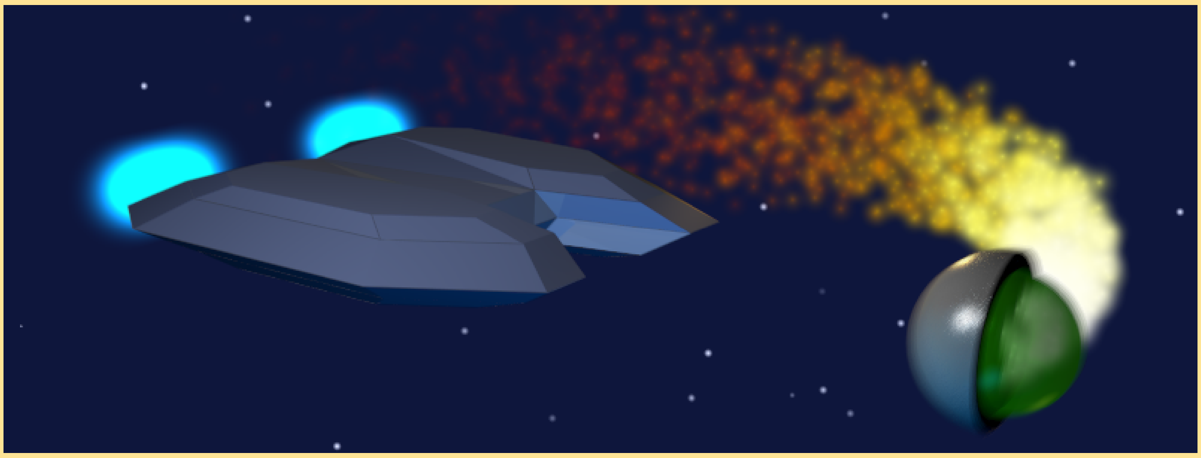
Lecture 10 - The Mythic Last Lecture

Written by Carl Gustafsson



Goal of the lecture

This lecture is targeted at answering all the questions and suggestions that I have received from the students. Or, at least, most of them.

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 is the final lecture in the Game Maker course. It will cover the subjects of some of the questions that have been made by the students. The following subjects will be covered:

- ~~Exclusive graphics mode~~
- Trigonometric functions (sine and cosine)
- Drawing with Game Maker's draw_..... functions
- Particles

Please note that in order to complete this last lecture, you will have to have a registered version of Game Maker.

~~Exclusive graphics mode~~

NOTE for Round 3 of the course: In Game Maker 6.x, the Exclusive Graphics Mode is no longer applicable. This entire chapter has therefore been skipped.

~~Introduction~~

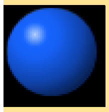
~~**NOTE: It appears that Exclusive Graphics Mode is no longer available in the next version of Game Maker (6.0). I am not sure why, but that is how it is.**~~

~~Well, what can be said about exclusive graphics mode? Here is an excerpt from the manual:~~

Exclusive graphics mode
In exclusive mode, the game has the full control over the screen. No other applications can use it anymore. It makes the graphics often a bit faster and allows for some special effects (like gamma settings). If you want to make sure the computer of the player is and stays in the right screen resolution, you best use exclusive mode. A warning is in place though. In exclusive mode no other windows can show. This e.g. means that you cannot use actions that display a message, ask a question, show the highscore list, or show the game information. Also no errors can be reported. In general, when something goes wrong in exclusive mode the game ends, and sometimes this does not help and the player
has no other option than to restart the computer. So make sure your game works absolutely correct. You cannot run your game in debug mode when using exclusive mode.

~~So, basically, Exclusive Graphics Mode means that your game will have all the control over the computer graphics. This usually means that things go faster, because the operating system does not need to be concerned with other windows and applications. The downside with this is, as the manual states, that you can not use any features that need to display other windows (pop-up windows and similar).~~

~~An example~~



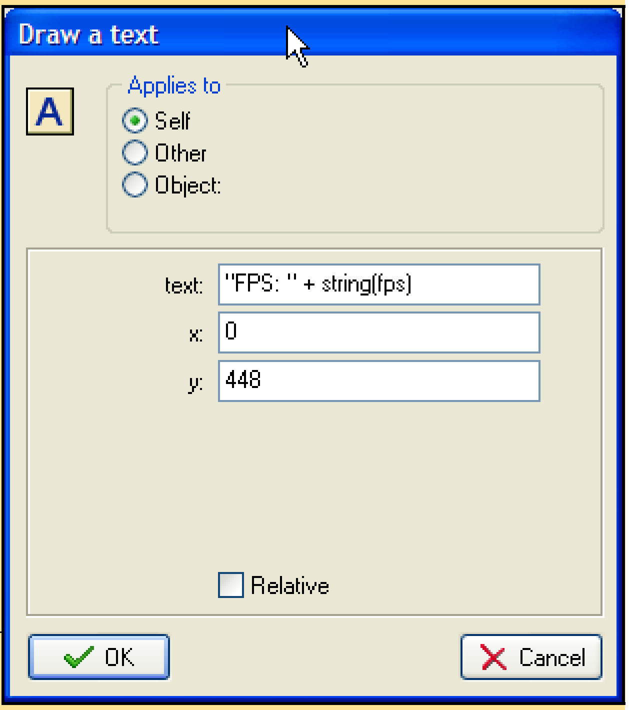
~~In order to make a little example and speed test of exclusive graphics mode, we do like this: Create a new game and add the sprite **sprBlueBall** using the image **BlueBall.png** from the resources to this lecture. Then add two objects: **objController** and **objBlueBall**. Make the object **objBlueBall** use the sprite **sprBlueBall**.~~

