

# Sokoban

Sokoban is a tile-based puzzle game in which the player is placed in a maze and must push all crates placed around the level in their designed locations. Only one crate may be pushed at a time, and crates cannot be pulled. The level is completed when all crates are placed in their designed locations, trying to do it with the lowest number of moves and pushes possible.

It's a tile-based game in which the player can interact and modify the game field (pushing crates) turning a solvable level into an unsolvable one. For this reason, most Sokoban games allow the player to undo a move or at least to reset the level. This is not considered cheating in Sokoban as the player cannot die. He just can solve the level or not.

During this chapter, you will create a complete Sokoban game prototype using these main techniques:

- Inserting values in arrays in various ways
- Using the `switch` statement
- Getting the absolute value of a number
- Detecting keys pressed
- Passing variables by value or by reference

Also, this is the longest chapter of the book, so prepare for a long journey that will bring you to the creation of Sokoban.

## Defining game design

Sokoban is the first game of this book featuring a character controlled by the player. This means controls will be more complex than a simple "select a card/mine/column with the mouse" you have used to manage until now.

To move a character around a tile-based environment, the best device is the keyboard. And the golden rule when you're about to program a keyboard game is "make everybody happy". What does it mean?

In some games, the character is controlled by arrow keys. In some other games, by the so-called "WASD" keyboard, which replicates the same functionalities of arrow keys with W (up), A (left), S (down), and D (right) keys. Some people prefer arrow keys to play, others use only WASD, especially when playing on netbooks.

Not every game on the scene is offering both ways to control the character. This game will include both arrow and WASD key controls. It's an easy way to make everybody happy.

The keys used in this game will be:

- Arrow keys or WASD to move the character
- U to undo a move
- R to restart a level
- N and P to play respectively next and previous level

I have to say, I hate skipping levels on a game. I would like players to use the brain to solve levels rather than just say "oh well" and skip to the next level. Anyway, Flash games are in most cases played during a coffee break by people that want to relax for a couple of minutes, so forcing them to solve tricky puzzles will let them quit your game.

It can be a good idea in a puzzle game like this to offer both a collection of easy, skippable levels and a "pro" mode with harder levels players have to beat before trying the next ones.

## Thinking about levels

The game will be structured to virtually have an infinite number of levels. Giving a puzzle game this kind of open structure is very important when you need to update the game with a new level pack.

Anyway, the creation of Sokoban levels is not the aim of this chapter, so the game we'll make will feature only three levels. It's up to you to add as many levels as you want.

The main actors of the Sokoban game are:

1. The wall
2. The crate
3. The crate goal
4. The player

Every tile can have seven situations:

1. An empty tile
2. A wall
3. A crate goal
4. A crate
5. The player
6. A crate over the crate goal
7. The player over a crate goal

A good practice would be to reduce this list as you don't have to deal with a large number of situations.

Let's take a look at the actors list. We can manage all seven situations using only that list, building the list this way:

- The empty tile is marked with zero as there isn't any actor over it.
- The wall is marked with 1 because there's the actor 1 (the wall).
- The crate is marked with 2 because there's the actor 2 (the crate).
- The crate goal is marked with 3 because there's the actor 3 (the crate goal).
- The player is marked with 4 because there's the actor 4 (the player).
- The crate over the crate goal is marked with 5 because there are both the crate goal (2) and the crate (3) and  $3+2=5$ . You can say 5 can also be obtained by 4 (player) + 1 (wall), but since the player can't stand over a wall, it's an illegal value, so the only way to have a 5 would be adding the crate to the crate goal.
- The player over the crate goal is marked with 6 because there are both the crate goal (2) and the player (4). There isn't any other legal combination of actors which can return a 6.

This way we can manage all tile situations just with a combination of main actors.

## Drawing the graphics

There are thousands of basic Sokoban games around the web, most of them using a couple of different colored tiles to represent the entire level.

**The idea:** We'll try to make something more accurate, giving the game some kind of personality. Nobody wants to move a "blue square" pushing "light blue squares" into "dark blue squares". Our Sokoban version will feature a bulldozer pushing crates to their spots. Crates will be red when out of their place, and green when correctly placed over their goal. Moreover, the bulldozer will rotate according to the directions it's moving in.

**The development:** Following the idea explained previously, there are two kinds of objects:

1. Objects representable with only one frame (the wall, the floor, and the crate goal)
2. Objects representable with more than one frame (the crate and the bulldozer)

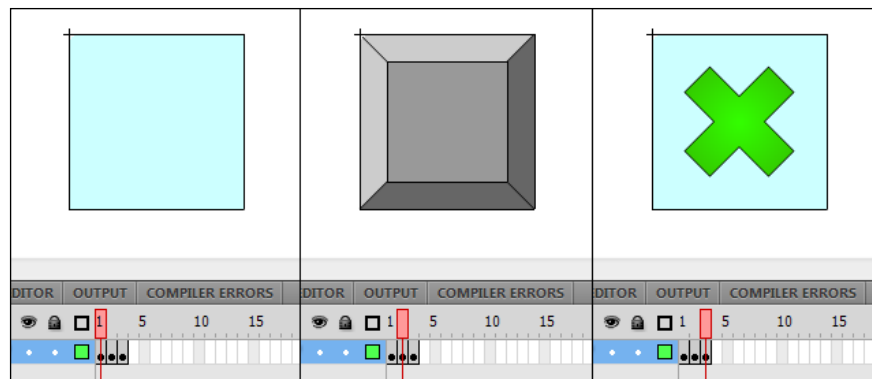
We will create a single movie clip with all objects represented in a single frame, one per each frame, and one movie clip for every object representable with more than one frame, with all frames in its timeline.

## Floor, wall, and crate goal

Create a new file (**File** | **New**) then from **New Document** window select **Actionscript 3.0**. Set its properties as width to 640 px, height to 480 px, background color to #FFFFFF (white), and frame rate to 30. Also define the Document Class as `Main` and save the file as `sokoban.fla`.

In `sokoban.fla`, create a new Movie Clip symbol called `tiles_mc` and set it as exportable for ActionScript. Leave all other settings at their default values, just like you did in previous chapters.

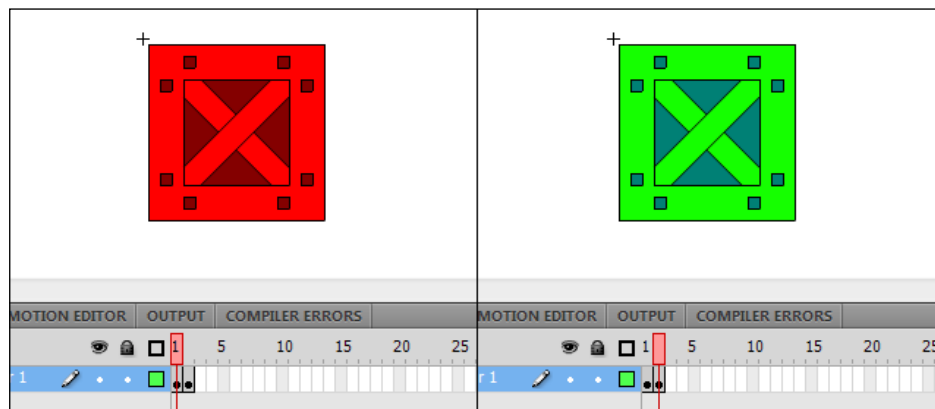
Draw 64x64 pixels tiles with registration point at 0, 0 representing the floor, the wall, and the crate goal respectively at frames 1, 2, and 3.



You can change the sizes and order of these tiles as you want, but during this chapter I will refer to sizes and timelines I am explaining here.

