TILE BASED GAMES

written by
TONYPA

Index

Welcome

Here you can read some tutorials about making tile based games with Macromedia Flash.

These tutorial are heavily based on OutsideOfSociety tutorials by Klas Kroon. Go and read them anyway, those are one of best you can find.

These tutorials expect you to know something about actionscript and Macromedia Flash. They might be too hard to understand, if you never before made any games with Flash. And as my English is not very good, there are probably lots of mistakes and some parts might not be well explained. Im sorry, I do my best. If you find bugs or mistakes, let me know.

I have used Flash5 to create the code and source files. Mainly because I am used to Flash5 and I work fast with it. The code and source files work well in FlashMX (ver6) too, you shouldnt have problems with them. The FlashMX2004 (ver7) with new AS2 is not compatible with older actionscript, so the code might fail. It might work too, but generally, if you really want to use AS2, you should write new code yourself. The principles, of course, you can still use, so it might be worth look through the tutorials anyway.

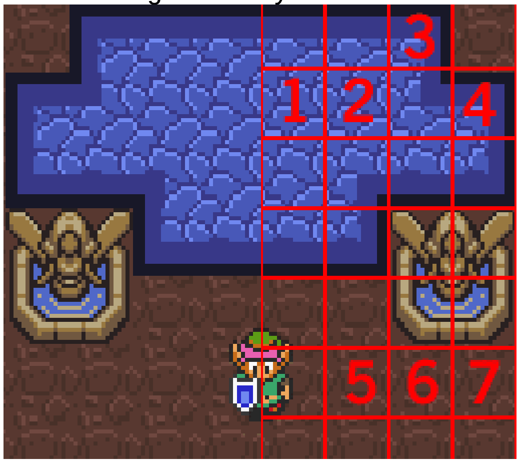
Examples and code and source files presented here are free to use. You can modify them as you wish, you can use them anyway you like.

I hope someone finds something useful here :)

Thanks
Tony / 2003 / 2004

Why tiles

Before diving into coding the game, lets talk little about the tile based games. Why would you want to use tiles anyway? Are tile based games easier to make or perhaps they are more complex then art based games? Is Flash good for tile based games?
Tiles were already used long-long time ago for making games. It was the time, when computers didnt have speeds of GHz and hundreds of MB memory. Slow speed and limited amount of memory meant game makers had to use their brains and create clever ways to make games look better and run faster.
So, you want to put nice background into your game, but the picture would be too large and make game very slow. What to do? Slice the picture into tiles!



In the picture you can see that parts of picture are exactly same. 1 is same as 2, 3 is same with 4 and parts 5-7 are all same thing. If you slice up the picture and reuse same parts in different areas, you have created the tiles. The big picture has much bigger filesize then tiles.
Other nice feature about tiles is, when you might want to replace part of your background, then you dont have to redraw everything, you can only replace 1 tile. You can reuse the tiles with different objects too. For example you might have tile with grass and another tile with flower on the grass, then you can take same grass background and only draw the flower.

Flash and tiles

As we all know Flash is vector based, so Flash files have small size and you can resize them. So, you wont need tiles at all to create game? Well, you can easily do art based games in Flash, but when your game area gets bigger and you want more features, you might be in trouble. Some things are so much easier to do in tile based games (isometric view, pathfinding and depth sorting to name few). Dont forget, tile based games have been around for a long time and much of the theory is usable with Flash too.
Sad part about tile based games in Flash is, that we wont benefit much from the drawing or timeline parts, our game is made with actionscript and basically we just have bunch of code to create, move and modify images on the stage.
Its also good idea to use bitmap images as tiles graphics. Yes, we could draw everything inside Flash and have the vector graphics, but when the game is run, player has to calculate the vectors on screen and we dont want anything to slow down our game. Bitmaps are pre-rendered and usually they look better too. If you want to import bitmap tiles into Flash, its usually best to save the images as GIF files with transparent background (for objects).
Enough boring talk, lets make something :)
First, we will see how to store our tile based maps.

Map format

We will hold our maps in nice format Flash provides: arrays. If you dont know, what is an array, open up Flash help and read first.

Two dimensional array

We need two-dimensional array for map. No, its not something out of other dimension, its only array with array as every element. Confused? Lets see.
Normal, simple array that normal people can make:

```
myArray=["a", "b", "c", "d"];
```

Thats was easy. You can get value of first element with myArray[0], which is "a", second element myArray[1] has value "b", and so on.
Now the clever part! What if we dont put "a", "b" and "c" into array, but we put other arrays there? Yes, we can do that. Here, lets make some arrays:

```
a=["a1", "a2", "a3"];
b=["b1", "b2", "b3"];
c=["c1", "c2", "c3"];
myArray=[a, b, c];
```

Now we have declared array myArray and each element in there is also an array. So, the value of first element myArray[0] is a and a is array of ["a1", "a2", "a3"], second element has value ["b1", "b2", "b3"]. If you write:

```
myVar=myArray[2];
```

then myVar gets value ["c1", "c2", "c3"].
Ok, so what, you might ask. We dont have to stop here. If you write:
myVar=myArray[2][0]; then it gets value of first element of third element in myArray "c1".
Lets practise more. myVar=myArray[0][1] takes first element of myArray (a) and second element from that ("a2").
myVar=myArray[1][0] gets value "b1"
You get the picture?

Making the map

First we write the map array that will hold information about every tile:

```
myMap = [
[1, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 0, 0, 0, 0, 1],
[1, 0, 0, 0, 0, 1, 0, 1],
[1, 0, 0, 0, 0, 0, 0, 1],
[1, 1, 1, 1, 1, 1, 1, 1]
];
```

As you can see our map has 6 rows and 8 columns. If our hero would start from top left corner, he could move 8 steps right and 6 steps down before going out from the map and wondering into unknown space.
But some smart people already have raised the important question: "What do those numbers in the map array mean?". Well, we will use some OOP (thats Objects, but dont run away, they are not so frightening as they sound) to create the tiles and manage our game (look into Links section for OOP tutorial for Flash). We declare several tiles first, they are like templates for other tiles we actually put into game. Then we loop through the map array and pick up the numbers in every position.
If for example we get number 1, then we create new tile from Tile1 template. Then in the game, when we reach that tile, we will check the properties of that tile objects. It can have many properties, most basic tiles have only 2 properties, walkable and frame.
Walkable is property that shows if any character can walk into that tile (then we have set walkable=true) or it can not do that (false). We do not use hitTest as hitTest is slow and it is not cool to use it with tile based game.
Frame is property that tells us what frame of tiles movie clip we have to show in that position. It is used when placing the tiles on the screen. As we use same tiles movie clip for every tile by attaching it over again, they all would show frame 1 by default. More about this on Creating Tiles section.
So, if we declare following tile:

```
Tile1= function () {};
Tile1.prototype.walkable=false;
Tile1.prototype.frame=2;
```

then we make similar object every time there is 1 in the map array (Tile1), we also say this tile cant be stepped on (walkable=false) and in that spot tile movie clip should show frame 2.

More Maps

You might wonder, why exactly have I chosen this kind of map format. I cant say this is absolutely best way to go, I cant say this map format creates maps fastest or creates smallest file size. I can only say that after couple years of messing with tile based games, I have found this format to suit best my needs. But lets look at other possible ways to hold your map data.

JailBitch method

The original OutsideOfSociety tutorials use very simple map format. Its saved same way in two dimensional array and every number gives us frame number to show in that spot. Every time you would need to check if next tile is wall (or something to pick up, or door or almost anything), you would look up the number from map array.
When looking for collision, you determine the section of frames that count as walls (or pick-ups, or doors). For example, you can make up your mind and say, that all the tiles from frame 0 to 100 are walkable tiles, all tiles from 101 to 200 are walls and tiles >200 are special tiles.
When you have few different tile types and tiles wont change much, this is good and easy way.

Tree in the Desert

Some maps have many different tiles, some have very few. For example, imagine the desert, where for miles and miles there is nothing but sand, if you are lucky, you can see few oasis. Or the sea, there is water and water and more water and finally small island.
If your map is made up from mostly same kind of tiles (sand) and you have only some small variation (trees), then two dimensional array is not good choise. It will hold too much dead information, rows of zeros before some other frame shows up. In this case you might be better to declare all the non-sand objects separately and let everything else be sand.
Lets suppose you have 100x100 map and you have 3 trees there. You can write:

```
trees = [[23,6], [37,21], [55,345]]
```

When creating the map, you step through trees array, place the trees and let every other tile show sand image. Thats much more simpler then writing down 100x100 two dimensional array.
Of course, when you make more objects (trees, bushes, grass, stones, water), this method loses much of its speed and it might become hard to remember what tiles are placed where.

S, M, XXXL

If you have Flash MX or later, you have probably heard of magic shortcut XML. It is similar format to HTML, that allows declaration of many things. You can use XML to hold your map data.
Following XML map explanation is based on the Jobe Makar's book "Macromedia Flash MX Game Design Demystified".

Lets look at the sample map in XML:

```
<map>
 <row>
  <cell type="1">
  <cell type="1">
  <cell type="1">
 </row>
 <row>
  <cell type="1">
  <cell type="4">
  <cell type="1">
 </row>
 <row>
  <cell type="1">
  <cell type="1">
  <cell type="1">
 </row>
</map>
```

Here we have set 3x3 map. First there is header "map". Then 3 "row" nodes is set. Each of them has 3 "cell" nodes.
To load maps from external files, XML might be good solution as most of XML parsing can be done with Flash MX built-in functions. Loading two dimensional arrays from text files is not that easy, you always get string from loading variables and you have to split the string into array, which again, is very slow.
You can also see the disadvantages of XML: it leads to much bigger filesize and you need Flash 6 player for it.
All the following examples use two dimensional array to hold map data and use objects when creating tiles on the screen like explained in the chapter "Map format".

Creating tiles

Like you saw from chapter "Map format", we will have our map in two-dimensional array. Now we will make the tiles appear on the screen, place them in correct position and make them show correct frame.

First we declare some objects and variables:

```
myMap = [
[1, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 0, 0, 0, 0, 1],
[1, 0, 0, 0, 0, 1, 0, 1],
[1, 0, 0, 0, 0, 0, 0, 1],
[1, 1, 1, 1, 1, 1, 1, 1]
];
game={tileW:30, tileH:30};
game.Tile0= function () {};
game.Tile0.prototype.walkable=true;
game.Tile0.prototype.frame=1;
game.Tile1= function () {};
game.Tile1.prototype.walkable=false;
game.Tile1.prototype.frame=2;
```

So, we have our map in the variable myMap. Next line after the map declares object called "game". We will use this object to hold all the stuff needed, we could hold all the stuff in the _root or anywhere, but its cleaner to put stuff in one certain place so we always know where it is.
Notice how we also give that game object two properties tileW=30 and tileH=30. Thats how wide and how tall all our tiles will be. Tiles dont have to be squares, you could also use wide or tall rectangles. Whenever we want to know width or height of any tile, we can write:

```
game.tileW;
game.tileH;
```

And when we want to change size of tiles, we only have to change numbers in one line.
Next we set the tile prototypes inside our game object:

```
game.Tile0= function () {};
game.Tile0.prototype.walkable=true;
game.Tile0.prototype.frame=1;
```

First line game.Tile0= function () {} declares new object prototype. When we get 0 from map array, we will use Tile0 as template to make new tile object on that position.
Next 2 lines give that Tile0 object and every object created with Tile0 prototype some properties. We will set every such object having property walkable=true (that meaning we can walk over it) and frame=1 (it will show frame 1 from the attached tiles movie clip).

Make my map show

Are you ready to make some tiles? We will make function buildMap that will handle all the tiles placement. If you want to create more levels, you can use same function with different map array. buildMap function will do following:
+ attach container movie clip
+ loop through map array
+ create new object for each tile
+ attach all the tiles clips
+ place the tiles in correct position
+ show correct frame in each tile
Here is the code for it:

```
function buildMap (map) {
 _root.attachMovie("empty", "tiles", ++d);
 game.clip=_root.tiles;
 var mapWidth = map[0].length;
 var mapHeight = map.length;
 for (var i = 0; i < mapHeight; ++i) {
  for (var j = 0; j < mapWidth; ++j) {
   var name = "t_"+i+"_"+j;
   game[name]= new game["Tile"+map[i][j]];
   game.clip.attachMovie("tile", name, i*100+j*2);
   game.clip[name]._x = (j*game.tileW);
   game.clip[name]._y = (i*game.tileH);
   game.clip[name].gotoAndStop(game[name].frame);
  }
 }
}
```

First line declares the function and also we set the argument of the function to be variable map. When we call the function, we will pass the map array to it, so variable map will be two-dimensional array.
Next line does attach container movie clip on the stage:

```
_root.attachMovie("empty", "tiles", ++d);
```

You will need empty movie clip (no graphics inside it) in the library. Right click on that movie clip in the library, choose "Linkage..." check "Export this symbol" and write "empty" in the Identifier box. Now the attachMovie command will look for movie clip with linkage name "empty" in the library. It will then make new instance of this movie clip on the stage and give this new mc name "tiles". That movie clip will hold all the tiles we place on stage. Nice thing about using container movie, is that when we want to remove our tiles (like when game ends), we only have to remove "tiles" movie clip and all the tiles will disappear. If you attach all the tiles directly into _root level, and you go to next frame (like game end frame) then the attached tiles wont disappear, you have to delete all of them with actionscript.
Once we have movie clip for all the tiles, we also link it to our game object game.clip = _root.tiles. Now when we need to access tiles movie clip, we can use game.clip. Thats handy, if we ever need to place tiles somewhere else, we only have to rename this line and not go through all the code.
Then we make two new variables mapWidth and mapHeight. Those we will use in the loop to step through map array. mapWidth has value of length of first element in the map array map[0].length. Look back in the "Map format" chapter if you forgot how the map array looks like. First element of map array is another array [1, 1, 1, 1, 1, 1, 1, 1] and mapWidth will have the value of its length or number of elements. We now know how wide will our map be.
Same way mapHeight will have value of map.length, thats number of rows in the map array. And thats how many rows we will need to make.
We will loop through the map array using lines:

```
for (var i = 0; i < mapHeight; ++i) {
for (var j = 0; j < mapWidth; ++j) {
```

We start variable i from 0 and will add +1 to it until it is less then height of our map.

Variable j loops from 0 to width of our map.
Variable "name" from the line var name = "t_"+i+"_"+j gives us name of out new tile object. Lets suppose i=0 and j=1, then name = "t_0_1". If i=34 and j=78, then name has value "t_34_78".
Now we create new tile object:

```
game[name]= new game["Tile"+map[i][j]]
```

In the left side game[name] will show that new tile object is placed inside game object, like all our stuff. Value of map[i][j] gives us number from map array depending what are i and j values. We then use keyword "new" to create new tile object from the prorotypes we declared earlier. Now we have new object in the game object representing current tile.
In next lines we attach new movie clip on stage and use game.clip[name] to access it. Movie clip will be placed on correct x/y position using j and i variables multiplied by width or height of tiles. As our new tile object inherited "frame" property from its prototype, we use it to go to correct frame with gotoAndStop command.
When we want to create tiles from map, we call out buildmap function like this:

```
buildMap(myMap);
```

You can download the source fla with all the code and movie set up here.

More tiles

Since we have set up our tiles as objects, we could also use some advantages objects give. Nice thing about objects (beside being cute, fluffy and loved all around the world) is how they can inherit properties from each other. Hopefully you did actually read the last chapter and you noticed how we wrote the tiles prototypes:

```
game.Tile0= function () {};
game.Tile0.prototype.walkable=true;
game.Tile0.prototype.frame=1;
```

This allows us to write properties for each type of tile only once and not every time new tile is created. Same way we could take the logic further and save us from the trouble typing out all the different tile types.
Lets declare general Tiles class:

```
game.TileClass = function () {};
game.TileClass.prototype.walkable=false;
game.TileClass.prototype.frame=20;
```

Here we have assumed, that each and every tile in the game is not walkable by default and it shows frame 20. Of course many tiles are actually walkable or we couldnt move around, and they also might show some other frame (not that frame 20 has anything wrong with it, its fine, healthy frame). While the problems might look overwhelming, dont be worried, be happy, we can make this system work and world better place.
Now we make some tile types:

```
game.Tile0 = function () {};
game.Tile0.prototype.__proto__ = game.TileClass.prototype;
game.Tile0.prototype.walkable=true;
game.Tile0.prototype.frame=1;
game.Tile1 = function () {};
game.Tile1.prototype.__proto__ = game.TileClass.prototype;
game.Tile1.prototype.frame=2;
game.Tile2 = function () {};
game.Tile2.prototype.__proto__ = game.TileClass.prototype;
```

By using this clever little thing called "__proto__", we dont have to write all the same properties over and over again, our tiles get all necessary stuff from the TileClass. So, when we have made new tile:

```
game.Tile2 = function () {};
game.Tile2.prototype.__proto__ = game.TileClass.prototype;
```

all tiles created later from Tile2 template inherit the properties walkable=false and frame=20 from TileClass. Thats very nice of them, isnt it? But fun is not over yet, we can also change the properties tiles got from TileClass. You dont believe me? Its true! Here:

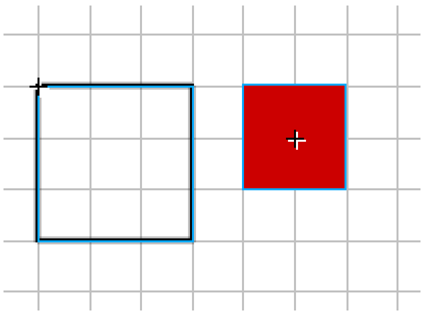
```
game.Tile0 = function () {};
game.Tile0.prototype.__proto__ = game.TileClass.prototype;
game.Tile0.prototype.walkable=true;
game.Tile0.prototype.frame=1;
```

Tile0 got walkable=false from the TileClass, but since we have declared new prototype walkable=true for it, the old walkable property is overrun and all tiles made from Tile0 are now absolutely walkable. We have also declared new frame for the Tile0, it wont show frame20 anymore, it will show frame1 instead.
All this is not perhaps too much for now, we only have couple of tiles and few properties, but if you want to make game with many tile types, each of them with many properties, then typing them all will become boring very soon.

The Hero

No game can exist without the hero. The hero will save the world and princess and beat up the bad guys. We will add hero too. He wont save the world yet, he wont do anything useful, but he is there.

The hero is red square :) What, he doesnt look mighty? You can draw your own hero. His movie clip is in the library named "char" and its also been set up to be exported as "char". You should not make the hero movie clip larger then tiles.
Also note, how hero movie clip (red square) has registration point centered and tiles movie clip has registration point in the upper left corner:



Want some code? Add the line after tiles definitions:

```
char={xtile:2, ytile:1};
```

This code declares new "char" object. The char object will hold all the information about our character, how he moves, how he feels, what he eats.
This time we will give char object only two properties, xtile and ytile. Those give us the tile our hero stands on. When he moves around, we will update the xtile/ytile properties and we will always know what tile is under the hero. For example, when xtile=2 and ytile=1 (like we wrote), that means hero is standing on the tile "t_1_2". When you look at the example movie, you see that hero stands on the tile 3 positions left and 2 positions down from the upper left corner of our game. All the tiles start counting from 0. We will add more properties to hero later.
To place hero movie clip on stage, add following lines to the buildMap function after the for loops:

```
game.clip.attachMovie("char", "char", 10000);
char.clip = game.clip.char;
char.x = (char.xtile * game.tileW)+game.tileW/2;
char.y = (char.ytile * game.tileH)+game.tileH/2;
char.width = char.clip._width/2;
char.height = char.clip._height/2;
char.clip._x = char.x;
char.clip._y = char.y;
```

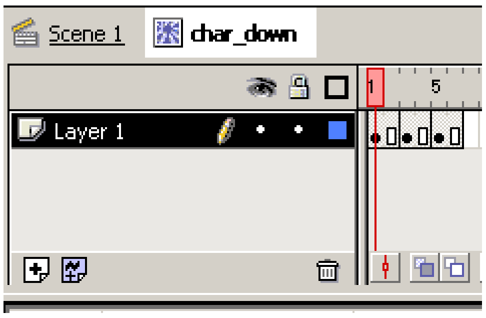
First line does attach new movie clip from the library in the game.clip (you remember that we saved path to the _root.tiles as game.clip in the last chapter) and give this instance name "char".
Then we save the path to the char movie clip into char object, so every time we want to access the movie clip, we can use simpler char.clip instead of typing this movie clip's full path _root.tiles.char. It also saves us from going through all the code if we might need to

move char movie clip to somewhere else.
Next we will calculate two properties in the char object: x and y. You may wonder, what for we need more properties, we already have xtile and ytile. Remember, xtile/ytile count the number of tiles, not actual pixels. The x/y properties will hold the pixel coordinates of our char movie clip. Its good idea to have coordinates in the variables before placing movie clip, you may need to change position because hero has hit the wall or has lost the balance and changing variables is easier then changing _x/_y properties.
We will calculate the actual position of our hero by multiplying the tile number he stand on with size of the tiles and adding half the tile size to place char on the center of tile. So, char.xtile * game.tileW gives us tiles number on horisontal multiplied by width of tile taken from the game object.
Next we save the half the width and height of our hero movie clip into char object. Those will become very useful, when calculating where are the boundaries of hero. Note that you can create your own boundaries, you dont have to use width and height of movie clip. Some heros might have long puffy hair which can collide with walls, then declare your own width and height variables.
Last two lines place the char movie clip char.clip to the coordinates we calculated earlier as x and y.
You can download the source fla with all the code and movie set up here.

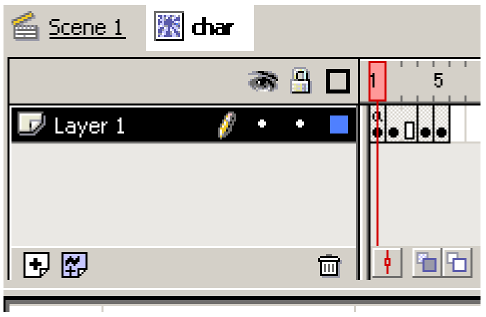
Keys to Move

In this chapter we are going to move our hero using arrow keys in 4 direction. He will face in the direction of movement and show walking animation while moving. Animation is stopped when standing still.