~~In **objController**, add the **Keyboard / <SPACE>** event. To that event, add the **Repeat next action** action and then a **Create and instance of an object** action. The **Repeat next action** should repeat **10** times. The instance that should be created should be **objBlueBall** and it should be created at the following coordinates:~~
~~x: random(room_width - 48)~~
~~y: random(room_height - 88)~~

~~This should, when space is pressed, add 10 instances of **objBlueBall** to the room in random locations. The **-48** and **-88** are there to make sure that the entire blue ball sprite is inside the room when created and that some place is left at the bottom of the room for some text that we will write out later.~~

~~Add the **Draw** event to the **objController** and, in that event, add a **Set a font for drawing text** action. Choose a font for the text. I chose **Verdana, 10** points and **black**.~~

~~After the font setting, add a **Draw a text** action and draw the text **"FPS: " + string(fps)**. This will draw the text "FPS: " followed by the current fps count (frames per second). The default room setting is 30 frames per second, and we will leave it at that. The position of the text should be: x = **0** and y = **448**.~~



[✓ Self] Other Object:

text: "FPS: " + string(fps)
x: 0
y: 448
Relative

~~Add a new **Draw a text** action and draw the text **"instances:" + string(instance_number(objBlueBall))** . This draws the text "instances: " followed by the current number of instances of the object **objBlueBall** that exist in the room. Position: x = **0**, y = **460**.~~

~~Time to look closer at the object **objBlueBall**. Add the **Create** event and, there, add a **Set direction and speed of motion** action. The direction should be **random(360)** and the speed should be **random(15)**.~~

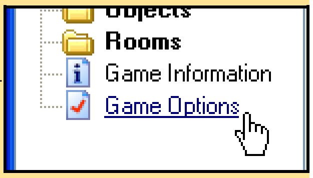
~~Add the **Step** event to the **objBlueBall** object and, in that event, add an **Execute a piece of code** action. The following code should be executed:~~

```
if (x + sprite_width > room_width or x < 0)
hspeed = -hspeed;
if (y + sprite_height > room_height - 30 or y
< 0) vspeed = -vspeed;
```

~~This will keep the blue ball inside the room. As soon as it reaches the border, it will bounce against it. Create a new room and add the **objController** to it. This is all that you need. Save the game.~~

~~Benchmarking~~

~~Now, start the game. It will only display the empty room, and the fps and instances text. Press **SPACE**. This will add some blue balls to the room. You can see the instance count increase as more blue balls are created. Try holding down **SPACE** until there are so many balls bouncing around that the frame count drops to **20 Frames per Second**. As you can see, when the frame count starts dropping, it first decreases a bit when you hold down **SPACE**, and then it goes up again. That is probably because the creation of instances takes a bit of extra processor time with memory allocation and all (at least that is what I guess). On my computer I got up to somewhere between 3900 - 4000 instances before the frame count drops below 20 fps. Make a note of your own instance count at 20 frames per second. (Well, don't misunderstand me here. You do not have to write down the number 20 times per second... :) Just once is good enough.)~~



~~Time to go to exclusive graphics mode. Open the **Game Options** window (by double-clicking on it in the **Resources tree**). Select the **Resolution** tab in the window that appears. Put a checkmark in the checkbox **Set the resolution of the screen**. This brings up a lot of different parameters for the resolution and color depth of the game. It is also possible to set the frequency of the screen (which I have not found any use for yet...).~~

~~A bit further down in this window is a checkbox called **Use exclusive graphics mode**. Put a checkmark in that box too. This will give you a little warning that popup windows such as question windows and highscore windows can not be used in this mode. Select **640 x 480** as resolution. This is the default setting of a room in Game Maker, and, unless you have altered it, this should be the size of the only room in our "game".~~

~~Save the game and run it again. This time the screen should flicker for a split-second, and then the game should start in full-screen, exclusive graphics mode. Now, do the same thing as above, i.e. press space to increase the instance count until the frame rate drops below 20 fps. On my computer I can get up to about 5400 instances before the frame rate drops below 20 fps. So, this would, in my case, mean a speed increase of **35 %**. Might be useful if your game runs slow, but then, you can not use popup windows. It is up to you to decide which you need. Personally I prefer games that fill the entire screen instead of just a window, although that can also be implemented without going to exclusive mode. I think that fullscreen games present a deeper immersion.~~

~~It is possible to set the resolution and exclusive mode at run-time, i.e. when the game is running, as opposed to at design-time that we used above. Setting the graphics mode is done with the function **set_graphics_mode**. Check it up in the manual if you are interested in how it works.~~

~~Another thing about exclusive mode is that debug mode does not work. That is because the pop-up debug window can not be viewed in exclusive mode. So one option is to develop the game using a normal window and then switching to exclusive mode once there are no more bugs in the game (yeah, right... no more bugs...hmm...).~~

Drawing functions and trigonometry

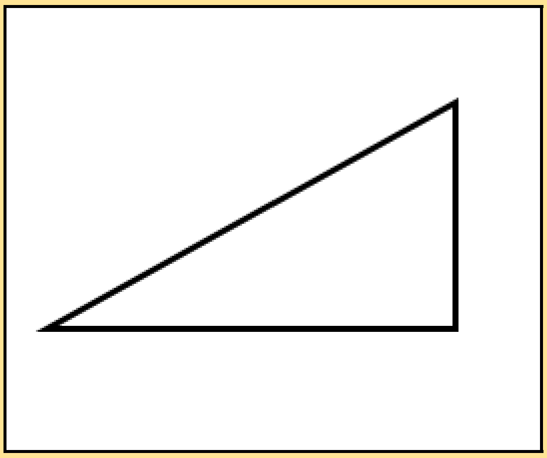
Game Maker is designed mainly for the use of sprites in games. Sprites are two-dimensional images, bitmaps, that are fast to draw on the screen. There is however a possibility to use other kinds of graphics, for example lines, arcs and text. These graphic types are a bit slower than the sprite functions and may therefore slow down the flow of a game if used excessively.

