been 1, 2, 8 or 16, but none of these suites very well. If any other, arbitrary number is used for the speed, the person object might never be exactly on a grid intersection, and the player will not be able to change the direction of the object. We **could** also choose to set the speed like **32/4** or **32/9** or something, but 4 is a suitable speed.

Add actions similar to the ones above for the other three cursor keys too. We also need a way to **stop** the person. This is solved through adding similar actions to the **<No key>** event that is also found under the **Keyboard** events. In that event, click the middle square in the **Start moving in a direction** action to stop the person there. Don't forget to add the **If instance is aligned with grid** action first.

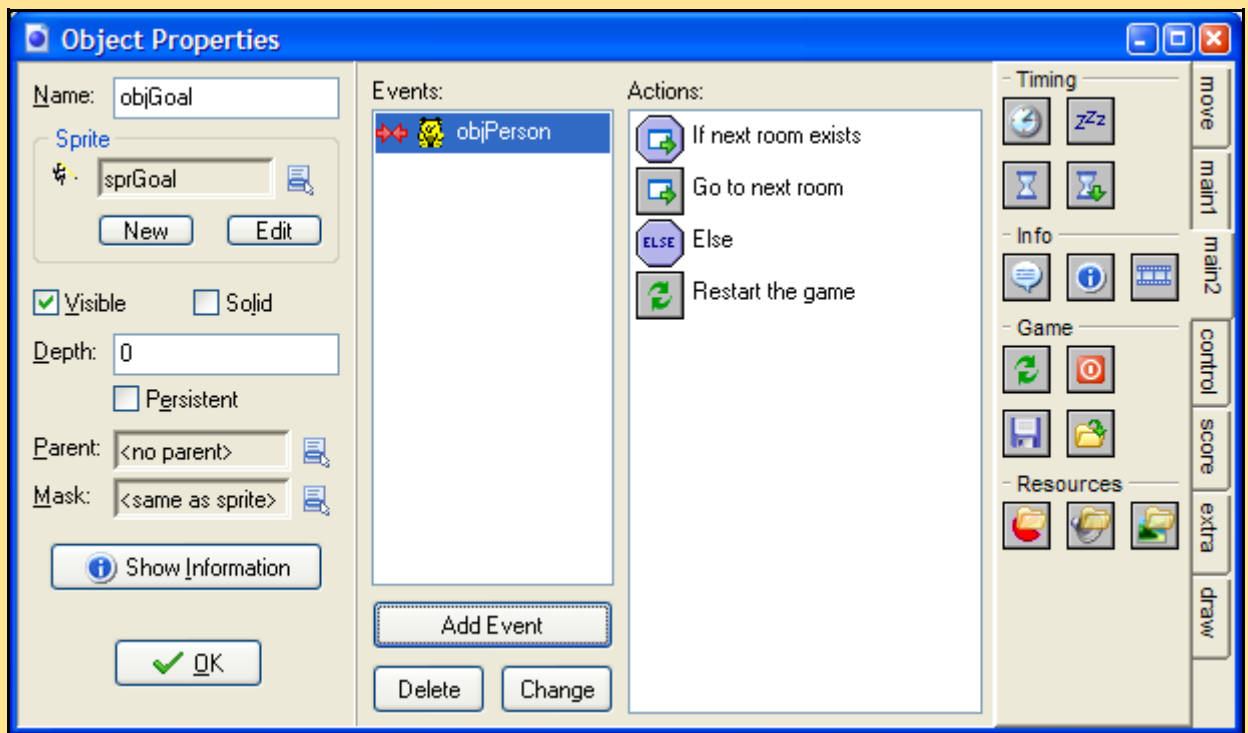
Now only one thing remains for the person object so far. If the person collides with a wall, it should stop. So, add a collision event with the wall object. In the action list for that event, stop the object like in the **<No Key>** event (without the grid check first). The problem that **might** occur here is that the object might be too close to the wall when it is stopped and that when stopped it is no longer aligned to the grid, and thus it is no longer possible to move with the cursor keys. That can be fixed through adding a **Snap to a grid** action to the action list, after the stopping action. In the new action, set the grid to 32 x 32, just as in the test action. This should, after collision and stopping, move the object to be aligned with the grid again, making it possible to move.

The goal

Next, create an object that we call **objGoal**. This object should be represented by the goal flag sprite. The goal object should **not** be solid, because the player needs to move on top of it. When the player "collides" with the goal object, the game should continue to the next room. So, add a collision event to the **objGoal** object (collision with the person object). We could now add the action to go to the next room. There is a little problem though. If the current room is the last room, then there is no next room to go to, and Game Maker will generate an error. We thus have to test if there really **is** a next room to go to before trying to go there.

In the **main1** tab there are actions for this. First, add the action **If next room exists**, then **Go to next room**. Good. But what if this now **is** the last room, then nothing will happen here. How about **restarting** the game if this is the last room? That could be done like this: Add an **Else** action (**control** tab) and, after that a **Restart the game** action (**main2** tab). **Else** works like this: if the last **test** that was conducted resolved to **false**, then the action following the **Else** action is executed, otherwise it is not. It is kind of an inverted test. (If you are a programmer I am absolutely sure you know of if - else statements).

The event with actions will then look like this:

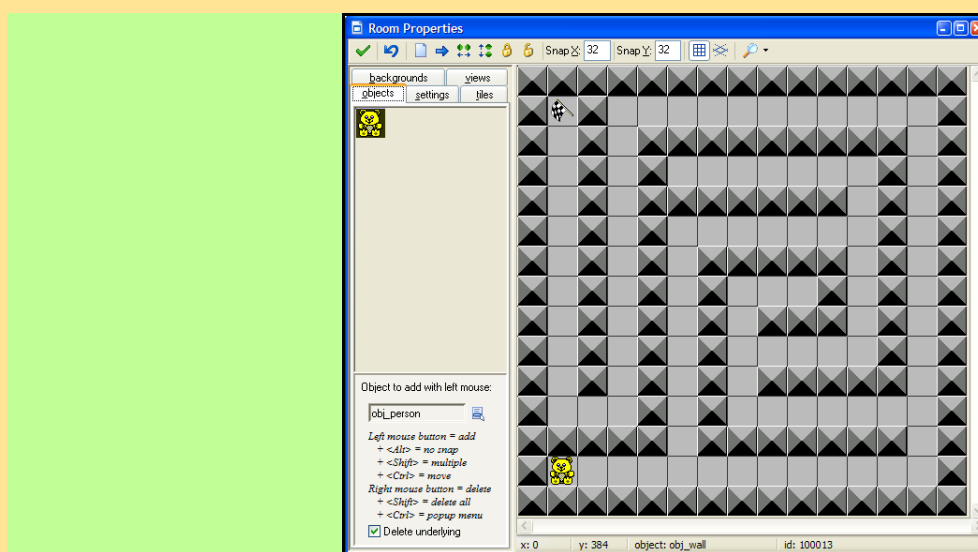


Of course this does not work so good in the final game. It would be better with a highscore entering or something, but we will come to that later on.