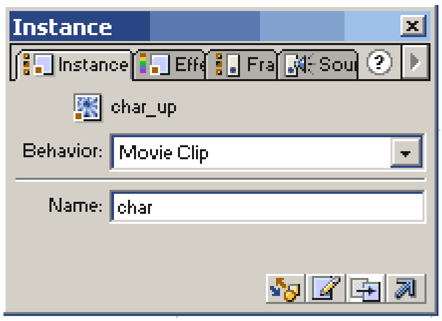
There is not collision detection, so hero can walk outside from stage, but dont worry about that, we will fix it in next chapter.
First lets set up our hero character. Create 3 new movie clips. You will need one movie clip for character moving left (or right, I chose left), 1 movie clip for character moving up, and final movie clip for moving down. Inside those movie clips place the animations of your character moving.



There is no code needed for those movie clips.
Now edit your "char" movie clip and create total of 5 keyframes inside it:



In keyframe 1 place the "char_up" movie clip, in keyframe 2 place "char_left", keyframe 4 gets "char_right" and keyframe 5 "char_down". You can use same movie clip for both left and right movement, you only flip one of the instances horisontally. Now make sure every instance with movement animation has instance name "char". Check the frames 1, 2, 4 and 5 again. They all are named "char"? Dont worry, if you dont understand yet why certain movement is suppose to be in certain frame, we explain it when we look at the movement code.



char_up
Behavior: Movie Clip
Name: char

Ok, time to script the actions.

Coding

Our hero will move and movement needs speed, so add speed property to hero object:

```
char={xtile:2, ytile:1, speed:4};
```

Speed is the number of pixels our hero will move on the screen, higher speed means he moves faster and slower speed makes him move like a snail. Its good practise to use integer as speed or you might get weird results and you wont see any difference between 10 pixels and 10,056873 pixels anyway.
As you remember (if you dont remember, please look in the previous chapter), we have created object _root.char to hold char info, and we have placed "char" movie clip in the "tiles" movie clip. To make our hero wake up and start to move, we need two more functions and controller movie clip to check for keys in each step.
Drag instance of "empty" movie clip to the stage. You can place it outside of visible area. Its only going to call the function so it doesnt matter where it stands. Write code to this mc:

```
onClipEvent (enterFrame) {
   _root.detectKeys();
}
```

You can see that in every frame we are calling function detectKeys. Now lets write it:

```
function detectKeys() {
  var ob = _root.char;
  var keyPressed = false;
  if (Key.isDown(Key.RIGHT)) {
    keyPressed=_root.moveChar(ob, 1, 0);
  } else if (Key.isDown(Key.LEFT)) {
     keyPressed=_root.moveChar(ob, -1, 0);
  } else if (Key.isDown(Key.UP)) {
    keyPressed=_root.moveChar(ob, 0, -1);
  } else if (Key.isDown(Key.DOWN)) {
    keyPressed=_root.moveChar(ob, 0, 1);
  }
  if (!keyPressed) {
    ob.clip.char.gotoAndStop(1);
  } else {
    ob.clip.char.play();
  }
}
```

First we declare two variables. We set variable ob to point to _root.char (remember, thats where we hold all the info about hero) and we set variable keyPressed to false. Variable keyPressed we use to check if in the end of the function some arrow keys have been pressed or not.

Next we have 4 similar if statements. Each of them checks if one of arrow keys is pressed down. If key is down, they call another function moveChar, using lines like:

```
keyPressed=_root.moveChar(ob, 1, 0);
```

This line calls the function moveChar using 3 arguments. First argument is the variable ob, that points to our char object. Last two arguments are always set -1, 1 or 0. Those determine if we should move hero horisontally by changing its x coordinate (second argument) or vertically by changing y coordinate (third argument). Last we set the return value of function moveChar to the variable keyPressed. You can soon see that moveChar function always returns "true", so if any of the arrow keys are pressed, variable keyPressed will be set to true.
Last piece of code checks, if variable keyPressed is false, meaning no arrow keys has been pressed, in which case we stop the movement animation using gotoAndStop(1). If variable keyPressed is however true, we continue to play the movement animation.
Now the second function:

```
function moveChar(ob, dirx, diry) {
  ob.x += dirx*ob.speed;
  ob.y += diry*ob.speed;
  ob.clip.gotoAndStop(dirx+diry*2+3);
  ob.clip._x = ob.x;
  ob.clip._y = ob.y;
  return (true);
}
```

See, moveChar function accepts 3 arguments, variable ob will be the object to move, dirx and diry are values to move in x or y direction. This is very universal function, we can use it to move everything on the game. If we had bullet flying, we could call moveChar with bullets direction, if we had enemy walking, we can again use same function to move it.
Next two lines adds value of ob.speed to objects x or y variable. Again, if we had different objects (bullet, enemy), they can each have their own speeds. So, when we detected right arrow key and called the moveChar function using 1, 0 then dirx=1 and diry =0. Now the x will be increased by speed while y remains the same. If we called moveChar function using 0, -1 (thats what up arrow uses), the y would be decreased by speed and x remains same.
Note that if we would have more movement stuff, like collision or gravity, we would calculate the values of x and y right here, before even replacing the actual movie clips. Thats much better way then using simple mc.hitTest method.
The line

```
ob.clip.gotoAndStop(dirx+diry*2+3);
```

sends character movie clip to correct frame to face to the direction it moves. You can calculate all the variations of dirx/diry (there are 4 different pairs so it wont take long to check) and you see that character movie clip is sent to right frame just like we did set it up earlier.
Dont have calculator? Lets see if it works: right arrow key was pressed, dirx = 1, diry = 0. Find the frame. diry * 2 = 0. dirx + 0 + 3 = 1 + 3 = 4. Its going to show frame 4. And frame 4 is where we did put our char_right animation.
Next two lines set the _x/_y properties of character movie clip to the values of x/y variables.

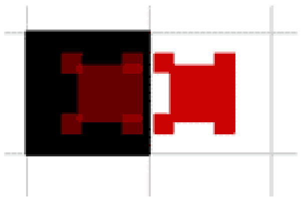
And finally we return "true" so we get the variable keyPressed to have correct value.
Collision with walls follows in the next chapter. Thats fun :)

You can download the source fla with all the code and movie set up here.

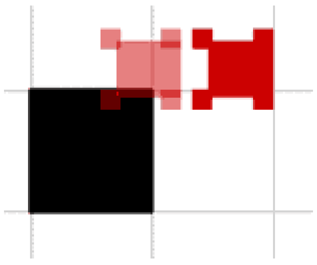
Hit the wall

Its no fun having hero that can walk, but cant hit the wall. We will make our hero to feel the power of solid brick wall. Or any other tile we decide to be not walkable.

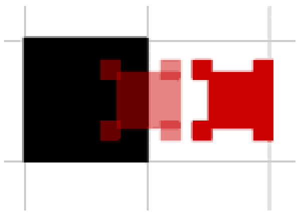
In the first chapter we did set our tiles to have property "walkable". When object representing tile in current position has property walkable set to "false", hero cant go there. But then again, if the walkable property is "true", then hero can walk there (thats called "logic", people learn it in the school, some even in the university, poor, poor devils).
In order this magic to work, we will do following: after arrow key has been pressed, we check if the tile, where char will walk, is walkable. If it is, we will move the hero. If the tile is not walkable (the hard brick wall type), then we will ignore the arrow keys pressed. This is perfect collision with wall:



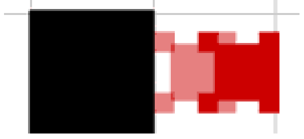
Hero stands next to wall and in next step he would be inside the wall. We cant let it happen, so we wont. No moving, man! But world is not perfect, what if only part of hero would be colliding:



That requires us to check for collision between hero and wall with all 4 characters corner points. If any of hero's corners (lower left corner in this example), would be inside the wall, we will stop the hero.
Or if hero is not next to wall yet, but would still go inside the wall if you allow him to step there:



We will have to place hero by the wall:



"Oh, no!" you might cry, "All this is impossible to do!" Not to worry, its not actually very hard.

Give me my Corners

We dont want parts of our character go inside wall so we have to check collision between hero and the unwalkable object using not one, but four points. We will use corner points, expecting most of heroes look like rectangles (they DO!).
For the purpose lets make new function called getMyCorners:

```
function getMyCorners (x, y, ob) {
  ob.downY = Math.floor((y+ob.height-1)/game.tileH);
  ob.upY = Math.floor((y-ob.height)/game.tileH);
  ob.leftX = Math.floor((x-ob.width)/game.tileW);
  ob.rightX = Math.floor((x+ob.width-1)/game.tileW);
  //check if they are walls
  ob.upleft = game["t_"+ob.upY+"_"+ob.leftX].walkable;
  ob.downleft = game["t_"+ob.downY+"_"+ob.leftX].walkable;
  ob.upright = game["t_"+ob.upY+"_"+ob.rightX].walkable;
  ob.downright = game["t_"+ob.downY+"_"+ob.rightX].walkable;
}
```

This function accepts 3 arguments: x/y position of the center point of object on stage (pixels) and name of the object. Wait, we already know x/y position of the object, we have it saved inside the char object, you may wonder. Thats true, but we have saved the CURRENT position of the char, here we are dealing with the position char WOULD BE if it would move.
First we calculate the tiles where character extends. Its center might be on one tile, but its left side might be on other tile, its highest point might be on third tile. Adding variable y with the height of hero and dividing it with height of tile, we will get the number of tile where objects lowest point (downY) will stand.
Last 4 lines use points we calculated to get the value of walkable property in each tile on the corners. For example upleft corner uses upY and leftX variables. As you can see all the points are also saved in the ob object and we can access them later when moving the char. I would again like to point out, how getMyCorners function will work with any moving object, not only the hero.

Move

When we know the types of tile each corner of character will be on, we can easily write movement for the char: if all the corners are walkable, then move, else dont move. More work is needed to place the hero right next to wall if the collision would happen. Our modified moveChar function to handle all 4 possible directions might look a bit confusing, but most of it is written 4 times over for each direction. Lets look at the function:

```
function moveChar(ob, dirx, diry) {
  getMyCorners (ob.x, ob.y+ob.speed*diry, ob);
  if (diry == -1) {
    if (ob.upleft and ob.upright) {
      ob.y += ob.speed*diry;
    } else {
      ob.y = ob.ytile*game.tileH+ob.height;
    }
  }
  if (diry == 1) {
    if (ob.downleft and ob.downright) {
      ob.y += ob.speed*diry;
    } else {
      ob.y = (ob.ytile+1)*game.tileH-ob.height;
    }
  }
  getMyCorners (ob.x+ob.speed*dirx, ob.y, ob);
  if (dirx == -1) {
    if (ob.downleft and ob.upleft) {
      ob.x += ob.speed*dirx;
    } else {
      ob.x = ob.xtile*game.tileW+ob.width;
    }
  }
  if (dirx == 1) {
    if (ob.upright and ob.downright) {
      ob.x += ob.speed*dirx;
    } else {
       ob.x = (ob.xtile+1)*game.tileW-ob.width;
    }
  }
  ob.clip._x = ob.x;
  ob.clip._y = ob.y;
  ob.clip.gotoAndStop(dirx+diry*2+3);
  ob.xtile = Math.floor(ob.clip._x/game.tileW);
  ob.ytile = Math.floor(ob.clip._y/game.tileH);
  return (true);
}
```

Like before, moveChar function gets object and directions from the detected keys. The line:

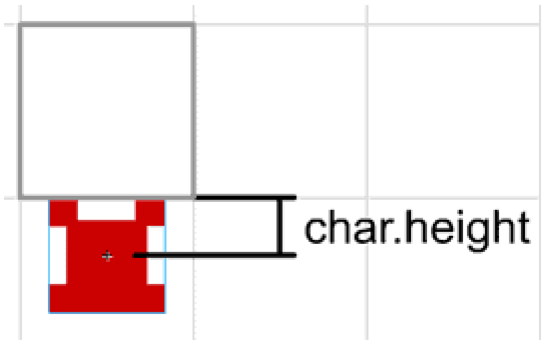
```
getMyCorners (ob.x, ob.y+ob.speed*diry, ob);
```

calculates the corner points for vertical movement (when diry is not equal to 0).

After we have calculated the corners, we can use the values from each tiles walkable property to check if hero can step there:

```
if (diry == -1) {
  if (ob.upleft and ob.upright) {
   ob.y += ob.speed*diry;
  } else {
   ob.y = ob.ytile*game.tileH+ob.height;
  }
}
```

This block of code works for up movement. When up arrow key was pressed, value for diry = -1. We use values of ob.upleft and ob.upright calculated in the getMyCorners function, if they are both "true" meaning both tiles are walkabale, we let char move like we did before adding speed*diry to char's y property.
But if one of corners happens to be inside the wall and so value of ob.upleft or ob.upright is "false", we place object near the wall. For char to be next to wall above it, its center point must be placed below the current tiles upper border by char.height.



ob.ytile*game.tileH would place character's center on the line between two tiles, we add height property of object to move it further down. Same way moveChar function goes through movements for down (diry == 1), left (dirx == -1) and right (dirx == 1).
Last lines place actual clip of character on the position calculated, make character show correct frame with animation and calculate new values for characters center point (xtile, ytile). Just like before, function returns "true".

You can download the source fla with all the code and movie set up here.

}

Open the door

How long can you stay in one room? How long will you look at same picture? Yep, we need more rooms to explore. That means a way to change the map, create new room from tiles and place the hero in correct position.

To make two rooms, we declare two maps:

```
myMap1 = [
[1, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 0, 0, 0, 0, 1],
[1, 0, 0, 0, 0, 1, 0, 1],
[1, 0, 0, 0, 0, 0, 0, 2],
[1, 1, 1, 1, 1, 1, 1, 1]
];

myMap2 = [
[1, 1, 1, 1, 1, 1, 1, 1],
[1, 0, 0, 0, 0, 0, 0, 1],
[1, 0, 1, 0, 0, 0, 0, 1],
[1, 0, 0, 0, 0, 1, 0, 1],
[3, 0, 0, 0, 0, 0, 0, 1],
[1, 1, 1, 1, 1, 1, 1, 1]
];
```

In the game object we also set the number of currently used map:

```
game={tileW:30, tileH:30, currentMap:1}
```

So, we start exploring in the myMap1. To call buildMap function, we calculate the map needed with _root["myMap"+game.currentMap] when currentMap is 1, we will get myMap1:

```
buildMap(_root["myMap"+game.currentMap])
```

Next we need new object to represent the doors:

```
game.Doors = function (newMap, newcharx, newchary) {
  this.newMap=newMap;
  this.newcharx=newcharx;
  this.newchary=newchary;
};
game.Doors.prototype.walkable = true;
game.Doors.prototype.frame = 3;
game.Doors.prototype.door = true;
game.Tile2 = function () { };
game.Tile2.prototype = new game.Doors(2, 1, 4);
game.Tile3 = function () { };
game.Tile3.prototype = new game.Doors(1, 6, 4);
```

As you probably already guessed, door tile can be stepped on, it shows frame 3 and it has new property "door" set to "true". We will use this property to determine if hero is standing on the door.
The doors use something called "inheritance", which may sound terrible, but is actually a good thing. The "doors" objects are all created using "Doors" template and all the tiles containing doors inherit all the properties of "door" object, for example they all are walkable and show frame 3.
Every door we crate must have following information: what number of map to show next, what is the new x and y position of our character. If you dont move the character to new position, then tiles will be changed, but hero stays at the same spot and this doesnt look correct. Word of caution about new coordinates for the hero. You should avoid placing hero in the next map also on the door tile as if he just stepped in from the wall. If new x/y position is also door, then as soon player moves, this next door in new map is detected and hero is sent back. Remember, place your hero next to door tile in new map!
When new door object is made, we pass 3 variables to it: newmap, newcharx, newchary. And new door object will have those values attached to its properties. So, when number 2 is set in the map array, we know its going to be made from Tile2 template and since Tile2 objects are created from Doors templates, it will have all the properties of Doors object. Tile2 object passes newmap=2 to the Doors template, so all the Tile2 objects will send hero to the map2. You can have more then 1 similar door in the game. You may want to put Tile2 type doors in several maps, and they all send hero to the map2.

More actions

In the moveChar function add this code to the end, just before returning true:

```
if (game["t_"+ob.ytile+"_"+ob.xtile].door and ob==_root.char) {
  changeMap (ob);
}
```

Here after we have moved the char (or some other moving object) we will check if the tile char is standing now, is door. As we dont want map to change when bullet or enemy steps on the door tile, we will also check if current object is the hero. We will use changeMap function for the map change:

```
function changeMap (ob) {
  var name = "t_"+ob.ytile+"_"+ob.xtile;
  game.currentMap = game[name].newMap;
  ob.ytile = game[name].newchary;
  ob.xtile = game[name].newcharx;
  ob.frame = ob.clip._currentframe;
  buildMap(_root["myMap"+game.currentMap]);
}
```

This code should be pretty obvious, get the values from the door tile and update variables of currentMap, ytile and xtile. The new property ob.frame will save current direction of the hero. Without this trick, our hero would start in frame1 every time he enters new map. You would also need to add line in the buildMap function after you have placed the char clip:

```
char.clip.gotoAndStop(char.frame);
```

Thats it, make some maps and play with doors.

You can download the source fla with all the code and movie set up here.

This chapter was rewritten in 13/dec/2003 to implement idea of door objects and getting rid of doors array.

Jumping

Lets turn our game from up view to the side view so we can add jumping. In this example we are looking to the game from side and our hero can walk left and right with arrow keys. He will also be able to jump with space key.

Jump basics

Any jump begins with push up. As you remember, up in the Flash stage means y coordinate will decrease. So, we calculate new_y=current_y-jumpspeed. If we do it only once, then hero moves up by jumpspeed and stops there. Yes, we have to continue calculating new y position as long the hero jumps, but we also have to change the jumpspeed or our hero will just fly away to the sky and never returns.
For changing jumpspeed we will declare new variable "gravity". Gravity is pulling hero back to the ground, thats down. In every step we add gravity to the jumpspeed: jumpspeed=jumpspeed+gravity. You can change the value of gravity, when you make less gravity, hero will fly higher (balloon-type), when you increase gravity, hero will pop down sooner (stone-type). As we have lotsa objects and character is also an object, you can actually give different gravity to different objects.
Lets look at one example. Jumpspeed starting value is -10 and gravity is 2. First step, hero is moved up by 10 pixels and jumpspeed gets value of 8. Next step, moving up by 8, speed=6. After couple of steps, jumpspeed gets value of 0, meaning hero wont move up anymore. Next step jumpspeed has positive value and hero starts to move down.
But what to do when hero hits the solid tile (wall) while jumping. If hero is jumping up and hits wall above, we will set the jumpspeed to 0 and hero starts to fall down. If hero hits solid tile below, then he has landed and jump is over.
In tile based game its always important not to let speed get bigger then size of tile. If hero has too high speed, he will not check the next tile, and might move through walls. While some magicians can move through walls, in normal game thats just a bug.
As you can see, jumping wont affect horisontal movement. Its still done exactly same way as before. We only have to check after moving hero left/right, if he has walked off the solid tile below and might start to fall down.

Be my hero

We add some properties to our character:

```
char={xtile:2, ytile:1, speed:4, jumpstart:-18, gravity:2, jump:false};
```

The property speed will set the speed when moving left or right. Jumpstart is the starting value of jumpspeed. Gravity will pull hero back to the ground and property "jump" will be used to check if hero is currently jumping (then jump=true) or standing/walking/running/sitting/fighting on solid ground (jump=false).
Next line to change is in the buildmap function, when we set the starting position of hero. In examples before we did put hero in the center of tile, but that will look weird since hero will always start falling down after new map is created. We will make hero stand in the bottom of its starting tile (dont forget to move that line after the line declaring char.height):

```
char.y = ((char.ytile+1)*game.tileH)-char.height;
```

Functions changeMap and getMyCorners doesnt need any change.

Give me the Wings

Lets starts with detectKeys function. We need to add code to check for space key and we can remove up/down arrow keys.

```
function detectKeys() {
  var ob = _root.char;
  var keyPressed = false;
  if (Key.isDown(Key.SPACE) and !ob.jump) {
    ob.jump = true;
    ob.jumpspeed = ob.jumpstart;
  }
  if (Key.isDown(Key.RIGHT)) {
    keyPressed=_root.moveChar(ob, 1, 0);
  } else if (Key.isDown(Key.LEFT)) {
    keyPressed=_root.moveChar(ob, -1, 0);
  }
  if (ob.jump) {
    keyPressed=_root.jump(ob);
  }
  if (!keyPressed) {
    ob.clip.char.gotoAndStop(1);
  } else {
    ob.clip.char.play();
  }
}
```

Note how we wont let character to jump again while he is already jumping (!ob.jump). Space key will be counted for starting new jump only if variable jump is false. But if space key is pressed and hero was not yet jumping, we will set variable jump to true and give hero starting speed.
After left/right arrow keys we will check if variable jump is true and if it is, we will call new "jump" function (function "jump" is not same thing as variable "jump", bad choice of names from me, sorry). This function will be called every step until variable jump is true, so our hero continues jumping even after the space key is released.

