

Lecture 7 - Time Lines and Paths

Written by Carl Gustafsson

Goal of the lecture

After this lecture the reader should be familiar with the entities **Time Lines** and **Paths** in Game Maker.

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

This lecture will build upon the two previous lectures (5 and 6) and we will continue developing the space shooting game. The entities **Time Lines** and **Paths** will here be introduced and we will look at some ways of using them. There will also be other additions to the space shooter game.

Levels and stars

The first thing we will do is to make use of the Time Lines feature of Game Maker. To do that, however, we need to add a few things to the game first.

Levelling up

I thought that we would add the Time Lines to the second level in the space shooter. Hmm. It is still called "space shooter". We should come up with a good name for it. In memorial of a little game I started on while in college (or whatever the school is called in English where you go when you are about 17-19 years old), writing in Pascal. Let's call the game "Total Holocaust" :) (Yes, this was years **before** "Total Annihilation". Guess they got the inspiration from my game. :) :) NOT).

Anyway. The change of a level is usually made either after the player has completed some task, like destroying a certain number of enemies or achieving a certain score, or a level change can be triggered after a certain amount of time. I think the best here would be to decide that after a certain amount of time the player reaches the next level.

In the script **EnemyControllerInit**, set alarm 1 to 10 seconds (that is 300 steps with default speed). This is a bit short time, but if we make it any longer, we would have to wait so long each time we try out the game. Actually, you can make it even shorter, if you like to.

How about this: at the end of each level, the player craft starts moving forward across the screen and disappears at the top? Create a new object, **objPlayerLevelEnd**. Give it the **sprPlayer** sprite. We should change the player plane instance into this object type just so that the player can not control it with the keyboard anymore and there will not be any collisions with enemies or bullets. The thought is that we do this in the **Alarm 1** event of the **objEnemyController**. There is a little problem though: there is a slight chance that there does not exist any player instance because it might just have been shot down. So, if the player DOES NOT exist, it should be changed into the **objPlayerLevelEnd** as soon as it appears. There might be many ways to solve this problem, but here is one.

Create another object and call it **objPlayerLevelWait**. It does not need any sprite. Make it **Invisible**. It should wait for the player instance to appear and then change it into the **objPlayerLevelEnd** object. So, in its **Step** event, add the following code (**Execute a piece of code** action):

```

if (instance_exists(objPlayer))
{
    // Change the player instance into a level end instance.
    with (objPlayer) instance_change(objPlayerLevelEnd, false);
    // Set the alarm timer for the "real" level end.
    alarm[0] = 150;
}

```

The **instance_change** function changes an instance of an object type into an instance of another object type. All local variables and such remain intact. The "false" at the end of the function call is there because we do not want the **Destroy** action of the **objPlayer** to execute (it does that otherwise), because it would create an explosion object where the **objPlayer** is.

Go to the **objEnemyController** object. In its **Alarm 1** event, create an instance of the **objPlayerLevelWait** object. The coordinates do not matter. Also, set the **Alarm 0** to **-1** here. That will stop that alarm from triggering anymore so that no more enemies are created.

After the instance changing in the **objPlayerLevelWait** object (see code above) the **Alarm 0** is set to 150 steps (5 seconds). When that alarm triggers, we simply go to the next room. So, in the **Alarm 0** event, add an action to **Go to the next room**.

Now, we should make sure that the **objPlayerLevelEnd** moves straight upward. In the **Step** event of that object, set the speed and direction to: **direction = 90, speed = 10**.

Finally, we need a level display object that tells the player which level he is currently on. In the **Create** event of the **objPlayerController**, set the variable **global.level** to **0**, and, in the **Room Start** event of that same object, add this code:

```

global.level += 1;

```

Now, this introduces a little problem. You see, if we add the **objPlayerController** to each level (room) in the game, it will be created as a new instance each time. That means that the level count is reset each time. This is not what we want. In order to get around this, we can make the **objPlayerController** object **Persistent**. Making an object **Persistent** means that it will live on, with all its variables intact, even though the room is changed. It will continue to "live" for as long as the game goes on, until it is explicitly destroyed (**instance_destroy**). This also means that we will not have to add an instance of the object to each room. It is enough to add it to the first level room. Otherwise there will be lots of duplicates of the object as the player moves from room to room.

So, in order to make the **objPlayerController persistent**, put a checkmark in the **Persistent** checkbox in its object window. Now we only need to have an instance of the controller in the first level room. It will then live on and exist in all the other levels as well.

Let's create a new object, called **objLevelDisplay**. It will be used to display which level we are currently on when it is started. In its **Create** event, set **Alarm0** to **10**, and set the variable **count** to **10**. The **count** variable is just a made-up variable that we will use to make the instance go visible and invisible five times (five "ons" and five "offs" are ten "events"). Now, the event **Alarm0** should make the object visible if it is currently invisible, and if it is already visible, it should be made invisible. This could be solved with a simple **if...else** statement. But we are going to make it even smarter. Write like this in the **Alarm0** event:

```

visible = not visible;

```

Uh, what? What does the above statement do? It seems like a paradox, or some ridiculous philosophical statement. Is this some kind of zen?