## The crate

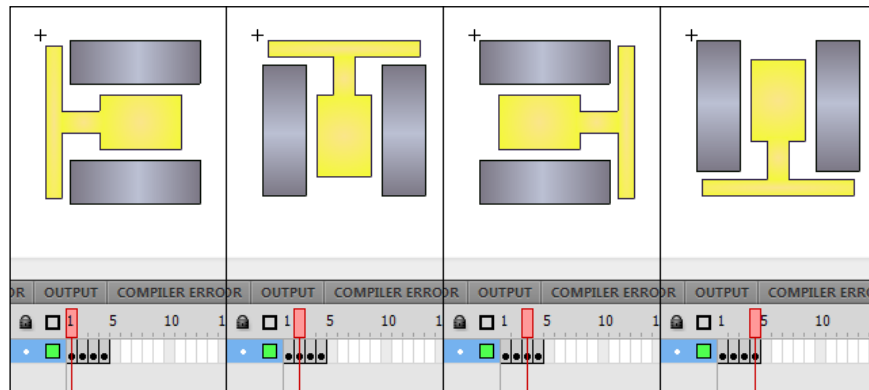
In `sokoban.fla`, create a new Movie Clip symbol called `crate_mc` and set it as exportable for ActionScript. Make it a little smaller than 64x64 pixels but center it in a hypothetical 64x64 pixels square.



Draw the generic crate at frame one, and the correctly placed crate at frame two.

## The bulldozer

In `sokoban.fla`, create a new Movie Clip symbol called `bulldozer_mc` and set it as exportable for ActionScript. As for the crate, make it a little smaller than 64x64 pixels but centered in a hypothetical 64x64 pixels square.



Draw the bulldozer facing left, up, right, and down respectively at frames 1, 2, 3, and 4.

This way we'll make something better than a bunch of pixels moving around the screen.

## Designing levels

In my opinion the best Sokoban levels are tiny but tricky. There are thousands of levels around the web, I suggest you to Google for "sokoban levels" and choose the ones you prefer, paying attention to licenses as not all levels are freely distributable.

**The idea:** Each level can be stored in a two-dimensional array, so the array that contains all levels is a three-dimensional array, where the first index is the level number.

**The development:** Without closing `sokoban.fla`, create a new file and from **New Document** window select **ActionScript 3.0 Class**. Save this file as `Main.as` in the same path you saved `sokoban.fla`. Then write:

```
package {
    import flash.display.Sprite;
    public class Main extends Sprite {
        private var levels:Array=new Array();
        public function Main() {
```

---

```

        setupLevels();
    }
    private function setupLevels():void {

        levels[0]=[ [1,1,1,1,0,0,0,0], [1,0,0,1,1,1,1,1], [1,0,2,0,0,3,0,1],
        [1,0,3,0,0,2,4,1], [1,1,1,0,0,1,1,1], [0,0,1,1,1,1,0,0] ];

        levels[1]=[ [0,0,1,1,1,1,0,0], [0,0,1,0,0,1,0,0], [1,1,1,0,0,1,1,1],
        [1,0,3,5,2,4,0,1], [1,0,0,0,0,0,0,1], [1,1,1,1,1,1,1,1] ];

        levels[2]=[ [1,1,1,1,1,1,1,1], [1,0,0,1,0,0,0,1], [1,0,0,0,0,0,0,1],
        [1,3,3,3,1,1,1,1], [1,6,2,0,2,1,0,0], [1,1,1,1,1,1,0,0] ];
        trace("My levels: "+levels);
    }
}

```

Test the movie and you will see:

```

My levels: 1,1,1,1,0,0,0,0,1,0,0,1,1,1,1,1,0,2,0,0,3,0,1,1,0,3,0,0,2
,4,1,1,1,1,0,0,1,1,1,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,0,0,1,0,0,1
,1,0,0,1,1,1,1,0,3,5,2,4,0,1,1,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1
,1,1,1,1,1,0,0,1,0,0,0,1,1,0,0,0,0,0,0,1,1,3,3,3,1,1,1,1,1,6,2,0,2,1,0
,0,1,1,1,1,1,1,0,0

```

That is the representation of the array we just created. Note how we are inserting values in the array: for the first time we aren't pushing a value at once, but directly filling `levels` with all values.

Let's take the creation of the first level (level zero):

```

levels[0]=[ [1,1,1,1,0,0,0,0], [1,0,0,1,1,1,1,1], [1,0,2,0,0,3,0,1],
[1,0,3,0,0,2,4,1], [1,1,1,0,0,1,1,1], [0,0,1,1,1,1,0,0] ];

```

This could also be defined this way:

```

levels[0]=new Array();
levels[0][0]=[1,1,1,1,0,0,0,0];
levels[0][1]=[1,0,0,1,1,1,1,1];
levels[0][2]=[1,0,2,0,0,3,0,1];
levels[0][3]=[1,0,3,0,0,2,4,1];
levels[0][4]=[1,1,1,0,0,1,1,1];
levels[0][5]=[0,0,1,1,1,1,0,0];

```

by populating the two-dimensional array, and the whole three-dimensional array could be defined this way:

```
levels=[[ [1,1,1,1,0,0,0,0], [1,0,0,1,1,1,1,1], [1,0,2,0,0,3,0,1],  
[1,0,3,0,0,2,4,1], [1,1,1,0,0,1,1,1], [0,0,1,1,1,1,0,0]],  
[[0,0,1,1,1,1,0,0], [0,0,1,0,0,1,0,0], [1,1,1,0,0,1,1,1], [1,0,3,5,2,4,  
0,1], [1,0,0,0,0,0,0,1], [1,1,1,1,1,1,1,1]], [[1,1,1,1,1,1,1,1], [1,0,  
0,1,0,0,0,1], [1,0,0,0,0,0,0,1], [1,3,3,3,1,1,1,1], [1,6,2,0,2,1,0,0]  
,[1,1,1,1,1,1,0,0]]];
```

completely declaring it with a single definition.

I would suggest you to use the way that makes you feel more comfortable and makes the code more readable.

## Adding walls and floor

Let's first create the level structure by adding walls and floor, leaving bulldozer and crates for later.

**The idea:** We need a variable to hold the number of the current level, then we have to scan `levels` array looking for 0 (floor), 1 (wall), and 2 (crate goal), and create the proper `DisplayObjects`.

**The development:** First we'll crate three class level variables:

```
private var levels:Array=new Array();  
private var currentLevel:uint=0;  
private var tile:tiles_mc;  
private var level_container:Sprite;
```

`currentLevel` represents the level we are currently playing on, `tile` will be used to create instances of `tile_mc` and `level_container` is the `DisplayObject Container` of each level.

We also need to delegate the level construction to a function to be called after `setupLevels` in `Main` constructor:

```
public function Main() {  
    setupLevels();  
    drawLevel(currentLevel);  
}
```



`drawLevel` function will handle level construction, according to its `level` argument that represents the number of the level we are actually building:

```
private function drawLevel(level:uint):void {
    level_container = new Sprite();
    addChild(level_container);
    for (var i:uint=0; i<levels[currentLevel].length; i++) {
        for (var j:uint=0; j<levels[currentLevel][i].length; j++) {
            switch (levels[currentLevel][i][j]) {
                case 0 :// floor
                    tile=new tiles_mc(1,i,j);
                    level_container.addChild(tile);
                    break;
                case 1 :// wall
                    tile=new tiles_mc(2,i,j);
                    level_container.addChild(tile);
                    break;
                case 2 :// crate goal
                    tile=new tiles_mc(3,i,j);
                    level_container.addChild(tile);
                    break;
            }
        }
    }
}
```

The first new feature I want to show you is the `switch` statement, that allows us to evaluate an expression (in this case `levels[currentLevel][i][j]`) and directly jump on the block of code where `case` is the result of the expression. This means this code:

```
if(a==1){
    // a is equal to 1
} else {
    if(a==2){
        // a is equal to 2
    } else {
        if(a==3){
            // a is equal to 3
        }
    }
}
```

Is equal to this one:

```
switch (a) {
    case 1:
        // a is equal to 1
        break;
    case 2:
        // a is equal to 2
        break;
    case 3:
        // a is equal to 3
        break;
}
```

Also, note how I am ending each case with a `break`. It's mandatory if you don't want the execution of the script to continue to next case regardless whether it does not match the `switch` expression.