Jump function will add gravity to the current jumpspeed. It will then check if jumping speed has grown too big and if it is, will set the speed equal to the tile size. Last lines will call moveChar function:

```
function jump (ob) {
  ob.jumpspeed = ob.jumpspeed+ob.gravity;
  if (ob.jumpspeed>game.tileH) {
    ob.jumpspeed = game.tileH;
  }
  if (ob.jumpspeed<0) {
    moveChar(ob, 0, -1, -1);
  } else if (ob.jumpspeed>0) {
    moveChar(ob, 0, 1, 1);
  }
  return (true);
}
```

We also need to change moveChar function. In the previous chapters we used ob.speed to change objects position, but now we also have the jumpspeed, which is changing in every step. Change the start of moveChar function:

```
function moveChar(ob, dirx, diry, jump) {
  if (Math.abs(jump)==1) {
    speed=ob.jumpspeed*jump;
  } else {
    speed=ob.speed;
  }
```

The jump argument will be 1 or -1 only if the moveChar function is called from jump function and then variable speed will have value from jumpspeed. When it is called from left/right keys detection variable speed will be equal to ob.speed. Change lines in the moveChar functions that used previously variable ob.speed to "speed" so correct value is used.
In the going up code we change jumpspeed to 0 if we hit the wall above:

```
ob.y = ob.ytile*game.tileH+ob.height;
ob.jumpspeed = 0;
```

And in the down part we set jump to false if we detect wall below:

```
ob.y = (ob.ytile+1)*game.tileH-ob.height;
ob.jump = false;
```

In the left and right movement, we add line to check for the situation, when hero walks over the edge of platform and should start falling down:

```
ob.x += speed*dirx;
fall (ob);
```

So, the last new function we will need, is fall:

```
function fall (ob) {
  if (!ob.jump) {
    getMyCorners (ob.x, ob.y+1, ob);
    if (ob.downleft and ob.downright) {
      ob.jumpspeed = 0;
      ob.jump = true;
    }
  }
}
```

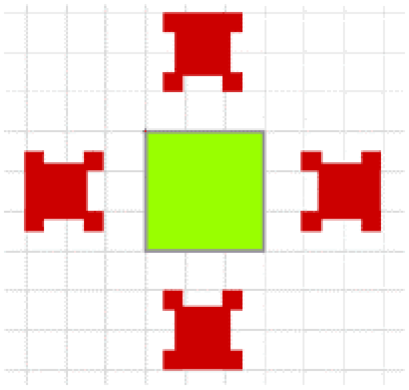
We cant start falling down if we already are jumping, so first we check if variable jump is false (hero currently stands). If we are standing, we will call the getMyCorners function to get corner points of hero. We use coordinate ob.y+1 to check if point 1 pixel lower then characters current position is walkable. If both corner points below hero (downleft and downright) are walkable (have value of true) that will mean our dear hero is standing on the air.
To correct the "thou-shall-not-stand-in-the-air" situation, we will force hero to jump by setting variable jump=true. But unlike when space key was pressed, we will set the starting speed of jump to 0, so hero will start to fall down.

You can download the source fla with all the code and movie set up here.

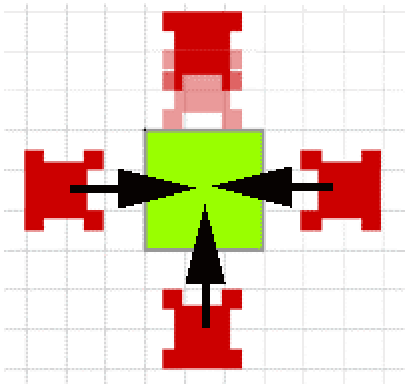
Clouds

So far we have hit our hero against the wall. And that was fun. But the brick wall is not the only type of wall we could make. Many games have "cloud" type of walls, they allow hero to move through them from left or right and also jump up, but when hero is falling down, he will stand on it.

So, did you notice the difference? Lets look at the pictures just in case. Here we have normal brick wall type of tile. Hero cant enter that tile from any direction.



But this is cloud. Hero can enter the tile from any other direction, except from above. If hero is dumb enough to try entering from above, we place him just above to the cloud:



We will first set up some tiles with the property "cloud". If the tile has "cloud" set to true, it is obviously cloud type. Declare some prototypes:

```
game.Tile4= function () {};
game.Tile4.prototype.walkable=true;
game.Tile4.prototype.cloud=true;
game.Tile4.prototype.frame=4;
```

The tile has "walkable" property set to true, so yes, hero can walk into it. In order to make hero stand on it, when falling, we create new function.

```
function checkIfOnCloud (ob) {
          var leftcloud = game["t_"+ob.downY+"_"+ob.leftX].cloud;
          var rightcloud = game["t_"+ob.downY+"_"+ob.rightX].cloud;
          if ((leftcloud or rightcloud) and ob.ytile != ob.downY)
                    return(true);
          } else {
                    return(false);
          }
}
```

We use the bottom right and left corner points to check if one of those is placed on the tile, which cloud property is true. If one of them actually is on cloud, we return true. If no cloud is found, return value is false.
Now we need to call this function from two places: from moveChar function when checking for going down and from fall function when checking if hero still stands on solid tiles or should he start to fall.
Locate this line in the moveChar function right after if (diry == 1):

```
if (ob.downleft and ob.downright) {
```

Add check for cloud:

```
if (ob.downleft and ob.downright and !checkIfOnCloud (ob)) {
```

Same way in the fall function replace line:

```
if (ob.downleft and ob.downright) {
```

with

```
if (ob.downleft and ob.downright and !checkIfOnCloud (ob)) {
```

So, before we used to check only if both left/right bottom points are on tile which has walkable property set to true (we calculated values for ob.downleft and ob.downright in the getMyCorners function). Now we only add check if those points are not inside cloud tile.
Enjoy the clouds. And sun. And stars :)

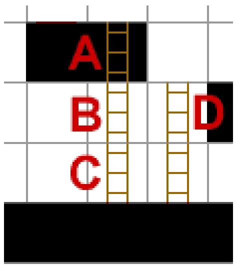
You can download the source fla with all the code and movie set up here.

Bugfix at 22.06.2004. Big thanks to Carlos for spotting the bug in the cloud detection code. I have updated the tutorials and flas, but if you happen to see code somewhere where clouds are detected in the checkIfOnCloud function:
if (leftcloud or rightcloud) {
instead of
if ((leftcloud or rightcloud) and ob.ytile != ob.downY) {
please fix them.

Ladders

Ladders are common form of movement in platform games. Hero can use ladders to climb up or down (I bet you didnt know that). We will make character climb when up or down arrow key is pressed and character stands near the ladder.

While ladders seam to be easy enough, there are some things to consider. First, what kind of ladders are there?



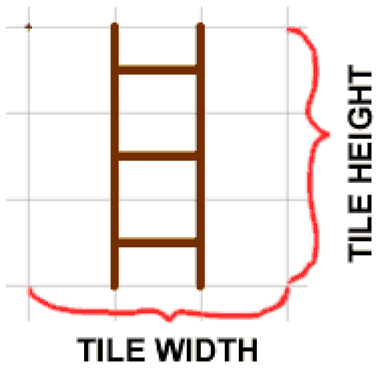
In the picture, there are 4 different types of ladders. In tile A ladder is inside wall tile, which normally is not walkable. What can hero do in tile A? He can climb up and down, but he shouldnt be able to walk left or right or he will be stuck in the wall. Ask anyone, who has been stuck in the wall and they all say it doesnt feel good.
In tile B ladder tile itself is walkable and tile above it also have ladder, so hero should be able to climb up and down. Hero can also move left or right, but when he does, he should fall down after leaving the ladder.
In tile C there isnt ladder below and hero shouldnt climb down, he can only climb up or walk left/right.
Tile D is not available in all games. Some think thats just a bad level design, ladder doesnt lead anywhere, it ends in the air. Should hero be able to climb above it and stand on the ladder? Can he walk then to the right on the solid tile next to ladder?
Those are just couple of examples, there are many more possible types of ladders, but I hope you can see how important it is to have strict definition before attempting to write code. As games are not all similar, then something fitting perfectly in one game, is waste of time, energy and world peace in other game.

The rules

Lets write down our rules for ladders and movement for hero:
1. Hero can climb on the ladder using up and down arrow keys
2. Hero can climb up if there is ladder at his current up or down center points
3. Hero can climb down if the tile his down center point ends up, has ladder
4. Hero can move left/right from the ladder if none of his corner points will end up in the wall
5. Hero cant jump from the ladder
That should do it.

One ladder, please

We will use separate movie clip with ladder graphics that will be attached in the tile when tile has ladder. That way we dont have to draw different graphics for every ladder on different backgrounds. Make sure your ladder movie clip has "Export this movie" checked and it is linked as "ladder".



In the ladder movie clip draw ladder graphics same height as tile and place them in the center of tile horisontally.
As with every other tile, we will declare new type of tile for ladders:

```
game.Tile4= function () {};
game.Tile4.prototype.walkable=false;
game.Tile4.prototype.frame=2;
game.Tile4.prototype.ladder=true;
game.Tile4.prototype.item="ladder";
game.Tile5= function () {};
game.Tile5.prototype.walkable=true;
game.Tile5.prototype.frame=1;
game.Tile5.prototype.ladder=true;
game.Tile5.prototype.item="ladder";
```

Those two types of ladder have different frame number to show, but they both have property "ladder" set to true (we will use it to check if hero is anywhere near the ladder) and they both have property "item" equal to "ladder" (we will use this to attach ladder graphics to the tile).
Attach the ladder movie to the tile in the buildMap function after sending tile to correct frame:

```
game.clip[name].gotoAndStop(game[name].frame);
if (game[name].item != undefined) {
  game.clip[name].attachMovie(game[name].item, "item", 1);
}
```

This code checks if property "item" in the current tile has non-empty value. If "item" has value, then we attach movie clip linked with the name as value of "item" property to the current tile and it will have instance name "item". You can attach any other items same way, just dont try to put many items in the same tile.

For not typing same code twice, lets move end of moveChar function and make separate function of it naming this new function updateChar. moveChar function will end with:

```
updateChar (ob, dirx, diry);
return (true);
```

and updateChar function will have:

```
function updateChar (ob, dirx, diry) {
  ob.clip._x = ob.x;
  ob.clip._y = ob.y;
  ob.clip.gotoAndStop(dirx+diry*2+3);
  ob.xtile = Math.floor(ob.clip._x/game.tileW);
  ob.ytile = Math.floor(ob.clip._y/game.tileH);
  if (game["t_"+ob.ytile+"_"+ob.xtile].door and ob==_root.char) {
    changeMap (ob);
  }
}
```

At the fall function add

```
ob.climb = false;
```

Modify detectKeys function for the arrow keys:

```
if (Key.isDown(Key.RIGHT)) {
  getMyCorners (ob.x-ob.speed, ob.y, ob);
  if (!ob.climb or ob.downleft and ob.upleft and ob.upright and ob.downright) {
    keyPressed=_root.moveChar(ob, 1, 0);
  }
} else if (Key.isDown(Key.LEFT)) {
  getMyCorners (ob.x-ob.speed, ob.y, ob);
  if (!ob.climb or ob.downleft and ob.upleft and ob.upright and ob.downright) {
    keyPressed=_root.moveChar(ob, -1, 0);
  }
} else if (Key.isDown(Key.UP)) {
  if (!ob.jump and checkUpLadder (ob)) {
    keyPressed=_root.climb(ob, -1);
  }
} else if (Key.isDown(Key.DOWN)) {
  if (!ob.jump and checkDownLadder (ob)) {
    keyPressed=_root.climb(ob, 1);
  }}
```

After we have detected left or right arrow key, we will check if hero is not climbing (!ob.climb) or in case he is climbing, we will check that none of his corner points will be in the wall.
For up and down arrow keys we first check if hero is not jumping (!ob.jump) and the conditions for climbing are met using two new functions: checkUpLadder and checkDownLadder. If everything is fine, we call new function "climb" to move our hero.

Climbing functions

We will make 3 new functions for climbing, 1 to check if it is fine to climb up, 1 to check if we can climb down and last function to move character.

```
function checkUpLadder (ob) {
  var downY = Math.floor((ob.y+ob.height-1)/game.tileH);
  var upY = Math.floor((ob.y-ob.height)/game.tileH);
  var upLadder = game["t_"+upY+"_"+ob.xtile].ladder;
  var downLadder = game["t_"+downY+"_"+ob.xtile].ladder;
  if (upLadder or downLadder) {
    return (true);
  } else {
    fall (ob);
  }
}
```

This code calculates first up and down center points of our hero. If one of the tiles in those points has ladder property set to true, we can climb up. If there isnt ladder up or down, we check if hero should fall.

```
function checkDownLadder (ob) {
  var downY = Math.floor((ob.speed+ob.y+ob.height)/game.tileH);
  var downLadder = game["t_"+downY+"_"+ob.xtile].ladder;
  if (downLadder) {
    return (true);
  } else {
    fall (ob);
  }
}
```

To check for climbing down, we need "ladder" property of the tile below hero. But unlike climbing up, we have to look for the tile, where hero will be standing after he moves (ob.speed+ob.y+ob.height).

```
function climb (ob, diry) {
  ob.climb=true;
  ob.jump=false;
  ob.y += ob.speed*diry;
  ob.x = (ob.xtile *game.tileW)+game.tileW/2;
  updateChar (ob, 0, diry);
  return (true);
}
```

In the climb function we first set the flags climb to true and jump to false. Then we calculate new y position for the hero. Next we will position the hero horisontally in the center of ladder:

```
ob.x = (ob.xtile *game.tileW)+game.tileW/2;
```

Hero can start climbing as long his center is in the tile with ladder, but it would look weird if he would climb in the left or right side of the ladder.
Last we update the actual position of character using same updateChar function.

You can download the source fla with all the code and movie set up here.

Stupid enemy

We have worked hard to get our hero moving, but he has no challenge. We need something more. No, we dont need food and drinks and pretty girls, what we need, is some enemies. Enemies are like salt in the soup, without it everything tastes lame. Good games have smart enemies, but we will start with simple dumb enemies. All they do, is walk back and worth and check if they have touched the hero.

So far we have had two types of objects: hero and tiles. Hero is moved by the player, tiles wont move. The enemies will be like hero, only player cant move them, we will give them brains for moving. We will make two different enemies, first will walk up and down, second will walk left and right. They both will turn around when they hit the wall.
Before you start to think your game needs very-very complex enemies, think again. Many games dont actually use smart enemies, or if they do, not all of them are too bright. You dont have unlimited resources with Flash, so your 100 smart enemies using complex A* pathfinding will slow down your game too much. If you can, make some enemies dumb and some smarter, in the end player might not even notice the difference. Beside, we all know how we like to be smarter then others, so let the player feel that joy too :)

Prepare the Enemy

Create enemy movie clips same way you made hero (look at the tutorial "The Hero" if you forgot). They should have 4 keyframes with left/up/down/right animations. They should also be set exported as "enemy1" and "enemy2". Now lets add enemies array:

```
myEnemies = [
[0],
[[1, 6, 1]],
[[2, 1, 3]]
];
```

So, for the map1 we have declared 1 array representing 1 enemy. The numbers give us the type of enemy (woohooo! we have more then one type of dumb enemies) and its starting position. When map is built, the enemy is placed on the tile x=6, y=1. Same way we have 1 enemy in map2, but thats different type. You can add more enemies into one map, but you have to place them into walkable tiles, not inside the walls.
Lets declare some templates for enemies:

```
game.Enemyp1= function () {};
game.Enemyp1.prototype.xMove=0;
game.Enemyp1.prototype.yMove=1;
game.Enemyp1.prototype.speed=2;
game.Enemyp2= function () {};
game.Enemyp2.prototype.xMove=1;
game.Enemyp2.prototype.yMove=0;
game.Enemyp2.prototype.speed=2;
```

They look similar, but they behave different way. Enemyp1 will walk vertically because its property yMove is 1, but Enemyp2 will walk left and right. You can set enemies xMove/yMove properties into 1 or -1 or 0. Those are same values we pass to the

moveChar function from detect keys function. Please avoid setting both xMove/yMove to non-zero unless you want your enemies to move diagonally.
If you make both xMove and yMove equal to 0, you will have non-moving enemy. Thats not much fun, but you never know what you might need.
The speed property determines how fast enemy will move. Different enemies can have different speeds too.

Place the Enemy

In the buildMap function add after doors and before char following piece of code:

```
var enemies = myEnemies[game.currentMap];
game.currentEnemies = enemies.length;
for (var i = 0; i<game.currentEnemies; ++i) {
  var name = "enemy"+i;
  game[name]= new game["Enemyp"+enemies[i][0]];
  game.clip.attachMovie("enemy"+enemies[i][0], name, 10001+i);
  game[name].clip=game.clip[name];
  game[name].xtile = enemies[i][1];
  game[name].ytile = enemies[i][2];
  game[name].width = game.clip[name]._width/2;
  game[name].height = game.clip[name]._height/2;
  game[name].x = (game[name].xtile *game.tileW)+game.tileW/2;
  game[name].y = (game[name].ytile *game.tileH)+game.tileH/2;
  game[name].clip._x = game[name].x;
  game[name].clip._y = game[name].y;
}
```

Whats happening here? First we get enemies array for current map. Next we will set variable currentEnemies to the length of enemies array so we can always know how many enemies we have created. Then we loop through the enemies array (remember, we currently used only 1 enemy in each map, but there can be more).
Variable name will have the name of our new enemy, so they are named "enemy0", "enemy1", "enemy2"... Then we make new enemy object from the templates we declared earlier:

```
game[name]= new game["Enemy"+enemies[i][0]];
```

from the enemies array we get the number in first position of first enemy. In our example there is number 1, so, new enemy is created using Enemy1 template. In map2 the enemy is made using Enemy2 template because in myEnemies array there is number 2.
Next lines will add starting coordinates, width and height to the enemy object. Then its position is calculated and enemy is placed into correct position.
But, you may cry out loud, but he doesnt move! No worry, we will make him move.

Move the Enemy

Like most people, most enemies need brains. And so, lets write enemyBrain function:

```
function enemyBrain () {
for (var i = 0; i<game.currentEnemies; ++i) {
  var name = "enemy"+i;
  var ob = game[name];
  getMyCorners (ob.x+ob.speed*ob.xMove, ob.y+ob.speed*ob.yMove, ob);
  if (ob.downleft and ob.upleft and ob.downright and ob.upright) {
    moveChar(ob, ob.xMove, ob.yMove);
  } else {
    ob.xMove = -ob.xMove;
    ob.yMove = -ob.yMove;
  }
  var xdist = ob.x - char.x;
  var ydist = ob.y - char.y;
  if (Math.sqrt(xdist*xdist+ydist*ydist) < ob.width+char.width) {
    removeMovieClip(_root.tiles);
    _root.gotoAndPlay(1);
  }
}
}
```

As you can see, we again loop through all the enemies using variable game.currentEnemies. ob reffers in every step to the current enemy object. When i=0, ob=enemy0 and so on.
We then call getMyCorners function to check if enemy will step into wall. If he wont go into wall, all the variables upleft, downleft, upright and downright will be true saying those tiles are walkable. Its safe to call moveChar function using xMove and yMove properties (which are -1, 0 or 1) same way we called moveChar from key detection to move char, but since we pass enemy object to moveChar, the enemy will be moved. Told you we can reuse same function to move many objects :)
In case enemy would hit the wall tile, we reverse the xMove and yMove so enemy will start to move into opposite direction. If yMove was 1, it will become -1, but if it was 0, it will remain 0.
Last part of brains checks if enemy is close enough to the hero and if they are too close, game ends. Of course, game doesnt usually end so easily, you would reduce heros life or do other tricks, but thats up to you. We use simple equation called "Pythagoras theorem" to calculate the distance between enemy and hero. If you really-really want pixel perfect collision, you could create complex hitTest here, but I dont see reason for it. Dont get too close or you die!
To call this function in every step, add line in the end of detectKeys function:

```
_root.enemyBrain();
```

Thats it, you can add simple enemies to be avoided. Next chapter we make our enemies behave more like human being, that is, run around until hitting the wall, then change the direction and run more.

You can download the source fla with all the code and movie set up here.

Big thanks to kuRTko for spotting annoying bug in the enemies code. I have updated the tutorials and flas, but if you happen to see code somewhere where enemies are declared as:
*game.**Enemy1**= function ()*
*instead of game.**Enemyp1**= function (), please fix them. Note, the bug was that enemy proprotypes used same name "enemy+number" as actual enemy objects. The prototypes should be named as "enemyp+number". Dont forget to check the line in the buildmap function too, it should be after update:*
game[name]= new game["Enemyp"+enemies[i][0]];

Enemy on platform

If you want to have enemies in the side view jumping game, you only need to change couple of lines. The enemy will walk on the platform and detect the end of platform. In the end enemy will turn around and walk back. This requires enemy to check for platform **below** the next tile he would be on:

```
getMyCorners (ob.x+ob.speed*ob.xMove, ob.y+ob.speed*ob.yMove+1, ob);
if (!ob.downleft and !ob.downright) {
```

Important number to notice here, is 1 in the ob.y+ob.speed*ob.yMove+1. That will check for wall below next tile. Also note how if statement will be tru only if both upleft and upright corner are walls, if one of them is walkable tile, enemy would walk into air.
You can download the source fla with all the code and movie set up here.

Teaching the enemy some tricks

What if our enemy could change direction, not only reverse the direction.

Lets modify enemyBrain function. When we last time just reversed ob.xMove and ob.yMove, now we will choose randomly new direction to move:

```
} else {
        ob.xMove = random(3)-1;
        if (ob.xMove) {
                ob.yMove = 0;
        } else {
                ob.yMove = random(2)*2-1;
        }
}
```

When enemy would hit the wall, xMove will get random value. random(2) will have value 0 or 1. If xMove was 0, we set yMove randomly to 1 or -1.
random(2) is 0 or 1.
random(2)*2 is 0 or 2.
random(2)*2-1 is -1 or 1.
In case xMove had value 1, w
e now set yMove to 0 and get random 1 or -1 for xMove.
You can download the source fla with all the code and movie set up here.
Thats much nicer, but if we want to make enemy better, we should avoid reversing the last direction.