No, no. Not at all. Remember that the "equal sign", when used like this, does not **mean equal**. Rather it means **is set to**. So, "visible" **is set to** "not visible".

This statment works about the same as a standard pen mechanism. You know, the one you used to write with before you got a computer :) . When you push the button at the end of the pen, the writing point is extended or retracted every second time. But how on earth does the mechanism know if it should extract the point, or retract it? Easy. It depends on its current state. If the state is "retracted", it should be

"extended" when pushing the button, and vice versa. Back to our statement above:

If the variable "visible" is **true**, which means that the instance is visible, then it is set to **not true**, which is **false**, and the instance is invisible. If it however is **false**, it is set to **not false**, which is **true**, and the instance is visible again. If this still feels like magic, read through it again. It should make sense, I hope. :) Otherwise, I we have just wasted a few minutes of our lives, and can go back to using the **if** statement: if (visible = true) then visible = false else visible = true;

Now, in the **Alarm0** event, also set the **Alarm0** to **10** again, and, finally, decrease **count** by one. Check if it is zero or less and, if so, destroy the instance. I think you can do that on your own by now. :)

In order to write something on the screen, we need to, as usual, add a **Font** resource to the game. You can add any font you like. I have chosen the font "Cockpit" and therefore I called the font resource **fntCockpit**. Use your own name for the font. Then go back to the **objLevelDisplay** object.

Add the **Draw** event to the object. In that, add this code:

```
draw_set_font(fntCockpit);
draw_set_halign(fa_center);
draw_set_color(c_white);
draw_text(room_width / 2, room_height / 2, "Level " + string(global.level));
```

The **font** functions speak for themselves, I think. **fa_center** means that the font is center-aligned when written. The **string** function converts a number (**global.level**) to a string. This is necessary, because the **draw_text** command expects a string as the last parameter, and gets quite upset if that is not the case. So, in effect, this little piece of code will draw the current level with the text "Level " in front of it on the screen. The "plus" sign works as a "string concatenator", that is, it is used to add strings together. Since the object is turned visible/invisible all the time, the text will blink a few times before it disappears.

In order to make sure that the level object is on top of everything else, set its **Depth** to **-10**.

What is left is now just to create an instance of this new object at the correct time. Do that in the **Room Start** event of the **objPlayerController** object. It does not matter where you put the instance, since the text is drawn in the middle of the room anyway.

There remains one problem, and that is to carry over the **energy** of the player to the next level. The energy is stored in a local variable in the instance of the **objPlayer** object. This problem has occurred because of the way that these lectures are built. The game started as a way of teaching GML, and I did not foresee everything that was going to happen on the way here. For example, when I started writing, the version of Game Maker that existed did not have any "health" feature, which was introduced in version 5.3. One solution would of course be to change all references to the energy into health instead. But that is not fun to do.

So, let's do like this instead: Make both the **objPlayer** object and the **objPlayerLevelEnd** object **Persistent**. Then, in the **Room Start** event of the **objPlayerLevelEnd** object, set its location to **(room_width / 2, 400)**, the speed should be set to **0**, and then change it into an **objPlayer** instance (either with the **instance_change** function or with its corresponding drag-and-drop action), **NOT** performing **CREATE** and **DESTROY** actions.

Finally! Add a new room, call it **Level2** and change the name of the first room to **Level1**.

If you now try out the game, you will encounter an error when the levels change (at least on my computer). I find it very hard to explain this error, but it apparently has something to do with changing the instance in the **Room Start** event. So, this is a good place for a little workaround (I think I fought this problem for about an hour before finding its source!!). It **appears** that we have stumbled on a little bug in Game Maker (might have been fixed in the latest GM release).

To get around it, do like this:

Instead of changing the instance in the **Room Start** event of the **objPlayerLevelEnd** object, set the **Alarm0** to **1** step. Then, in the **Alarm0** event, change the instance to the **objPlayer** object. This makes it work better.

An array of stars

I am sure that you have noticed that the second level is still just a gray background. Extremely boring. We could use the parallax trick of the first level here too, of course, but we are here to learn, right? So let's do it in a different way. Time to learn a bit about **arrays**.