The rest of the script just creates new `tiles_mc` instances, so I would focus only on the arguments passed to the constructor. We'll find them in `tiles_mc` class we are going to create.

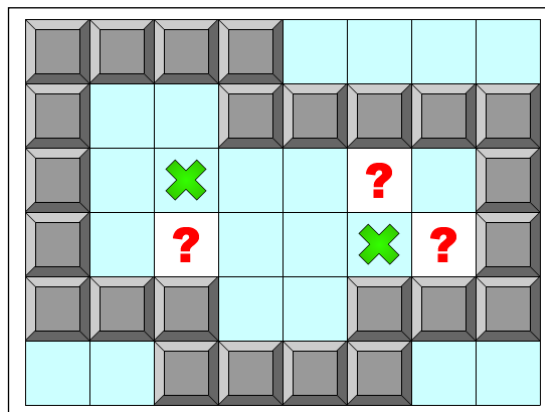
Without closing `sokoban.fla`, create a new file and from **New Document** window select **ActionScript 3.0 Class**. Save this file as `tile_mc.as` in the same path you saved `sokoban.fla` and `Main.as`.

Then enter this code:

```
package {
    import flash.display.MovieClip;
    public class tiles_mc extends MovieClip {
        public function tiles_mc(frm:uint, row:uint, col:uint) {
            gotoAndStop(frm);
            x=col*64;
            y=row*64;
        }
    }
}
```

There's really nothing to explain since it's the same concept applied to disc creation during the making of Connect Four. Tiles are just placed in their position and the proper frame is shown, according to the tile's type.

Test the movie and you will see the first level:



Anyway, there's something wrong with it. There are three "holes" in the game field, marked in the picture with a "?". Obviously they are tiles to be occupied by the bulldozer and the two crates, but anyway there can't be tiles without the floor.

## Adding bulldozer and crates

When we add the crates and the bulldozer, we must keep in mind we have to add both the crate/bulldozer and the floor.

**The idea:** In the same way we scanned `levels` array looking for floors, walls, and crate goals, we'll look for the remaining actors, such as the bulldozer and the crates, both on a normal tile and on the crate goal.

**The development:** We need two new class level variables to be added in `Main.as` to create instances of `crate_mc` and `bulldozer_mc`.

```
private var levels:Array=new Array();
private var currentLevel:uint=0;
private var tile:tiles_mc;
private var level_container:Sprite;
private var crate:crate_mc;
private var bulldozer:bulldozer_mc;
```

`crate` is the variable that we'll use to create `crate_mc` instances while `bulldozer` will create `bulldozer_mc` instances.

At this time we have to add the new cases to the `switch` statement in `drawLevel` function:

```
case 3 :// crate
    tile=new tiles_mc(1,i,j);
    level_container.addChild(tile);
    crate=new crate_mc(1,i,j);
    level_container.addChild(crate);
    break;
case 4 :// bulldozer
    tile=new tiles_mc(1,i,j);
    level_container.addChild(tile);
    bulldozer=new bulldozer_mc(i,j);
    level_container.addChild(bulldozer);
    break;
case 5 :// crate over the goal
    tile=new tiles_mc(3,i,j);
    level_container.addChild(tile);
    crate=new crate_mc(2,i,j);
    level_container.addChild(crate);
    break;
case 6 :// bulldozer over the goal
    tile=new tiles_mc(3,i,j);
    level_container.addChild(tile);
    bulldozer=new bulldozer_mc(i,j);
    level_container.addChild(bulldozer);
    break;
```

Look how I add a floor tile before adding the crate or the bulldozer, in cases 3 and 4. In cases 5 and 6, I am adding the crate goal tile rather than the floor as the crate/ bulldozer is meant to be over a crate goal.

Then, in the same way we created a new class for previous tiles, we need to create a class for the bulldozer and one for the crate.

Without closing `sokoban.fla`, create a new file and from **New Document** window select **ActionScript 3.0 Class**. Save this file as `bulldozer_mc.as` in the same path you saved `sokoban.fla` and `Main.as`.

Then write this code:

```
package {
    import flash.display.MovieClip;
    public class bulldozer_mc extends MovieClip {
        public function bulldozer_mc(row:uint,col:uint) {
            gotoAndStop(1);
        }
    }
}
```

```
        x=col*64;  
        y=row*64;  
    }  
}
```

The function just places the bulldozer and makes it head left.

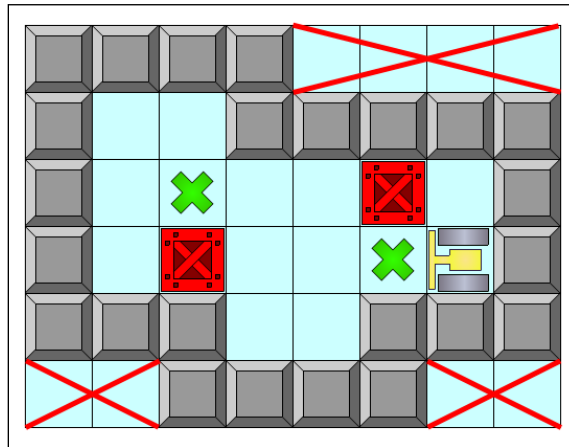
Without closing `sokoban.fla`, create a new file and from **New Document** window select **ActionScript 3.0 Class**. Save this file as `crate_mc.as` in the same path you saved `sokoban.fla` and `Main.as`.

Then write this code:

```
package {  
    import flash.display.MovieClip;  
    public class crate_mc extends MovieClip {  
        public function crate_mc(frm:uint,row:uint,col:uint) {  
            gotoAndStop(frm);  
            x=col*64;  
            y=row*64;  
        }  
    }  
}
```

This works the same way, showing a red crate if the crate isn't on a crate goal, or a green crate otherwise.

Test the movie and you will see the full level:



This is a good representation of the level, but we can do even better. There are eight floor tiles that the bulldozer will never be able to walk since they are outside the walls. It would be nice if the floor tiles marked with a red cross could not be drawn.

## Removing unnecessary floor tiles

Normally this situation can be prevented by assigning a value to floor tiles which do not need to be drawn, such as -1, but if you're looking for Sokoban levels around the web, you won't find external tiles marked in a different way from internal ones.

That is, in this particular case it's better to do it from inside the code.

**The idea:** Assuming we are dealing with a well designed level, the bulldozer must be placed inside the walls. That is, every tile that cannot be reached by the bulldozer (in other words, outside the walls) does not need to be drawn.

**The development:** The process of drawing the level needs to be changed a bit. First, we won't place floor tiles as soon as we have found a floor in `levels` array, but we'll add them once we find the bulldozer.