Another concept that is very good to know of when making games is math. There are many different areas of math, and I will here try to cover one widely used area in game design: trigonometry. Trigonometry involves calculation with angles (not to be confused with **angels**, whose help is required in another math area: transformation theory. ;)) and the use of sine, cosine and other related functions. Since there are many resources on the Internet for learning trigonometry, I will only cover the theory behind it briefly here. Trigonometry is very handy when it comes to different kinds of circular features in a game.

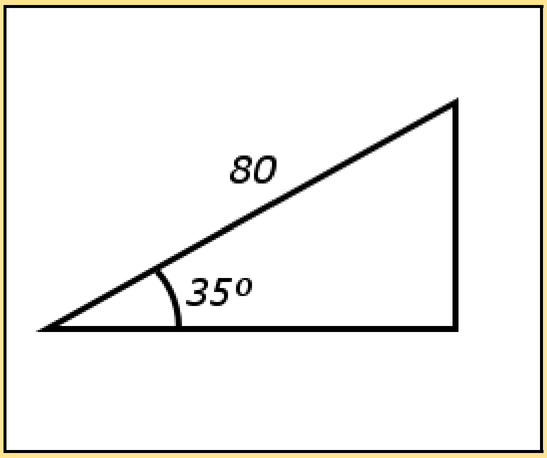
Trigonometry

So, how can you learn something that you can't even pronounce? Well, we will give it a try, even though I am not completely certain on some English expressions. I am not sure how to start this, but here we go.

Suppose you have a right-angled triangle, that is, a triangle in which one of the angles is 90°. Just like this one:



We then measure the lower left angle and find it to be 35° (just guessing). Then we measure the longest side in the triangle (the upper left, slanted side). This is called the hypotenuse. Suppose it is 80 units long. Since in Game Maker most measurements are in pixels, we could call the length units "pixels" here too, just to have a name on them.



It is then possible to calculate the length of the other two sides of the triangle through using the trigonometric functions **sine** and **cosine**. They work something like this:

*length of opposite side = length of hypotenuse * sine of the angle*

*length of nearest side = length of hypotenuse * cosine of the angle*

There is a little additional trick here though. In Game Maker (and lots of other mathematical programs) the angle that should be put into the **sin** and **cos** functions (as they are abbreviated) should be measured in **radians**. **Radians** is a little different way of measuring the angle. It is proportional to common **degrees** in this way:

*1 radian = 180 / pi * degrees*

*1 degree = pi / 180 * radians*

This is because in one, complete, revolution, there are 360 degrees and 2 * pi radians. Here we have another concept that might be new to some of you: **pi**. **Pi** is just a number, but an **irrational** one. This means that it can **not** be properly written using either decimals or fractions. One can however write an estimate of it, which starts with **3.1415**. The number comes from the proportion between the diameter of a circle compared to its circumference. The circumference of a circle always equals the diameter times **pi**. Fortunately, Game Maker comes with two useful functions for converting angles from degrees to radians and vice versa: **degtorad** (**deg**rees-**to**-**rad**ians) and **radtodeg** (**rad**ians-**to**-**deg**rees).

So, back to the trigonometry. Suppose we want to know the length of the "opposite" side. That is, the vertical line that is opposite to the angle that we have "measured". This is calculated using the formula above:

80 * sin(degtorad(35));

The immediate conversion from degrees to radians using the **degtorad** function above might look strange, but that is how it works. We could also have written like this, if we pleased:

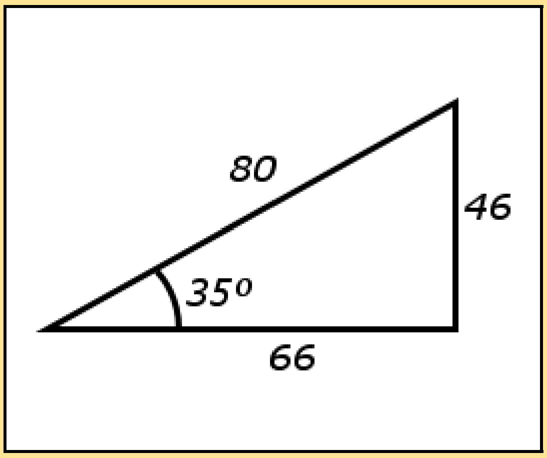
80 * sin(35 * pi / 180);

That is about the same thing, but I suspect that the first function works faster and is more obvious. Likewise, the length of the nearest side (the horizontal line) can be calculated using the **cos** function:

80 * cos(degtorad(35));

Worth noting is that the result of a **sin** and **cos** function can **never** be higher than **1** or lower than **-1**. This is because none of the two sides can **ever** be longer than the **hypotenuse** (the diagonal).

So, what is the result then?
The length of the vertical line is 80 * sin(degtorad(35)), which is about 46 pixels (rounded).
The length of the horizontal line is 80 * cos(degtorad(35)), which is about 66 pixels (rounded).



If we now know about **Pythagoras**, we can make a test to determine if these values are correct. According to Pythagoras, the sum of the squared length of the two sides equals the square of the hypotenuse Let's try that:

46 * 46 + 66 * 66 = 6472
80 * 80 = 6400

Well, well. Close enough for jazz. (a quote from a friend of mine, tuning his guitar) ;)
The difference from the check comes from the rounding errors made earlier. If we had used all the decimals places that actually are there, the check result would be much better.