The rooms

Now the time has come for the level design. Create at least two rooms. To make it easier when designing the levels, set the **Snap X** and **Snap Y** in the top of the room window to **32**. Add instances of the wall object to the rooms in such a fashion that they become mazes. Add a goal instance at the end of the maze, and add the person instance at the beginning of the maze. Then, save the game and run it. It should now be possible to move the person across the maze using the cursor keys. When colliding with the goal flag, the next room should be entered. Be sure to add walls completely enclosing the entire room, otherwise the player might be able to walk outside the game area, which is not intended here.

If you are completely out of imagination when it comes to making a maze, here is an example from Mark Overmars' tutorial:



Note that the room has a different size than the default. The size of the room can be changed in the **settings** tab in the **Room Properties** window. The room above is 480 x 480 pixels large. Make sure you choose a size that is a multiple of 32, otherwise the grid will not end even along the edges of the room.

Another thing that can be set in the **settings** tab is the **caption** for the room. The caption is what shows in the window's title bar. Set it to something witty that has to do with the room design.

Now that the astonishment from the test of the first maze game version has settled a bit it is time to add a few goodies to the game. A game that has levels that contains the same thing over and over again is not interesting for long, even though the form of the maze changes. There is definitely a need for surprises and bonuses in order to make the game exciting to play.

Diamonds are a person's best friend

For a starter, it was initially meant that the person should be collecting diamonds, and that, in order to complete a room, all diamonds in that room must be collected. The diamonds are no problem, but how do we make sure that the goal flag can not be reached before all diamonds are taken? There are, as I see it, two solutions to that:

- A door is placed so that it blocks a passage to the goal flag. When all diamonds are taken, the door opens (disappears).
- The flag is not placed in the room until all diamonds are taken. This is solved through adding a flag **marker** object. This object is invisible, but when all diamonds are taken, it creates a goal flag object where it is, then it destroys itself.

In order to be truly pedagogic, I choose to implement both methods in the game. We thus need three new sprites and three new objects. Add the sprites **sprDoor**, **sprDiamond** and **sprGoalMarker**. To these three sprites, load the images **door2.gif**, **diamond.gif** and **goalmarker.gif** respectively. Remove the **Transparent** checkmark from the **sprDoor** sprite, since the image does not have any parts that should be transparent, just like the walls.

The game also is a bit dull without sound effects, don't you think? Let us add some sounds to the game. There should be a sound for picking up a diamond, for opening the door respectively revealing the flag (all diamonds picked up) and for reaching the goal flag. Create three sound entities and call them **sndDiamond**, **sndDoor** and **sndGoal**. To these sounds, load the sound files **Diamond.wav**, **Door.wav** and **Goal.wav** respectively.

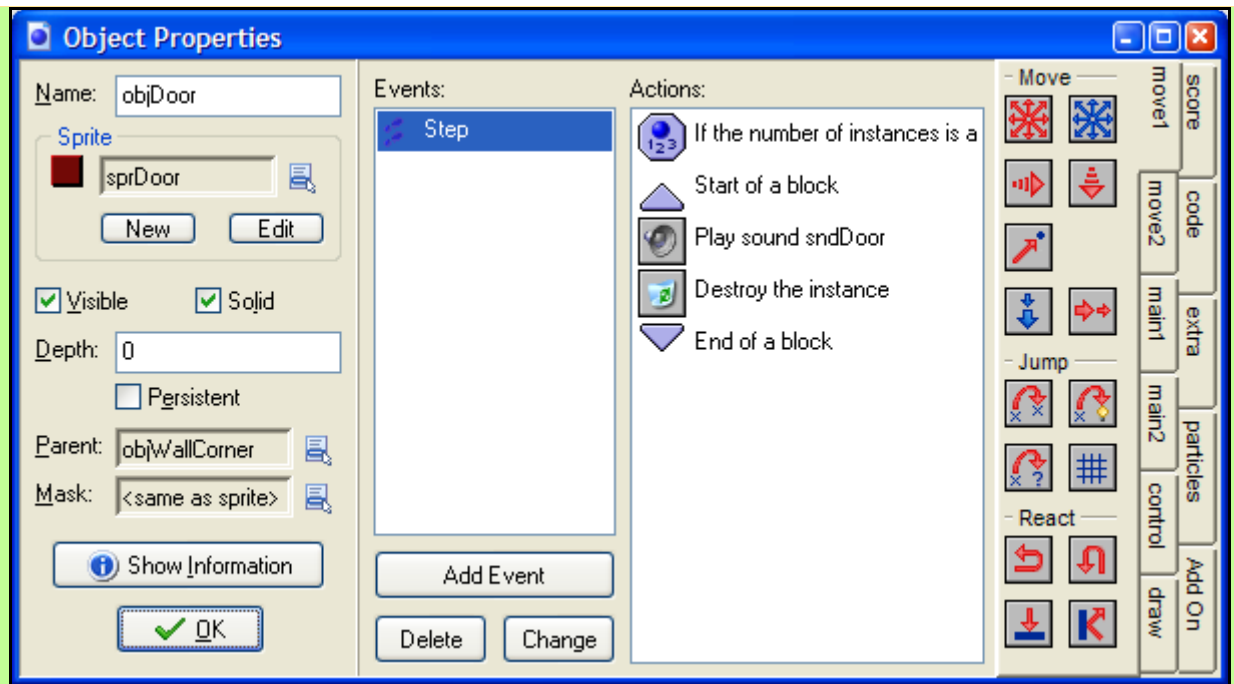
Now, add three objects - **objDoor**, **objDiamond** and **objGoalMarker**. Make them use the correct sprites.

Start out with the simplest object - the diamond. The only event it should have is a collision event with the person object. It should then play the diamond sound and destroy itself. The **Destroy the instance** action should be set to **Self**, nothing else.

The door object should act as a normal wall and thus be made **solid**. In the person object, add a collision event with the door object, stopping the person and aligning it with the grid, exactly as with the wall object. Back to the door object. When all diamonds are picked up, the door should play a sound and destroy itself, just like the diamonds. The problem is, **when**, should it do that?

There is an event called the **Step** event. That event is executed once for each **step** of the game. That usually means that it is executed 30 times per second. It is very useful for things that should be checked often in the game. Add that event to the door object. Now we need to check if there are any diamonds left. That is done through adding the **If the number of instances is a value** action (**control** tab) (phew, what a long action name). In the properties for this action it is possible to set which kind of instances to check (**object**) and for how many to check (**number**). Set the **object** to **objDiamond** and the **number** to **0**. It is also possible to choose which operation to carry out here, but **Equal to** will be fine in this case.