Change `drawLevel`'s switch statement this way:

```
switch (levels[currentLevel][i][j]) {
    // remove case 0
    case 1 :// wall
        ...
    case 2 :// crate goal
        ...
    case 3 :// crate
        crate=new crate_mc(1,i,j);
        level_container.addChild(crate);
        break;
    case 4 :// bulldozer
        bulldozer=new bulldozer_mc(i,j);
        level_container.addChild(bulldozer);
        flood(i,j);
        setupLevels();
        break;
    case 5 :// crate over the goal
        ...
    case 6 :// bulldozer over the goal
        tile=new tiles_mc(3,i,j);
        level_container.addChild(tile);
        bulldozer=new bulldozer_mc(i,j);
```

```
        level_container.addChild(bulldozer);  
        flood(i,j);  
        setupLevels();  
        break;  
    }
```

What's happening? As said, we removed the lines that placed floor tiles. The case 0 (floor) has been completely removed, as well as the lines to place floor tiles in cases 3 (crate) and 4 (bulldozer).

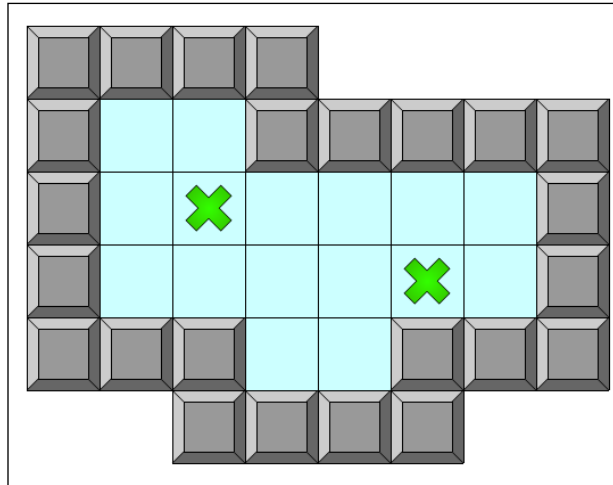
But there's something more: in the cases seeing the bulldozer to be added, case 4 (bulldozer) and 6 (bulldozer over the goal), there's a call to a function called `flood` whose arguments are the current row and column, then `setupLevels` is called again.

As the name of the new function should remind you, `flood` will perform a recursive flood fill to place floor tiles, just as we did during the creation of Minesweeper when it was time to remove all empty tiles.

Add this new function to `Main.as` file:

```
private function flood(row:uint,col:uint) {  
    if (Math.abs(levels[currentLevel][row][col])!=1) {  
        if (levels[currentLevel][row][col]!=2) {  
            if (levels[currentLevel][row][col]<5) {  
                tile=new tiles_mc(0,row,col);  
                level_container.addChild(tile);  
            }  
        }  
        levels[currentLevel][row][col]=-1;  
        flood(row-1,col);  
        flood(row+1,col);  
        flood(row,col-1);  
        flood(row,col+1);  
    }  
}
```

Test the movie and although the crates and the bulldozer seem to be lost, we aren't displaying unnecessary floor tiles anymore.



flood function is responsible for everything, so let's have a deeper look at it:

```
if (Math.abs(levels[currentLevel][row][col])!=1) { ... }
```

The entire function is executed only if the absolute value of the current tile is different to 1, that is if the current tile value is different to 1 (wall) and -1 (at the moment there isn't any tile -1 value, but we'll meet it in a moment).

Note how `Math.abs` method is used to return the absolute value of a number.

```
if (levels[currentLevel][row][col]!=2) { ... }
```

This `if` statement checks if the current tile is different from two, that is it's not a crate goal as crate goals are placed separately.

```
if (levels[currentLevel][row][col]<5) { ... }
```

And this `if` statement checks if the tile value is less than five, that is, it can't be a crate over the goal and a bulldozer over the goal.

The latest two `if` statements could have been merged into a single one, but I kept them separated for a layout purpose.

```
tile=new tiles_mc(0,row,col);  
level_container.addChild(tile);
```



Finally the floor tile is added. But when is it added? When the current tile is not a wall, not a crate goal, not a crate over a crate goal, and not a bulldozer over a crate goal. And it's not the "mysterious tile" with a -1 value.

It's time to explain what this tile is:

```
levels[currentLevel][row][col]=-1;
```

As you can see, we are marking every tile visited by `flood` function with -1. This is just a quick and dirty way to avoid `flood` function running into an infinite recursion because it keeps checking the same tiles again and again.

So the first `if` statement now sounds like: if the current tile value is different to 1 (wall) and it's the first time I met this tile (different than -1), then execute the function.

The remaining four lines just execute flood recursively to neighbor tiles:

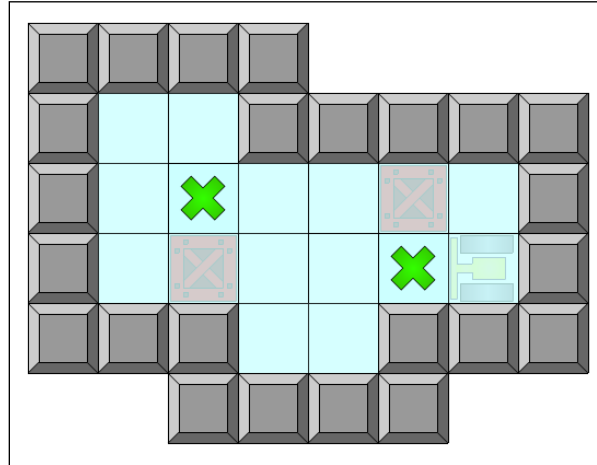
```
flood(row-1,col);
flood(row+1,col);
flood(row,col-1);
flood(row,col+1);
```

At this time it should also be clear why `setupLevels` function is executed after `flood` function: marking tiles with -1 actually changes the levels, so we need to restore them.

Now we should need to know where the crates and the bulldozer have gone. Add this line to `flood` function:

```
private function flood(row:uint,col:uint) {
    if (Math.abs(levels[currentLevel][row][col])!=1) {
        if (levels[currentLevel][row][col]!=2) {
            if (levels[currentLevel][row][col]<5) {
                tile=new tiles_mc(0,row,col);
                level_container.addChild(tile);
                tile.alpha=0.8;
            }
        }
        levels[currentLevel][row][col]=-1;
        flood(row-1,col);
        flood(row+1,col);
        flood(row,col-1);
        flood(row,col+1);
    }
}
```

We are just giving a little transparency to floor tiles. Test the movie:



As we see in the screenshot, crates and bulldozer are buried under the floor. Can you figure out why it's happening? Remember the DisplayObjects hierarchy you learned during the making of Minesweeper: DisplayObjects added first are overlapped by DisplayObjects added later, and in our case, floor is added after the crate and the bulldozer.

Did we take a wrong turn? No, this can easily be fixed by putting everything into layers.

## Putting everything into layers

The real world is made of layers. I am writing on the third floor, so I am on a higher layer than people on the first and second floor, who are higher than people on the street and in the subway.

Following this concept, you should plan DisplayObjects hierarchy during the making of your games. Walls, crates, and bulldozer are over the floor, so we have to assign them to a different layer.

**The idea:** We have to create a DisplayObjectContainer to place all floor tiles. All other assets will be placed above the container, that is, above all floor tiles placed before and after them.

**The development:** First, let's create the DisplayObjectContainer for all floor tiles: add a new class level variable to Main class:

```
private var levels:Array=new Array();
private var currentLevel:uint=0;
private var tile:tiles_mc;
private var level_container:Sprite;
private var crate:crate_mc;
private var bulldozer:bulldozer_mc;
private var floor_container:Sprite;
```

floor\_container will contain all floor tiles. Now in drawlevels function we need to add it as a child of level\_container. Change drawLevel this way:

```
private function drawLevel(level:uint):void {
    level_container = new Sprite();
    addChild(level_container);
    floor_container = new Sprite();
    level_container.addChild(floor_container);
    ...
}
```

Now we are ready to add floor tiles in level\_container. Change the content of the switch statement in drawLevel function this way, adding crate goal as a child of floor\_container.