Now, what on earth can all this be used for? The thing is that sometimes when you want to determine the location of an object (in a game, for example), you do not know the location by x and y coordinates, but instead by angle and distance from another, "reference" object. This is where trigonometric functions come in real handy; for converting from **polar** coordinates (angle and distance) into **rectangular** coordinates (x and y location). Using the method described above, you can for example solve the following example:

You have a spaceship at location (132, 421). Then you want to put a little pod at an angle, 150 degrees and distance 120 pixels from the spaceship. The coordinates of the pod would then be calculated like this:

x: 132 + 120 * cos(degtorad(150))
y: 421 - 120 * sin(degtorad(150))

Hmm, what are those values in the calculation? Firstly: 132 is the x coordinate of the spaceship. 120 is the distance to the pod, and 150 is the angle from the spaceship to the pod. Not too hard. Calculating the y coordinate is made in a similar way, **but**, when calculating y coordinates, the **sin** function result should be **subtracted** instead of **added** to the original coordinate. That is because the y axis in Game Maker (and most computer contexts) is pointing **downwards** instead of **upwards**, which is the normal mathematical direction.

However, these trigonometric functions have now been much simplified in Game Maker through the recent addition of the **lengthdir_x** and **lengthdir_y** functions to the function list. As arguments (or "parameters") they take the length and the direction that should be transformed into rectangular coordinates (x and y). In the above spaceship-pod problem, the x and y coordinates can now be calculated like this:

x: 132 + lengthdir_x(120, 150)
y: 421 + lengthdir_y(120, 150)

So, there is no longer any need for taking the negative y axis into consideration or converting from degrees to radians. All that is neatly fixed by the new functions. It might however be good to know about the trigonometric background.

Practicing trigonometry: rotating a pod

We could build up a little example with the trigonometry we learned above. Let us **make** that spaceship with a pod rotating around it.

Create a new game in Game Maker. Add the spaceship and pod sprites found in the resources for this lecture. "Center" the sprites by setting the sprite origin of the spaceship to **(32, 32)** and the origin of the pod sprite to **(15, 15)**. Then make an object for each sprite (**objSpaceship** and **objPod**). In the spaceship object, make the object follow the mouse position by entering this into the **End Step** event:

```
x = mouse_x
y = mouse_y;
```

This makes sure that the spaceship is always at the mouse location in the game. Just for testing out. Create a new room, add the spaceship object to it and make the background color of the room black. You could now save and try the "game". Move the mouse across the game window. The spaceship should follow it tightly. In order to make it look better, we could remove the mouse cursor from the game by going into the **Global Game Settings** and unchecking **Display the cursor**. It now looks better, without a white arrow (or whatever mouse cursor you use) in front of the spaceship.

Now, add the **Create** event to the spaceship object. In the **create** event we want to create an instance of the pod object as well as setting some variables that we use for the pod movement:

```
// Create a pod instance that will orbit the spaceship.
myPod = instance_create(x, y, objPod);

// Define the distance and initial direction to the pod.
podDistance = 80;
podDirection = 0;

// Define the rotational speed of the pod (degrees per step).
podRotSpeed = 5;
```

That will firstly create the pod and secondly set a few variables. **podDistance** determines the distance from the center of the spaceship to the center of the pod. **podDirection** is a variable that keeps the current direction from the spaceship to the pod. This variable will change all the time to rotate the pod around the ship. The **podRotSpeed** sets the rotational speed of the pod. In our example the speed is set to **5** degrees per step, which means that the pod will orbit once every 360 / 5 = 72 steps.

Now, go back to the **End Step** event. Time to add the rotational functionality. Here is the code, the explanation will come below:

```
// Move the spaceship with the mouse.
x = mouse_x;
y = mouse_y;

// Increase the pod rotation direction.
podDirection += podRotSpeed;

// Place the pod according to distance and direction.
myPod.x = x + podDistance * cos( degtorad( podDirection ) );
myPod.y = y - podDistance * sin( degtorad( podDirection ) );
```

First we have the mouse following code as before. Then we increase the **podDirection** variable. This is because we want the direction to the pod to increase a bit each step in order to make it rotate around the spaceship.

The last two lines of this script set the x and the y coordinates of the pod using the trigonometric functions **cos** and **sin**. Let us take apart each line from inside and out, which is the way the calculation is actually made.

First, the **podDirection** variable is converted from **degrees** to **radians** with the **degtorad** function. That is what the **cos** and **sin** functions want. Then the **cos** function is run with the **podDirection** in radians. The result is the **cosine** value of the direction, which in this application is the **x** component factor. The **cosine** value is multiplied with the **podDistance** variable in order to get the correct distance to the location. Now the location is correct in relation to the spaceship's location. Finally, the **x** value of the spaceship instance is added in order to make the pod **x** value correct in the room.

The **sine** function line works about the same. The difference is that the **sine** function gives the **y** component, or, rather, the **inverted y** component. That is why it is **subtracted** from the spaceship's **y** coordinate instead of **added**.
Save and try the game. If everything is done correctly, the pod should now be orbiting the spaceship, no matter how the ship is moved around in the room.