Now, we need to **both** play a sound **and** destroy the object if the test resolves to true (no diamonds left). How can we do that? An **if** action just controls the **next** action, not the following ones. This is where the **block** actions come in. There is one action called **Start of block** and one action called **End of block** (look like triangles). They are used if more than one action should be affected by an **if** action. After the **If the number of instances is a value** action, add a **Start of block**. Then add an action to play the door sound and an action to destroy the door instance. Finally, add an **End of block** action. Now the actions that should be controlled by the **if** test are confined within the blocks. That is how they work. It should look like this:



Finally, the goal marker object. This is actually very simple. First, make the object invisible through unchecking the checkbox **Visible**. Then, use the same event and actions as for the door object, **but**, instead of the **Destroy the instance** action that destroys the door, use a **Change the instance** action (**main1** tab). Set **Change into** to **objGoal**. This will change the goal marker into the "real" goal object. **Perform events** can be left at **not**. This has to do with the **Destroy** and **Create** event of the respective objects. They should normally not be executed when changing objects like this (depending on the specific case of course, but usually not). The goal marker can now be placed in the room where the goal should later appear when all the diamonds are gone.

Another brick in the wall

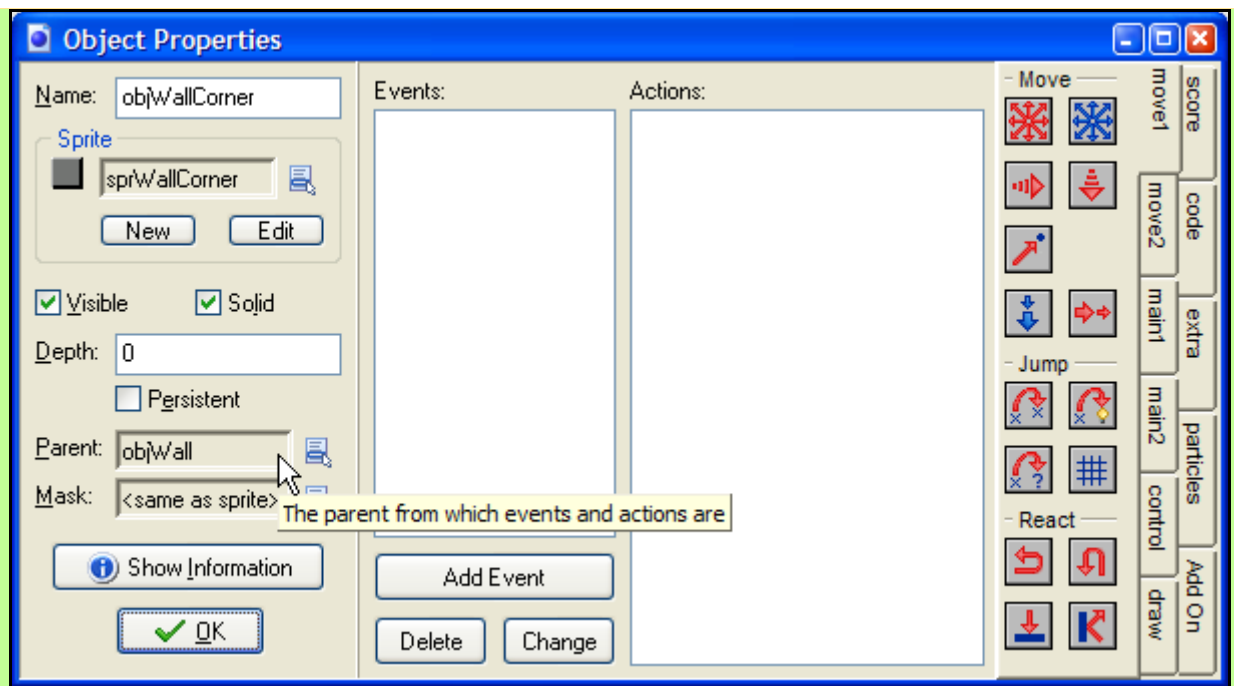
Now we can make other parts of the game a bit nicer. That can be done in many ways.

For a starter, it does not look so good when the wall objects all look the same. It would be better if there were different objects for corners and for vertical and horizontal sections of the wall. There are such sprites in the resource folder. Make three new sprites and three new wall objects for this. You can leave the old wall object the way it is. Make sure to remove the **Transparent** checkmark from the new wall sprites, just as with the old wall.

Now, before starting to add any event and actions to these new objects, we will take a look at the Game Maker feature **Parent and children**. If you know anything about object-oriented programming, you ought to know about inheritance and polymorphism. One could say that Game Maker implements parts of these concepts to a certain degree.

What am I jibbering about? Well, in Game Maker it is possible to make an object the **parent** to other objects. This means that the children objects **"inherit"** all properties of the parent object. Such things as events and actions are inherited, but it also works when defining e.g. collision events from other objects. If an object is set to look for collisions with a parent object, it will also detect collisions with objects that are children to that parent. This is extremely useful if you have a large number of just slightly different objects that almost work in the same way. Ofcourse it is possible to "override" the parent's behaviour on certain events if that is desired.

We will now utilize that behaviour for the wall objects. The "old" wall object already exists in a collision event in the person object (the person stops when colliding with the wall). Instead of adding such collision checks with all the new wall objects, we make the old wall the parent of the new walls. In the **Object Properties** window of the new wall objects there is a property called **Parent**. There it is possible to set an object that will be the parent of this object. Select the old wall object there. Do this for all the new wall objects. Also make them **solid**. That property is not inherited.



Now, you can do the same with the door object and then remove the collision with the door object from the person object. The only collision event that should be left in the person event now is the collision with the **objWall** object. The door object will now also be a child to the **objWall** object, **but** it has some extended behaviour, namely in its step event where it checks the diamonds as before.


The walls in the room should now be changed to the new objects so that they resemble this image:



As you can see in the image above, a background has also been added (dark greenish blue or something). Add a background using e.g. the image **background2.bmp**. Make sure it is tiled when selected in the room background settings.

Heeeee scores!

