

2DIY ActionScript FAQ

For the latest version of this document please go to

http://support.2simpleweb.com/public/docs/FAQ/2DIY_ActionScript.pdf

Q: Where can I find examples of using Actionscript code with 2DIY?

Q: Can I make the snake have a shorter tail?

Q: Can I add objects to the snake's tail when it eats them?

Q: How do I make a runaway apple?

Q: How do I make a monster chase me faster?

Q: How do I change the Journey keys so the car can move backwards?

Q: Can I make the player fly in the platform game?

Q: Where can I find examples of using Actionscript code with 2DIY?

- View the latest 2DIY user guide (which contains some code examples) here: www.2simple.com > support > downloads & updates. Also:

- <http://www.2simple.com/2diy/examples/>
- www.2diyarchive.co.uk, a site (independent from 2Simple) created by Simon Widdowson from Porchester Junior School in Nottingham.
- Download examples from here: http://support.2simpleweb.com/public/2DIY/as_examples.zip

Advanced option - you can get more insight into how 2DIY uses ActionScript as follows:

- Go to C:\Program Files\2Simple Software\2DIY and edit the "2diy.ini" file using Notepad.
- Set "outputCodeAsFile=True" (near the bottom of the list). Save and close the ini file.
- Open 2DIY and start an activity or game and press the play button.
- Go to your My Documents folder and you will now find a file named "2diy code.txt". This file contains some of the ActionScript code which 2DIY uses when creating your swf file. You will find some of the variables and functions 2DIY defines, and you can use these in your own code blocks as well.

Q: Can I make the snake have a shorter tail?

A: Yes. In the Snake game, drag a sun object to the main area on screen. Edit the sun's animation and choose "Adv". Add the following ActionScript code.

```
for(n=6;n<=20;n++) {  
    ob=eval('_root.p' add n);  
    ob._x=-100;  
    ob._y=-100;  
};
```

This will hide all except the first 5 segments of the snake's tail. The first line provides a loop which cycles through from n=6 to n=20. The 2nd line sets which segment of the tail is being changed. There are 19 tail segments in total, named p2, p3...p20. As with all other variables, they need to be accessed with "_root." in front. p2 is closest to the snake's head. The 3rd and 4th lines set the x and y position of the tail segments to negative numbers, ie off the screen. The above code could have been placed in the start-up section, for code to run once only at the start of the game. This would have worked for the first life of the snake, but when the snake dies, the code would not get executed again and the snake would have a full length tail again. This is why I placed the code in the sun animation; it is not ideal because its inefficient to set the position at every moment, but because it is only for a small number of objects, the effect is not noticeable. An alternative to setting the tail segment x and y positions to be off screen could be to set the segment to be invisible or have width zero. This does effectively hide the segment, but the snake will still die when it collides with the invisible segment.

Q: Can I add objects to the snake's tail when it eats them?

A: Below are some approaches you could take. The code samples described below can be downloaded from <http://support.2simpleweb.com/public/2DIY/snake/> . Please note that the code below is quite complex and definitely not for beginners.

1. The first approach is to leave the tail as is, and cause any eaten objects to copy the original tail segment's movements.

- a. Add to the start-up code block:

```
var numCaught:int = 0;
var caughtMonsters:Array=new Array();
```

The first is a variable which stores the number of objects the snake has caught. The 2nd is an array of objects which have been caught.

- b. Add to the collision animation of each monster object you add to the canvas:

```
var alreadyCaught = false;
for (n=0;n<_root.numCaught;n++) {
    if (_root.caughtMonsters[n] == this) {alreadyCaught = true;}
}
if (!alreadyCaught) {
    _root.caughtMonsters[_root.numCaught] = this;
    _root.numCaught++;
}
```

We use monsters as objects for the snake to collect because monsters have a collision animation code block but suns and apples do not. When the snake collides with a monster, 2DIY loops through the caughtMonsters array and checks if the monster has already been caught previously. If it has been caught previously, then it is ignored. If it has not been caught previously, it gets added to the caughtMonsters array.

- c. Add to a sun animation:

```
for (n=0;n<_root.numCaught;n++){
    segToFollow = n+2;
    ob = eval('_root.p' add segToFollow);
    _root.caughtMonsters[n]._x = ob._x;
    _root.caughtMonsters[n]._y = ob._y;
}
```

This loops through all the caught objects, and makes them copy the movement of the existing tail segments.

2. The second approach ignores the tail segments completely and causes the caught objects to follow the snake's head using their own follow algorithm.