All computer languages I know of have support for some kind of arrays. An **array** is like a long list of variables with the same name, but with a (usually) numerical index that tells them apart. Suppose that you have a need to lots of very similar variables, like the location of different stars in our example. We could write it like this:

```
star_x_1 = 10;
star_y_1 = 20;
star_x_2 = 20;
star_y_2 = 20;
star_x_3 = 30;
star_y_3 = 25;
```

Very well. Now, suppose that we want to move all the stars downwards one pixel. Well, here we go:

```
star_y_1 += 1;
star_y_2 += 1;
star_y_3 += 1;
```

Now, imagine that we have 100 stars instead of three. Oh, the pain... Seems like there ought to be a way of making this easier to write. That is correct! We can use **arrays** instead of single variables. Array variables are written like this:

```
variableName[ind
ex]
```

So, applying this to our stars example above would be:

```
star_x[1] = 10;
star_y[1] = 20;
star_x[2] = 20;
star_y[2] = 20;
star_x[3] = 30;
star_y[3] = 25;
```

Seems like very much the same thing? Yes, that is correct, but that is just because there is no good pattern in how the stars are initially placed. Now look at how we move them down one pixel:

```
for (i = 1; i <= 3; i
+= 1)
{
    star_y[i] += 1;
}
```

Hmm. Four lines instead of three? What have we won?? Yes, but this is just a small example. Suppose there were 100 stars now. It would still only require those four lines! Just change the **3** in the **for** statement into **100** and it works for 100 stars. (We learned the **for** statement in the previous lecture, remember?).

Now, if we want to place the stars randomly in the beginning, instead of manually putting each star into place, we could use a similar technique as the movement in the **for** statement above. This is what we are going to do:

- Make the background color for the Level2 room completely black.
- Create a new object, **objStarField**.
- Add it to the Level2 room, **it does not matter where in the room you place it.**

- In the **Create** event, set up some initial variables and star positions (in the arrays):

```
nStars = 100;           // The complete number of stars that will be displayed.
starMaxSpeed = 5;      // The speed of the fastest stars.
starMinSpeed = 1;     // The speed of the slowest stars.
// Create the star array with random star positions and speed.
for (i = 0; i < nStars; i += 1)
{
    star_x[i] = random(room_width);           // Star x coordinate
    star_y[i] = random(room_height);         // Star y coordinate
    star_dy[i] = starMinSpeed + random(starMaxSpeed - starMinSpeed); // Star speed
}

```

(A little notice: "dy" means "delta-y" and refers to the amount the y coordinate is changed for each step. This is, in effect, the speed of the star. Each star is given a random speed).

- In the **Step** event, add this code:

```
// Go through all stars and move them.
// Also check if they are below the bottom of the screen. If so, create move it up
again.
for (i = 0; i < nStars; i += 1)
{
    star_y[i] += star_dy[i]; // Increase the y coordinate with the star's speed.
    if (star_y[i] > room_height) // If the star is below the screen...
    {
        star_y[i] = 0; // ... move it up again ...
        star_x[i] = random(room_width); // ... and give it a new x coordinate.
    }
}

```

- In the **Draw** event, add this code to draw all the stars on the screen:

```
// Go through all stars and draw them on the screen.
draw_set_color(c_white);
for (i = 0; i < nStars; i += 1)
{
    draw_point(star_x[i], star_y[i]);
}

```

As you can see, each **for** statement loops through all the stars in the star arrays and performs some kind of action on all of them. This is called "iterating" an array. The problem with this approach to the starfield is that I **think** that it takes more processing power to do this, but I am not sure. It seems to have improved since earlier GM versions.

There is one thing that could be better though. The depth feeling is not quite right. It could be made better if the stars that are moving slowly were drawn with a darker color. Here we will use a little mathematics to determine which color the star should have. Suppose that the darkest stars should be drawn with a brightness of 100 and the brightest with a brightness of 255 (on a scale from 0 to 255, which is common for computer colors). Here is how we can change that:

In the **Create** event:

```
nStars = 100;           // The complete number of stars that will be displayed.
starMaxSpeed = 5;      // The speed of the fastest stars.
starMinSpeed = 1;      // The speed of the slowest stars.
// Create the star array with random star positions and speed.
for (i = 0; i < nStars; i += 1)
{
    star_x[i] = random(room_width);           // Star x coordinate
    star_y[i] = random(room_height);         // Star y coordinate
    star_dy[i] = starMinSpeed + random(starMaxSpeed - starMinSpeed); // Star speed
    brightness = 100 + (255 - 100) * (star_dy[i] - starMinSpeed) / (starMaxSpeed - starMinSpeed);
    star_col[i] = make_color(brightness, brightness, brightness);
}
}
```

And, in the **Draw** event:

```
// Go through all stars and draw them on the screen.
for (i = 0; i < nStars; i += 1)
{
    draw_set_color(star_col[i]);
    draw_point(star_x[i], star_y[i]);
}
}
```

Note that the **brightness** variable is just temporarily used to calculate the color for each star. It is not stored in any array. What is stored is the **color** of the star (**star_col[i]**). A color is made up of red, green and blue ingredients. To make gray colors, use an equal amount of all the ingredients: "brightness".

Now, the brightness calculation might seem a bit hard to understand, and I understand you. Perhaps, if you have not read as much math, you will find it hard to follow, but basically it maps the speed of the stars (1 to 5) onto a brightness scale (100 to 255). Let's check if it works. Say that the speed of the star is 1, which is the slowest star speed. The result of the calculation would then be:

$100 + (255 - 100) * (1 - 1) / (5 - 1) = \dots\text{calculating}\dots = 100$, which is the darkest brightness.