Write code:

```
} else {
  if (ob.xMove == 0) {
    ob.xMove = random(2)*2-1;
    ob.yMove = 0;
    getMyCorners (ob.x+ob.speed*ob.xMove, ob.y+ob.speed*ob.yMove, ob);
    if (!ob.downleft or !ob.upleft or !ob.downright or !ob.upright) {
      ob.xMove = -ob.xMove;
    }
  } else {
    ob.xMove = 0;
    ob.yMove = random(2)*2-1;
    getMyCorners (ob.x+ob.speed*ob.xMove, ob.y+ob.speed*ob.yMove, ob);
    if (!ob.downleft or !ob.upleft or !ob.downright or !ob.upright) {
      ob.yMove = -ob.yMove;
    }
  }
}
```

This time we first check the current direction. If for example we moved vertically (xMove==0) then we choose randomly 1 or -1 for xMove and set yMove to 0. But if enemy moves into corner, his new direction might send him again into wall. Thats why we get the corner points with new direction and if we detect wall, we reverse the new direction.
You can download the source fla with all the code and movie set up here.
Ok, enemy moves better since player cant predict where enemy is going to step next. But as you can notice, enemy keeps hugging the walls, he always moves until hitting the wall, then and only then choose another direction. If your map contains large empty areas, player can be sure enemy never comes there. Good example is second room, until hero stays in the center, enemy will never catch him.
We will add a chance for enemy to change direction even when he doesnt hit the wall.

I havent figured good description for ability to change direction while walking, so lets add each enemy new property called "turning":

```
game.Enemy1.prototype.turning=5;
game.Enemy2.prototype.turning=5;
```

Turning will represent the chance to randomly change direction in each step. Value of 0 will mean enemy never changes direction, value 100 will make him choose new direction in each step (thats funny, you should try that out).
And to make enemy choose new direction, add to the if statement:

```
if (ob.downleft and ob.upleft and ob.downright
 and ob.upright and random(100)>ob.turning) {
```

In case random(100) will have value less then value of ob.turning, we will choose new direction even when we could continue same way.
You can download the source fla with all the code and movie set up here.

Update at 23.06.2004. Thanks to qhwa for suggesting simpler method of choosing new random direction.

Shoot him

You can kill the enemies in many ways. You can use sword, gun or words (extremely powerful weapon that takes long time to master). Lets see how we can shoot the enemy.

When I say "bullet", I mean any object that is flying from the hero looking to kill baddies. It can be cannon ball, arrow, ice cream, penguin etc.
First, we again should think of, what is shooting suppose to do and how will our bullets behave. When key is pressed (SHIFT key), bullet object and movie clip are created on the correct side of the hero. The bullet should start moving straight in the direction hero is facing. If bullet hits the wall or enemy, it is destroyed. If it hits enemy, then enemy is also destroyed.
The speed of bullet should be higher then speed of hero unless you wish to give hero some way to stop the moving bullets. Usually the dumb enemies dont see bullets coming, but you could create enemies that try to avoid bullets. You could also make enemies shooting at the hero.

Prepare to Shoot

Draw bullet movie clip and make sure its set to be exported as "bullet" so we can attach it to the stage. The graphics inside bullet mc should be aligned to the center.
Lets declare bullet object:

```
game.Bullet= function () {};
game.Bullet.prototype.speed=5;
game.Bullet.prototype.dirx=0;
game.Bullet.prototype.diry=-1;
game.Bullet.prototype.width=2;
game.Bullet.prototype.height=2;
```

Bullets will move with the speed of 5 pixels per tick. They have width/height of 2 pixels, thats enough damage to the enemy.
Now the properties dirx/diry will make bullet moving. They are same things we used in the moveChar function. If dirx=1, bullet will move right, diry=-1 makes it move up. We will actually take the values of dirx/diry from the char object, but at the start of the game, when char hasnt been moved yet, but player wants to shoot, we will use default values to make bullet move up.
Add two new propeties to game object:

```
game={tileW:30, tileH:30, currentMap:1, bulletcounter:0};
game.bullets= new Array();
```

The bulletcounter will be counting the number of bullets we have used and helps to give each new bullet new name. First bullet we shoot in the game, will be named bullet0, then bullet1 and so all the way up to bullet100. Then we reset the bulletcounter. We could in theory let it raise forever, but who knows what kind of nasty things can happen then.
game.bullets in an array which will hold reference to all the bullets we have flying around. At the beginning its empty array.
For the char object add shootspeed property for making him stop between the shots:

```
char={xtile:2, ytile:1, speed:4, shootspeed:1000};
```

Higher shootspeed value makes the char shoot slower and lower value faster. Value of 1000 is exactly 1 second between shots.
For the enemies to die, we have to remove them from the game. Change the enemies creation part in the buildMap function:

```
game.currentEnemies = [];
for (var i = 0; i<enemies.length; ++i) {
  var name = "enemy"+i;
  game[name]= new game["Enemy"+enemies[i][0]];
  game[name].id=i;
  game.currentEnemies.push(game[name]);
```

And in the enemyBrain function replace the line:

```
var name = "enemy"+i;
```

with

```
var name = "enemy"+game.currentEnemies[i].id;
```

We will use currentEnemies to hold the names of all the enemies on the stage. When enemy is killed, we will remove him from the array. The new property "id" helps us to destroy the enemy object placed in the enemies array.
In the detectKeys function add code after checking for arrow keys:

```
if (Key.isDown(Key.SHIFT) and getTimer()>ob.lastshot+ob.shootspeed) {
  _root.shoot(ob);
}
```

If SHIFT key is pressed and enough time has passed for hero to shoot again, we will call shoot function.
In the beginning of moveChar function add two lines to save the direction of current object:

```
ob.dirx=dirx;
ob.diry=diry;
```

We will use those to determine which way our bullets will move.

Shoot

For creating the bullets and giving bullets all the data they need for successful deadly flights, we will use new function called "shoot":

```
function shoot (ob) {
 ob.lastshot=getTimer();
 game.bulletcounter++;
 if (game.bulletcounter>100) {
  game.bulletcounter=0;
 }
 var name = "bullet"+game.bulletcounter;
 game[name]= new game.Bullet;
 game[name].id=game.bulletcounter;
 game.bullets.push(game[name]);
 if (ob.dirx or ob.diry) {
  game[name].dirx= ob.dirx;
  game[name].diry= ob.diry;
 }
 game[name].xtile= ob.xtile;
 game[name].ytile= ob.ytile;
 game.clip.attachMovie("bullet", name, 10100+game.bulletcounter);
 game[name].clip=game.clip[name];
 game[name].x = (ob.x+game[name].dirx*ob.width);
 game[name].y = (ob.y+game[name].diry*ob.height);
 game.clip[name]._x = game[name].x;
 game.clip[name]._y = game[name].y;
}
```

First we have passed object to the function. In this case it is char object as shoot was called from detectKeys function, but if bullet would be shot by enemy, enemy object would be passed.
We use getTimer() function to save the time this shot was fired in the lastshot property.
Next we add 1 to the game.bulletcounter property and if it is >100 we set bulletcounter back to 0.
Now we create new bullet using bulletcounter to give new bullet unique name and we also save this number into bullet object. We will add reference to the new bullet to the game.bullets array.
The if condition with dirx/diry checks if char has been moved. If player hasnt moved the char yet, the char object doesnt have dirx/diry properties and we will have the default dirx/diry from the bullet template. However, if the char has been moved, we set bullets dirx/diry equal to the chars.
To make bullet appear by the char, we need to save chars position. ob.xtile and ob.ytile are copied to the bullet.

Last part of the code creates new movie clip for the bullet, calculates its position on the screen and sets it there. Interesting part might be how exactly is bullets position found:

```
game[name].x = (ob.x+game[name].dirx*ob.width);
```

First we take chars position (ob.x), thats where the center of char is. As bullets usually dont come out from the center of hero, we add width of char to that. But since width is multiplied by the value of dirx, the bullet will be placed on the left from char (dirx=-1), right from char (dirx=1) or in the center (dirx=0). Uh, you may wonder, not in the center? But yes, dirx can be 0 only if diry is either 1 or -1, so the bullet ends up above or below char.

Kill!

In the end of detectKeys function add line to call second new function that will move the bullet and look if we have killed something:

```
_root.moveBullets();
```

And the function itself:

```
function moveBullets () {
 for (var i = 0; i<game.bullets.length; ++i) {
  var ob=game.bullets[i];
  getMyCorners (ob.x+ob.speed*ob.dirx, ob.y+ob.speed*ob.diry, ob);
  if (ob.downleft and ob.upleft and ob.downright and ob.upright) {
   moveChar(ob, ob.dirx, ob.diry);
  } else {
   ob.clip.removeMovieClip();
   delete game["bullet"+game.bullets[i].id];
   game.bullets.splice(i,1);
  }
  for (var j = 0; j<game.currentEnemies.length; ++j) {
   var name = "enemy"+game.currentEnemies[j].id;
   var obenemy = game[name];
   var xdist = ob.x - obenemy.x;
   var ydist = ob.y - obenemy.y;
   if (Math.sqrt(xdist*xdist+ydist*ydist) < ob.width+obenemy.width) {
    obenemy.clip.removeMovieClip();
    delete game["enemy"+game.currentEnemies[j].id];
    game.currentEnemies.splice(j,1);
    ob.clip.removeMovieClip();
    delete game["bullet"+game.bullets[i].id];
    game.bullets.splice(i,1);
   }
  }}}
```

This function loops through all the bullets in the bullets array.
Using getMyCorners we will know if the current bullet will hit the wall or not. If none of its corners hit the wall, we will move the bullet with moveChar function.
Now if the bullet would hit wall, we have to destroy it. There are 3 things we need to do in order to get rid of bullet:
+ remove the bullet mc (using removeMovieClip)
+ remove bullet object (using delete function)
+ remove current bullet from bullets array.
We could leave the bullet object in the game, since its without movie clip you cant see it and when removed from bullets array it wont be accessed again, but then after 100 bullets the game will look like scrapyard. Its not nice to leave trash behind you.
When we have successfully moved the bullet and it hasnt hit any walls yet, we start to check if it has hit some enemy. Looping through all the enemies in the currentEnemies array, we calculate the distance between current bullet and current enemy. If they get too close, we destroy both of them.
If you want enemies to die forever, meaning them not to resurrect after leaving the map and returning, place 1 line after the distance calculation:
myEnemies[game.currentMap][obenemy.id]=0;
You can do several things to make shooting more inetresting:
+ limit the amount of available bullets. You could set variable in the beginning and every time bullet is shot, reduce it by 1, only allowing shooting if its >0.
+ limit only 1 bullet on stage. You could do this by checking game.bullets and if its length is >0 do not allow shooting.
+ make enemies shoot bullets too. It would be easy to make them shoot at random times in random directions, same way they change the movement.
+ make different weapons to choose from. You could declare several bullet templates and assign different damage values to each, so you can get better weapons and kill enemies faster.
Happy shooting :)
You can download the source fla with all the code and movie set up here.
Shooting with jumping side view.

Bugfix at 20.12.2003. In the moveBullets function the line:
var name = "enemy"+j;
should be
var name = "enemy"+game.currentEnemies[j].id;
If you see old version anywhere, please fix it. Thanks to Christian and Travis for finding the bug.
Bugfix at 1.03.2004. In the enemyBrain function the line:
var name = "enemy"+i;
should be
var name = "enemy"+game.currentEnemies[i].id;
Stupid me :(Thanks to Ian for pointing it out.

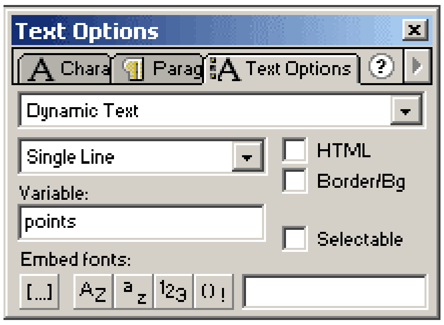
Getting items

In this chapter we will look how to make our hero to pick up some items from the ground. You know the stuff: crystals, coins, dead spiders, healing potions, ammunition.

Items are different in what they actually do. Some items increase your score, some increase your health or give you more bullets. In this example all items do only one thing - they give you more points. Its up to you to create other kind of items.
We will start with the movie from the chapter 6 Open the door so we dont have too much code to make things hard to understand.
First draw your items movie clip. Place graphics of different items in the frames inside it. Like hero and enemies, all items should be aligned in the center. In the first frame add "Stop" action to prevent mc from playing all the lovely graphics. Make sure this mc linkage is set to "Export this symbol" and its identifier is "items". Having items in the separate movie clip allows us to place same item on any background tile without the need to draw tiles again.

Whats your point?

To show the collected points draw dynamic text box on the screen. Position this text box outside from tiles area so tiles wont cover it. The dynamic text box should show variable "points":


Text Options
Dynamic Text
Single Line
HTML
Border/Bg
Variable: points
Selectable
Embed fonts:

The variable "points" will need to remember the current points even when we change maps. We can safely attach it to the game object. In declaring game object add property points and set it to 0 (most games start with 0 points).

```
game = {tileW:30, tileH:30, currentMap:1, points:0};
```

Something to pick up

Like with everything else, first we will declare the items objects and create array to hold positions of items in different maps:

```
myItems = [
[0],
[[1,1,1],[1,1,2],[2,1,3]],
[[2,1,3],[2,6,3],[1,5,4]]
];
game.Item1= function () {};
game.Item1.prototype.points=1;
game.Item2= function () {};
game.Item2.prototype.points=10;
```

myItems array is built same way as enemies array (chapter 9). It has array for each of the maps. We havent used map0 so first array is empty. For map1 we have set 3 items: [1,1,1],[1,1,2],[2,1,3]. Each item has 3 numbers, first number is the type of item (1 or 2 here) and it is also the number of frame to be shown in the attached items movie clip. Second and third number determine tile it will be placed. Lets look at the last item: we know it is type 2 and it will be placed on the tile x=1, y=3.
Last part of code declares two types of items. Currently they only have one property "points". Thats how much points player gets for picking the item. Type1 item will add 1 to the players score, type2 item gives whopping 10 points.
Now lets modify the buildMap function to add items when map is created. Add this code before char creation:

```
game.items = myItems[game.currentMap];
for (var i = 0; i<game.items.length; ++i) {
		var name = "item"+game.items[i][2]+"_"+game.items[i][1];
		game[name]= new game["Item"+game.items[i][0]];
		game[name].position= i;
		game.clip.attachMovie("items", name, 10001+i);
		game[name].clip=game.clip[name];
		game[name].clip._x = (game.items[i][1]*game.tileW)+game.tileW/2;
		game[name].clip._y = (game.items[i][2]*game.tileH)+game.tileH/2;
		game[name].clip.gotoAndStop(game.items[i][0]);
}
_root.points=game.points;
```

First we make copy of the myItems array for the current map. This array called "game.items" will hold the information about items on the current map. We then loop through all the elements in the items array.
From the line:

```
var name = "item"+game.items[i][2]+"_"+game.items[i][1];
```

we will get name for new item. Its name will follow the same rules as names of our tiles, item on the tile x=1, y=3 will be named "item3_1".
After creating new item object from the templates we made earlier, we save position into that new object. What position is that? Its counter "i" and by saving it in the item object we will know which item in the items array this object represents. This will be very handy when we start to pick up items. Lets see: when we make map1 and i=1, we are creating item from the data [1,1,2], thats second element for the map1 items array. Item will be named "item2_1" and we can access its position in the array with item2_1.position. More about this when we remove items.
After that we place new instance of items movie clip on stage and place it in the correct coordinates. Last line in the loop sends new movie clip to the frame equal with the type of item. All type1 items will show frame 1 for example.
Finally we update the points variable to show correct number of points player has. In the beginning of the game he probably has 0 points, but when changing maps we still use same function and by that time player might have been lucky and collected some points.

Find it

We have items, we have hero, now we have to make sure hero knows when he has stepped on something. Add to the end of moveChar function following code:

```
var itemname=game["item"+ob.ytile+"_"+ob.xtile];
if (itemname and ob == _root.char) {
        game.points=game.points+itemname.points;
        _root.points=game.points;
        removeMovieClip(itemname.clip);
        game.items[itemname.position]=0;
        delete game["item"+ob.ytile+"_"+ob.xtile];
}
```

The variable itemname will have value based on the current position of hero. When hero stands on the tile x=4, y=9, it will look for "item9_4". If such object exists, itemname will refer to that object, but in the unlucky case of not having item in that tile, itemname will have no value.
If we have found item object and we are moving hero, we will first add points from the item object to the game.points and then update variable _root.points too to show them.
Now its time to remove item from the stage. Each item has to be removed from 3 places: its movie clip, from items array and its object. Currently we dont delete it from the items array, we just set 0 in the position of this item. And dont forget, items array is only copy of myItems array, if we leave the map and return, all items would appear again. For prevent such bad idea, we will update the changeMap function.
Add to the beginning of changeMap function:

```
var tempitems=[];
for (var i = 0; i<game.items.length; ++i) {
  if(game.items[i]) {
    var name = "item"+game.items[i][2]+"_"+game.items[i][1];
    delete game[name];
    tempitems.push(game.items[i]);
  }
}
myItems[game.currentMap]=tempitems;
```

Here we use temporary array "tempitems" to copy items not yet picked up back from items array to the myItems array. When element in the position i is not 0, that means item was not picked up and we will save it to be shown next time player enters this map. But to make sure the item object wont appear in next map, we have to delete it first. However when item was picked up, it will not appear again when players comes back.
You can download the source fla with all the code and movie set up here.
And here I have set up side scroller with everything talked so far:

Bugfix in 31/12/2003. Thanks to José for finding the bug in the items code when changing maps. The problem was, that if item was not picked up, then item object was not deleted in the map changing process and it existed in the next map too, but it didnt have any movie clip to show. So, you ended up with invisible items. I have updated the fla's and code, but please make sure you have lines:

```
var name = "item"+game.items[i][2]+"_"+game.items[i][1];
delete game[name];
```

in the changeMap function.

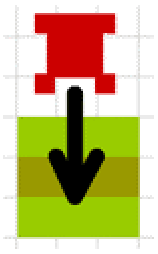
Moving tiles

Gather up, boys and girls, for today I am going to tell you a little story about moving tiles. Some of you may know it already by the name "moving platform", dont be fooled by the name, its still same cute thing.
Once, long time ago, in the land of tile based games, lived a young tile. He was a happy tile. One fine day a hero came to him and asked: "Young tile, why dont you move?"
"I dont know how to move," said little tile.
"Thats sad," sighed the hero, "since I would love to stand on you and move with you to the places I cant reach by my own."
That day young tile realised his life was not so happy anymore.
But we can help him to move:

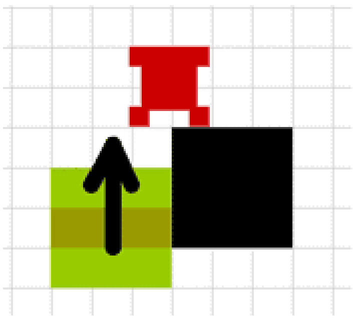
Before we actually start the coding, we have to make some rules. Rules, while annoying, are very helpful. What are we doing? Where to go? Whats the meaning of life? Rules to answer:
* moving tiles are cloud type (for explanation please look here)
* moving tiles can move horisontally and vertically
* hero can land on the moving tile only from above
* when hero is standing on moving tile, he moves with the tile
* hero on moving tile still cant go into wall

Stepping on moving tile

So, how can hero land on the moving tile. First and simplest way is to jump.



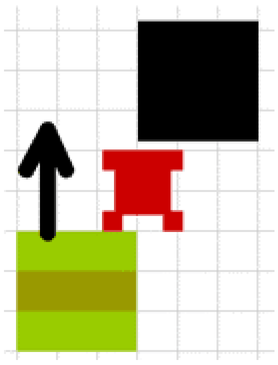
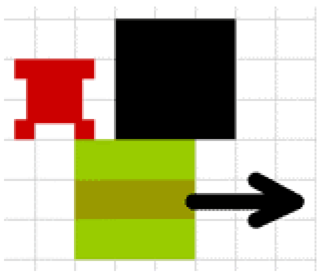
On the picture hero is moving down and in the next step he would be inside the moving tile. We will place him standing on the tile. Note that hero MUST be above the moving tile and hero MUST be moving down. In any other case hero wont land on moving tile.
But thats not the only way for hero to get on the moving tile. In the next picture hero is standing still on the piece of wall and he is not moving anywhere.



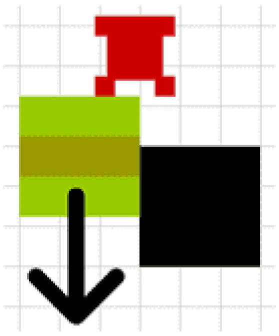
But the moving tile is moving up and in the next step, hero would be inside the moving tile. Yes, we again have to make sure the hero will be catched by the moving tile and he starts to move up with the moving tile.

Stepping off

Once we have the hero standing on the moving tile, we also need to create some ways for the hero to get off. First, he can jump off. He can walk over the edge of moving tile. And also couple of possible situations exist in the next pictures:

When hero stands on the moving tile moving up and in the next step hero would hit the wall above, he should drop off the moving tile or he would be squashed. When hero is standing on the moving tile moving horisontally, hitting the wall left/right in next step, he should be placed near the wall and in case the moving tile keeps moving, hero should fall off.



In this picture hero is moving down while standing on the moving tile. When the hero hits the wall tile, he should stay on the wall. Moving tile continues going down, but without the hero.