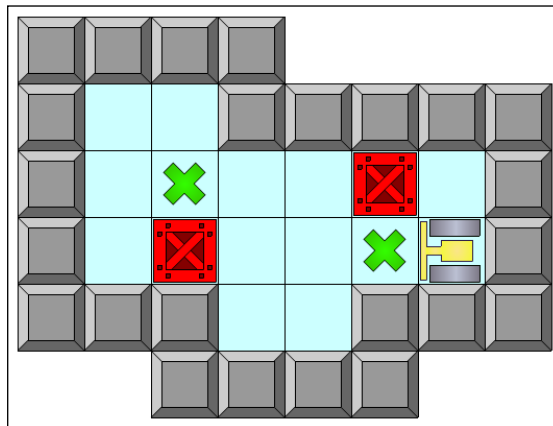
```
switch (levels[currentLevel][i][j]) {
    case 1 :// wall
        ...
    case 2 :// crate goal
        tile=new tiles_mc(3,i,j);
        floor_container.addChild(tile);
        break;
    case 3 :// crate
        ...
    case 4 :// bulldozer
        ...
    case 5 :// crate over the goal
        tile=new tiles_mc(3,i,j);
        floor_container.addChild(tile);
        crate=new crate_mc(2,i,j);
        level_container.addChild(crate);
        break;
    case 6 :// bulldozer over the goal
        tile=new tiles_mc(3,i,j);
```

```
        floor_container.addChild(tile);
        bulldozer=new bulldozer_mc(i,j);
        level_container.addChild(bulldozer);
        flood(i,j);
        setupLevels();
        break;
    }
```

Floor tiles also need to be added in floor\_container when created with flood function:

```
private function flood(row:uint,col:uint) {
    if (Math.abs(levels[currentLevel][row][col])!=1) {
        if (levels[currentLevel][row][col]!=2) {
            if (levels[currentLevel][row][col]<5) {
                tile=new tiles_mc(0,row,col);
                floor_container.addChild(tile);
            }
        }
        levels[currentLevel][row][col]=-1;
        flood(row-1,col);
        flood(row+1,col);
        flood(row,col-1);
        flood(row,col+1);
    }
}
```

Test the move and finally you'll see the level ready to be played:



Now we can make the player control the bulldozer.

## Detecting keyboard input

The bulldozer is controlled with the keyboard, so it's time to work on a way to get keyboard inputs.

**The idea:** We need a way to detect when the player presses a key, and recognize which key has been pressed.

**The development:** AS3 provides a class to deal with keyboard input. `KeyboardEvent` class provides an event called `KEY_DOWN` that is triggered when the user presses a key. Listening on the Stage will globally detect key events.

This couldn't be easier. Import `KeyboardEvent` class in Main class:

```
import flash.display.Sprite;
import flash.events.KeyboardEvent;
```

Then add the listener to the Stage in Main function:

```
public function Main() {
    setupLevels();
    drawLevel(currentLevel);
    stage.addEventListener(KeyboardEvent.KEY_DOWN, onKeyD);
}
```

Now, every time a key is pressed, `onKeyD` function will be executed. At the moment, let's just print some text in the output window:

```
private function onKeyD(e:KeyboardEvent):void {
    trace("you pressed: "+e.keyCode)
}
```

`keyCode` property returns the code value of the key pressed. Press LEFT, UP, RIGHT, and DOWN arrow keys in this sequence, and in the output window you will see:

```
you pressed: 37
you pressed: 38
you pressed: 39
you pressed: 40
```

These are the code values for arrow keys. Write them down, as you will be using them in a while, or google for "AS3 keycodes" for a complete list.



KeyboardEvent class manages user input through the keyboard.  
 KeyboardEvent.KEY\_DOWN listener is triggered when a key is pressed.  
 keyCode property returns the code value of the key pressed.

Now you can allow players to move the bulldozer with the keyboard.

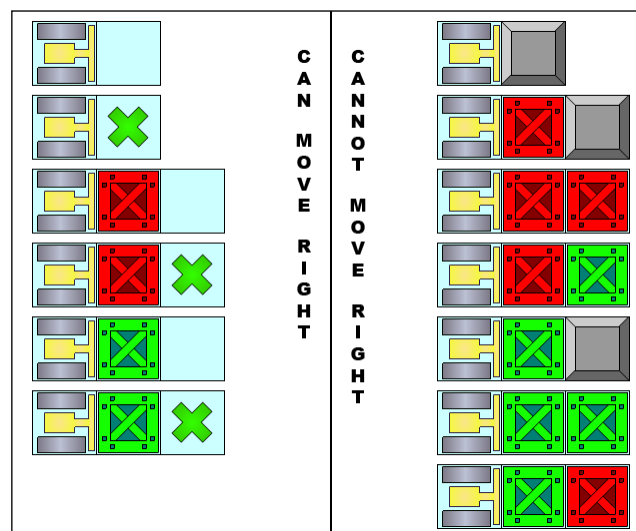
## Moving the bulldozer

Triggering keys to move the bulldozer is just the beginning. You still have to check if the bulldozer can move, then you should move it, update the level, and eventually push a crate.

**The idea:** When the player wants to move the bulldozer in a direction, the first thing you need to do is check whether a move is legal or not. Let's recap on legal moves in a Sokoban level. The bulldozer can move to an adjacent tile in one of four directions if:

- The tile is walkable—neither a wall nor a crate. The bulldozer moves to the walkable tile.
- The tile is a crate, but next to the crate, in the same direction, there is a walkable tile. The bulldozer moves to the tile with the crate over it and the crate is pushed to the walkable tile.

This picture displays the possible cases:



Obviously the bulldozer should also turn in the direction it's moving (or trying to move).

**The development:** The first thing to do is make a couple of functions to see if a tile is walkable and if there's a crate over a tile.

Add this function to Main class:

```
public function isWalkable(row:uint,column:uint):Boolean {  
    return (levels[currentLevel][row][column]%2==0);  
}
```

isWalkable function wants the row and the column of a level as arguments, and returns true if the tile is walkable, otherwise it returns false.

Since a tile is walkable when it's an empty tile (value 0) or if it's a crate goal (value 2), we can say a tile is walkable when its value can be divided by 2. You may say 4 (the bulldozer) and 6 (the bulldozer over a crate goal) can also be divided by 2, although they aren't walkable. You're right, but since there is only one bulldozer in a level, you will never find a bulldozer in your way.

Now, another function to check is whether there's a crate in our way. Add this in Main class:

```
public function isCrate(row:uint,column:uint):Boolean {  
    var tileValue:uint=levels[currentLevel][row][column];  
    return (tileValue%2==1&&tileValue>1);  
}
```

isCrate function wants the row and the column of a level as arguments, and returns true if there is a crate, otherwise it throws false.

Following a similar concept as the one explained before, a tile has a crate over it if it's an empty tile with a crate (value 3) or it's a tile goal with a crate over it (value 5). So we can say a tile has a crate over it when it cannot be divided by 2. You may say 1 (the wall) cannot be divided by 2 but it has nothing to do with a crate. That's why we want the value to be greater than 1. Also notice both functions are public because we are going to use them also outside Main class.

If these two ways to return values gave you a headache, simply replace them respectively with:

```
return (levels[currentLevel][row][column]==0 || levels[currentLevel][row]  
[column]==2);
```

and

```
return (levels[currentLevel][row][column]==3 || levels[currentLevel][row][column]==5);
```

but, other than making me write code that does not fit in a row (I'm absolutely trying to prevent it for the sake of readability), you are definitively refusing to optimize the code.