In order to try out the **lengthdir** functions mentioned above, we will now alter the last two lines of the script. Or, rather, we will comment them out and add two new lines. Here is the result:

```
// Move the spaceship with the mouse.
x = mouse_x;
y = mouse_y;

// Increase the pod rotation direction.
podDirection += podRotSpeed;

// Place the pod according to distance and direction.
// myPod.x = x + podDistance * cos( degtorad( podDirection ) );
// myPod.y = y - podDistance * sin( degtorad( podDirection ) );
myPod.x = x + lengthdir_x(podDistance, podDirection);
myPod.y = y + lengthdir_y(podDistance, podDirection);
```

Run the game again. The pod should now orbit in the exactly same way. Perhaps using the **lengthdir** functions is easier than using the **sine** and **cosine** functions. Note also that the **lengthdir** functions handle the inverted **y** component correctly, so the developer does not need to care about that. The **lengthdir** functions effectively carry out the same work as the **sine** and **cosine** functions, but without much of the fuss.

Drawing functions

Apart from the sprites in Game Maker one can also use Game Maker's built-in drawing functions for drawing various geometrical shapes, e.g. lines, rectangles and ellipses. It is also possible to draw single pixels.

We will here take a little look at how to draw some lines that move around a bit on the screen. The result of this little exercise will be a rotating square in the game. We will continue to work on the Game Maker file that we made for the spaceship and pod.

Firstly, add a new object, called "objSquare". This will be used for drawing the lines. It should not have any sprite, since it should draw lines instead. Add it to the middle of the room somewhere about 1/3 of the room's height up from the bottom of the room. Since the object does not have any sprite associated with it, it will look like a small blue circle with a question mark in the room editor.

In the **CREATE** event we define some variables that will be used when we later draw the square:

```
// The "radius" of the square (distance from middle to a corner).
radius = 20;
```

Thus we only define what I call the "radius" of the square. That is the distance from the center of the square to each corner. What we are going to do next is calculate the four corners of the square and then draw it on the screen. That is done in the **DRAW** event:

```
// Calculate the four corners of the square.
// Upper left:
x1 = x + radius * cos(degtorad(135));
y1 = y - radius * sin(degtorad(135));
// Upper right:
x2 = x + radius * cos(degtorad(45));
y2 = y - radius * sin(degtorad(45));
// Lower right:
x3 = x + radius * cos(degtorad(315));
y3 = y - radius * sin(degtorad(315));
// Lower left:
x4 = x + radius * cos(degtorad(225));
y4 = y - radius * sin(degtorad(225));

// Set the drawing color
draw_set_color(c_green);

// Draw the lines
draw_line(x1, y1, x2, y2);
draw_line(x2, y2, x3, y3);
draw_line(x3, y3, x4, y4);
draw_line(x4, y4, x1, y1);
```

The calculation should not be too hard to follow if you have understood the use of sine and cosine above. After the corner calculation, we set the drawing color in Game Maker to a green color. Finally the lines are drawn using the **draw_line** command to draw the four lines.

Save the game and try it out. You should now see a green square sitting in the middle of the room. Time to make it do something.

Add some more variables to the **CREATE** event (additions shown as **bold** below) :

```
// The "radius" of the square (distance from middle to a corner).
radius = 20;

// The rotation speed. (degrees per step)
rotSpeed = -4;
// The current angle.
angle = 0;
```

The **rotSpeed** variable determines the speed of rotation of the square. Yes, we will make it rotate using those trigonometric functions. A negative rotation speed means that the rotation will be **clockwise**. The **angle** is used to store the current angle of the square.

What we now should do is to **change** the calculation of the four corners of the square in the **DRAW** event . We alter it so that when calculating each corner, the **angle** variable is added to the argument in the **cos** and **sin** functions. Like this:

```
// Calculate the four corners of the square.
// Upper left:
x1 = x + radius * cos(degtorad(135 + angle));
y1 = y - radius * sin(degtorad(135 + angle));
// Upper right:
x2 = x + radius * cos(degtorad(45 + angle));
y2 = y - radius * sin(degtorad(45 + angle));
// Lower right:
x3 = x + radius * cos(degtorad(315 + angle));
y3 = y - radius * sin(degtorad(315 + angle));
// Lower left:
x4 = x + radius * cos(degtorad(225 + angle));
y4 = y - radius * sin(degtorad(225 + angle));
```

Finally, we alter the **angle** variable itself, by adding the **rotSpeed** to it. We then check it so that it does not increase above 360. If it does, we subtract 360 from it, making it start over from the beginning. In a similar way, we add 360 to it if it is less than 0. Add this to the **DRAW** event:

```
// Rotate the square
angle += rotSpeed;
if angle >= 360 angle -= 360;

if angle < 0 angle += 360;
```

Run the game again. The square should now be rotating. Now we have used the trigonometric functions **and** the drawing functions in the same feature. But we will not stop there. Let us get the square moving too. That is done pretty simple, since the square is now drawn **relative** to the instance of the object that draws it. That is, the **x** and **y** coordinates of the instance are added to the square corner coordinates when they are calculated.

Add this to the **CREATE** event to add some gravity and horizontal speed to the square:

```
// The gravity and initial horizontal speed.
gravity = 0.3;
gravity_direction = 270; // Downwards.
hspeed = 5;
```

Then add this to the **DRAW** event to keep the square inside the room:

```
// Keep the square inside the room.
if x >= room_width - radius or x <= radius then hspeed = - hspeed;
if y >= room_height - radius then vspeed = - vspeed;
```

This will make the square bounce against the left and right borders of the room, or, actually, against a border that is a bit ("radius") away from the actual room borders. The square will also bounce in the same way against the bottom of the room.

To top it all off, we make sure that the square rotates in the "correct" direction with regard to its horizontal speed. Add this to the end of the **DRAW** event:

```
// Rotate the square according to horizontal speed:
rotSpeed = -hspeed;
```

Well, that will be enough of the square. Let us leave it alone for now. If you let it "run" for a while, it will slowly move nearer to the floor and eventually just "float" along the floor. That is probably because when bouncing on the floor, the square loses a little bit of speed. It could be fixed by changing the **y** coordinate check line and moving the square a bit up from the floor after the bounce, exactly the same amount as the vertical speed. But let us skip that.