Preparing

Draw moving tile movie clip. You can have many different moving tiles, put them in different frames inside "movingtiles" mc. Make sure mc in the library is linked as "movingtiles".
Declare movingtiles objects:

```
game.MovingTilep1= function () {};
game.MovingTilep1.prototype.speed=2;
game.MovingTilep1.prototype.dirx=0;
game.MovingTilep1.prototype.diry=1;
game.MovingTilep1.prototype.miny= 0;
game.MovingTilep1.prototype.maxy=2;
game.MovingTilep1.prototype.width=game.tileW/2;
game.MovingTilep1.prototype.height=game.tileH/2;
game.MovingTilep2= function () {};
game.MovingTilep2.prototype.speed=2;
game.MovingTilep2.prototype.dirx=1;
game.MovingTilep2.prototype.diry=0;
game.MovingTilep2.prototype.minx= -2;
game.MovingTilep2.prototype.maxx=2;
game.MovingTilep2.prototype.width=game.tileW/2;
game.MovingTilep2.prototype.height=game.tileH/2;
```

We have 2 kinds of moving tiles: MovingTilep1 will be moving vertically (its diry property is set to non-zero value) and MovingTilep2 is horisontal mover (since its dirx is set so). The speed property, as you might of already guessed, sets how many pixels tile moves in each step.
The miny/maxy/minx/maxx properties will set the boundaries for the movement. We could write some absolute values too, but then doing changes will be confusing. Instead, we have the boundaries set as relative to the starting position of the moving tile. We can place the moving tile in any position and it will still move correctly between its boundaries. Remember, moving tiles dont check for any wall tiles and its your job to place them so they dont move through walls. Or place them so they move through walls if you want so. You are making the game, so you are the god.
Lets look at the example. Movingtile starts in the position x=2, y=5. It has vertical movement and miny=-1, maxy=4. How far will it move then? Starting y-miny=5+(-1)=4, so minimum position it goes, is x=2, y=4. Maximum position is 5+4=9 or the postion x=2, y=9.

To give moving tiles starting position, we use array similar to the enemies array:

```
//moving tiles array is in the order [tile type, xtile, ytile]
myMovingTiles = [
[0],
[[1, 4, 2]],
[[2, 4, 4]]
];
```

In the map1 we have declared 1 moving tile. Its type 1 (made from the template MovingTile1) and its placed at the starting position x=4, y=2. Map2 also has only 1 moving tile. You can put more then 1 moving tile in each map.
Next we need to add moving tiles in the buildMap function. Write after the enemies code:

```
game.movingtiles = myMovingTiles[game.currentMap];
for (var i = 0; i<game.movingtiles.length; ++i) {
	var name = "movingtile"+i;
	game[name]= new game["MovingTilep"+game.movingtiles[i][0]];
	game.clip.attachMovie("movingtiles", name, 12001+i);
	game[name].clip=game.clip[name];
	game[name].clip.gotoAndStop(game.movingtiles[i][0]);
	game[name].xtile = game.movingtiles[i][1];
	game[name].ytile = game.movingtiles[i][2];
	game[name].x = game[name].xtile *game.tileW+game.tileW/2;
	game[name].y = game[name].ytile *game.tileH+game.tileH/2;
	game[name].clip._x = game[name].x;
	game[name].clip._y = game[name].y;
	game[name].minx=game[name].minx+game[name].xtile;
	game[name].maxx=game[name].maxx+game[name].xtile;
	game[name].miny=game[name].miny+game[name].ytile;
	game[name].maxy=game[name].maxy+game[name].ytile;
}
```

We take the array for current map from the moving tiles array. Variable game.movingtiles will be holding info about how many moving tiles we have on stage and where they start at. Then we create new object, place mc on stage in correct position and send it in correct frame. Note that type1 moving tile will be sent to the frame1, type2 to the frame2. Last part of the code calculates values for the boundaries for each moving tile. While the names are again miny/max/minx/maxx, the properties are not relative anymore. Each moving tile object will have absolute values for its boundaries so we dont have to contsantly calculate them over and over to check if its time to turn back.

In the moveChar function we have to add 1 line in the beginning to save current y position:

```
ob.lasty=ob.y;
```

We will also have to rewrite the code in the moveChar function for movement down

```
if (diry == 1) {
  if (ob.downleft and ob.downright and !checkMovingTiles(speed*diry)) {
    ob.y += speed*diry;
  } else {
    ob.jump = false;
    if(ob.onMovingTile){
      ob.y=ob.onMovingTile.y-ob.onMovingTile.height-ob.height;
    }else{
      ob.y = (ob.ytile+1)*game.tileH-ob.height;
    }
  }
}
```

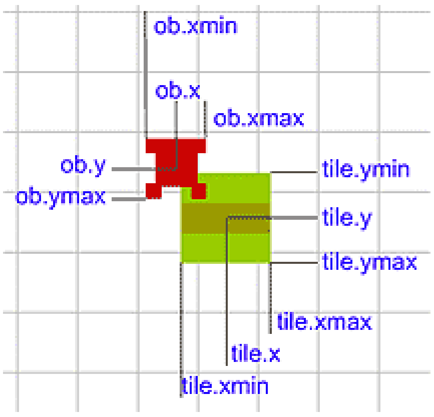
We have added the call for checkMovingTiles function, which will return true in case hero will land on moving tile. And if we have landed on it, we set the y coordinate of the hero so it stands just above the tile.

Is hero on moving tile?

You already guessed from the addition in the moveChar function, yes, its time to make new function for the checking if char is standing on the moving tile. checkMovingTiles function not only finds the answer, but it also saves the name of the tile hero currently moves on, into the char object.

```
function checkMovingTiles (y) {
  if(char.diry<>-1){
    var heroymax=char.y+char.height+y;
    var heroxmax=char.x+char.width;
    var heroxmin=char.x-char.width;
    foundit=false;
    for (var i = 0; i<game.movingtiles.length; i++) {
      var ob=game["movingtile"+i];
      var tileymax=ob.y+ob.height;
      var tileymin=ob.y-ob.height;
      var tilexmax=ob.x+ob.width;
      var tilexmin=ob.x-ob.width;
      if(char.lasty+char.height<=tileymin){
        if (heroymax<=tileymax and heroymax>=tileymin) {
          if (heroxmax>tilexmin and heroxmax<tilexmax) {
           char.onMovingTile=ob;
           foundit=true;
           break;
          } else if (heroxmin>tilexmin and heroxmin<tilexmax) {
           char.onMovingTile=ob;
           foundit=true;
           break;
         }
        }
       }
      }
      return(foundit);
      }
     }
```

Lets see whats happening here. If char is not moving up (diry is not -1), we calculate the boundaries for the char. Then we start to loop through all the moving tiles. ob will be moving tile object we are currently dealing with. We also calculate boundaries for current tile to figure out if they are colliding:



The if statement with "lasty" property secures that chars last position was above the moving tile and other if statements find collision between char and tile. If we have found moving tile, then the onMovingTile will be holding reference to the tile object.

It shall move

Be prepared for the ugliest, biggest, meaniest function in the history of mankind! It is big, because it does many things, first, it moves around all the moving tiles, it then checks if the tiles need to have their movement direction reversed, and if thats not hard work enough, this function also handles the movement of hero standing on the moving tile and checks if hero should fall off from it.

```
function moveTiles () {
 for (var i = 0; i<game.movingtiles.length; i++) {
   var ob=game["movingtile"+i];
   getMyCorners (ob.x + ob.speed*ob.dirx, ob.y + ob.speed*ob.diry, ob)
   if (ob.miny>ob.upY or ob.maxy<ob.downY) {
     ob.diry=-ob.diry;
   }
   if (ob.minx>ob.leftX or ob.maxx<ob.rightX) {
     ob.dirx=-ob.dirx;
   }
   ob.x = ob.x + ob.speed*ob.dirx;
   ob.y = ob.y + ob.speed*ob.diry;
   ob.xtile = Math.floor(ob.x/game.tileW);
   ob.ytile = Math.floor(ob.y/game.tileH);
   ob.clip._x = ob.x;
   ob.clip._y = ob.y;
   if(ob.diry==-1){
     checkMovingTiles(0);
   }
 }
 //check if hero is on moving tile
 if(char.onMovingTile){
   getMyCorners (char.x, char.y+char.onMovingTile.speed*char.onMovingTile.diry, char);
   if (char.onMovingTile.diry == -1) {
     if (char.upleft and char.upright) {
       char.y=char.onMovingTile.y-char.onMovingTile.height-char.height;
     } else {
       char.y = char.ytile*game.tileH+char.height;
       char.jumpspeed = 0;
       char.jump = true;
       char.onMovingTile=false;
     }
   }
   if (char.onMovingTile.diry == 1) {
     if (char.downleft and char.downright) {
       char.y=char.onMovingTile.y-char.onMovingTile.height-char.height;
     } else {
       char.onMovingTile=false;
       char.y = (char.ytile+1)*game.tileH-char.height;
     }
   }
   getMyCorners (char.x+char.onMovingTile.speed*char.onMovingTile.dirx, char.y, char);
   if (char.onMovingTile.dirx == -1) {
     if (char.downleft and char.upleft) {
       char.x += char.onMovingTile.speed*char.onMovingTile.dirx;
     } else {
       char.x = char.xtile*game.tileW+char.width;
       fall (char);
     }
   }
   if (char.onMovingTile.dirx == 1) {
     if (char.upright and char.downright) {
       char.x += char.onMovingTile.speed*char.onMovingTile.dirx;
     } else {
       fall (char);
       char.x = (char.xtile+1)*game.tileW-char.width;
     }
   }
   updateChar (char);
 }
}
```

Well, like told before, the first part moves moving tiles. It loops through all the moving tiles and compares their next position with miny/maxy(minx/maxx properties. If moving tile has moved too far, its movement is reversed.
With the following code:

```
if(ob.diry==-1){
  checkMovingTiles(0);
}
```

we check if we have picked up the hero. Notice, that this is only possible if hero is standing still on the edge of wall and tile is moving up (diry is -1).
The part of function from the line:

```
if(char.onMovingTile){
```

deals with hero. When onMovingTile property is not false, that means char is standing on one of them and its onMovingTile refers to the right moving tile object. The code here is very similar to the code in moveChar function. We calculate corners for next position of hero using getMyCorners function, if we havent hit any walls, we move the hero with tile, but if some hard wall has been found, then hero cant move there.

Wrapping it up

In the detectKeys function add line to the beginning to move all the tiles before hero gets chance to move too:

```
moveTiles();
```

and in case hero will jump, we have to set its onMovingTile property to false:

```
ob.onMovingTile=false;
```

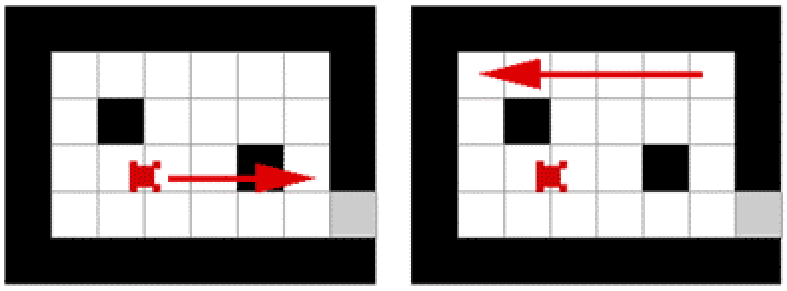
You can download the source fla with all the code and movie set up here.

Scrolling

Before we start to scroll, we must make one thing very clear. Flash is slow. Flash is extremely slow. Scrolling means moving hundreds of tiles on screen and it should happen 20-30 times in second. Too many scrolling tiles means Flash cant draw them in time and slows down. Your game will crawl like a sleeping snail.
"What?" you might wonder, "no scrolling? And how did snail fall to sleep?".
You can make scrolling tiles, but you should be careful not to make scrolling area too big and not to scroll too many tiles. How big is too big and how many is too many, those answers you have to find out yourself. Remember, that Flash games are played mostly on the browser, probably many windows are opened same time, many programs are running on background and players doesnt always have latest power computers. Test your game in old crappy computers and if it feels slowing down, make it smaller.

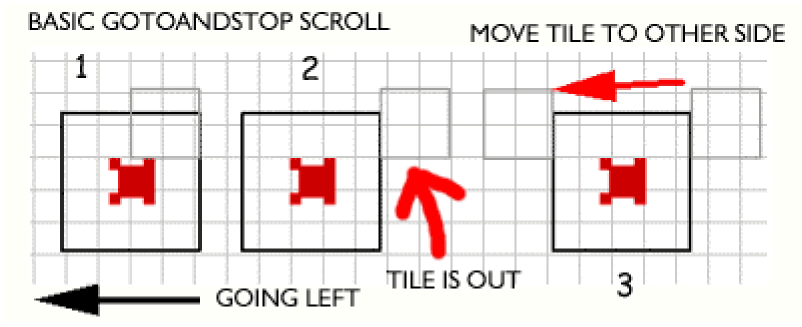
Theory

In the left picture is non-scrolling game. Hero moves right, everything else stays where it was. Now, in the right picture we have scrolling game. Hero is also suppose to move right, but to make it look like scrolling, we actually keep hero in its current position and move everything else to the left.



So, theory is easy: when hero is suppose to move, move all the tiles in the opposite direction. But since we have used to place hero inside the tiles movie clip with all the background tiles, hero would move in opposite way with them. To fix it, we still move everything in opposite direction AND we will move hero same amount in correct direction.
Time for example. Scrolling game. Hero is suppose to move 10 pixels right. First, we move all the tiles (including hero) left by 10 px, and we move hero right by 10 px. In the end, it looks like hero has stayed on place, but other tiles have scrolled left.
Easiest way to scroll tiles, is to place all the tiles on screen, but to show only small portion of them, then we simply move them all around. This might make your game very slow, since thousands of tiles outside visible area still need resources. Next idea is to remove tiles when they go off from visible area and attach tiles when they come visible again. Thats better, but Flash spends too much time removing/attaching movie clips.
Our last hope is to place only visible tiles on stage and when they go off, we move the same tiles to the opposite side, rename them and reuse same movie clips.

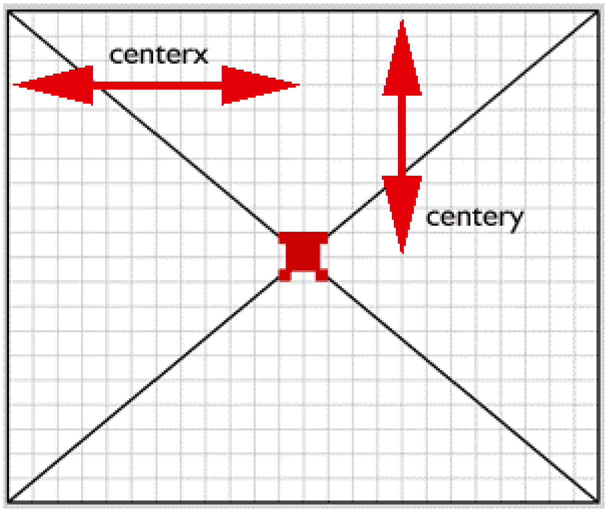
Thats called "gotoAndStop" scrolling engine:



Like seen on the picture, when tile goes off from right, we move the tile to the left. We also have to rename the movie clip, since all our movie clips have names like "t_3_4" which means it is placed in the y=3, x=4. And tile in the new position probably has to show different frame (graphic), that why we need to send it to correct frame with correct graphics and thats why this method is called "gotoAndStop".

Preparing to scroll

In most scrolling games, hero is always staying on the center of screen.



You can see how number of tiles left from hero is equal to the number of tiles right from him. That means your number of columns is 3,5,7,9,11 etc, but never 2,4,6,8. Same goes for rows too.
Lets declare game object:

```
game = {tileW:30, tileH:30, currentMap:1, visx:7, visy:5, centerx:120, centery:90};
```

visx property is number of visible columns and visy is number of rows. We also need properties centerx/centery to mark center point of our movie.
When we scroll the tiles, we might have to show tiles that are not declared in the map array. Like when hero happily walks left until he stands on the leftmost tile declared with map array, we still have some tiles left from him to show. For those kind of tiles we create new tile type with no graphics to show (to make it easier for Flash). In the tiles movie clip make new empty keyframe in frame 20. Add the code to declare this type of tile:

```
game.Tile4 = function () { };
game.Tile4.prototype.walkable = false;
game.Tile4.prototype.frame = 20;
```

You might also want to cover some tiles partly ouside of visible area. For that purpose make new movie clip with linkage name "frame", that has hole in the middle, showing only the tiles through that hole.

Build the world of scroll

Lets start with buildMap function.

```
function buildMap (map) {
  _root.attachMovie("empty", "tiles", 1);
  game.halfvisx=int(game.visx/2);
  game.halfvisy=int(game.visy/2);
```

We will calculate 2 new properties for the game object. halfvisx and halfvisx will hold information about number of columns and rows from the chars position to the edge of visible area. When total number of columns visx=5, then halfvisx=2, meaning 2 columns are right from hero, 2 columns are left and 1 is straight in the center with the hero.

```
game.clip = _root.tiles;
game.clip._x = game.centerx-(char.xtile*game.tileW)-game.tileW/2;
game.clip._y = game.centery-(char.ytile*game.tileH)-game.tileH/2;
```

We have to place movie clip holding all the tiles and the hero based on 2 variables: center point of the game declared in the game object by centerx/centery and position of the hero declared in the char object by xtile/ytile. When hero is placed on the x coordinate (char.xtile*game.tileW)+game.tileW/2, then tiles mc have to be in the opposite direction, thats why we subtract that number from the centerx.

```
for (var y = char.ytile-game.halfvisy; y<=char.ytile+game.halfvisy+1; ++y) {
  for (var x = char.xtile-game.halfvisx; x<=char.xtile+game.halfvisx+1; ++x) {
    var name = "t_"+y+"_"+x;
    if(y>=0 and x>=0 and y<=map.length-1 and x<=map[0].length-1){
      game[name] = new game["Tile"+map[y][x]]();
    }else{
      game[name] = new game.Tile4();
    }
    game.clip.attachMovie("tile", name, 1+y*100+x*2);
    game.clip[name]._x = (x*game.tileW);
    game.clip[name]._y = (y*game.tileH);
    game.clip[name].gotoAndStop(game[name].frame);
  }
}
```

This loop creates all the visible tile objects and also attaches movie clips for the tiles. As you can see, the loop doesnt start from 0 anymore, it starts from the ytile-halfvisy. Same way it doesnt run until the end of map array, it runs until ytile+halfvisy+1. The if condition then checks if the tile to be created is in the map array. If it falls outside the map, we use empty tile from the tile4 template.
In order to create seamless scroll, we have to use one extra row and one extra column of tiles in the right and bottom edge. Those tiles ensure, that even when we move half of tile to the other side, there wont be any empty space.

```
_root.attachMovie("frame", "frame", 100);
```

Last line attaches frame to cover areas of movie you dont want to be seen.

```
char.clip = game.clip.char;
char.x = (char.xtile*game.tileW)+game.tileW/2;
char.y = (char.ytile*game.tileW)+game.tileW/2;
char.width = char.clip._width/2;
char.height = char.clip._height/2;
char.clip._x = char.x;
char.clip._y = char.y;
char.clip.gotoAndStop(char.frame);
char.xstep=char.x;
char.ystep=char.y;
```

The char is created exactly same way as before. Only difference are 2 new properties xstep and ystep, which we will use later to check for right time to move rows or columns of tiles to the other side.

Scroll! Scroll!

Now that we have set up the world, we need to actually scroll it. In the end of moveChar function after we have calculated the ob.xtile/ob.ytile for hero, add code to scroll:

```
game.clip._x = game.centerx-ob.x;
game.clip._y = game.centery-ob.y;
if(ob.xstep<ob.x-game.tileW){
  var xtile = Math.floor(ob.xstep/game.tileW) + 1;
  var xnew=xtile+game.halfvisx+1;
  var xold=xtile-game.halfvisx-1;
  for (var i = ob.ytile-game.halfvisy-1; i<=ob.ytile+game.halfvisy+1; ++i) {
    changeTile (xold, i, xnew, i, _root["myMap"+game.currentMap]);
  }
  ob.xstep=ob.xstep+game.tileW;
} else if(ob.xstep>ob.x){
  var xtile = Math.floor(ob.xstep/game.tileW);
  var xnew=xtile+game.halfvisx+1;
  var xold=xtile-game.halfvisx-1;
  for (var i = ob.ytile-game.halfvisy-1; i<=ob.ytile+game.halfvisy+1; ++i) {
    changeTile (xold, i, xnew, i, _root["myMap"+game.currentMap]);
  }
  ob.xstep=ob.xstep-game.tileW;
}
if(ob.ystep<ob.y-game.tileH){
  var ytile = Math.floor(ob.ystep/game.tileH)+1;
  var ynew=ytile+game.halfvisy+1;
  var yold=ytile-game.halfvisy-1;
  for (var i = ob.xtile-game.halfvisx-1; i<=ob.xtile+game.halfvisx+1; ++i) {
    changeTile (i, yold, i, ynew, _root["myMap"+game.currentMap]);
  }
  ob.ystep=ob.ystep+game.tileH;
} else if(ob.ystep>ob.y){
  var ytile = Math.floor(ob.ystep/game.tileH);
  var yold=ytile+game.halfvisy+1;
  var ynew=ytile-game.halfvisy-1;
  for (var i = ob.xtile-game.halfvisx-1; i<=ob.xtile+game.halfvisx+1; ++i) {
    changeTile (i, yold, i, ynew, _root["myMap"+game.currentMap]);
  }
  ob.ystep=ob.ystep-game.tileH;
}
return (true);
```

First we move game.clip holding all the tiles and the hero to new position based on center point and coordinates of the hero. Then we have 4 similar blocks of code, each for 1 direction. When hero has moved more then the size of tiles, those loops call changeTile function with correct variables. When loop has ended and tiles have been moved/renamed/changed, we update ystep/xstep properties.
Now lets make that changeTile function:

```
function changeTile (xold, yold, xnew, ynew, map) {
        var nameold = "t_"+yold+"_"+xold;
        var namenew = "t_"+ynew+"_"+xnew;
        if(ynew>=0 and xnew>=0 and ynew<=map.length-1 and xnew<=map[0].length-1){
                game[namenew] = new game["Tile"+map[ynew][xnew]]();
                game.clip[nameold]._name = namenew;
                game.clip[namenew].gotoAndStop(game[namenew].frame);
                game.clip[namenew]._x = (xnew*game.tileW);
                game.clip[namenew]._y = (ynew*game.tileH);
        }else{
                game[namenew] = new game.Tile4();
```

```
                    game.clip[nameold]._name = namenew;
                    game.clip[namenew].gotoAndStop(game[namenew].frame);
          }
}
```

So, we will receive 2 old coordinates and 2 new for our tile. To check if tile is still inside map array, we also have passed the map. "nameold" will be our tiles old name and "namenew" will be new name. After we have created new tile object with new and fresh name, the line:

```
game.clip[nameold]._name = namenew;
```

renames tiles movie clip. When new movie clip is empty, then we dont need to place it to the new _x/_y, it can remain in its old position.