## Detecting keys and taking decisions

When the player presses an arrow key, we must detect which key he pressed and where to try to move the bulldozer. A `switch` statement in `onKeyDown` function will perfectly fit our needs:

```
private function onKeyDown(e:KeyboardEvent):void {
    switch (e.keyCode) {
        case 37 ://left
            bulldozer.moveBy(currentLevel,0,-1);
            break;
        case 38 :// up
            bulldozer.moveBy(currentLevel,-1,0);
            break;
        case 39 ://right
            bulldozer.moveBy(currentLevel,0,1);
            break;
        case 40 ://down
            bulldozer.moveBy(currentLevel,1,0);
            break;
    }
}
```

For each of the four possible directions (remember: the bulldozer cannot move diagonally) we'll call a function with different arguments according to the direction the player wanted to move the bulldozer.

`moveBy` function, which we will create in `bulldozer` class, is the core of bulldozer's movements.

It wants three arguments: the current level, the row movement offset (-1 if going up, 1 if going down) and the column movement offset (-1 if going left, 1 if going right). Basically it's just as if the player was moving the bulldozer around the array representing the level, and that's what we need.



So, for instance, if the player presses LEFT arrow, the script will execute this case:

```
case 37 ://left
    bulldozer.moveBy(currentLevel,0,-1);
    break;
```

moving left, the bulldozer will remain on the same row, and moves back by one column, that's why `moveBy` function has the last two arguments at 0, -1.

You may wonder why I am talking about the way I am passing arguments to a function that does not exist yet. Believe it or not, at the time of writing this page I still have to write it. That's because when you are dealing with complicated tasks, you should split them into simpler sub-tasks to maintain control over your application.

In this case, I split the problem assuming I had the perfect function to manage bulldozer movement, and calling it with the right arguments at each arrow key pressed.

Now it's time to develop such a function, but before all that we have to prepare `bulldozer_mc` class to interact with its parent class, `Main`.

First, we need some class level variables to let us use the current row and column position of the bulldozer around the class. As seen during the making of Connect Four, we'll also need a variable to work with `Main` class.

Add these class level variables to `bulldozer_mc` class.

```
private var curRow:uint;
private var curCol:uint;
private var parentMovieClip:Main;
```

Then, just like what you've seen in Connect Four, we delegate all initialization stuff to a function to be executed when the bulldozer is added to stage.

Change `bulldozer_mc` function this way:

```
public function bulldozer_mc(row:uint,col:uint) {
    curRow=row;
    curCol=col;
    addEventListener(Event.ADDED_TO_STAGE,onAdded);
}
```

now it just assigns class level variables `curRow` and `curCol` the values of function arguments, respectively `row` and `col`. The rest of the initialization is made by `onAdded` function, that's called when `Event.ADDED_TO_STAGE` event is triggered.

This is onAdded function:

```
private function onAdded(e:Event) {  
    gotoAndStop(1);  
    parentMovieClip=this.parent.parent as Main;  
    x=curCol*64;  
    y=curRow*64;  
}
```

As you can see, it's basically the same content you put in `bulldozer_mc` function earlier. The only difference is the way `x` and `y` properties take their values, this time from the class level variables `curCol` and `curRow` since `row` and `col` aren't available in this function.

Also, notice how `parentMovieClip` variable takes `Main` datatype:

```
parentMovieClip=this.parent.parent as Main;
```

There is a double `parent` call because `bulldozer_mc` is a child of `level_container` which is a child of `Main`.

## Moving the bulldozer and updating game field

Finally it's time to create `moveBy` function. You already know its arguments. It's time to explain how it should work.

```
public function moveBy(level:uint,row:int,col:int):void {  
    if (parentMovieClip.isWalkable(curRow+row,curCol+col)) {  
        // moving the bulldozer  
        moveBulldozer(level,row,col);  
    } else {  
        if (parentMovieClip.isCrate(curRow+row,curCol+col)) {  
            if (parentMovieClip.isWalkable(curRow+2*row,curCol+2*col)) {  
                // moving the bulldozer and pushing the crate  
                moveBulldozer(level,row,col);  
                // end moving the bulldozer and pushing the crate  
            }  
        }  
    }  
    gotoAndStop((3+row)*Math.abs(row)+(2+col)*Math.abs(col));  
}
```

First, we need to know if the tile we are going to move is walkable:

```
if (parentMovieClip.isWalkable(curRow+row,curCol+col)) { ... }
```

will do the check for us. Note how function arguments are added to `curRow` and `curCol` to determine the destination tile.

If the tile is walkable, then we delegate to `moveBulldozer` function the task of moving the bulldozer and updating the game field. We are going to write this function in a couple of minutes. This is how we call it:

```
moveBulldozer(level,row,col);
```

The arguments represent level number, and the horizontal and vertical offset, in tiles.

If the destination tile is not empty, we must see if it contains a crate, and if the next tile is walkable:

```
if (parentMovieClip.isCrate(curRow+row,curCol+col)) { ... }
```

This `if` statement checks if the destination tile is a crate, while this one:

```
if (parentMovieClip.isWalkable(curRow+2*row,curCol+2*col)) { ... }
```

checks if the tile next to the crate in the same direction is a walkable tile. To do this we must look two tiles away from the current tile, and that's why `row` and `col` arguments are multiplied by two.

In this case, at the moment we just move the bulldozer just like we found an empty tile.

Let's focus on `moveBulldozer` function:

```
private function moveBulldozer(level:uint,row:int,col:int):void {
    parentMovieClip.levels[level][curRow+row][curCol+col]+=4;
    parentMovieClip.levels[level][curRow][curCol]-=4;
    curRow+=row;
    curCol+=col;
    x+=col*64;
    y+=row*64;
}
```

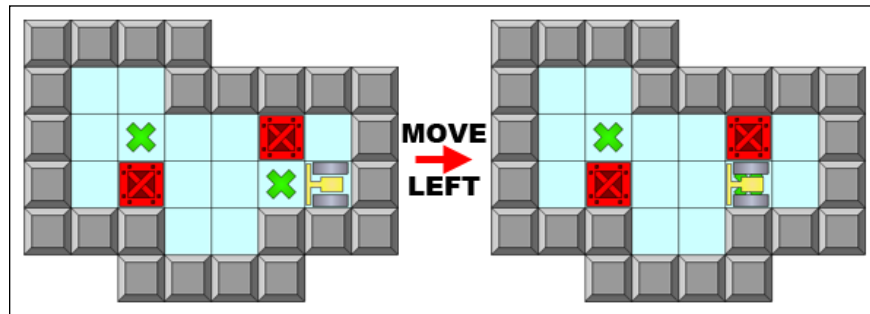
This function just updates everything that belongs to the bulldozer: first, the destination tile in `levels` array is incremented by 4 as this is bulldozer's value. At the same time we have to decrease the tile the bulldozer just left by 4, as the bulldozer is no longer over there.

Then we update `curRow` and `curCol` values, and finally `x` and `y` properties are updated too.

Obviously since `levels` array belongs to `Main` class, it should be declared as `public` in `Main` class level variables.

```
public var levels:Array=new Array();
private var currentLevel:uint=0;
private var tile:tiles_mc;
private var level_container:Sprite;
private var crate:crate_mc;
private var bulldozer:bulldozer_mc;
private var floor_container:Sprite;
```

Move the bulldozer around the level, and everything should work fine.



Unfortunately, at the moment you can't push any crate.

## Pushing the crates

Pushing crates is not that different from moving the bulldozer, as it's just another routine to update `levels` array and adjust `DisplayObjects` position.

**The idea:** We will move the crate in exactly the same way we moved the bulldozer, just paying attention to if the crate is over a crate goal. In this case we will show the green crate, showing the red crate otherwise.