Now, test with the fastest star (5):

$100 + (255 - 100) * (5 - 1) / (5 - 1) = \dots\text{calculating}\dots = 255$, which is the brightest brightness. Great!!

Probably the other values go in between in a linear fashion (I promise you, they do! :))

This is a very general calculation, and for the specific settings we have (star speeds, brightness scales and such), it could be much optimized. Though, since it is only calculated **once** for each star, speed does not matter much. It is better to have a general algorithm like this and be able to change the speed settings without needing to change this calculation at all.

Now, if you run the game, the slower stars are darker than the faster stars, and we have a lot more depth feeling to the starfield. Oh, and speaking of depth, better set the **Depth** of the **objStarField** object to **10** or something, just so that it does not appear in front of any other object.

Great! Now we have learned about arrays and star fields and such things.


Time Lines

Finally! Time to take a look at those **Time Lines**. A time line is like a kind of manuscript that tells Game Maker when something should happen and what should happen. It looks pretty much like an object when you are working with it, but instead of defining **Events**, you are defining **Moments**. To a **Moment** a list of actions are added, in the same manner as with an **Event**.

Create a new **Time Line** and call it **timLevel2**. This **Time Line** will control the appearance of enemies on the second level of the space shooter game.

Add a **Moment** by clicking on the **Add** button in the **Time Line** window. This brings up a small window where you can enter the time at which the moment should occur. The default setting for time lines is that the time units are equal to the speed of the room (the steps), although it is possible to alter the "speed" of the time line. So, enter **60** here. This should set the moment to occur at about 2 seconds after the time line has started (default room speed is 30 steps per second).

Now it is time to add some actions for the moment. Let's add some instances of **objEnemy1** to the game. Drag the action **Create and instance of an object** to the action list in the new moment. Select **objEnemy1** to be created and set **x** to **90** and **y** to **-64**. An enemy will be created above the screen and at x coordinate 90. Now, add another **Create and instance of an object** action to add another **objEnemy1** instance at coordinates **x: room_width - 90** and **y: -64**.

Before the actions in the time line are executed, the time line must be assigned to an actual object. So, add a new object, **objEnemyController2**. In the **Create** event of that object, drag the action **Set a time line** () from the **main2** tab to the action list. In that action, select the timeline **timLevel2**. Let the **position** of the time line be **0**. Finally, add an instance of the **objEnemyController2** to the room **Level2**.

If you now try out the game, you will see that two seconds after the start of level 2, two enemy aircraft appear at the top of the screen. The problem is just that they were supposed to look as if they were flying in some kind of formation, but since they are given random speeds in their **Create** event, the formation is not upheld. Time to create a new kind of enemy. We will use the same sprite, just for simplicity.

Create a new object and call it **objEnemy2**. Make it have the object **objEnemy1** as parent, and use the sprite **sprEnemy1**. This makes it inherit all the properties of the first enemy object. The only thing we are actually going to change is its **Create** event. Add a create event to the new enemy object. The **Create** event of the already existing enemy just calls the script **Enemy1Init**. Look at that script. It gives the enemy a random speed, some energy, and sets its **Alarm0** event to a random value. We want the same thing to happen to the second enemy, except for the random speed thing. So, to the **Create** event of the object **objEnemy2**, add the action **Execute a piece of code** and enter the following:

```
vspeed = 5;
myEnergy = global.enemy1MaxEnergy;
alarm[0] = random(60);
```

Adding a **Create** event to the **objEnemy2** object will override its inherited **Create** event. Since everything else is inherited from the **objEnemy1** event, we do not need to define new collision events for the player or the shooting stuff. This is a very good thing. Now, change the moment at step 60 in **timLevel2** so that the instances that are created are of the object type **objEnemy2** instead of **objEnemy1**.

If you now run the game again, you will see that the enemies that appear in level two look as they are flying in some kind of formation.

Continue adding a few more **moments** to the time line **timLevel2**. Make two enemies appear in each of those moments, according to this table:

Time	Enemy coordinates
80	(150, -64) (room_width - 150, -64)
100	(200, -64) (room_width - 200, -64)
120	(250, -64) (room_width - 250, -64)
140	(300, -64) (room_width - 300, -64)

There you are! A nice little upside-down "V" formation that catches the player in a trap if he is not fast enough! :)

So, time lines are good for formations. We will add more to this time line later on. But first, let's take a look at...