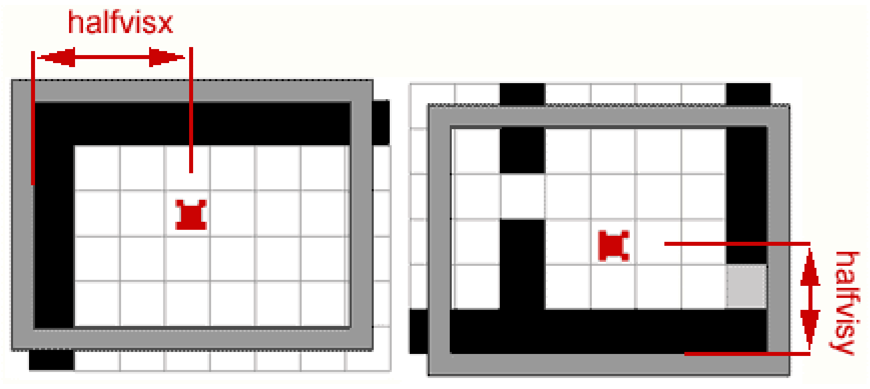
You can download the source fla with all the code and movie set up here.
Bugfix at 29/jul/2004. The scrolling didnt work correctly with speeds higher then half the tile width. Thanks to Chris for spotting the bug and providing better solution. In the moveChar function for checking if rows/columns should be moved, there should be lines to calculate localvariable xtile/ytile and not old ob.xtile/ob.ytile.

More scrolling

Keeping the hero in the center is all fine until we move at the border of map, then we start to see some ugly background outside of the map. You can make this problem disappear, if you build wall tiles inside your map, preventing heros approaches to the edge. But that will need additional planning in the maps, and it adds unnecessary empty area around them. Much better idea is to scroll the background only when hero is not near the edge.

How far is too far?

Hero will always move, only difference is, that when he reaches the edge of map, we wont scroll the background tiles anymore, making it not scroll. In the left picture hero is from the left edge "halfvisx" number of tiles away. If hero would move more left and the map would scroll, it would reveal an area not covered by the map and the tiles:



In the right ricture hero is from the bottom edge "halfvisy" number of tiles away. No more scrolling should happen when hero moves down.
We have to consider the new positions of hero also when we first build the map. If hero starts near the map edge, he cant be placed in the center. What actually happens, is that we will shift all the tiles, included hero by certain amount when placing them with buildMap function.
After placing tiles clip and calculating values for halfvisx/halfvisy in the buildMap function add code:

```
game.mapwidth=map[0].length;
game.mapheight=map.length;
```

game.mapwidth will have be the number of horisontal tiles on current map and game.mapheight is number of vertical tiles. We will use those to determine, if hero is near the right or bottom edge of map. Now lets calculate how much we have to move tiles when hero starts near the edge:

```
if(game.halfvisx>char.xtile){
  var fixx=char.xtile-game.halfvisx;
  var fixx1=0;
}else if(char.xtile>game.mapwidth-game.halfvisx-1){
  var fixx=char.xtile-game.mapwidth+game.halfvisx+1;
  var fixx1=1;
}
if(game.halfvisy>char.ytile){
  var fixy=char.ytile-game.halfvisy;
  var fixy1=0;
}else if(char.ytile>game.mapheight-game.halfvisy-1){
  var fixy=char.ytile-game.mapheight+game.halfvisy+1;
  var fixy1=1;
}
```

fixx and fixy will have value of number of tiles needed to shift everything to make hero appear in the correct position. First line checks if hero is near the left edge, it happens only when xtile is less then halfvisx and then we move tiles by the amount of xtile-halfvisx. In the right edge hero stands only if xtile is more then mapwidth-halfvisx. Variables fixx1 and fixy1 (sorry for not using clear and understandable variable names) will be used in the loop when going through the tiles. Without those the code would still trying to place tiles outside of the map area, when hero stands near the right or bottom edge.
Now we add fixx/fixy to the position of tile:

```
game.clip._x = game.centerx-((char.xtile-fixx)*game.tileW)-game.tileW/2;
game.clip._y = game.centery-((char.ytile-fixy)*game.tileH)-game.tileH/2;
```

and we also have to add fixing variables to the loops for creating visible tiles.
To make sure we start to count from the right tile, add those fix variables to the xstep and ystep too:

```
char.xstep=char.x-(fixx+fixx1)*game.tileW;
char.ystep=char.y-(fixy+fixy1)*game.tileH;
```

Thats about building the map. Now lets move on to move the hero.

Moving on the edge

In the moveChar function we first calculate the fixx and fixx1 variables same way as in buildmap function. Then we put the scrolling part of the code into if statements, which is only true in case hero is far enough from the edge of the map. Same way we modify vertical movement too.
Since sometimes changeTile function is still called to change tile that doesnt actually exist, we apply quick and dirty fix. If the old tile doesnt exist, return without doing anything:

```
if(game[nameold].walkable==undefined){
  return;
}
```

Thats all from the scrolling department, next we will look at the depth of movie clips and something very scary called "z-sorting".

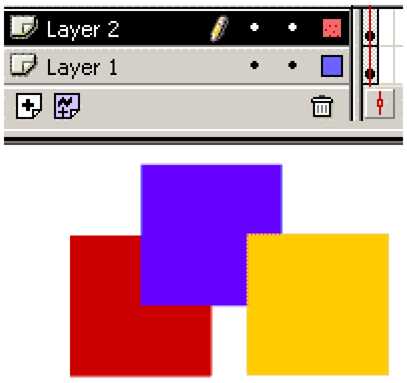
You can download the source fla with all the code and movie set up here.
This chapter was rewritten in 9/apr/2004 because the original code failed in the right and down edge of the map. Thanks to Dave for finding the bug.

Depth

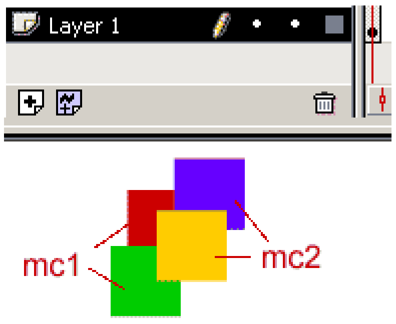
So far we have kept our game stricktly two-dimensional. That means if some tile is in front of other tile, it will remain in front no matter what happens in the game. And for those poor unhappy tiles placed in the background will never make it to the front row. Luckily for the tiles on back and maybe not so luckily for the tiles currently in front, we can change the situation. To make our game look better, we will bring in the "depth", creating illusion of objects being closer or further. Here is an example (move the hero below and above same black wall tile):

Many levels of Flash

Im sure you have noticed how some things you draw in the Flash cover up other things. If objects are on the same layer, then things you have drawn later are placed in front. You can also change the order of objects on the same layer with "bring to front/send to back" commands from the Modify>>Arrange menu.
Flash has also been kind enough to provide us layers. Layers are great to arrange objects, when you draw something on the layer above other layer, it will always remain in front of objects on the other layer.



In the picture blue square and red square are both drawn on layer1. Blue square was drawn last, so its in front of red square. Yellow square is on layer2 and since layer2 is above layer1, yellow square covers both blue and red square.
In the next picture red and green squares are in the movie clip "mc1", blue and yellow squares are inside movie clip "mc2".



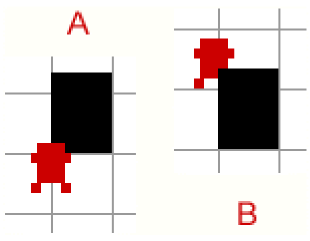
Blue and yellow square always cover red and green because the parent mc2 is above mc1. Inside movie clips you can arrange the order of squares, but you cant place blue square in front of red and same time yellow behind red.
So far everything we have looked at, are placed directly in Flash environment. As you know, the tile based games dont have tiles drawn directly on stage. Nonono, we cant do anything so simple! Instead, we are using "attachMovie" command to place movie clips on stage dynamically. attachmovie command has syntax

```
anyMovieClip.attachMovie(idName, newname, depth);
```

You see, it has depth written in its syntax already. What is exactly that "depth" in its syntax? Everything you draw directly in the stage, are placed on _level0. Thats the furthest it will go. **Everything you attach dynamically** (using attachmovie, duplicateMovieClip or loadMovie) **will be placed in the higher levels, above _level0**. From now on we will only deal with movie clips attached to the stage. The other levels in Flash where only brought up to refresh your memory and make you more confused.

Finding right depth

In picture A hero is covering part of wall, so hero looks like being closer to viewer then wall, but in the picture B part of the wall covers hero making hero standing further.

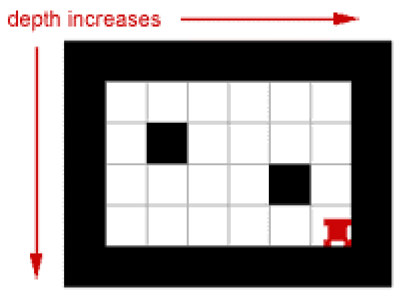


Now, whats the difference between those two pictures beside hero/wall coverup? The hero and wall are same. The wall is in the same spot. Yes, the hero has moved up. Now we will come to the very important conclusion: **object moves closer when its y-coordinate increases**. Its also worth to remember that we will consider higher depth being closer to viewer.
How could we change the depth of hero? Flash has a command for it: "swapDepths". We can change depth of any movie clip until we know what its depth should be. From the picture above we can say that in most basic level objects depth is its y coordinate:

```
mc.swapDepths(mc._y)
```

Since we have rows of tiles, all in exactly same y coordinate, we also should take into account the x coordinate, for no two movie clips cant share same depth in Flash. If you swapDepth movie clip to the depth, where another movie clip already sits in, they swap depths (hm, thats where the name of swapDepths comes from) and we dont want that.

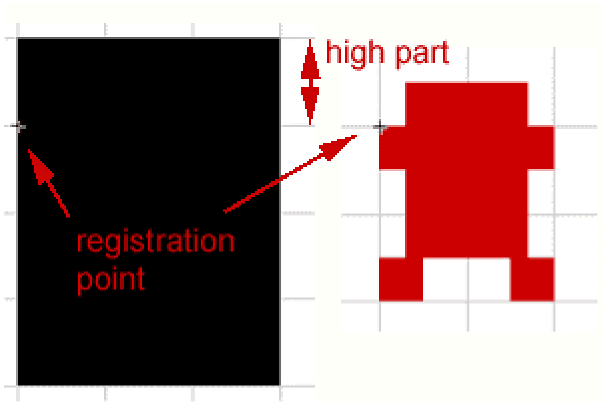
```
mc.swapDepths(mc._y*movie_width+mc._x)
```



Tile in the left upper corner will be furthest, the depth of tiles increases (they will be closer) when moving right and down.

z-sorting

z-sorting is fancier name for what we have done so far with depths. It comes from the 3D world, where every point has x, y and z coordinate. z coordinate determines how close or far the point is from viewer. We will make our tiles all have right depth and when the hero moves around, we will change its depth according to where he stands.
Please make sure that both tiles and hero have registration point in the left top corner. Draw the parts you want to overlap other tiles above registration point.



Now lets declare char object:

```
char={xtile:2, ytile:1, speed:4, width:20, height:20};
```

The width and height should be predetermine and not read using _height because we want the hero to walk partly over the walls.
Now to the buildmap function:

```
function buildMap (map) {
  _root.attachMovie("empty", "tiles", 1);
  _root.tiles.attachMovie("empty", "back", 0);
```

We will attach extra movie clip to the tiles mc. Thats where we will place all the ground tiles which cant overlap the hero. The tiles like walls and trees which can be also in front of hero, will be placed in the tiles mc directly. Since "back" movie clip is at level0 inside "tiles" mc, it will be always behind every other object, icluding hero and trees and walls. And everything inside "back" movie clip will also be behind all the other objects placed directly into "tiles" mc.
In the loop to create tiles use this code:

```
var name = "t_"+i+"_"+j;
game[name]= new game["Tile"+map[i][j]];
if(game[name].walkable){
  var clip=game.clip.back;
}else{
  var clip=game.clip;
}
game[name].depth= i*game.tileH*300+j*game.tileW+1;
clip.attachMovie("tile", name, game[name].depth);
```

```
clip[name]._x = (j*game.tileW);
clip[name]._y = (i*game.tileH);
clip[name].gotoAndStop(game[name].frame);
```

After we have created tile object, we check if it is walkable tile (ground type) and if it is, we place it in the back movie clip. Next we calculate the depth of the tile. We will add 1 to the depth to avoid tile at x=0, y=0 being attached to the level0, where we have our "back" movie clip. If we would allow this, then "tile0_0" would replace "back" movie clip. Now lets calculate the depth of hero too:

```
ob.depth=ob.y*300+ob.x+1;
game.clip.attachMovie("char", "char", ob.depth);
```

And since the hero will change its depth in every step, add to the end of moveChar function:

```
ob.depth=ob.y*300+ob.x+1;
ob.clip.swapDepths(ob.depth);
return (true);
```

Now each time moveChar function is run, the depth of object is updated. If you would happen to use other moving objects, they too would update their depth to correct value. But if you add more moving objects, you should make sure they cant have exactly same x/y coordinate.

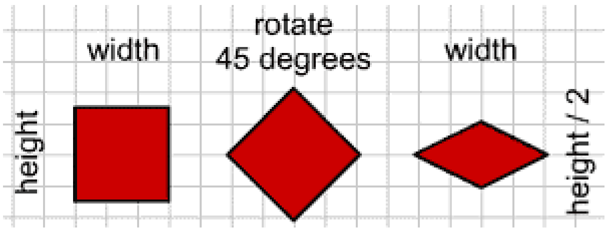
You can download the source fla with all the code and movie set up here.

Isometric view

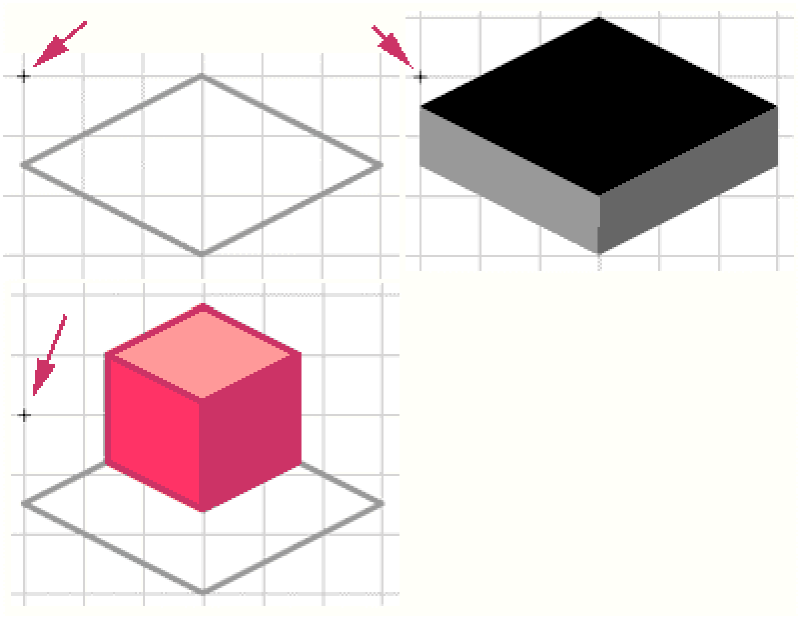
Isometric view is great to add depth into your game. Many famous games use this vie, because its simple to do, but looks good. And best thing about isometric view is how easy it is to create based same old tile based approach.

Theory

First you should know, that actual isometric view (from the mathematics) is little more complicated and never used in games. Now that you know it, forget it. From here on, we only talk about simple isometrics. Maybe best way to imagine isometric, is to look, what happens to the normal square, when its transformed into isometric view:



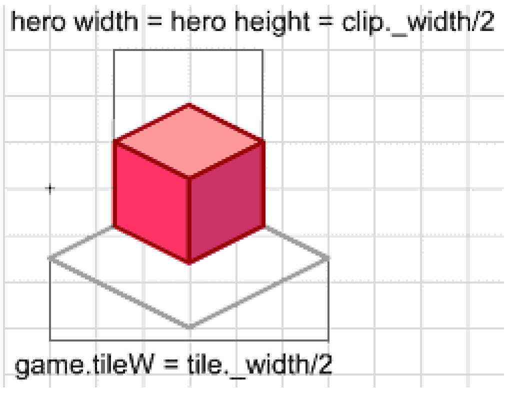
First we rotate the square by 45 degrees and then we make its height half the width. That was simple. Now lets create our tiles and hero:



Its important to place graphics to the registration point (little cross, where Flash starts to count coordinates from) like shown in the picture. The wall tile you can draw as high as you want. For the hero, I have left the tile in the picture so you hopefully understand its position better, dont place that rectangle in the final graphics of hero. It would only look strange, if hero walks around, rectangle around him.

Changes to the code

First thing to change, is size of tiles and size of hero. But wait, what size? So far it was clear, width and height of movie clip, but in isometric view the height of movie clip can be almost anything.


hero width = hero height = clip._width/2
game.tileW = tile._width/2

So, value of tileW variable is half the width of actual tile graphics. For the hero, its width and height properties are equal and half of its graphics actual width. Write it down:

```
game={tileW:30};
char={xtile:2, ytile:1, speed:4, width:16, height:16};
```

Biggest change in the code will be actual placement of movie clips on screen. In normal view, we used clip._x=x and clip._y=y, to make same x/y appear in isometric view, we will use:

```
clip._x=x-y;
clip._y=(x+y)/2;
```

In the buildMap function change placement of tiles:

```
clip[name]._x = (j-i)*game.tileW;
clip[name]._y = (j+i)*game.tileW/2;
```

Tiles depth is calculated same way as before, but we wont use x/y, we will use tiles new coordinates in isometric view:

```
game[name].depth=(j+i)*game.tileW/2*300+(j-i)*game.tileW+1;
```

And same thing for the hero:

```
var ob=char;
ob.x = ob.xtile*game.tileW;
ob.y = ob.ytile*game.tileW;
ob.xiso = ob.x-ob.y;
ob.yiso = (ob.x+ob.y)/2;
ob.depthshift=(game.tileW-ob.height)/2;
ob.depth=(ob.yiso-ob.depthshift)*300+ob.xiso+1;
game.clip.attachMovie("char", "char", ob.depth);
ob.clip=game.clip.char;
ob.clip._x = ob.xiso;
ob.clip._y = ob.yiso;
```

As you can see, new properties xiso and yiso will hold the coordinates of char in the isometri view, but we still have properties x/y. Variable depthshift is needed since our char is not drawn on the center of the tile (look the picture with char), it is shifted up. **All the collision and movements are calculated in normal way, but in the end position of movie clip is converted into isometric view.** In the end of moveChar function change the char placement same way:

```
ob.xiso = ob.x-ob.y;
ob.yiso = (ob.x+ob.y)/2;
ob.clip._x = ob.xiso;
ob.clip._y = ob.yiso;
ob.clip.gotoAndStop(dirx+diry*2+3);
ob.xtile = Math.floor(ob.x/game.tileW);
ob.ytile = Math.floor(ob.y/game.tileW);
ob.depth=(ob.yiso-ob.depthshift)*300+(ob.xiso)+1;
ob.clip.swapDepths(ob.depth);
```

After we have moved the char, detected the collisions with walls and probably placed char near the wall, properties x and y will hold correct position in non-isometric view. Now we only calculate new values for isometric view.

You can download the source fla with all the code and movie set up here.

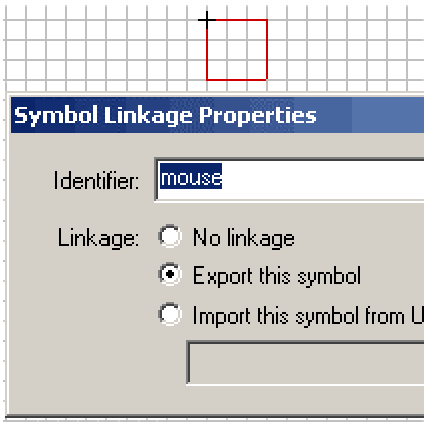
Mouse to move

Its time to let go the keyboard and grab that little furry thing on your desk. Not that, the computer mouse. While moving hero around with the keys is fine and has been happening for a long time, we should also understand why mouse is so popular. Mouse is so damn convenient, click here, click there.

Before we jump on the mouse, one thing should be made clear. So far our hero was able to move pixel perfect, not so with the mouse. Mouse control also means hero will be stepping from center of one tile to the center of next and he cant stop somewhere between two tile, look around, whistle nice tune. Moving from one tile another is much simpler then pixel perfect positioning. We dont need collision detection for example, when we want to walk on next tile, we only check if its walkable and then its all happy walking until we reach the center.

Making the mouse

Draw rectangle, which will represent the mouse, make it movie clip and set its linkage name to "mouse". Make sure you have aligned mouse graphics same way as tiles - left upper corner.