**The development:** Inside moveBy function in bulldozer\_mc class, rewrite the code between // moving the bulldozer and pushing the crate and // end moving the bulldozer and pushing the crate this way:

```
// moving the bulldozer and pushing the crate
var crateName:String=(curRow+row)+"_"+(curCol+col);
var crateToMove:crate_mc;
crateToMove=parentMovieClip.level_container.getChildByName(crateName)
as crate_mc;
moveBulldozer(level,row,col);
crateToMove.moveCrate(level,row,col);
// end moving the bulldozer and pushing the crate
```

As you can see I only detected which crate I have to move according to its name, built upon its position, and called a moveCrate function in the same way I called moveBulldozer, with the only difference moveCrate is located in crate\_mc class.

Now we have to work on crate\_mc.as, adding the class level variables to determine crate's current row and column:

```
package {
    import flash.display.MovieClip;
    public class crate_mc extends MovieClip {
        private var curRow:uint;
        private var curCol:uint;
        ...
    }
} curRowcurCol
```

Now crate\_mc constructor needs to be changed a bit to assign the crate a name according to its row and column (the same name we used before in moveBy function) and to assign curRow and curCol the values of crate's row and column.

```
public function crate_mc(frm:uint,row:uint,col:uint) {
    gotoAndStop(frm);
    curRow=row
    curCol=col
    x=col*64;
    y=row*64;
    name=row+"_"+col;
}
```

And finally let's create moveCrate function. As said it's almost identical to moveBy, but this time we are increasing and decreasing levels values by 3 as the crate is assigned the value of 3.

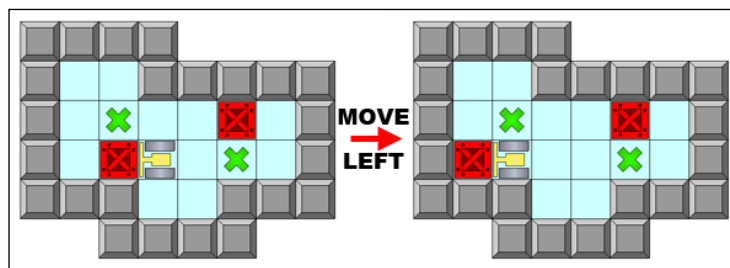
```
public function moveCrate(level:uint,row:int,col:int):void {  
    var parentMovieClip:Main=this.parent.parent as Main;  
    parentMovieClip.levels[level][curRow+row][curCol+col]+=3;  
    parentMovieClip.levels[level][curRow][curCol]-=3;  
    curRow+=row;  
    curCol+=col;  
    if (parentMovieClip.levels[level][curRow][curCol]==5) {  
        gotoAndStop(2);  
    } else {  
        gotoAndStop(1);  
    }  
    x+=col*64;  
    y+=row*64;  
    name=curRow+"_"+curCol;  
}
```

The other interesting differences are the way we change name of the crate according to its new position in the last line of the function, and how we decide if we have to show whether it's a red or a green crate according to the value of the tile the crate is on. If tile value is 5, then we have a crate over a crate goal so we'll show frame two.

To avoid errors, since we accessed Main's `level_container` variable from inside `bulldozer_mc` class in `moveBy` function, we have to set it as public.

```
public var levels:Array=new Array();  
private var currentLevel:uint=0;  
private var tile:tiles_mc;  
public var level_container:Sprite;  
private var crate:crate_mc;  
private var bulldozer:bulldozer_mc;  
private var floor_container:Sprite;
```

Test the movie and this time you will be able to push crates.



The basic prototype is completed; anyway we can improve it with some other features.

## Adding smooth movement

The Eighties are over and players expect at least a smooth movement while at the moment our bulldozer jumps from tile to tile.

**The idea:** We need to move the bulldozer, and eventually the crate it's pushing, from one tile to another by small steps, to give the idea of a smooth movement. The player won't be able to change bulldozer direction while he's moving from one tile to another.

**The development:** To move the bulldozer, we need to know two things:

1. is the bulldozer already moving?
2. in which direction is the bulldozer moving?

We'll store this information in three level class variables in `bulldozer` class:

```
private var curRow:uint;
private var curCol:uint;
private var parentMovieClip:Main;
private var isMoving:Boolean=false;
private var x_dir:int;
private var y_dir:int;
```

`isMoving` is `true` if the bulldozer is moving, and `false` if it's not moving. Its starting value is `false` because at the beginning of the game the bulldozer is not moving.

`x_dir` and `y_dir` will hold the horizontal and vertical direction of the bulldozer.

We also need an enter frame listener to create smooth movements, like you already saw during the creation of Connect Four. Add the enter frame listener to `bulldozer_mc` function:

```
public function bulldozer_mc(row:uint,col:uint) {
    curRow=row;
    curCol=col;
    addEventListener(Event.ADDED_TO_STAGE,onAdded);
    addEventListener(Event.ENTER_FRAME,onEnterF);
}
```

`onEnterF` function will handle all animations, in a very similar way disc animations was managed during the creation of Connect Four game. Add this function to `bulldozer_mc` class:

```
private function onEnterF(e:Event) {
    if (isMoving) {
        x+=x_dir;
        y+=y_dir;
```

```
        if (x%64==0&&y%64==0) {  
            isMoving=false;  
        }  
    }  
}
```

The whole function content will be executed only if `isMoving` is `true`, because if the bulldozer is not moving, there's no need to show any animation. These two lines:

```
x+=x_dir;  
y+=y_dir;
```

move the bulldozer horizontally and vertically by respectively `x_dir` and `y_dir` pixels. At the moment you don't know yet how to set these values, but it's a matter of minutes, and it's not that important at the moment since this function only needs to know the bulldozer must move. When should the bulldozer stop moving? Once it's completely over a tile, and you can determine it this way:

```
if (x%64==0&&y%64==0) {  
    isMoving=false;  
}
```

Since game tiles are 64x64 pixel squares, when the remainder of bulldozer `x` and `y` positions divided by 64 are equal to 0, we can say the bulldozer is exactly over a tile, so there's no need to move the bulldozer anymore and `isMoving` is set to `false`. Obviously this will work only if `x_dir` and `y_dir` perfectly divide 64. We must keep this in mind when assigning these values.

As said before, you must allow the player to move the bulldozer only if it's not already moving, so prevent the content of `moveBy` function to be executed if the bulldozer is moving this way:

```
public function moveBy(level:uint,row:int,col:int):void {  
    if (!isMoving) {  
        ...  
    }  
}
```

Now the content of `moveBy` function (executed when the player presses an arrow key) is executed only if the bulldozer is not moving.



Last but not least, let's define `x_dir` and `y_dir` values. Modify `moveBulldozer` function removing the lines that directly change bulldozer `x` and `y` properties and replace them with this one:

```
private function moveBulldozer(level:uint,row:int,col:int):void {
    parentMovieClip.levels[level][curRow+row][curCol+col]+=4;
    parentMovieClip.levels[level][curRow][curCol]-=4;
    curRow+=row;
    curCol+=col;
    // remove x+=col*64 and y+=row*64;
    isMoving=true;
    x_dir=col*8;
    y_dir=row*8;
}
```

Now when it's time to move the bulldozer, it won't jump directly to its destination but will set `isMoving` to `true` to let the content of `onEnterFrame` function be executed.

Also notice `x_dir` and `y_dir` can be set to -8, 0, or 8 according to `row` and `col` values.

8 is a number which perfectly divides 64. You can use any number among 1, 2, 4, 8, 16 and 32. The higher the number, the less frames needed to move from one tile to another. Find a good compromise between smoothness and speed. Remember players won't be able to interact with the game while the bulldozer is moving.