In order to make the player play a little longer, he/she should be rewarded in some way. That is easily done with score. Everytime the player does something good, the reward is a little more score. (This reminds me of teaching dogs to obey commands. :))

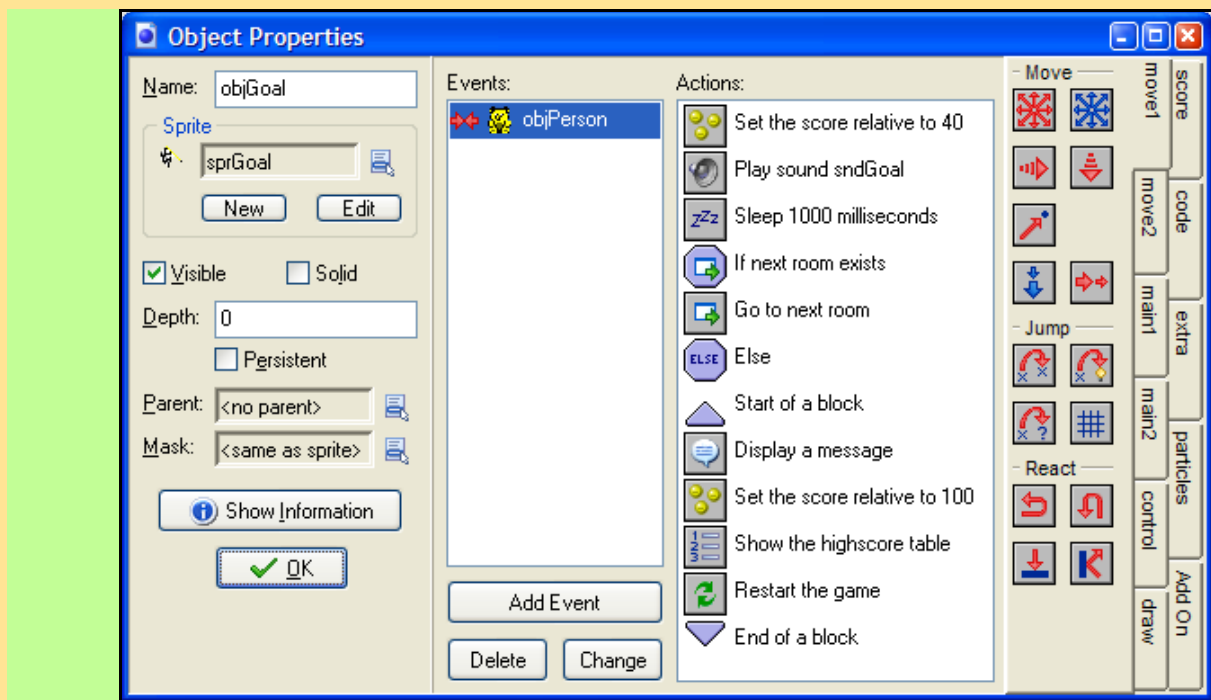
Each time the player picks up a diamond, the reward should be 5 points. For reaching the goal, 40 points. This is done through adding the **Set the score** action () to both the diamond and the goal object, both in their collision event with the person object. In the properties for the score action, set the **new score** to 5 and 40 respectively, and make sure to mark the **Relative** checkbox. Otherwise the score will only be **set** to 5 or 40, when what we want is for it to be **added** to the already existing score.

Now, what good is a score if there is no highscore list? Fortunately Game Maker includes such a feature. The highscore list in Game Maker is very easy to handle. We will use the goal object to show the highscore list. Let us first think about what the **goal** object should do in the collision event with the person.

First it should add 40 points to the score and play a sound. It then ought to pause for a little while before checking if there are any more rooms in the game. If there are more rooms, it should go to the next room. If there, however, are no more rooms, it should perhaps add a little completing-the-game-bonus before

showing a message: "Congratulations on completing the game", and then the highscore table. Finally, it should restart the game.

Instead of instructing you in everything, I will just show you how the action list should look like, and then you do your best to make it look the same:





The sleep (pause) is just to let the sound play to the end before going to the next room and to not startle the player too much with this operation.

Currently the score is automatically displayed in the window caption. That does not look very professional. We will remove it from the caption soon, but first we make our own score showing object.

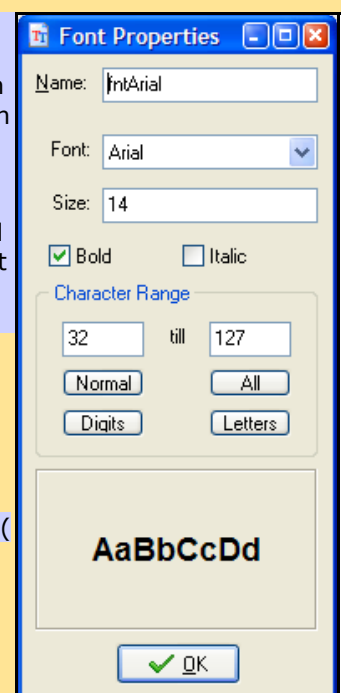
Create a new object called **objControllerMain**. A **controller** object is usually an object that is not visible, or at least does not use a sprite. Instead it is used to manage something (often invisible) in the game. This controller will be managing the display of the score in the game, and therefore it **should be visible**. Add the **DRAW** event to the new object. The **DRAW** event is triggered every step when the instance of the object is about to be drawn on the screen. If the **DRAW** event is not specified, Game Maker automatically draws the sprite that is assigned to the object. This is what always has happened so far. It is now time to fiddle a bit with that.


Now, in order to draw text in the game, it is necessary to add a **font** to the game. This is new for version 6.0 of Game Maker. Add a new **font** entity and call it **fontArial**. Select the font "**Arial**", which is a common font that I think exists on all systems. Set the **size** to **14**, and put a checkmark in the **Bold** checkbox. Then it is possible to select a **Character Range**. In order to save memory, the game author can choose to only include a selected range of characters from the font. What we want to do is to write "**Score: 1234**", so we only need "normal" letters and digits. Therefore the "Normal" ASCII range is sufficient. Click on the **Normal** button in order to select that range. It is also possible to manually enter the start and end of the series of characters. The **Normal** range is from **32** to **127**. If you want to know what characters this includes, you can look at Asciitable.com.

As soon as anything is added to the **DRAW** event, the normal, automatic, drawing of the assigned sprite is skipped. Instead the actions in the **DRAW** event are carried out. Add a **Set the font** action (). In the action it is