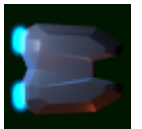
Paths

Don't stray from the path

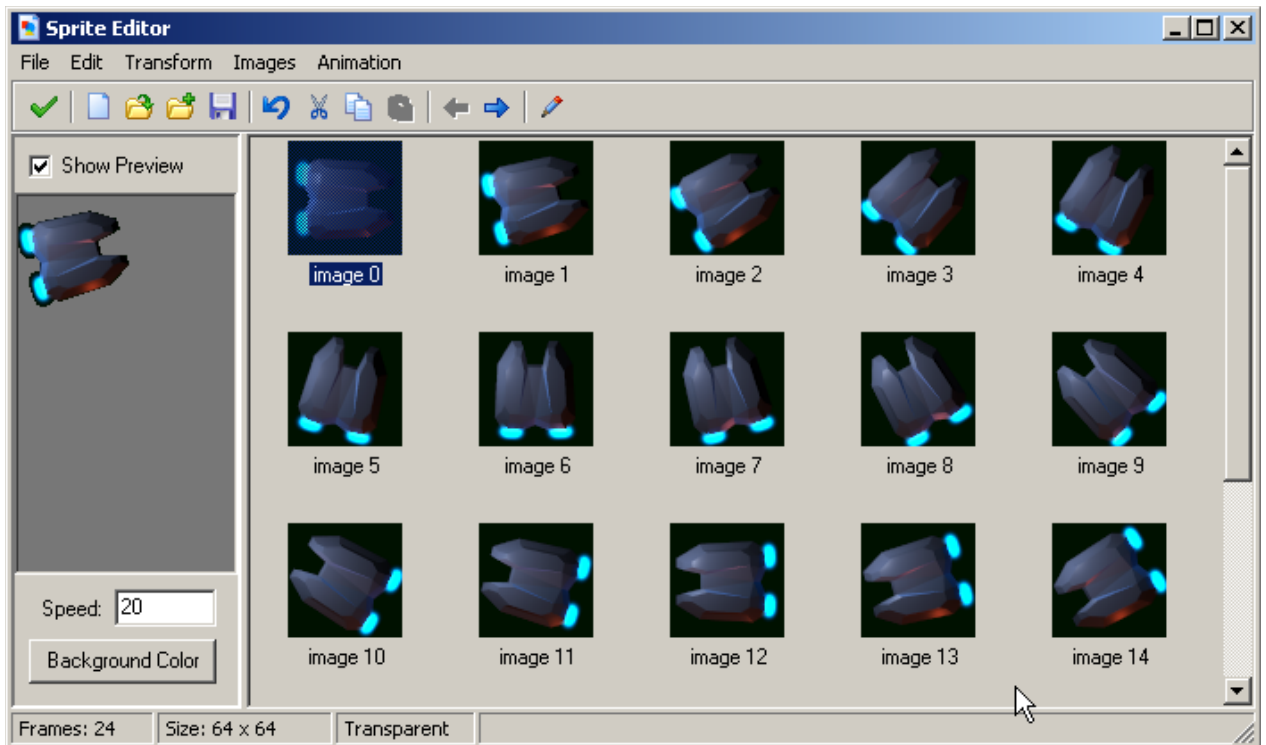
Paths are lines or "splines" that define the movement of an object. We will here use them for the movement of incoming enemy planes.

Say, I am getting tired of the MIG plane. Time to add something new.

I played around with Blender for about half an hour, and this is what I came up with. It is not anything revolutionary when it comes to 3D design, but I reckon it looks good enough for our game. The reason I did this animated rotating spaceship is because at the same time as I want to introduce you to paths, I intend to show you how to make objects that can rotate and point in the direction in which they are moving. (If you are really nice to me and it seems like there are enough people interested in it, I might try and put together a little document displaying how I made the ship in Blender (www.blender.org)).