- a. We start by hiding ALL the original tail segments of the snake, using the code mentioned above, but this time with n starting at 2 (add this to a sun animation.)

```
for(n=2;n<=20;n++) {
    ob=eval('_root.p' add n);
    ob._x=-100;
    ob._y=-100;
};
```

- b. Add the same start-up code and monster code as in the first approach above, but change the sun animation code for what the caught objects should do about following the snake. Use the code below:

```
for (n=0;n<_root.numCaught;n++){
    _root.caughtMonsters[n]._x += ((_root.player._x - Math.cos(_root.player._rotation
* Math.PI/180)*25*(n+1)) - _root.caughtMonsters[n]._x) / (8+n*8);
    _root.caughtMonsters[n]._y += ((_root.player._y - Math.sin(_root.player._rotation
* Math.PI/180)*25*(n+1)) - _root.caughtMonsters[n]._y) / (8+n*8);
}
```

```
}
```

This code works out the x and y position of each caught monster. It starts off from the position of the snake head (“_root.player”). It then takes the cos or sin of the angle of rotation of the snake, and multiplies this by the order in which the object was caught. The effect this has is to produce a line of objects behind the snake’s head, with the line having the same angle as the snake’s head itself. This is the target position for the caught object, but it is not moved to that position immediately; rather, it is moved there gradually, by taking a difference between the target position and the current position, and dividing that by a number. The number that is used for the division is also dependent on the order in which the object was caught; the further away an object is from the snake’s head, the longer it will take to get to its target position, giving a “snake-like” movement.

3. The third approach is almost exactly the same as the second approach, except we can use the same idea in the journey screen and not have to worry about the original tail at all. The movement of the car is also more fluid and suited to the new tail algorithm we have created. Use the same startup code and monster collision code as above, but change the sun animation to the following (just replaces “player” with “car”):

```
for (n=0;n<_root.numCaught;n++){
    _root.caughtMonsters[n]._x += ((_root.car._x - Math.cos(_root.car._rotation *
Math.PI/180)*25*(n+1)) - _root.caughtMonsters[n]._x) / (8+n*8);
    _root.caughtMonsters[n]._y += ((_root.car._y - Math.sin(_root.car._rotation *
Math.PI/180)*25*(n+1)) - _root.caughtMonsters[n]._y) / (8+n*8);
}
```

Q: How do I make a runaway apple?

A: The code below will make an apple run away from the player. Drag an apple to the canvas and add the code to its animation.

```
if (( Math.abs(_root.player._x - this._x) < 100) and ( Math.abs(_root.player._y - this._y) < 100)) {
    var angle = Math.atan2(this._y - _root.player._y, this._x - _root.player._x);
    this._x += Math.cos(angle)*2;
    this._y += Math.sin(angle)*2;
}
```

Explanation: You need to understand trigonometry for this one! The first step specifies that the apple should only run away if the player is closing in. The 2nd step works out the angle between the 2 objects by taking the arctan of the difference in y over the difference in x. The next two lines calculate the x and y position of the apple by taking the cos or sin of the angle. (Replace _root.player with _root.car for Journey.)

Q: How do I make a monster chase me faster?

A: You can assign a “chase” action for a monster without using actionscript by choosing the ready-made animation in the list. However if you want the monster to chase you faster, you need to use actionscript for this. The basic code, which you assign to the “animation” action of the monster, is:

```
var angle = Math.atan2(this._y - _root.player._y, this._x - _root.player._x);
this._x -= Math.cos(angle)*2;
this._y -= Math.sin(angle)*2;
```

This is similar to the “runaway apple” code. The first line works out the angle between the 2 objects by taking the arctan of the difference in y over the difference in x. The next two lines calculate the x and y position of the monster by taking the cos or sin of the angle. (_root.player marked in red to remind you to replace it with _root.car for Journey games.) Experiment with values other than 2 in the 2nd and 3rd lines to achieve a different monster speed – making the value larger, such as 4, will make the monster move considerably faster.

(2) The above code does not rotate the monster to face the player when it is chasing it. To achieve this effect, use the following code:

```
this._rotation= Math.atan2(_root.player._y-this._y,_root.player._x-this._x) *180/Math.PI;
this._x+=2*Math.cos(this._rotation/180*Math.PI);
```

```
this._y+=2*Math.sin(this._rotation/180*Math.PI);
```

The first line sets the rotation of the monster to be the same as the angle between the monster and the car. The next two lines calculate the x and y position of the monster by taking the cos or sin of the angle. Again, you can change the “2” in the last 2 lines to different values for different speeds. The 180 and Math.PI in each line appear because the rotation variable is measured in degrees but the trig functions work with radians.