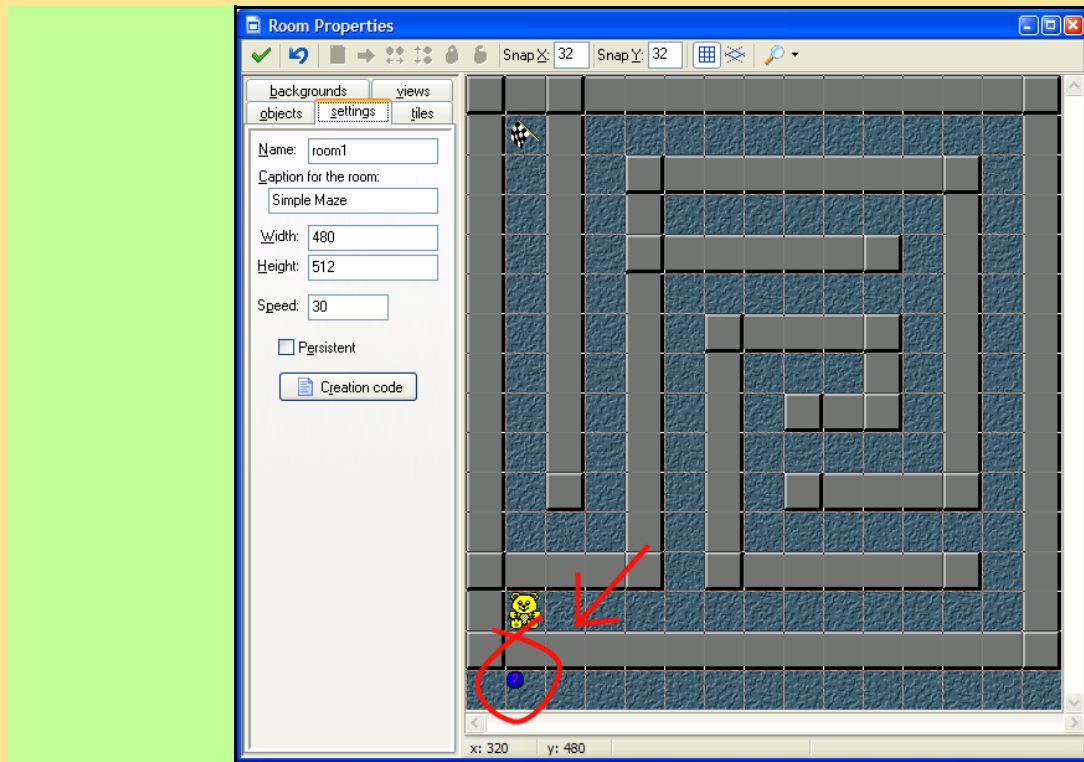
possible to set which font to use when drawing text. Choose the font we just added, **fontArial**. Set the color of the font to white with the **Set the color** action ().

Now that the font is set, we can draw some text on the screen. What we should draw now is the current score. That is done through adding the **Draw the value**



of score action (). As x and y coordinates for the text, enter **0**. Then put a checkmark in the **Relative** checkbox. This means that the x and y coordinates where the text will be drawn will be the same as the location of the instance that executes the action. So, we will have to place the instance of the **objControllerMain** object in the place in the room where we want the score to appear. Finally, enter **Score:** as the **caption**. Add a space character after **Score:**. This action is now done, click OK.

Now we need to add an instance of the **objControllerMain** object to the room where the score should show up. Go to the first room. Click on the **settings** tab and increase the room **height** by 32 pixels (making the room 512 pixels high if you used the size mentioned above). Now add an instance of the **objControllerMain** object to the lowest grid row in the room. You can put it one grid unit inside the room, which gives it the coordinates (32, 480). Since no sprite is assigned to the object (it is not needed), the object is represented in the room design view by a blue ball with a question mark in it. This is perfectly OK.



At times, save and test run the game to see that all the new features work as they should. :)

Starting screen

Are you used to playing games that instantly jumps to the action when you run them? No, me neither. As far as I know, **all** games have some kind of starting screen that welcomes the player, tells the name of the game and offers a way of starting it.

In Game Maker, the easiest way of making a starting screen is to simply create a background that has the same size as the room (e.g. 480 x 480 or 480 x 512). Then draw something nice as the background image using either the internal image editor or, if you have access to, some external graphics program.

(Side note: If you are looking for completely free graphics programs, I recommend two: Pixia and Gimp. Both are freeware. Search the internet for them if you are interested. Pixia is a bit easier to learn, since Gimp was first made for Linux and the user interface is a bit strange for Windows users, at least it was for me. :)

You can also use 3D graphics programs. Of them I prefer [POVRay](#) and [Blender](#) the most.)

Good things to have in the starting screen image are game name, creator and instructions for starting the game, e.g. "Press SPACE to start".

Make a new room and use the new starting screen background as the background for the room. Now, in order for that room to appear when starting the game, drag it to the first position in the rooms list in the **Resource Explorer**.

We now need a little controller object that senses when the player presses space in the starting screen. Call the object **objControllerStart** and make it invisible through unchecking the **Visible** checkbox. Add it to the starting room somewhere, does not matter where. It does not require any sprite either.

In the new object, add a **Create** event. This is where we are going to make some initial settings to the game. First, set the score to 0. Then it should start playing some music. Add a music entity and call it e.g. **sndBackgroundMusic**. You can use the **.midi** file that is in the resources for this lecture if you want to, or use any midi file you like. In the object we just created, add an action to play the background music after the score is reset. Set the music to loop so it does not end suddenly. We should also tell Game Maker **not** to display the score in the window caption here. This is done with the **Set the window caption info** action in the **score** tab. In the settings for the action, set **Show score** to **Don't show**. That was all that is needed in the **Create** event.

Now add an event that detects **<Any key>** on the keyboard. Just add an action to go to the next room in that event. That is it. The start screen is now done.

More levels (or "rooms")

Now, go on and create a few more levels (one or two will do). Use your imagination. Add some diamonds and use either the blocking door method or the hidden goal flag method to stop the player from going on without picking up all the diamonds.

Save the game as **maze_2.gmd**.

Maze 3

Now it is time for version 3 of the maze game. We are going to add monsters, lives, bonuses and a few new puzzle elements.

Lives

Let us start with the simplest thing - to add lives to the game. In Game Maker there is a built-in feature to control lives for the player. The actions for lives control can be found in the **score** tab. In the **objControllerStart**, make sure that lives are set to **3** when the instance of the object is created (**Create** event).

Then, the only thing we should do so far is to draw the number of lives on the screen. There are two main ways of doing this - either as a figure, or as a number of images representing the number of lives left. The second approach looks better, I think.

So, make a **Duplicate** of the person sprite and call it **sprLife**. Edit the new sprite. In the image editor, choose in the menu **Transform->Stretch**. Set **Width** and **height** to **50 %**. Mark **Excellent** and click **OK**. This will scale the image to half its original size, which makes it good for displaying lives. Then, in the **Draw** event of the **objControllerMain**, we are going to make some changes.

First, change the **x** value of the action that draws the score to **300**. This means that the score will be drawn a bit further to the right on the screen. This is in order to leave place for the life images to be drawn. Then, add the action **Draw a text** and draw the text **Lives:**. Set the x coordinate to 8 and put a checkmark in the **Relative** checkbox. Finally, add a **Draw the lives as image** action. In it, set **x** to **70**, select the **sprLife** as sprite and put a checkmark in the **Relative** box there too. All coordinates used here should be relative to the coordinates of the instance that carries out the drawing.