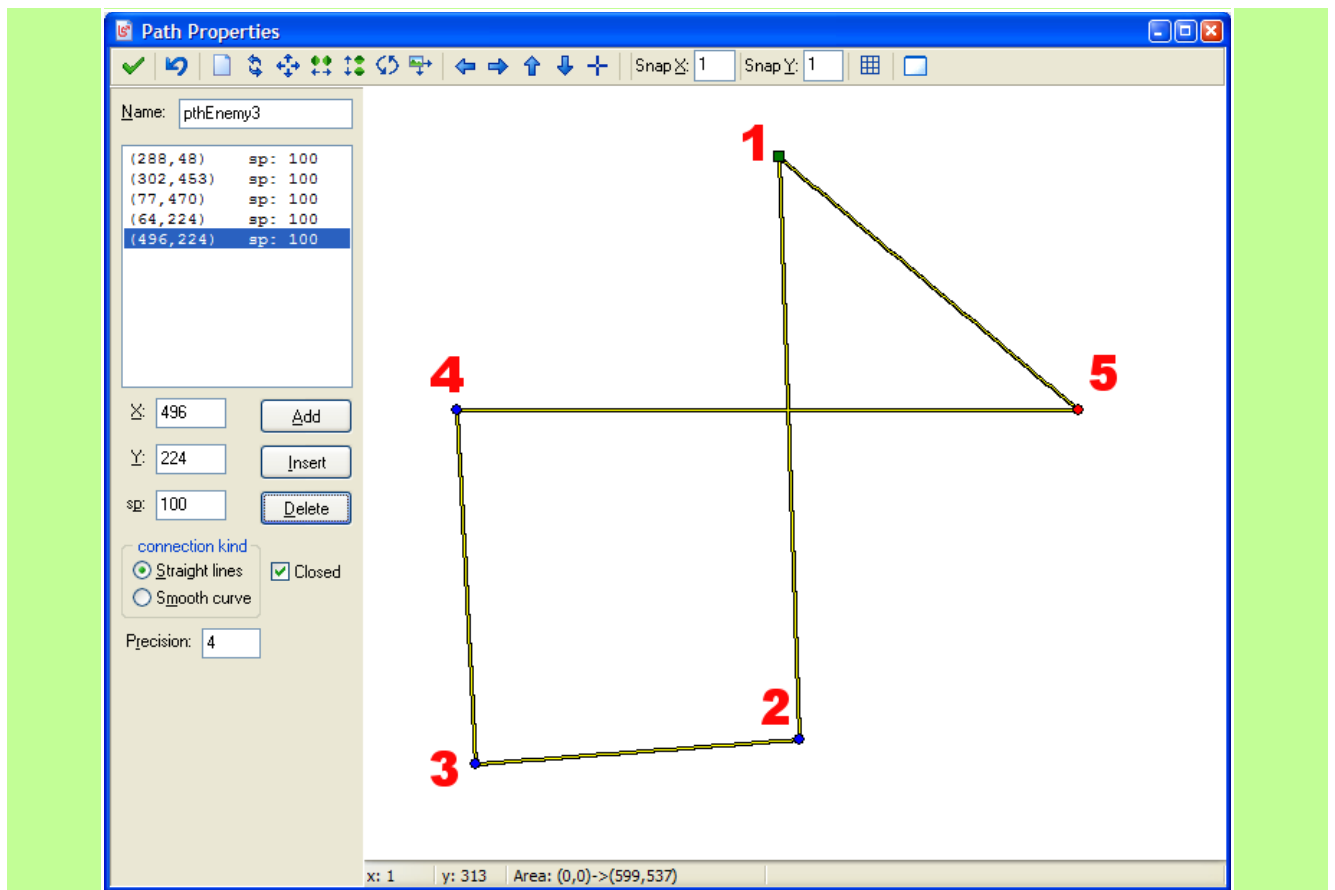


Create a new sprite: **sprEnemy3**. I know there is no enemy 2 sprite, but it would be very confusing if we called this sprite enemy 2 and used it for the object enemy 3, don't you think. In the sprite, load the first image from the Ship1 sequence (Ship10001.png) that is in the resources for this lecture. Then click the **Edit Sprite** button. You are now faced with the sprite editor of Game Maker. Here we will add the rest of the images that make up the rotating ship. Either click in the menu **File -> Add from file ...** or press **CTRL-A** on the keyboard (I prefer the keyboard). This brings up a file dialog where you are supposed to choose the second ship1 image (Ship10002.png). Once that is added, repeat the adding (CTRL-A) with all the rest of the images. (Actually, in the last version of GM, you can select multiple images for addition at once. Do whichever you please.) When you are done you should have 24 images in the sprite (0 to 23). Put a checkmark in the **Show Preview** checkbox to see the ship animate. Wow, what a beauty! :)



Add a new object for the enemy. Call it **objEnemy3**. Make it use the new enemy sprite.