(3) Finally, we can add the effect that occurs in the ready-made chase action;

```
if(Math.random() $<$ 0.05){  
    aR=Math.atan2(_root.player._y-this._y,_root.player._x-this._x);  
    this._rotation=Math.random()*60-30+ aR*180/Math.PI;  
};  
this._x+=2*Math.cos(this._rotation/180*Math.PI);  
this._y+=2*Math.sin(this._rotation/180*Math.PI);
```

This is based on the previous approach but only changes the direction of the monster some of the time (it takes a random number between 0 and 1 and only changes direction if the number is less than 0.05.). One other difference is that when the monster *does* change direction, it does not turn to directly face the player, but again adds a random element and turns to within 30 degrees of the player. Once again, you can change the number 2 in the last 2 lines to amend the monster speed.

(4) You could take this one step further and add a variable for the monster chase speed. Right-click the green triangle to access the initial code block and add the following:

```
_root.monsterChaseSpeed=2;
```

Then in the monster’s animation code, add the following

```
if(Math.random() $<$ 0.05){  
    aR=Math.atan2(_root.player._y-this._y,_root.player._x-this._x);  
    this._rotation=Math.random()*60-30+ aR*180/Math.PI;  
};  
this._x+=_root.monsterChaseSpeed *Math.cos(this._rotation/180*Math.PI);  
this._y+=_root.monsterChaseSpeed *Math.sin(this._rotation/180*Math.PI);
```

Try different values, but a chase speed of 4 will most likely be fast enough!

Q: How do I change the Journey keys so the car can move backwards?

A: The first step is to “unassign” the back key. Right-click the car image (the big image outside and above the canvas) and then click “keys”. For the “brake” key, choose any key other than “{Down Arrow}”. The second step is to use actionscript to code our own back movement. Drag a sun object to the canvas and add this to its animation code :

```
if(Key.isDown(40)==true) {  
    _root.v-=0.5;  
    if (_root.v * -1 > _root.maxCarSpeed) { _root.v= -1 * _root.maxCarSpeed;};  
}  
if (_root.v < 0) { _root.v +=0.05; }
```

This code changes the car’s velocity (_root.v) to be negative if the back arrow is pressed. It limits the negative velocity to be -1 * maxCarSpeed. It also applies a damping factor so that eventually the car will stop moving if no keys are pressed (this already happens when the velocity is positive.)

Q: Can I make the player fly in the platform game?

A: Gravity is a force which exerts constant downward acceleration on an object. For each second that an object is under the force of gravity, its downward speed will increase at a constant rate until other factors put a limit to this speed (such as air pressure or reaching the ground.) In 2DIY, the player’s vertical position is saved in the variable `_root.player._y`. Its downward speed is saved in the variable `_root.dy` (a *positive* value in dy means the player is currently moving downwards. A negative value means the player is moving up.) The maximum downward speed a player can achieve is stored in `_root.maxFallingSpeed`, which can be set in the start-up script section (right click the green triangle).

To try achieve flight, we could first try to set `maxFallingSpeed` to be 0. If we try this we will indeed see that the player does not fall down at all, however the flip side of this is that it is impossible to move downwards! We could nonetheless change the `maxFallingSpeed` to be a smaller than 16; for example 10. This would result in the player not falling as quickly as before.

Now add a sun object to the main screen area and add the following code to its animation:

```
if(Key.isDown(38)==true) { _root.dy-=2; }  
if(Key.isDown(40)==true) { _root.dy+=2; }  
if(_root.dy < -10) { _root.dy = -10; }
```

The first line checks if the up arrow has been pressed, and when this happens it increases the upwards speed of the player (remember it *decreases* the value of `dy` by 2, since 2DIY treats a positive `dy` value as downward speed.) The next line checks if the down arrow has been pressed and acts accordingly. The last line adds a limit to the upward speed of the player in the same way that `maxFallingSpeed` places a limit on the max downward speed. The above 3 lines do allow the player to fly, while at the same time maintaining a gravity force which acts on the player when no up button is pressed (like a “jet pack”).

The above code is not necessarily the only way to achieve flight. Another approach might be to use code like “`if(Key.isDown(38)==true) { _root.player._y -=20; }`”. This changes the actual `y` value of the player rather than the speed variable, however I have found that using the speed variable results in a smoother experience.

2Simple Software
support@2simple.com
020 8203 1781
Last updated 7 Feb 2011