Playing the game now should display the lives along the bottom of the screen as three miniature versions of the person object.

Monsters

Up to now the game has been way too easy to play. The only difficulty in the game has been to find the way to the goal flag, and that can be done by a three-year old child before breakfast. A lot of games become more interesting after the addition of **Monsters** to them. With the concept of monsters, it is possible for the player to loose "lives", and timing becomes a more important part of the game. Some kind of action is added to it.

We will here create three types of monsters that differ mostly in their moving behaviour. One monster will only move up and down, one will move from left to right, and the last one will move in any direction. When the player hits a monster the level is restarted and the player loses a life.

The first thing we do is add the three monster sprites to the game. Call them **sprMonster1**, **sprMonster2** and **sprMonster3**. As images, use the files **monster1_nice.gif**, **monster2_nice.gif** and **monster3_nice.gif**. If you look at the sprites after loading their images, you will see that they consist of 4 images each. Thus it was animated gif files that were loaded. The different images depict the monsters moving in four different directions. We will use that to make it look better in the game.

Make three new objects and assign the corresponding sprite to them. In the **Create** event of the first object, make it **start moving in a direction** and select both the **left** and the **right** direction arrows. This makes it choose randomly left or right when starting. Set the speed to **32/6**. Yes, you can write an "expression" as the value in an action property. **32/6** will not be an integer, but it is important in grid-based maze games like this that the speed is a multiple of the grid size. The speed is slightly faster than the speed of the person (which moves at $4 = 32/8$ pixels per step), just to make it a bit more interesting. Otherwise it would be possible to simply run away from the monsters.

In the collision event with the wall object that is the parent of all other wall objects, simply add the action called **Reverse horizontal direction**. Done! No, wait... the different sprite images also have to be fixed. If nothing is done, the poor monster will rotate its way through the game (you can add the monster to a room and see what happens). That is because the default setting is that a sprite animation is always played through in a loop. To make it look sane, we will set two instance properties that are called **image_index** and **image_speed**. Setting those properties (or "variables") in an instance defines the current sprite frame and animation speed. The image numbering starts at 0. Start out with adding a **Set the value of a variable** action to the **CREATE** event. The variable name is **image_speed** and it should be set to **0**. Then, in the **End Step** event (under the **Step** event button), we will add a series of actions from the **Variables** section on the **code** tab in the actions panel. First add an **If a variable has a value** action. Enter **hspeed** as **variable**, **0**, as **value** and select **smaller than** in **operation**. The **hspeed** variable in an instance is the value of the horizontal speed that the instance is currently having. If that speed is smaller than **0**, it means that the instance is moving left. It should thus display image number **3** of its sprite. So, add a new action, **Set the value of a variable**. The variable is called **image_index**, and the value should be **3**.

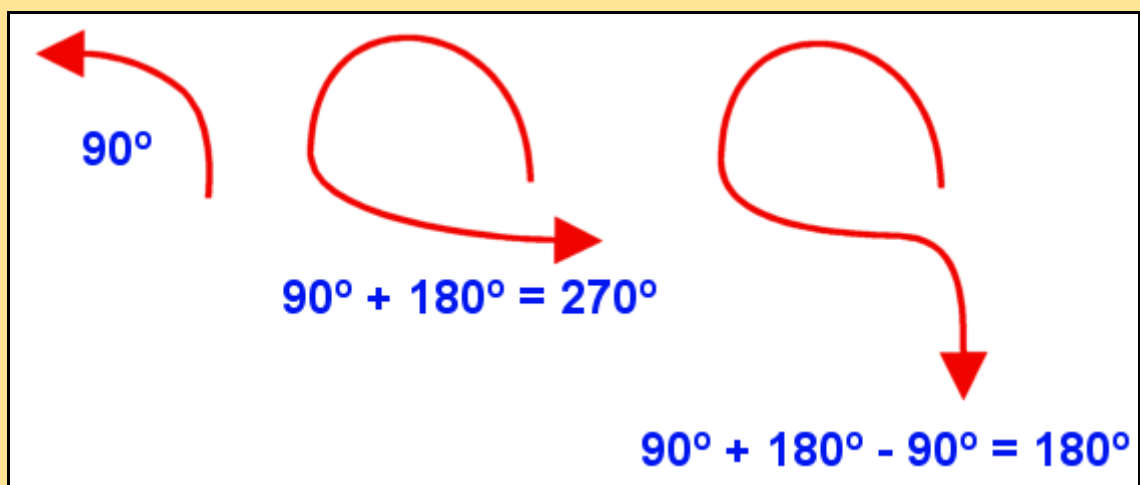
Continue adding this action sequence to the list. Check if **hspeed** is **larger** than **0**. If so, set **image_index** to **2**. Then check the vertical speed, **vspeed**. If that is smaller than **0**, set **image_index** to **1**, and finally, if **vspeed** is **larger** than **0**, set **image_index** to **0**. The action sequence for the **End step** event is now finished. We put this in the **End Step** event because we want to make sure that all collision events and such have been executed first. Now, make identical action sequences for the **End step** event of the other two monster objects too. (Hint: You can select all the actions in an event by shift-clicking them and the do a copy-paste to copy them to other objects).

The **second** monster should work the same, only it should move either up or down in the beginning, and in the collision with the parent wall object, it should change its **vertical** direction.

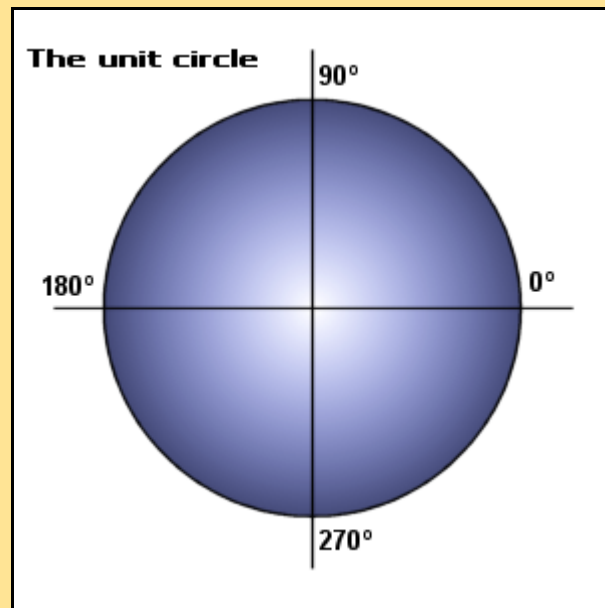
Now for the **third** monster. In its create event it should be set to go in any of the following directions: **up, down, left** or **right**. In its collision event for the wall, it should do this sequence:

- Check if it is possible to turn left 90 degrees. If it is, do that.
- If it was not possible to turn left, check if it is possible to turn right. If it is, do that.
- If that was not possible either, turn around completely.