Particles

The latest versions of Game Maker have had a new, interesting addition: Particle systems. The definition of a particle is perhaps best explained through an excerpt from the Game Maker manual:

*Particle systems are meant to create special effects. Particles are small elements (either
represented by a little sprite, a pixel or a little shape). Such particles move around according to predefined rules and can change color while they move. Many such particles
together can create e.g. fireworks, flames, rain, snow, star fields, flying debris, etc.*

So, particles are (usually small) entities that move and behave together in a group, according to specified rules regarding speed, direction, life time, etc. Particles are characterized by defining a **Particle Type**. The **Particle Type** describes the look, the movement and the lifetime of a group of particles. All particles are handled by a **Particle System**. The particles are kind of "contained" within such a system. It is possible to manually emit particles, but usually one would want to use a **Particle Emitter** for this. The **Particle Emitter** defines an area in which new particles are randomly created, and the amount of particles per game step.

Since this original lecture was written, "Simple particles" have been added to Game Maker. This makes it easier to make particles without writing any script. But we will look at the **script way** of doing it here.

A comet's tail

We will start as simple as possible. The plan is that the rotating pod that we created earlier should emit a stream of particles to give it a "tail" as it orbits the spaceship. We start by defining a **Particle Type**, a **Particle System** and a **Particle Emitter** in the **CREATE** event of the pod:

```
// Create a particle system.
myParticleSystem = part_system_create();


// Then, define a particle type.
myParticleType = part_type_create();
// Set the lifetime of the particles to between 30 and 50 steps.
part_type_life(myParticleType, 30, 50);

// Create a particle emitter for our particle system.
myParticleEmitter = part_emitter_create(myParticleSystem);
// Set the particle emitting region to an ellipse (a circle, actually) with Gaussian distribution
part_emitter_region(myParticleSystem, myParticleEmitter, x - 10, x + 10, y - 10, y + 10,
         ps_shape_ellipse, ps_distr_gaussian);
// Set the particle emitting stream to 5 particles per step.
part_emitter_stream(myParticleSystem, myParticleEmitter, myParticleType, 5);
```

There! First the particle system is created. That is very easy. When the function **part_system_create** is called, it creates a particle system and returns an **index** to that system. The index is just a number that can be used to refer to the particle system. We store that number in the variable **myParticleSystem**.

Next, a new particle type is created. This is done in the same way as the particle system. The particle type index is stored in **myParticleType**. The particle type is then further modified to set the lifetime of its particles to between 30 and 50 steps. Here we see how the particle type index is used to tell the function **part_type_life** which particle type we want to modify.

Lastly, a particle emitter is created. When creating an emitter, we have to tell Game Maker to which particle system it should belong when calling the **part_emitter_create** function. Again, an index is returned, that can later be used to refer to the newly created emitter. The emission region is defined to be in a 10 pixel radius around the coordinates of the current object (the pod). The shape is set to an ellipse (a circle in this case), and the distribution to Gaussian. This means that there will be more particles emitted from the **center** of the emitter than from the **perimeter**. The other distribution type is "linear", which distributes the particles evenly across the defined region.

Finally, the particle emitter is set to emit particles of the type we just created, and with a frequency of 5 particles per step.

We can now leave the particle creation for a little while. It will update itself every step.

Yes, that is enough in order to update the particle system. The system will then take care of creating, destroying and moving particles according to its definitions. It also draws itself automatically each step.

Now, at last, you can test run the game. If everything work as planned, you should now see the particle system in action. Well, it might not look that impressive right now. There should be some white pixels being created and destroyed at the location where you placed the pod in the room. It does not even follow the pod around. Pah! :)

Back to the code. We should try to make it follow the pod around as it moves. That can be done through changing the location of the emitter for each step, so that the particles are always emitted from the position where the pod is. Add a line to the **STEP** event of the pod so that it now looks like this:

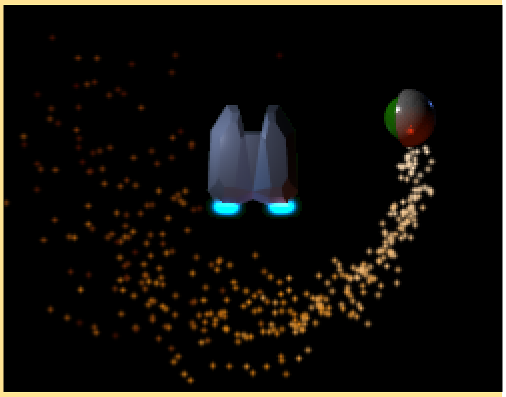
```
part_emitter_region(myParticleSystem, myParticleEmitter, x - 10, x + 10, y - 10, y +
          10, ps_shape_ellipse, ps_distr_gaussian);
```

This means that the region of the emitter is changed in each step to be exactly where the pod is. Run the game again. Yeah, that looks MUCH better. The pod now has a comet-like tail. There is a problem when the spaceship and pod are moved too fast. Then the particles will be created in distinct "puffs". But that will have to do. The only other way is to increase the room speed in order to make everything update more often, but let us not do anything with that.