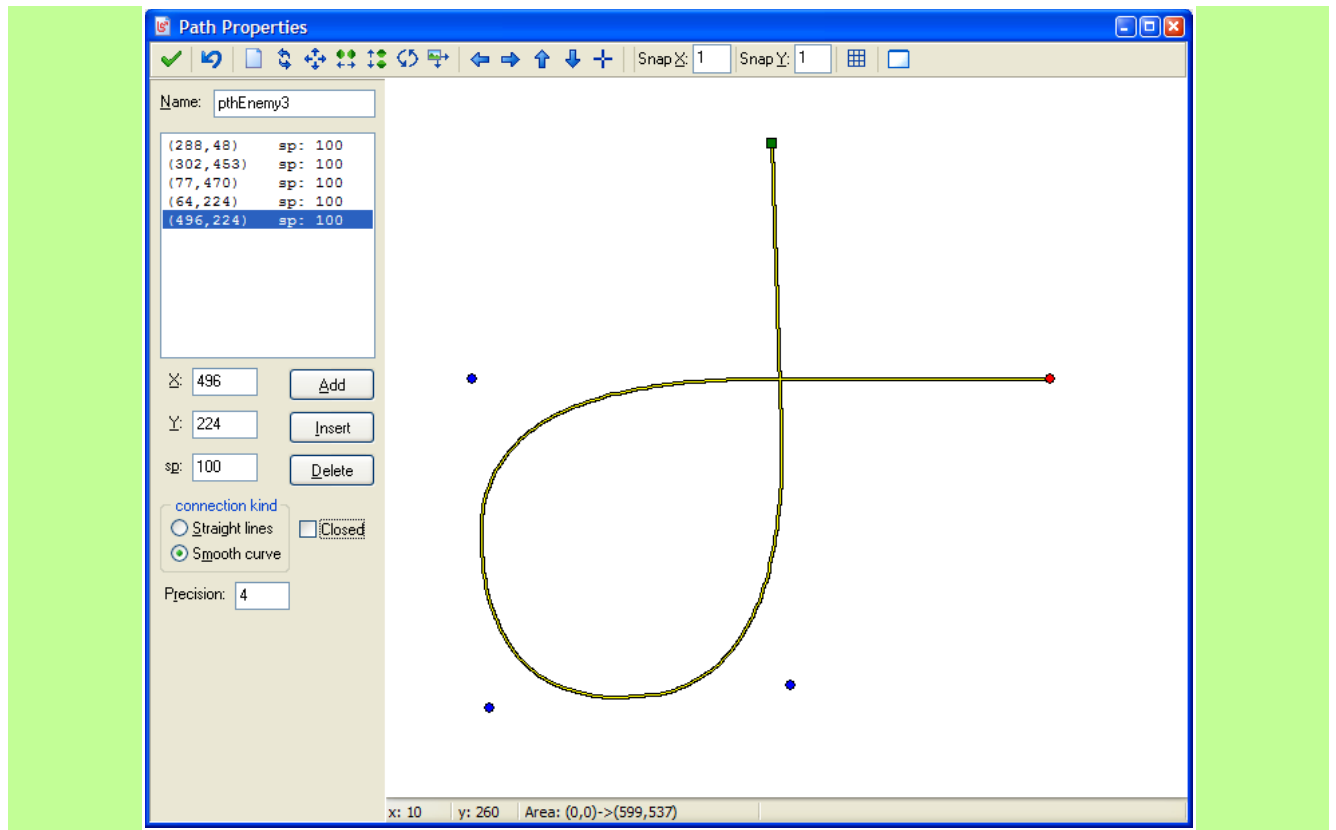
Finally, it is time to add a path. Call the path **pthEnemy3**. This will be the path that the new ship will follow when it is added to the level. Stretch out the path window so that it is almost as big as possible on the screen. I have also disabled the viewing of the grid, which can be done from the toolbar in the Path window, and set the grid size to 1 x 1. Now, click in the path area to add five path points according to the image below. Make sure that they are added in the order as indicated by the red numbers. It is not that important exactly where the points are, just as long as the general shape is somewhat as below.




If you are not satisfied with the position of the points, you can click-and-drag them. Another way is to select the individual points in the list to the left and manually enter the coordinates in the text boxes below the list.

The shape looks a bit strange, but the purpose is to have the enemy fly down, make a clockwise 270 degrees turn and fly out at the right edge of the screen. You can see that it is possible to set the **speed (sp)** for each point. This is however a speed that is **relative** to the speed that the instance has. So, if it says **100** here (which is the default), the instance will move at 100% of its set speed. Let all the points have their default speed (100).

The path above is a bit sharp though, don't you think? Click next to the words **Smooth curve** to make it smooth (bezier, I think). Much better. **Now, also uncheck the Closed checkbox.** This will open up the curve so that it has different start and ending points. It should now look like this:



Note that the **points** are still at the same locations. It is only the way the path follows these points that has changed. The path now looks like a smooth sweep across the screen. If you want to, you could alter the **precision** (which is **4** by default) and see what happens. Try setting it to 1, 2 or 3. The **Precision** determines the "smoothness" of the curve. Make sure it is **4** when you are done. Click **OK** to close the path window.

Now we also need to tell the new enemy object that it should follow this path. Open up the object **objEnemy3**. Add a **Create** event. To that even, add the action **Set a path for the instance** (). In the action, select the path **pthEnemy3**. Set the **speed** setting to **8**. **Let the other two settings be as default.**

Now, we have one more thing to do here. There is an event called **End of path**. It can be found under the **Other** events. Add it to the object. This event will trigger when the instance of this object reaches the end of its path. When this happens, we want the ship to simply continue in its current direction and fly out of the screen. So, first "disconnect" it from the path by adding a similar action as in the **Create** event (**Set a path for the instance**), but select **No Path** in the action. **Leave the speed at 0 and do not change the other settings either.** Then, add an action to set a variable and make it set the variable **speed** to **8**.

That is it. At the end of the path, the settings of the path will stop the instance, but the **End of path** event will thereafter disconnect the instance from the path and give it a speed again.

Open up the time line **timLevel2**. Add a new moment at **200** steps. Here, add the action to create a new instance of the **objEnemy3** object. Set the coordinates to **(400, -64)**.

Run the game and see what we have accomplished. Hmm. A spinning enemy. That is not how we want it to look. But it moves OK (I hope).

Rotational movement

So, we need to make sure that the correct image of the sprite is displayed, depending on in which direction the instance moves. Using the variable **image_index** we can decide which image should be displayed. The image that should be displayed is depending on in which direction the instance is moving, and that is stored in the **direction** variable. So, to make it work, we should "map" the **direction** onto the **image_index** in some way. We also need to stop the animation of the sprite, which is done by setting the variable **image_speed** to **0**.