This will work like this: First, change the direction 90 degrees left. If that is ok, exit this event. If not, turn 180 degrees more, which will in effect be a right turn ($90 + 180 = 270$). If that is ok, exit this event. If not, turn 90 degrees back, which in effect will be a complete turnaroud ($90 + 180 - 90 = 180$). This might sound complex, but see if this image clears it. It shows the angle that is **added** to the initial angle of the instance:



You should also know that in Game Maker, the direction is measured according to the standard unit circle, which starts at zero degrees to the right and then goes counter-clockwise. That means that 90 degrees is straight up, 180 degrees is straight to the left and 270 degrees is straight down.




Now, to the beginning of all the collision events in the monster objects, add the **Snap to grid** action and set the grid size to **32, 32**. This makes the monsters stay in the grid.

The action that is used for setting the new direction is called **Set direction and speed of motion** ()

For **direction**, enter **direction+90**. That means that the value 90 is added to the existing value of the **direction** variable. For **speed**, enter **32/6**. That is the monster's normal speed setting.

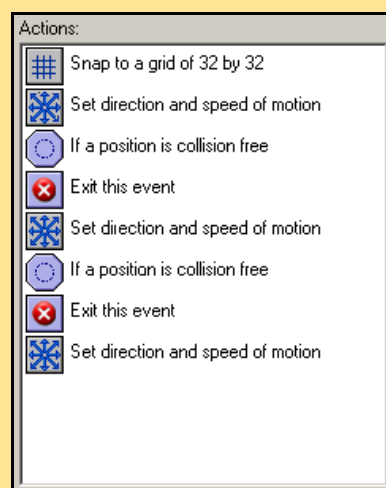
Next, add an action called **If position is collision free** ()


As **x**, write **hspeed**, and as **y**, write **vspeed**. This means that the speed of the instance is added to the coordinate of the instance, and thus the position that is checked for obstacles is the position in which the instance would be moved to in the next step. Also put a checkmark in the **Relative** checkbox.

Now, what should happen if the position that is checked is free from collisions? Nothing should happen then, because the direction has already been changed, and all is fine. So, add an **Exit this event** action ()

After that action, add the **Set direction and speed of motion** action again, this time with **direction+180** as the **direction**. Other things should be the same. Add the **If position is collision free** and the **Exit this event** action there too.

Finally, if none of the above tests work out it means that the monster has entered a dead end. Thus, the only thing to do is to turn completely around, which is done through adding another **Set direction and speed of motion** action and set the **direction** to **direction-90**. See the image above (second above) if there are still problems getting into these behaviours and directions. When all actions are added to the event, it should look like this:

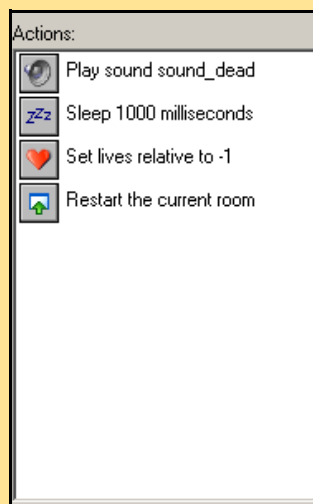


We will add yet another thing to this monster. At random intervals it should change to a random direction. That will make the monsters even harder to predict and avoid. To do this, add the event **Step** to the third monster. In that event, add the action **If instance is aligned with grid** and set the grid to **32 x 32**. Then add an action called **With a chance perform next action** (). This action will, with a certain

probability, execute the following action. The properties of the action is a little cryptic, I think. It says "sides". The **sides** property defines the number of sides of an imaginary die. The computer "rolls" the die, and if it lands on **1**, the next action is executed. Thus, if we enter **2** for **sides**, the next action would have a 50% probability of executing (how does a 2-sided die look? Like a coin?). If **3** is entered, the next action is executed with a probability of 33% (now, how does a **3-sided** die look like? I can't imagine a 3-sided shape... anyone?? Perhaps we shall make a contest? :) :)).

Sorry... anyway, enter **8** as **sides** (an "octaeder" :)). That gives the next action a probability of 1/8 (12.5%) to execute. The next action should just be like the first action in the **Create** event - a **Start moving in a direction** action with the four horizontal and vertical arrows selected. **Speed** should be **32/6**.

When the person collides with a monster, there should be some kind of "death" sound, the game should pause a bit, a life should be removed and the room (or "level") should be restarted. It is important to put the **Restart the current room** action last, since after that action, no more actions in this list will be executed. The room is immediately restarted. The action list in the person object (collision with all types of monsters) should look like this:



The sound that is played could for example be the sound file **Dead.wav**.

If there are no more lives, the highscore table should be shown, and the game should be restarted. This is done through adding those actions to the event called **No more lives**. It could be put into the **objControllerMain** object, since that object should exist in all rooms. The **No more lives** event can be found under the **Other** events button in the event selector.

Blocks and holes

Time to add yet an interesting twist to the game - blocks of rock that the player can push into holes. The holes are initially impassable, but pushing a block into a hole "fills" the hole, and the player can pass over it. Somewhat reminding me of **Sokoban**. Now the rooms can be even trickier with holes that must be filled with blocks in order to pass. It will also be possible to "catch" monsters with the blocks, locking them into tight areas.

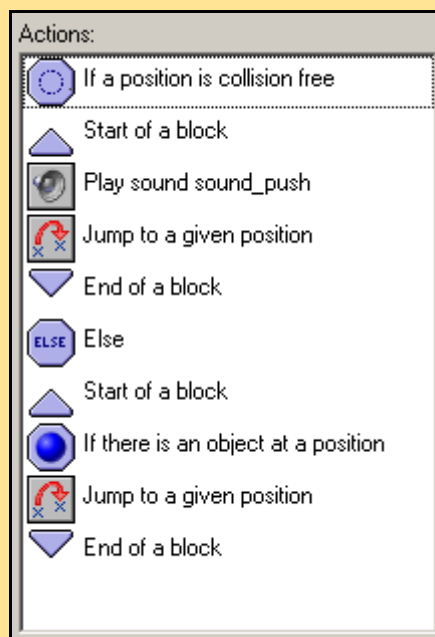
For the block sprite, use the **block.gif** file. For the hole sprite, just make a completely black square (32 x 32 pixels) and make it non-transparent. Add objects for these new entities too. The hole object could have as parent the **wall object** that is parent to the other wall type objects too. Make the hole solid. In the collision event with the block object, the hole should play a sound. How about **hole.wav**? (Make a sound entity, just a usual). Also, both the hole and the block that collides with it should be destroyed. Add two **Destroy the instance** actions after the sound action. Set one to destroy **Self**, and the other one to destroy **Other**. In collision events, **Other** refers to the other instance that is involved in the collision (the block in this case).