Symbol Linkage Properties

Identifier: mouse

Linkage: No linkage [✓ Export this symbol] Import this symbol from U

Now attach mouse to the stage, above all the tiles. In the buildMap function write.

```
_root.attachMovie("mouse", "mouse", 2);
```

Everything else in the buildMap function remains the same, so Im not going to go over it again. For mouse cursor to update its position correctly, lets create new function, called work:

```
function work() {
  game.xmouse=Math.round((_root._xmouse-game.tileW/2)/game.tileW);
  game.ymouse=Math.round((_root._ymouse-game.tileH/2)/game.tileH);
  _root.mouse._x=game.xmouse*game.tileW;
  _root.mouse._y=game.ymouse*game.tileH;
}
```

We calculate properties xmouse and ymouse in the game object, which will know where mouse is. To be exact, the tile number its over. In the char object we had similar properties called xtile and ytile. Last two lines place the mouse over that tile.
This function will be called on every frame from the controller movie clip on stage:

```
onClipEvent (enterFrame) {
  _root.work();
}
```

As we dont use keys, you can delete detectKeys function in case you have it left over from earlier chapter.
Now that we have mouse moving like it should, we add a way to click on stage. Write this to the controller mc:

```
onClipEvent (mouseUp) {
  _root.getTarget();
}
```

When left mouse button is released, we call function getTarget. By using clip event "mouseUp", we call the function only, when mouse button is released, not when its pressed down. If you prefer it other way, replace it with mouseDown event. Write the getTarget function:

```
function getTarget() {
  if(game["t_"+game.ymouse+"_"+game.xmouse].walkable){
    game.targetx=game.xmouse;
    game.targety=game.ymouse;
    char.moving=true;
  }
}
```

Here we make sure player actually clicks on walkable tile. You know how players are, they click everywhere and when you ask them politely "Why in the name of God did you have to click there?", they only say "I dunno, looked like nice spot". Anyway, we ensure they can still click anywhere they want to, but our function ignores all the clicks on non-walkable tiles. If, however, by some miracle, the player manages to click on walkable tile, we set properties "targetx" and "targety" to the tiles below the mouse. And we set variable "char.moving" to true.

Moving the hero to right tile

In the getTarget function we did set variable "moving" to true and as long that variable is true, our hero tries to move. Since we dont want the hero to move when game starts, set the moving variable to false in char object:

```
char={xtile:2, ytile:1, speed:2, moving:false};
```

Add some code in the end of work function:

```
var ob=char;
if (!ob.moving) {
  ob.clip.char.gotoAndStop(1);
} else {
  moveChar(ob);
  ob.clip.char.play();
}
```

Until moving variable is false, hero stands still and doesnt play any animations. But when moving is true, we play the walking animation and call function to move the char. Dont worry, for the movement from tile-to-tile our moveChar function is much simpler:

```
function moveChar(ob) {
  if((ob.x-game.tileW/2)%game.tileW==0 and (ob.y-game.tileH/2)%game.tileH==0){
    ob.xtile = Math.floor(ob.x/game.tileW);
    ob.ytile = Math.floor(ob.y/game.tileH);
    if(game["t_"+ob.ytile+"_"+(ob.xtile+1)].walkable and game.targetx>ob.xtile){
      ob.dirx=1;
      ob.diry=0;
    }else if(game["t_"+ob.ytile+"_"+(ob.xtile-1)].walkable and game.targetx<ob.xtile){
      ob.dirx=-1;
      ob.diry=0;
    }else if(game["t_"+(ob.ytile+1)+"_"+ob.xtile].walkable and game.targety>ob.ytile){
      ob.dirx=0;
      ob.diry=1;
    }else if(game["t_"+(ob.ytile-1)+"_"+ob.xtile].walkable and game.targety<ob.ytile){
      ob.dirx=0;
      ob.diry=-1;
    }else{
      ob.moving=false;
      return;
    }
  }
  ob.y += ob.speed*ob.diry;
  ob.x += ob.speed*ob.dirx;
  ob.clip._x = ob.x;
  ob.clip._y = ob.y;
  ob.clip.gotoAndStop(ob.dirx+ob.diry*2+3);
}
```

First line in the function checks if hero is currently standing in the center of tile and that would mean he is ready to move, if he only knew where to go and why to go there. The x position and y position are checked using modulo "%". This basically checks the remainder of dividing chars position by tile size. When x and y are in the center of tile, then dividing them with tileW gives remainder 0 and we start to pick new direction. Advanced pathfinding algorithms are available for Flash too, mainly A*. Most of them are going to take a lot of time to calculate path from point A to point B. Our method here is very basic and doesnt find any paths, hero moves in one direction until it reaches the target coordinate or wall. Then he moves vertically same way.
To know which way to turn, lets look at the going right decision:

```
if(game["t_"+ob.ytile+"_"+(ob.xtile+1)].walkable and game.targetx>ob.xtile){
```

We check if the tile 1 step right from the heros current tile, has walkable property set to true. And we also check if the mouse was clicked right from the hero.
Thats all for today about controlling the hero with mouse.

You can download the source fla with all the code and movie set up here.

Isometric mouse

Now that we know how to make isometric view and how to control the hero with mouse, we should go to sleep and forget all about those... Wait, no, I wanted to say we should now combine isometrics and mouse. Hey, wake up!

Nothing really new here, we take isometric tutorial and change keyboard control to the mouse control.

Converting from isometric

The only interesting question in this chapter is how to convert mouse coordinates from the screen to the tiles so we know which tile player has clicked. As you might remember from the previous chapter, we used:

```
game.xmouse=Math.round((_root._xmouse-game.tileW/2)/game.tileW);
game.ymouse=Math.round((_root._ymouse-game.tileH/2)/game.tileH);
```

If you ever wondered why we used those lines, then no, its not because the code looks good and is long enough to impress your girlfriend. The reason we used that code was mainly because we had placed the hero on correct place using lines:

```
char.x = (char.xtile *game.tileW)+game.tileW/2;
char.y = (char.ytile *game.tileW)+game.tileW/2;
```

Dont be shy, look at the two pairs. I can even rewrite the lines for clarity. Lets take the code with char.x, if we replace strange names with simple letters, it says:

```
a=b*c+d
```

Now for us to find mouse coordinates, we need to get letter "b" from that equation:

```
b=(a-d)/c
```

and this is exactly, what we have used for mouse. OK, but we were here to talk about isometric. In isometric view we cant get the tile clicked with mouse using same code because we have placed tiles in different way. All the tiles are placed in isometric using code:

```
xiso=x-y
yiso=(x+y)/2
```

In order to find out, what tile has been clicked, we need to find variables "x" and "y" from those equations. Lets rewrite first line:

```
x=xiso+y
```

Now replace the equation for x into second line:

```
yiso=(xiso+y+y)/2
```

which can be rewritten couple of times:

```
yiso=(xiso+2*y)/2
2*yiso=xiso+2*y
2*y=2*yiso-xiso
y=(2*yiso-xiso)/2
```

And we have created two lines to calculate tile in isometric space from the screen coordinates:

```
y=(2*yiso-xiso)/2
x=xiso+y
```

Actual code

In the work function use this code to find out the isometric tile under the mouse:

```
var ymouse=((2*game.clip._ymouse-game.clip._xmouse)/2);
var xmouse=(game.clip._xmouse+ymouse);
game.ymouse=Math.round(ymouse/game.tileW);
game.xmouse=Math.round(xmouse/game.tileW)-1;
```

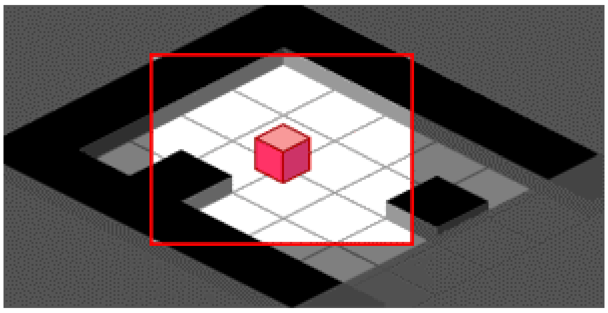
Im sure you can see the similarities with 2 lines we created before. Variables xmouse and ymouse have values where mouse would have been, if we wouldnt be silly enough to start with all this isometric stuff.
Remember, dont use _root._xmouse because our "tiles" movie clip was moved (game.clip._x=150) and we want to get mouse inside "tiles" movie clip. If you use _root._xmouse, then the tile with x=0 would be placed in the left side of the stage, but isometric view places that tile about in the center of stage.

You can download the source fla with all the code and movie set up here.

Isometric scroll

Isometric scrolling is no different from the normal top-down view. With very little trouble we can easily combine scrolling engine (chapter 13) with isometric (chapter 16). This will result into isometric scrolling with diamond shaped view.

We will count the movement of the char exactly like in the normal scroll and when tile has gone too far, we will move it to th other side. And after all the calculations are done and positions found, we convert it to the isometric view. Im not going into exact code here, as the idea is explained before and you can look it up from the fla too.
You can download the source fla with all the code and movie set up here.
Diamond view has 1 big flaw: its shaped like diamond. Yes, diamonds are girls best friend and I have nothing against them personally, just your monitor is not shaped like diamond (or if it is, could you please send me picture of it). Your game will be shaped like rectangle too. Surely you can just create more tiles to cover the whole stage area, plus lot of tiles outside the stage.

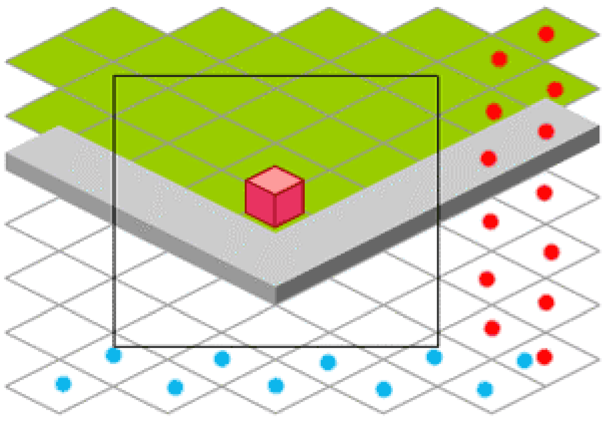


It doesnt look bad in the picture, but it will get worse when you make your game bigger. More extra tiles will be around to eat away those precious CPU cycles. If your game is small enough and you dont want to bother too much with the more complex system, use simple isometric scroll. Rest of us, lets move on.

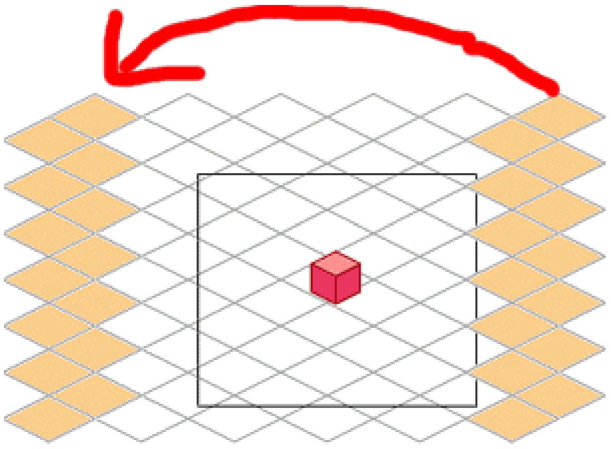
Shape it like rectangle

We can get rid of all the extra tiles with some more work. This movie doesnt create tiles in the same way as earlier examples, you can see the tiles being moved behind the gray frame.

The code behind this movie is getting long and complicated so I try to explain the idea with less code samples. Please download the fla for the working actionscript.
The basic idea here is to fill the visible area with isometric tiles and not to have more tiles, that cant be viewed:



There is one extra row of tiles (marked with blue dots) and one column (red dots) to make sure tiles can be moved and they still fill the visible area. Now when our tiles have been moved right enough, we will take the whole column and move it straight across the stage (not in isometric angle!) to the left side. Same way with up/down movement.



To achieve this, we will set up bunch of new objects to hold information about the tiles. When building the map in the beginning the tile in the center (where hero stands) becomes "t_0_0" no matter on which tile the hero actually stands on.

```
if(y%2!=0){
  var noisox = x*game.tileW*2+game.tileW+ob.xiso;
}else{
  var noisox = x*game.tileW*2+ob.xiso;
}
var noisoy = y*game.tileW/2+ob.yiso;
```

Because we want the isometric tiles to tile up nicely, we have to shift every other row to the right by half the actual tile width.
Next we will use the formulas from the iso mouse tutorial to find out which tile in isometric space should be shown it that position.

```
var ytile = ((2*noisoy-noisox)/2);
var xtile = (noisox+ytile);
ytile = Math.round(ytile/game.tileW);
xtile = Math.round(xtile/game.tileW);
```

We continue same way as in non scrolling isometric system, create new isometric object from the map data, find out its depth etc. But in addition we also create non_iso object, that remembers all the connections between isometric and normal objects and movie clips.
When moving the row or column of tiles over the stage with changeTile function, we use same method: move tile into new position, find out which tile should be shown there, create new objects and delete the old ones.
Now 2 warnings to look out for. First, since we have used another movie clip for the walkable tiles to avoid all the depth problems (game.clip.back), we might find that new position of the moved tile needs to be not walkable. Then we have to actually delete the old movie clip and attach new one in the game.clip. That is handled by saving the tiles _parent clip name with the no_iso object and comparing it later with new position for that tile movie clip.

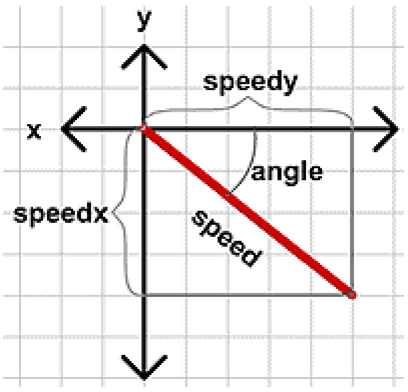
Second problem is that Flash doesnt like movie clips in the negative depth (read more here). Tiles can actually be placed in negative depths, but we wont be able to remove them later. Some of our tiles might end up in the depths like -3456. To avoid this problem I have added +100000 to each depth calculation (dont forget the hero too).

You can download the source fla with all the code and movie set up here.

Rotate the Hero

Hopefully you have noticed how our hero has been so far walking straight. He could walk straight to the left or straight to the right, but not in any angle. Now we will allow player to rotate the hero and walk in any direction. Play with this example, left and right arrow keys to rotate, up and down arrow keys to move.

For this kind of game we only need 1 view on the hero as he is always looked directly down. So, you can get rid of all other frames in the char movie clip except the one, where hero faces right. Moving right is default position for movement, when angle of rotation is 0. Dont forget, the hero is still looked straight down, but he should be placed so he looks at the right.
For movement in any direction, we will need 2 variables, the angle of the movement and the value of the movement. We will use rotation of the hero movie clip for the angle and variable "speed" in the char object for the value of movement. When we know angle and value, we can find out x and y components from those:



To project speed vector with the known length and angle to the x and y axis, we can use these lines:

```
speedx = speed*Math.cos(angle);
speedy = speed*Math.sin(angle);
```

Dont forget the weird thing in Flash, that _rotation property is given in degrees and Math methods expect the angle to be in radians. If we use _rotation as angle, we must first convert it into radians using:

```
anglerad = _rotation*Math.PI/180;
```

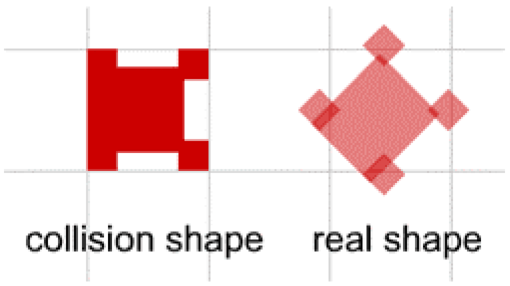
Ready to rewrite key detection function? What we want, is the char movie clip to rotate when left or right arrow keys are pressed and move when up or down arrow keys are pressed:

```
if (Key.isDown(Key.RIGHT)) {
  ob.clip._rotation +=5;
} else if (Key.isDown(Key.LEFT)) {
  ob.clip._rotation -=5;
}
if (Key.isDown(Key.UP)) {
  ob.speedx=ob.speed*Math.cos((ob.clip._rotation)*Math.PI/180);
  ob.speedy=ob.speed*Math.sin((ob.clip._rotation)*Math.PI/180);
  keyPressed = _root.moveChar(ob, ob.speedx, ob.speedy);
}else if (Key.isDown(Key.DOWN)) {
  ob.speedx=-ob.speed*Math.cos((ob.clip._rotation)*Math.PI/180);
  ob.speedy=-ob.speed*Math.sin((ob.clip._rotation)*Math.PI/180);
  keyPressed = _root.moveChar(ob, ob.speedx, ob.speedy);
}
```

When up or down arrow keys are pressed, we first calculate the x and y movement values and then we call the moveChar function using those. Since we are now passing both values to the moveChar function, we also have to change some code in that function. So far our moveChar function has always got only 0, 1 or -1 in the x or y directions (refresh your memory from tutorial 4).
The number 5 here is only as an example, you can use other numbers to change the rotation. Try to use numbers that create full circle in the end (360 degrees). So 2, 4, 6, 12 and 90 are good, but 13.5 and 7 would be bad choice.
Remove all the ob.speed parts from that function:

```
function moveChar (ob, dirx, diry) {
  getMyCorners(ob.x, ob.y+diry, ob);
  if (diry < -0.5) {
    if (ob.upleft and ob.upright) {
      ob.y += diry;
    } else {
      ob.y = ob.ytile*game.tileH+ob.height;
    }
  }
  if (diry > 0.5) {
    if (ob.downleft and ob.downright) {
      ob.y += diry;
    } else {
      ob.y = (ob.ytile+1)*game.tileH-ob.height;
    }
  }
  getMyCorners(ob.x+dirx, ob.y, ob);
  if (dirx < -0.5) {
    if (ob.downleft and ob.upleft) {
      ob.x += dirx;
    } else {
      ob.x = ob.xtile*game.tileW+ob.width;
    }
  }
  if (dirx > 0.5) {
    if (ob.upright and ob.downright) {
      ob.x += dirx;
    } else {
      ob.x = (ob.xtile+1)*game.tileW-ob.width;
    }
  }
  ob.clip._x = ob.x;
  ob.clip._y = ob.y;
  ob.xtile = Math.floor(ob.x/game.tileW);
  ob.ytile = Math.floor(ob.y/game.tileH);
  return (true);
}
```

I have added check for each movement value to be at least 0.5 pixels. When calculating speedx and speedy using sin and cos functions, Flash will return very small values instead of 0. Something like 0.00000000000000001255. You cant detect so small movement on the screen so we will ignore it.
As you can see, when the hero is rotated, it can move its corners by small amount into the wall tiles. This is happening because we are not taking into account its rotation when calculating the corner points. As long your hero is shaped like a square the corners wont go too much into walls, but when you use very tall rectangle for the hero the bug becomes obvious.



You can fix this easily by updating the width and height of the char object every time it is rotated. Place 2 lines in the left and right key detection code after changing the rotation:

```
char.width = char.clip._width/2;
char.height = char.clip._height/2;
```

This will take care of those pesky corners (yay!) but it will create another bug (bummer). You see, we only check for the corners when hero moves, but we dont check for corners when he rotates. This will allow the hero to rotate near the wall while standing so its corner goes into wall and when he starts to move finally, its position is updated and movement might look jumpy. Im not sure which is better way so in the fla I have let the corners go into walls.

You can download the source fla with all the code and movie set up here.
In the next chapter we will have even more fun by rotating entire background.

Rotate the background

In the last chapter our hero gained the ability to rotate and walk in any angle. While the hero is surely very thankful for it (and lets be honest, who wouldnt) we can make more interesting by scrolling and rotating the background. The hero will stay in place, but the background is rotated with left/right arrow keys and like rotatable background isnt enough to make human race happier, the background will also scroll with up/down arrow keys.

If you look at the fla in Flash preview, you see how whole map is actually drawn out, but only small portion is visible at any time. So be careful when experimenting with huge maps with big visible area, your game may be too slow to play. Dont forget that each tile Flash has to draw/move/rotate needs another cycle from poor old CPU and most people playing your game dont yet have latest 10GHz super-computers (no, I havent actually asked most people in the world what kind of computers they use, but I still believe they dont have latest computers. I could be wrong of course).
First we need to rotate the tiles. We do have "tiles" movie clip with all the tiles already and you might think we can simply rotate the "tiles" mc. Thats wrong! World is so unfair sometimes... The problem is, that all movie clips in the Flash are rotated around their center point. Our tiles movie clip has center point in the upper left corner and so it would be rotated around top left corner. Unless you want the game, which always rotates around left top corner (I havent seen such game so far), we need to add another holder movie clip. We will place "tiles" mc inside this holder mc and then we can rotate the holder mc without worrying about the left top corner anymore.
In the start of builMap function add code:

```
_root.attachMovie("empty", "rotate_tiles", 1);
_root.rotate_tiles._x=120;
_root.rotate_tiles._y=140;
_root.rotate_tiles.attachMovie("empty", "tiles", 1);
game.clip = _root.rotate_tiles.tiles;
```

The "rotate_tiles" is the movie clip, that will be rotated. It wont move around, so place it in the center of your stage horisontally. I have also placed near the bottom of the stage, so the hero will have more visible area in front of him and less behind. Thats up to you how you want to position the rotated tiles, fortunately you can change its position around as much as you want.
Because we will have scrolling background, our hero will stay in same spot, but the background (game.clip mc) is moved in opposite direction. So we have to place it in the beginning to. Add to buildMap function:

```
game.clip._x = -(char.xtile*game.tileW)-game.tileW/2;
game.clip._y = -(char.ytile*game.tileH)-game.tileH/2;
```

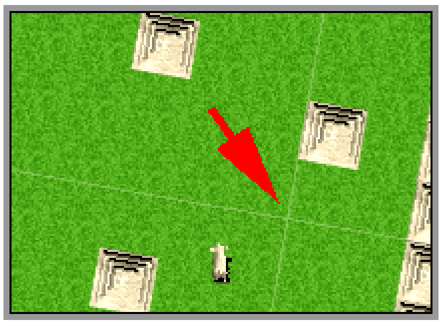
The (char.xtile*game.tileW)-game.tileW/2 is where the hero will be placed and tiles mc is moved by same amount to the completely other way.
We also need to add 2 lines in the very end of moveChar function to scroll the background after we have calculated the new position for the hero:

```
game.clip._x = game.centerx-ob.x;
game.clip._y = game.centery-ob.y;
```

Now for the actual rotation of the tiles, we simply add 1 line to rotate the added holder movie clip in other way. As we have declared game.clip inside this holder mc, we can use _parent property to access it:

```
if (Key.isDown(Key.RIGHT)) {
  game.clip._parent._rotation -=5;
  ob.clip._rotation +=5;
} else if (Key.isDown(Key.LEFT)) {
  game.clip._parent._rotation +=5;
  ob.clip._rotation -=5;
}
```

And thats it for today.
Tip: when rotated, your tiles might create some ugly-looking lines between them:



You can fix it with increasing the size of your tiles by 1 px. In our example the tiles would still be placed using 30px space, but their actual size would be 31px.