This is sometimes called "Rotational movement" or "360-degrees movement". In GM version 6 and later, it is possible to rotate the sprite image at runtime simply by setting the **image_angle** variable, but I think it is good to know of this way too. At least if you want to really have **different** images for the sprite, such as in this case where the spaceship is lit up by light from a certain direction when rotated. That can not be achieved with the **image_angle** variable.

Start out with setting the variable **image_speed** to **0** in the **Create event**.

The way the sprite images are laid out, the first image shows the ship pointing straight to the right. This is direction 0. The second image shows the ship pointing slightly upwards. The direction variable ranges from 0 to 359 and the images from 0 to 23. That gives 360 different directions (well, you **could** have partial degrees too, but leave them out for a while) and 24 different images. $360 / 24$ gives 15, which means that each image covers 15 degrees.

Thus, we could set **image_index = direction / 15**. If we put this in the **Step** event, Game Maker will evaluate the calculation in each step of the game and select the correct image to be displayed.

There is, however, a more general way of doing this. Game Maker keeps track of the total number of images in a sprite with the variable **image_number**. For the sprite **sprEnemy3**, **image_number** equals **24**. Remember that the number **15** above comes from dividing the number of degrees with the number of images?

So, we can write **image_index = direction / (360 / image_number)**. Now, if you know some math, you can move the **image_number** around a bit and get:

image_index = direction * image_number / 360

And that is exactly what we will put in the **Step** event of the **objEnemy3** object.

```
image_index = direction * image_number / 360;
```

In this way it is possible to change the number of images in the sprite and the game still works without needing to change this setting.

There is one more thing we need to consider - removing the enemy once it is out of the room. Add an **Outside Room** event and enter this:

```
if (x > room_width + sprite_xoffset) instance_destroy();
```

This means that as soon as the enemy is out of the room, a check is made to see if it is out on the right side of the room (where it disappears) also considering its origin (offset). If it is outside to the right, the instance is destroyed. If we do not do this, there will be lots of enemy instances flying around outside the room and the game will eventually go slower and slower.

There. Now add a few more enemies in the time line, for example at the times 220, 240, 260, 280, 300, 320 and 340. Use **objEnemy3** for them all and also give all the same starting coordinates (**400, -64**). (Note that it is possible to copy-and-paste an action so that you do not have to fill in the settings for the **Create an instance of an object** action all the time.)

Run the game. If all is correct, eight enemies of the new type should now appear, following each other along the swirling path.

A little intro

I was going to end here, but I thought I would display just one more little GM trick. Game Maker is able to display common movie files. I think it supports AVI and MPG formats. So, I made a little intro sequence (in Blender) making a little attempt to copy the famous Star Wars scroller (or "crawler" as it is also called). It is encoded with DivX, so you need that codec in order to view it (Download for free from www.divx.com).

So, get the zip file with the animation from the resources to this lecture. Or, if you do not feel like downloading a file that big (about 3 MB) use any other animation, you can even make your own if you please.

Unzip the file and get the AVI file. Put the AVI file in the same directory as the game file you have been creating for this lecture (the gmd file). Now, make a new room and call it **IntroRoom**. Make a new object, called **objIntroMovie**. Add that object to the new room. Move the room in the **Resource Explorer** so that it becomes the first room in the list.

In the **Create** event of the **objIntroMovie**, enter this code:

```
show_video("ScrollerAnim0001_0600.avi", true, false);  
room_goto_next();
```

The arguments in the **show_video** function are: file name, fullscreen (true/false), loop video (true/false). So, here the file ScrollerAnim0001_0600.avi is displayed in a fullscreen window, and it does not loop.

Just after the animation, the first level room is entered.

Sit back and enjoy! :)

Conclusion

Now you have learned about time lines and paths, and we even got a bit of rotational movement and arrays in here. Time to add some things to the game as assignments. :)

Good luck!

Carl

Assignments

Assignment 7 - Add features to Total Holocaust

Due date: Thursday, 1 September 2005, 08:00 AM

Maximum grade: 100

In this assignment you will add more things to Total Holocaust. Here is the list:

- There is a little bug in the level transfer that disables the shooting for the rest of the game if the player is shooting while the level transfer begins. Try to fix that bug.
- Make the new enemy (objEnemy3) shoot. It should shoot in about the same way as the first enemy, except that it should shoot straight ahead, in the direction it is currently facing.
- Make the new enemy (objEnemy3) affect the player in a similar way as the first enemy when it comes to collisions. The player could crash into it, the player could shoot it, etc. This includes giving the enemy some kind of "energy" and explosion too.
- Create an end level "boss" for the second level. After some time (in the time line) the boss is added. The boss should follow a simple path in a kind of horizontal movement along the top of the screen. The path could also lead it down a bit at some points.

The boss should simply be a little larger sprite than the other enemies, and it should take a little more hits than the others. Also, it should not disappear until shot down. Once shot down, it should trigger the transfer to the next level (so you have to add another room too, but you do not have to fill it with anything).

Good luck!

Carl