The block object is a little more advanced. It should be pushable by the person. For a starter, it should set the wall object as parent. Also, make the block solid. That is because monsters should react to it in the same way as to walls. This brings on a little problem. If the blocks are as walls, then the person object too will stop when hitting a block, and will never be able to push it. To remedy this, in the person object, we add a collision event with the block. This will "override" the default collision event (with the wall). In the event, just add a **Put some comment** action (**control** tab). Enter anything as comment, just so that you know that it overrides the wall collision event.

Now, back to the block object. Add a collision event with the person object here. To that event, first add an **If a position is collision free** action. But where should we check? We want to check a bit further on in the direction the block is being pushed. The direction of the push can be taken from the **other** instance (the person that collides with the block). Here we will use something that I would like to call instance dereferencing. As **x**, put **8*other.hspeed**, and as **y**, write **8*other.vspeed**. Thus we take the hspeed and vspeed from the **other** instance and multiply them with 8 to get the position where the block is being pushed to. The speed of the person is 4, so $8*4$ would give a movement aligned with the 32 x 32 grid. Also put a checkmark in the **Relative** checkbox.

Add a start block and then play the sound **push.wav**. After that, add the action **Jump to a given position** and enter the same coordinates as for the collision free checking action. Also check the **Relative** box. Then, end the block.

Add an **Else** action and start a new block. Here we will check for the special case that the instance that blocks the pushing movement is of the hole object type. If it is, the block jumps there anyway. The instance is checked with the **If there is an object at position** action. Enter the same coordinates once again and check the **Relative** box. Also, in the property **object**, select the hole object. After that action, copy the **Jump to a given position** from above and use it here too. Same properties. Finally add an **End block** action to close it all up. The entire action list should look like this:



With this new feature - blocks and holes - it is possible to construct completely new puzzles for the player to solve. It is however now possible to get locked into a level and not being able to either "die" or complete the room. To solve this, we need to add a "restart the room" key for the player to use in emergency situations. Kind of a suicide key. We could pick e.g. the **R** key for this.

In the **objControllerMain**, add an event to "listen" for the **R** key. In that event, play the death sound, decrease lives by 1 and restart the current room.

Another feature that would be very good for debugging the game is to also add events for the keys **N** and **P** here. Those keys would immediately take the player to the next or previous room respectively. Do not, however, tell the player of this functionality. It will then work kind of a "cheat code". You could of course remove these events once the game is tested enough and ready for publishing.

Now, add a few rooms where the player has to use the blocks in order to solve the game puzzle.

So far, so good

So far, I hope that you have been able to keep up with the course. The assignments for this lecture will be a bit tougher than previous assignments (of course :)). You will add a few features to the game, like some bonuses, more rooms, a power ring, explosives, etc.

Good luck!

Carl

Assignments

Assignment 3 - Add features to the maze game

Due date: Tuesday, 14 June 2005, 08:00 AM (32 days 18 hours)

Maximum grade: 100

In this assignment you should add some features to the Maze game that we developed in the third lecture.

The following features should be added or changed in the game:

- Make an info document, similar to the one for the Catch the Clown game. It should include game information and instructions for play. Do not forget to mention the R key for restarting a room. You should also tell about the save and load game keys, which we will mention later in this assignment.
- Make the start screen show information about how to view the help file and also offer to load a game with the load game key (see below).
- Make the person object have a better sprite that works similar to the monster sprites with four directions. You can use the image **person_nice.gif** for this if you want to. The image should thus change according to the direction in which the instance is moving, just like the monsters.
- Add two kinds of bonuses to the game: one that gives the player 100 points (use the **bonus.gif** image for this) and one that gives the player one extra life (using the **life.gif** image). Make those, especially the life bonus, rare in the game. Perhaps hidden behind holes or doors.
- Add information about the save/load game feature. How do we add that feature? Easy - it is already done. 😊
Game Maker includes an automatic save/load game feature. It is mapped to the F5 (save) and F6 (load) keys by default. Just mention it in the info and on the start screen (load game).
- Add bombs and triggers to the game to be used in some levels. You can use the **bomb.gif** and the **trigger.gif** graphics for this. When the person collides with the trigger, all bombs should explode (**explosion.gif**) thereby destroying all objects - walls, blocks, monsters, **persons** - in the nearese vicinity (radius of 32 pixels or something).
Hint: Check the **Destroy instances at position** action.
Also, be careful to place the explosion animation in the right place relative to where the bomb was. Their sprites differ in size.
- Make some new rooms (levels). For the highest rating, there should be at least 5 new rooms in the game. We will try to rate this assignment much on the level design. Try your best to be imaginative and make use of all the features there are in the game. Remember to put easy rooms in the beginning and from there increase the difficulty step by step. Also, do not put all the interesting objects in the first rooms. Give the player the new objects with a few rooms in between.

Just to make you happy, there will also be an optional bonus assignment for you all. There is not end to all the fun! 😊

As usual, zip the gm6 file and upload it here. If you do not have any zip program, I recommend www.filzip.com, which is completely free.

Bonus assignment 3 (optional) - More features to maze game

Due date: Tuesday, 14 June 2005, 08:00 AM (32 days 18 hours)

Maximum grade: 100

This assignment is optional and just meant for those who just can not get enough of work. You are all encouraged to try and complete the assignment, but do not be discouraged if you do not succeed.

Add the following features to the maze game:

- A ring object (**ring.gif**) that, when picked up by the person, makes all the monsters afraid, just like in Pacman! This should be indicated by changing their sprite a bit (perhaps adding a color to it, or "dissolving" it a bit).

When the monsters are afraid, their behaviour does not need to change, **but** when they collide with the person, they should "die" and jump back to their starting position in the room. They should then not be afraid anymore but act "normal".

Put the ring object at some locations in some levels where there are lots of monsters.

- One way streets. These are objects that may be represented by arrows or something in the game. When walking on them, the person is automatically moved in the direction they are pointing, without the possibility to affect the speed or direction with the cursor keys.

- Keys that open specific doors. E.g. yellow keys that open yellow doors, blue keys that open blue doors, etc.

- Transporters that make the person immediately jump from one location to another location in the same room.

- Even more rooms. 😊

You are of course welcome to add more features, all to your liking.

Read part of the GM Manual

Due date: Tuesday, 14 June 2005, 08:00 AM (32 days 18 hours)

Read the following sections of the Game Maker 6.0 manual:

- ADVANCED MODE
- MORE ABOUT SPRITES
- MORE ABOUT SOUNDS AND MUSIC
- MORE ABOUT BACKGROUNDS
- MORE ABOUT OBJECTS

(Nothing to submit here either...)

Carl