What we should do is to alter the color of the particles. The way it is now they look just too boring. How about if they start out bright yellow, blend into orange and then fade to dark red? That would look much better. In order to change the color of the particle type, the function **part_type_color** is used. The parameters to that function determine the color of a particle at "birth", "mid-life" and at "death". Add the following code to the **CREATE** event of the pod object, just after the particle type has been created.

```
// Set the colors of the particle type.
myBrightYellow = make_color_rgb(255, 230, 200);
myOrange = make_color_rgb(220, 130, 20);
myDarkRed = make_color_rgb(50, 0, 0);
part_type_color3(myParticleType, myBrightYellow, myOrange, myDarkRed);
```

Run the game again. A much better comet's tail is displayed.



Now, we would like the particles to move about just a little bit too, in order to make it more interesting. I want the all the particles to move slowly, say between 0 and 1.5 pixels per second and in any direction. In order to set the speed of the particles, we can use the **part_type_speed** function. It takes five parameters. The first parameter is simply the index of the particle type that we want to change. The next two parameters are the min and max speed of a particle. We thus set the min speed to **0** and the max speed to **1.5**. The last two parameters have to do with speed increase and "wiggling" during the lifetime of the particle. We set both those to **0**, since we do not want the particle to change its speed once it has been created.

The complete function call would then be: **part_type_speed(myParticleType, 0, 1.5, 0, 0);**

Now we also have to specify in which directions the particle can go. That is done using the **part_type_direction** function. It takes five parameters too. First, the index of the particle type, then the min and max directions for the particle, and, finally, a direction increase and direction random parameter. The last two parameters apply to the particle during its lifetime, which we do not want, so we will just set the min and max direction. Since we want the particles to go in any direction, we can set the min direction to **0** and the max direction to **360**. Here is the function call:

part_type_direction(myParticleType, 0, 360, 0, 0);

Both those function calls are made in the **CREATE** event of the pod. So, here is the entire **CREATE** event code, with the newest additions marked with blue:

```
// Create a particle system.
myParticleSystem = part_system_create();

// Then, define a particle type.
myParticleType = part_type_create();
// Set the colors of the particle type.
myBrightYellow = make_color_rgb(255, 230, 200);
myOrange = make_color_rgb(220, 130, 20);
myDarkRed = make_color_rgb(50, 0, 0);
part_type_color3(myParticleType, myBrightYellow, myOrange, myDarkRed);
// Set the speed for the particles
part_type_speed(myParticleType, 0, 1.5, 0, 0);
// Set the direction of the particles.
part_type_direction(myParticleType, 0, 360, 0, 0);
// Set the lifetime of the particles to between 30 and 50 steps.
part_type_life(myParticleType, 30, 50);

// Create a particle emitter for our particle system.
myParticleEmitter = part_emitter_create(myParticleSystem);
// Set the particle emitting region to an ellipse (a circle, actually) with Gaussian
          distribution.
part_emitter_region(myParticleSystem, myParticleEmitter, x - 10, x + 10, y - 10, y +
          10, ps_shape_ellipse, ps_distr_gaussian);
// Set the particle emitting system to 5 particles per step.
part_emitter_stream(myParticleSystem, myParticleEmitter, myParticleType, 10);
```

I have marked the additions in bold blue. Note that I have also altered the number of particles emitted per step in the last line and set it to **10**. You can do that too, if you like. I thought it looked better that way.

Run the game again. Neat! Now there is almost a kind of 3D feeling to the particle tail. :)
That will be enough of those particles.

A particle star field

Now that we know how to make particles, we should try to make a star field with a particle system. Basically, we will emit pixels from the entire top border of the room.

Start by adding a new object: **objStarField**. Set its **Depth** to **10** in order to make the particle field appear behind all other objects. Add it to the room too. The plan is that we should have particles emit from the top of the room and travel downwards. In order to make the illusion of depth in the starfield, we will make four different particle types with different color and speeds. The brightest particle will move the fastest, and the darkest particles will move the slowest.

In the **CREATE** event of the starfield, we define the particles in the following way:

```
// Create a star field particle system.
myStarField = part_system_create();

// Create four particle types that look like stars.
for (a = 1; a < 5; a += 1) {
    myStarType[a] = part_type_create();
    tempColComp = 255 - (a - 1) * 30;
    tempCol = make_color_rgb(tempColComp, tempColComp, tempColComp);
    part_type_color1(myStarType[a], tempCol);
    part_type_speed(myStarType[a], 5 - a, 6 - a, 0, 0);
    part_type_direction(myStarType[a], 270, 270, 0, 0);
    part_type_life(myStarType[a], 1000, 1000);
}

// Make a particle emitter for each star type.
for (a = 1; a < 5; a += 1) {
    myEmitter[a] = part_emitter_create(myStarField);
    part_emitter_region(myStarField, myEmitter[a], 0, room_width, -5, 0,
               ps_shape_rectangle, ps_distr_linear);
    part_emitter_stream(myStarField, myEmitter[a], myStarType[a], -4);
}
```

That was quite a lot of code at a time. But look through it slowly and you will find that it contains mostly things that we have done before.

First, the starfield particle system is created. Then a **for** loop is started, where the variable **a** is increased from 1 to 4 (less than 5). During each **for** loop, a particle type is created and its index stored in an **array** (**myStarType[]**). A color is calculated, starting at (255, 255, 255) and going down with 30 in each color in each loop. The color is then assigned to the entire life time of the particle type. A speed is set for the particle type, between (5 - a) and (6 - a), so it will be slower for each **for** loop. The direction for all particle types is set to 270, and the lifetime is set to 1000 steps.

In the last **for** loop, four particle emitters are created for the four particle types. The region for each emitter is set to be across the entire room length and five pixels high just above the top of the room. Each emitter streams a new particle every fourth step. That is what happens when we set the amount of particles to a negative number.

Now that we have defined the particle system, types and emitters. Let's see how the game looks now!

Running the game now should display a little starfield.