You can download the source fla with all the code and movie set up here.

Pathfinding

One thing common so far with all our examples is how the hero is dumb. If you tell him to move left, he would go left. Surely being dumb is nice when you are in the army, but our hero should have some sort of brains to find its way through the dangers of the world. So, lets put in some pathfinding.
Before we do, perhaps it would be good idea to say out loud and clear, what is pathfinding. Pathfinding is finding the path from tile A to the tile B. Phew, that went well. And didnt take days.

Many paths

Pathfinding is actually very complex thing. But its also very useful thing. We can use same system not only for the hero, but for every other object in the game too. Imagine the game with many enemies, they all want to find path to our hero and eventually kill the hero (they wouldnt be enemies, if they wouldnt want to kill the hero).
In this example we expect the hero to find path from the tile he currently stands on, to the path where the mouse was clicked. Dont be fooled to mix up any path with shortest path. Any path can be found even when we wonder around the map randomly (like our enemies did in the chapter 9). It might take time, but eventually the path from tile A to the tile B is found. Now the shortest path means we find path that takes less steps then any other path. Humans do that all the time, if you go home from work, you will find shortest path between work and home and use it. Unless you wish to visit the pub in your way home.
In some case tile might not be equal in number of steps they take to walk over them. Like you could have road and swamp and forest tiles and surely walking on the road tile takes less time then walking on the swamp. I wont go into this kind of situation too deep, but you will need an A* (A star) algorithm to find paths in such maps. Several A* codes for Flash are freely available in the internet, just search for them.
Pathfinding can take very long time. Specially on the bigger maps, with lots of tiles to search through, it can take several seconds to find the path. As you know, Flash is not extremely powerful and if your game stops every now and then to find paths, its not playable anymore. Before you add pathfinding to your game, think. Do you really need it? Can you make it faster? Can you cheat to create not-so-perfect-but-still-nice-paths?

Breath first

The pseudo code for the breadth-first search is very clearly explained in this link and please read the great article here. My example follows the pseudo code.
I will quote from the page above (skip it if you did read the explanation from the link):
"This is a very expensive, but relatively clear algorithm. It begins with a starting point Source, and finds a path to a goal Target. The basic approach is to take each neighbour of the starting point, and then take each neighbour of those neighbors, and then the neighbours of those neighbours, and so on, expanding the tiles (nodes) being considered until finally one of them is the Goal. Each time a node is looked at, its neighbours are pushed onto a queue; thus the order in which the nodes are considered is one neighbour after another. Where numbers indicate the order in which the nodes are considered, the pattern is:

```
 __ __ __ __ __
|  9|10|11|14|15|
|__|__|__|__|__|
|12|  1|  2|  3|16|
|__|__|__|__|__|
|13|  4|  0|  5|17|
|__|__|__|__|__|
|18|  6|  7|  8|19|
|__|__|__|__|__|
|20|21|22|23|24|
|__|__|__|__|__|
```

That is, we begin with 0. All of its neighbours 1-8 are put on the queue. Then the first element (1) is taken off the queue, and its neighbours 9-13 are put on the queue. The next element (2) is then taken off the queue, and its neighbours (i.e. 14) are put on the queue. Then (3) is taken off the queue, and its neighbours 15-17 are put on the queue; (4) is taken off, and its neighbours 18 are put on; and so on."

Coding breath first

We will use the example from the chapter 17 as the base. Build the map same way as before, using buildMap function. No changes needed in the work function either. First new thing will be in the getTarget function. After we have made sure walkable tile was clicked, we update the target tile and call new findPath function:

```
function getTarget() {
  if(game["t_"+game.ymouse+"_"+game.xmouse].walkable){
    game.targetx=game.xmouse;
    game.targety=game.ymouse;
    if(!char.moving){
      findPath(char.xtile, char.ytile, game.targetx, game.targety);
    }else{
      game.clicked=true;
    }
  }
}
```

We are checking if the hero is currently moving from one tile to another. Thats because we will let the player click any time on the stage, even when hero is currently moving from one tile to another, but hero will first finish its current movement and reach the center of the tile. When on the center, we will check for game.clicked and call findPath

function if user has clicked.
As you might have guessed by now, the findPath function is going to do main work in the finding path.

```
function findPath(startx, starty, targetx, targety){
 path={};
 path.Unchecked_Neighbours=[];
 path.done = false;
 path.name="node_"+starty+"_"+startx;
 path[path.name]={x:startx, y:starty, visited:true, parentx:null, parenty:null};
 path.Unchecked_Neighbours[path.Unchecked_Neighbours.length]=path[path.name];
 while(path.Unchecked_Neighbours.length>0) {
  var N = path.Unchecked_Neighbours.shift();
  if (N.x == targetx and N.y == targety) {
   make_path(N);
   path.done = true;
   break;
  }else {
   N.visited=true;
   addNode (N, N.x+1, N.y);
   addNode (N, N.x-1, N.y);
   addNode (N, N.x, N.y+1);
   addNode (N, N.x, N.y-1);
  }
 }
 delete path;
 if (path.done) {
  return true;
 }else {
  return false;
 }
}
```

Lets see what going on here. The function will receive 4 variables: x and y coordinates of both start and target tiles. Then we create new "path" object and Unchecked_Neighbours array. "path" object is our temprorary object to hold every kind of bits and pieces while we search for the path. In the end of pathfinding we delete "path" object and leave everything clear again.
"done" property is used in the end to check if we have actually found the path (done is true) or has player been foolishly clicking on some tile, which no path is reaching (done is false).
Then we create our first node on the start position and add it to the Unchecked_Neighbours array. Each node object will have several important properties. node.x will hold its x position and node.y its y position. node.visited will be true, when the node has been already processed, this way we dont go back to the tiles we already searched. node.parentx and parenty will refer to the node, where we reached on the

current node. If for example we stepped on the tile 1_2 from the tile 1_3, then its parentx will be 1 and parenty will be 3. Its important to remember the parent node, since that will allow us to actually build the path array later, when we have found the path.
Now we start the main while loop, which will continue until Unchecked_Neighbours array is empty. We then take first element from the Unchecked_Neighbours array (shift command) and check if that node is the target. If it is the target, we have found the path and will call function make_path, set "done" property to true and break the loop.
If unfortunately it wasnt the target, then we have to continue looking. We set node's "visited" property to true and add all its neighbours using addNode function.

```
function addNode (ob, x, y){
  path.name="node_"+y+"_"+x;
  if(game["t_"+y+"_"+x].walkable){
    if (path[path.name].visited != true) {
      path[path.name]={x:x, y:y, visited:false, parentx:ob.x, parenty:ob.y};
      path.Unchecked_Neighbours[path.Unchecked_Neighbours.length]=path[path.name];
    }
  }
}
```

The addNode function gets "ob" as the current node and x/y for the new node we are about to add. We only create new node if that tile is walkable and is not visited yet. New node will then get its parentx and parenty properties from the ob node.
To finish up our path, we have to actually create path array for our hero to use. Dont confuse this path array with the path object used in the findPath function. And if you wish to find many paths for several moving objects, it would be better to attach path array into each object and not into game object as we have done it here:

```
function make_path(ob){
  game.path=[];
  while (ob.parentx!=null){
    game.path[game.path.length]=ob.x;
    game.path[game.path.length]=ob.y;
    ob=path["node_"+ob.parenty+"_"+ob.parentx];
  }
  char.moving=true;
}
```

The while loop will continue until node's parentx is null, meaning it is the original starting node, where hero currently stands. We add each node's x and y position to the path array and then make its parent the current node. In the end we set char to move again.
SInce we add both x and y position separately, the path array will be holding information in some kind of this format:

```
[targetx, targety, many steps here, firststepx, firststepy]
```

We also have to use the path array in the moveChar function. After we calculate the tile where chars center is (values for xtile and ytile), change the code:

```
if(game.clicked){
  game.clicked=false;
  findPath(char.xtile, char.ytile, game.targetx, game.targety);
  return;
}
if(game.path.length>0){
  game.targety=game.path.pop();
  game.targetx=game.path.pop();
  if(game.targetx>ob.xtile){
    ob.dirx=1;
    ob.diry=0;
  }else if(game.targetx<ob.xtile){
    ob.dirx=-1;
    ob.diry=0;
  }else if(game.targety>ob.ytile){
    ob.dirx=0;
    ob.diry=1;
  }else {
    ob.dirx=0;
    ob.diry=-1;
  }
}else{
  ob.moving=false;
  return;
}
```

Here we check if the player has clicked on some tile while we were busy moving from one tile to another. If the player really has clicked (the players are like this, they click all the time), we call findPath function and return.
But if player hasnt clicked we check if we have some path left to continue moving. So, if path array contains some elements, we remove LAST element from the array using pop command and assign it to the targety. Please pay attention here, last element goes to y and then we take last element again and assign it to targetx. Now we compare current position of the hero with next tile he should step on and change dirx/diry to make hero move correctly.
Please do consider that this search is extremely slow. It might even crash the Flash player if you use it on the large maps. You might want to add some timer into findPath loop to break it when enough steps has been passed, but no path has been found yet. In large maps it might also be good idea to use waypoints with precalculated paths to spend less time on the search for the path.

More pathfinding

The breath-first search we created in last chapter, is not too fast. Thats why this time we look into faster algorithm, that would allow usage of larger maps without slowing down the game:.

Best-first

You may remember from the last chapter, how breath-first search expanded all the nodes in every direction. It had no idea, where the target was, it looked everywhere. Thats why it took forever to find the path. It did find shortest path every time, but lets be honest, whats more important, finding perfect path or making playable game?
So, lets introduce best-first search. This time we will try to see how far the target is from current node and we attach the estimated distance with each node.

```
cost = Math.abs(x-targetx)+Math.abs(y-targety);
```

We count how many steps is target from the current tile both in x and y direction and add steps from both directions.
The other difference is, that we keep our Unchecked_Neighbours array sorted from the node with lowest cost to the target (it should be near the target) to the node with highest cost. By sorting the array we always look into the direction of the target, before going in other directions. But be warned, while the best-first search always finds the path, its not always the shortest path. Its still much faster in most maps then A*.

Best-codes

Take code from last chapter (tut22) and change the findPath function. We need to add cost to the first node:

```
var cost = Math.abs(startx-targetx)+Math.abs(starty-targety);
path[path.name]={x:startx, y:starty, visited:true, parentx:null, parenty:null, cost:cost};
```

and we have to pass the target to the addNode function:

```
addNode (N, N.x+1, N.y, targetx, targety);
addNode (N, N.x-1, N.y, targetx, targety);
addNode (N, N.x, N.y+1, targetx, targety);
addNode (N, N.x, N.y-1, targetx, targety);
```

In the addNode function remember, that target is passed to it too:

```
function addNode (ob, x, y, targetx, targety){
  path.name="node_"+y+"_"+x;
  if(game["t_"+y+"_"+x].walkable){
    var cost = Math.abs(x-targetx)+Math.abs(y-targety);
    if (path[path.name].cost > cost or path[path.name].cost==undefined) {
      path[path.name]={x:x, y:y, visited:false, parentx:ob.x, parenty:ob.y, cost:cost};
      for(var i=0; i<path.Unchecked_Neighbours.length; i++){
        if (cost<path.Unchecked_Neighbours[i].cost){
          path.Unchecked_Neighbours.splice(i, 0, path[path.name]);
          break;
        }
      }
      if (i>=path.Unchecked_Neighbours.length) {
        path.Unchecked_Neighbours[path.Unchecked_Neighbours.length]=path[path.name];
      }
    }
  }
}
```

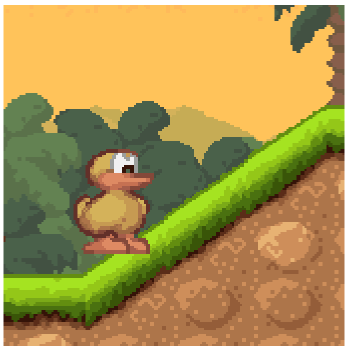
Cost is calculated same way. Then we check if the node is already created or if it is created, but has currently higher cost.
After making new node we start to loop through the Unchecked_Neighbours array and compare cost of current node with cost of each element in the array. We break the loop, if we have found node in the array, that has higher cost. We insert new node in that spot into array.
Last if statement checks if we have looped through entire array without finding any node with higher cost. That means we add our new node in the end of the array.

Faster, faster

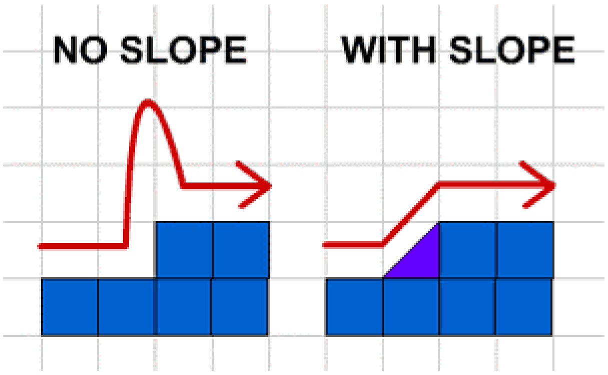
If the pathfinding still takes too long, because you have large maps, you might consider precalculating some parts of the maps or using waypoints to lead char from one part of the map into another part without looking through all the tiles.
Michael Grundvig has created extremely fast pathfinding system using precalculated paths. You can read about it here. He calculates all the paths from each tile to every tile and stores those paths with the maps. When char wants to go from one tile to another during the game, he only has to pick the path and walk.
Waypoints system would slice the map virtually up into smaller maps, connected with prededefined paths. Then you would need to find only which minimap starting tile and target tile belong and use memorised path to get from starting minimap to target. That allows paths for really-really big maps too.
Another way for calculation of larger paths without slowing down the game would be spreading the path calculation over several frames. You would need to break the pathfinding loop after certain steps, remember the current state, then run other code in the game and in the next frame continue finding the path. Andre Michelle has posted very nice example of this idea here.

Slopes

Many people have asked me "How do I make my hero walk on the sloped tiles?". And I usually reply "Why on earth would you want your hero to walk on sloped tiles? Are you not happy with the rectangular tiles? Cant your hero just jump on the higher tiles?" And then they say "No, I must have sloped tiles".
Perhaps you do not know, what sloped tiles are. In this picture hero (the Duck named Charlie) is walking on sloped tile:



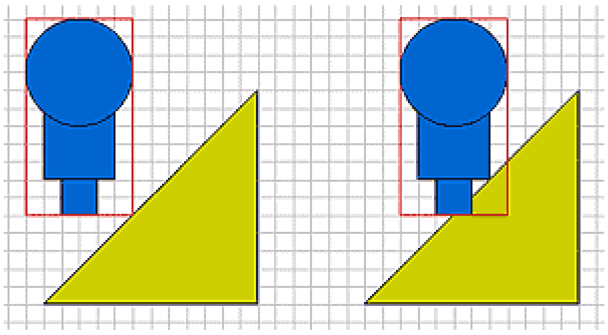
Screenshot from the game "CHARLIE II" by Mike Wiering / Wiering Software
The slope allows our hero to get on higher (or lower) height simply by walking right (or left) without jumping (or falling). So, the slope we will talk about, is connecting 2 tiles with different heights:



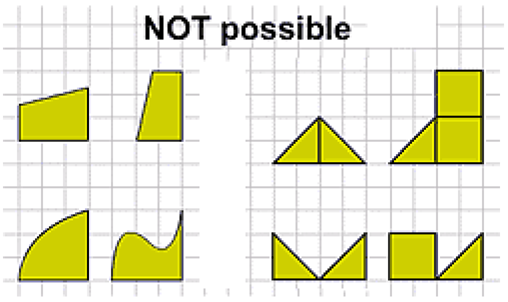
When hero in left picture wants to continue moving right, he has to jump on the higher tile. But thanks to the slope, hero on the right picture is not forced into jumping. Sure, if our hero is jumper-type of hero, he could still jump in the right picture too, but normal heroes are very happy, if they can avoid jumping.

Problems

As soon as we want to add sloped tiles, we will face several problems (I bet you didnt expect any problems). First, the placement of hero on the sloped tile. If you remember, our hero is basically rectangle, all the movement and collisions are based on his corner points. We cant use same idea on the slope, since most heroes would end up standing on the slope without their feet touching the ground. Instead, we have to place center of hero on the sloped ground and move part of hero inside the ground.



Next, the slopes in our game must always be from one corner of the tile to the opposite corner and you should be careful how you place the slopes. You cant make slopes with some other angle or shape and you cant make strange maps with slopes.



Code it

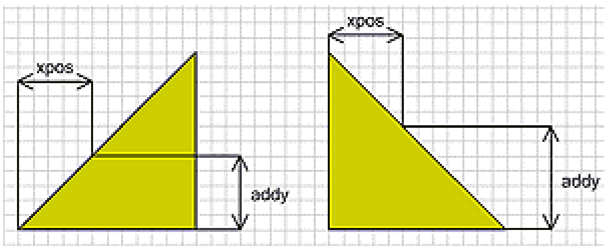
Start with the code from tutorial 7 Jumping.
Declare new tile prototypes:

```
game.Tile4= function () {};
game.Tile4.prototype.walkable=true;
game.Tile4.prototype.slope=1;
game.Tile4.prototype.frame=4;
game.Tile5= function () {};
game.Tile5.prototype.walkable=true;
game.Tile5.prototype.slope=-1;
game.Tile5.prototype.frame=5;
```

Tile4 has slope moving upward (/) and tile5 has slope moving down (\). Draw the slopes in frames you have set with frame property.
New functions are such nice things. They are fresh, smell good and do things you never new could be done. Lets make checkForSlopes function:

```
function checkForSlopes (ob, diry, dirx) {
  if (game["t_"+(ob.ytile+1)+"_"+ob.xtile].slope and !ob.jump){
    ob.ytile += 1;
    ob.y += game.tileH;
  }
  if (game["t_"+ob.ytile+"_"+ob.xtile].slope and diry != -1){
    if (diry == 1){
      ob.y = (ob.ytile+1)*game.tileH-ob.height;
    }
    var xpos = ob.x-ob.xtile*game.tileW;
    ob.onSlope = game["t_"+ob.ytile+"_"+ob.xtile].slope;
    ob.jump = false;
    if(game["t_"+ob.ytile+"_"+ob.xtile].slope == 1){
      ob.addy = xpos;
      ob.clip._y = (ob.ytile+1)*game.tileH-ob.height-ob.addy;
    }else{
      ob.addy = game.tileW-xpos;
      ob.clip._y = (ob.ytile+1)*game.tileH-ob.height-ob.addy;
    }
  }else{
    if((ob.onSlope == 1 and dirx == 1) or (ob.onSlope == -1 and dirx == -1)){
      ob.ytile -= 1;
      ob.y -= game.tileH;
      ob.clip._y = ob.y;
    }
    ob.onSlope = false;
  }
}
```

This function will be called from moveChar function with movement variables dirx and diry. First if statement checks for the slope on the tile below current tile. This is for the situation, where hero currently stands on unwalkable tile, but moves left or right and there is slope going down from his current height. If there is slope below hero, we increase the ytile and y. However, we will not check for it, if hero is jumping.
Next if statement checks for the slope on the tile hero currently is. The diry!=-1 part ignores the check, if SPACE key has been pressed and hero jumps up.
If we were falling down (diry==1), we will set the y property as if hero would of landed on tile below. We set jump property to false and onSlope property equal to the slope value on current tile (1 or -1).
xpos is the value of how far from the left edge of current tile center of our hero is:



If slope is going up, then we move hero up by the value of xpos, if its going down, then by the value of tileW-xpos. Note that if you dont use square tile, then you would need to find xpos as percentage from tileW.
Last part after else statement checks if we were standing on slope, but now we have moved off from it onto higher tile.

Next take moveChar function. Modify the check for going left and right:

```
//left
if ((ob.downleft and ob.upleft) or ob.onSlope) {


//right
if ((ob.upright and ob.downright) or ob.onSlope) {
```

Here we will basically ignore all collision checks for left/right movement as long hero is on the slope. Remember, while on slope, he is standing partially inside the wall, so we cant use normal collision with his corners.
After placing the hero movie clip, call checkForSlopes function:

```
ob.clip._x = ob.x;
ob.clip._y = ob.y;
checkForSlopes(ob, diry, dirx);
```

When we jump while standing on the slope, we have to update the y coordinate of the hero. Modify detectKeys function:

```
if (Key.isDown(Key.SPACE)) {
  if (!ob.jump){
    //if we were on slope, update
    if (ob.onSlope) {
      ob.y -=ob.addy;
      ob.ytile = Math.floor(ob.y/game.tileH);
    }
    ob.jump = true;
    ob.jumpspeed = ob.jumpstart;
  }
}else if (Key.isDown(Key.RIGHT)) {
  keyPressed=_root.moveChar(ob, 1, 0);
}else if (Key.isDown(Key.LEFT)) {
  keyPressed=_root.moveChar(ob, -1, 0);
}
```

If hero has onSlope property set to true, we will first update its y property and calculate new value for the ytile.

You can download the source fla with all the code and movie set up here.