Now there are **LOTS** of particles in the game. There are so many that my computer starts to stagger a bit. One of the problems is that the stars are "alive" much longer than needed. We could make that better through lowering their lifetimes, but we will try a different approach - the **Particle Destroyers**.

A particle destroyer is a defined area of the room. If a particle from the system that the destroyer belongs to enters this area, it is instantly destroyed. Sounds harsh...

What we will do is to add a destroyer for the star particles just below the bottom of the room. We will make it some 10 pixels high, since the highest speed of a star particle is less than 10 pixels and thus it will not "miss" the destroyer.

Here is what we add to the **CREATE** event of the **objStarField**:

```
// Create the destroyer
myDestroyer = part_destroyer_create(myStarField);
part_destroyer_region(myStarField, myDestroyer, 0, room_width, room_height, room_height
          + 10, ps_shape_rectangle);
```

A destroyer is created for the starfield, and its region is set to the entire room width (x = 0 to room_width) and a 10 pixel area below the room (y = room_height to room_height + 10). The shape of the destroyer is set to a rectangle type.

Running the program now should be a bit smoother, because stars are destroyed as soon as they hit the destroyer, just below the bottom of the room.

Into the Black Hole

The last thing we add to the starfield particle system is a black hole. The black hole will be created from a **particle attractor**. The attractor has a single coordinate position and a **force** defined. It would be a good idea to read the definition of a particle attractor in the GM manual.

Anyway, as I just mentioned, an attractor is defined by its position and by its force. We could for example put the black hole at location (200, 350) in the room. That is done with the **part_attractor_position** function. Secondly, we need to set the force of the attractor. The force is defined by its strength, its radius and its type. We could decide to have the force strength **3**, the radius **200** and that the force should be quadratic in its nature, which gravitational forces actually are in the real world (as opposed to linear or static).

Add the following code to the **CREATE** event of the **objStarField**:

```
// Create the black hole.
myBlackHole = part_attractor_create(myStarField);
part_attractor_position(myStarField, myBlackHole, 200, 350);
part_attractor_force(myStarField, myBlackHole, 3, 200, ps_force_quadratic, true);
```

If you run the game now, you will see that the star particles are drawn towards something on the left side of the room - the "black hole".

However, the stars are not destroyed as they are sucked into they hole, they simply pass through and out on the other side. It would be better if the stars were destroyed as they travel past the "event horizon" of the black hole. In order for that to happen, we create another destroyer and place it where the black hole is.

Add this to the **objStarField CREATE** event:

```
// Create the destroyer in the black hole.
myBlackDestroyer = part_destroyer_create(myStarField);
part_destroyer_region(myStarField, myBlackDestroyer, 165, 235, 315, 385, ps_shape_ellipse);
```

If you now look at the game, some of the stars disappear as they travel across the black hole. Great!

To be on the safe side, here is the entire **CREATE** event script for the starfield:

```
// Create a star field particle system.
myStarField = part_system_create();

// Create four particle types that look like stars.
for (a = 1; a < 5; a += 1) {
    myStarType[a] = part_type_create();
    tempColComp = 255 - (a - 1) * 30;
    tempCol = make_color_rgb(tempColComp, tempColComp, tempColComp);
    part_type_color1(myStarType[a], tempCol);
    part_type_speed(myStarType[a], 5 - a, 6 - a, 0, 0);
    part_type_direction(myStarType[a], 270, 270, 0, 0);
    part_type_life(myStarType[a], 1000, 1000);
}

// Make a particle emitter for each star type.
for (a = 1; a < 5; a += 1) {
    myEmitter[a] = part_emitter_create(myStarField);
    part_emitter_region(myStarField, myEmitter[a], 0, room_width, -5, 0,
             ps_shape_rectangle, ps_distr_linear);
    part_emitter_stream(myStarField, myEmitter[a], myStarType[a], -4);
}

// Create the destroyer
myDestroyer = part_destroyer_create(myStarField);
part_destroyer_region(myStarField, myDestroyer, 0, room_width, room_height, room_height
             + 10, ps_shape_rectangle);

// Create the black hole.
myBlackHole = part_attractor_create(myStarField);
part_attractor_position(myStarField, myBlackHole, 200, 350);
part_attractor_force(myStarField, myBlackHole, 3, 200, ps_force_quadratic, true);

// Create the destroyer in the black hole.
myBlackDestroyer = part_destroyer_create(myStarField);
part_destroyer_region(myStarField, myBlackDestroyer, 165, 235, 315, 385,
             ps_shape_ellipse);
```

Now, if you run the game, you should see some of the stars being dragged towards the black hole and disappear. Cool, huh? :) A few stars even stay alive as they make half an orbit around the hole and sprint away in a different direction.

Conclusion and thank you

Now you know about ~~exclusive graphics mode,~~ trigonometry, the drawing functions, and the particle system of Game Maker. Is there really anything more to learn now? Yes, of course there is!

This represents the end of my Game Maker introduction course. I want to thank everyone who has encouraged me to write it, and all you students for following it. I hope that I have managed to teach you something that you will find useful.

Now we will have to see if anyone else feels like writing a continuation course. I hope someone will.

Best regards and good luck!

Carl Gustafsson

Assignments

Assignment 10 - Change particles and vectors

Due date: Thursday, 5 January 2006, 08:00 AM
Maximum grade: 100

For the 10th and last assignment, I want you to do the following changes to the "game" that is developed in the 10th lecture:

1. Change the rotating and bouncing green square so that it instead is a **blue triangle**.

2. Change the starfield particle system so that the stars appear along the right side of the room and travel left (instead of appearing at the top and travel down).

Good luck!

Carl