## Adding smooth movement to crates

To add smooth movement to crates, we'll use the same concept. One neat feature of `Event.ENTER_FRAME` event is this event is simultaneously dispatched to all `DisplayObjects` listening for it, so we don't even need to synchronize animations.

Let's repeat all steps done for the bulldozer in `crate_mc` class. Add the three class level variables to manage movement.

```
private var curRow:uint;
private var curCol:uint;
private var isMoving:Boolean=false;
private var x_dir:int;
private var y_dir:int;
```

Then add `Event` class since you did not import it already.

```
import flash.display.MovieClip;
import flash.events.Event;
```

And add the listener calling `onEnterF` function directly in `crate_mc` function.

```
public function crate_mc(frm:uint,row:uint,col:uint) {
    gotoAndStop(frm);
    curRow=row
    curCol=col
    x=col*64;
    y=row*64;
    name=row+"_"+col;
    addEventListener(Event.ENTER_FRAME,onEnterF);
}
```

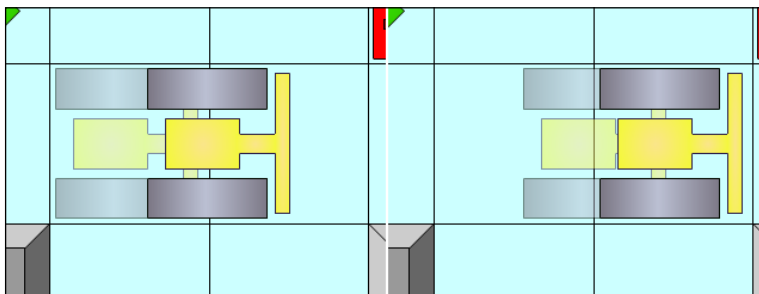
`onEnterF` function is exactly the same, you can copy/paste it from `bulldozer_mc` class.


```
private function onEnterF(e:Event) {
    if (isMoving) {
        x+=x_dir;
        y+=y_dir;
        if (x%64==0&&y%64==0) {
            isMoving=false;
        }
    }
}
```

Finally, when it's time to move the crate, remove the lines acting directly on `x` and `y` properties and add the code to move the crate, that is the same you used to move the bulldozer. Double check you are assigning the same values to `x_dir` and `y_dir` in both `crate_mc` and `bulldozer_mc` classes or the bulldozer and the crate will move at a different speed.

```
public function moveCrate(level:uint,row:int,col:int):void {
    var parentMovieClip:Main=this.parent.parent as Main;
    parentMovieClip.levels[level][curRow+row][curCol+col]+=3;
    parentMovieClip.levels[level][curRow][curCol]-=3;
    curRow+=row;
    curCol+=col;
    if (parentMovieClip.levels[level][curRow][curCol]==5) {
        gotoAndStop(2);
    } else {
        gotoAndStop(1);
    }
    // remove x+=col*64 and y+=row*64;
    isMoving=true;
    x_dir=col*8;
    y_dir=row*8;
    name=curRow+"_"+curCol;
}
```

Test the movie and push some crates. You should see a smooth animation.



[  Event. ENTER\_FRAME is simultaneously dispatched to all DisplayObjects listening for it. ]

And now bulldozer and crate movements are completely developed.

## Checking for victory

The aim of the game is placing all crates on their spots, so we need at least to prevent the player from making another move when all crates are correctly placed. But first, obviously, we need to check whether a level is solved or not.

**The idea:** We know the array representing a level contains an item with value 2 for every crate goal without anything over it and an item with value 6 to represent a bulldozer over a crate goal. And we also know in a completed level all crate goals must have a crate over them, so there can't be any item with a value 2 or 6 in the level array.

**The development:** We need a function to check if the level is completed, that is when there aren't items with value 2 or 6 in the level array. Add this function to Main class:

```
public function isSolved(level:uint):Boolean {
    var s:String=levels[level].toString();
    return s.indexOf("2")==-1&&s.indexOf("6")==-1;
}
```

isSolved function takes the level number as argument and returns true if the level is solved, or false elsewhere.

Since `levels` is a three-dimensional array, `levels[level]` is a two-dimensional array and using `indexOf()` method to search for it as seen in the making of Concentration game would be a bit tricky because you would need to use it on any array representing a row as it does not work on multi-dimensional arrays.

It would be better to convert this two-dimensional array into something simpler. This job can be easily done thanks to `toString()` method that converts the array into a string inserting each item starting from index zero and separating them by commas.

That is, this script:

```
var a:Array=[0,1,2];
var s:String=a.toString();
```

sets `s` as `1,2,3`. Exactly what we are looking for, since it works even with multi-dimensional arrays. That's why this line:

```
var s:String=levels[level].toString();
```

converts the level array into a string, which we can easily scan looking for a 2 or a 6 with `indexOf()` method that works with strings in the same way it works with arrays.

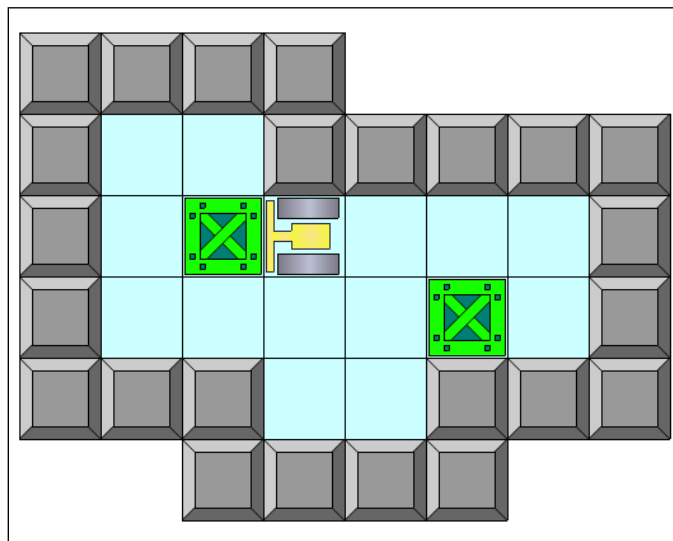
```
return s.indexOf("2")==-1&& s.indexOf("6")==-1;
```

This line will return `true` if there isn't any 2 or 6 characters in `s` string, and since `s` is the string representation of `levels[level]` array, we can say the level does not contain any crate goal without a crate over it.

At this time, we can prevent the bulldozer from moving when the level is solved adding a condition in the first `if` statement in `moveBy` function in `bulldozer_mc` class.

```
public function moveBy(level:uint,row:int,col:int):void {
    if (!parentMovieClip.isSolved(level)&&!isMoving) {
        ...
    }
}
```

Test the game and solve a level, you won't be able to move the bulldozer anymore.



`toString()` method converts an array into a string inserting each item starting from index zero and separating them by commas.  
`indexOf()` method works on strings in the same way it works on arrays.

In a commercial game, you should add some eye-candy congratulations effects and a jingle, but it's enough for this prototype.

## Adding Restart, Next, and Previous level options

With all game mechanics completed and working, it's time to add complementary functionalities such as Restart, Next Level, and Previous Level options.

**The idea:** When the player presses a key, we need to check if it's "R" (restart), "N" (next) or "P" (previous), remove the current level and draw the new one according to the options he selected. To keep things simple, when the player presses "R" to restart the level, we'll still remove the current level and redraw it. Also, we want to cycle through levels, so when the player selects Next from the last level, he will be brought to first level, and when he selects Previous from the first level he/she will be brought to the last level.

**The development:** We need to add three more cases to switch statement in onKeyD function, one for each new key we want to watch.

```
private function onKeyD(e:KeyboardEvent):void {
    switch (e.keyCode) {
        ...
        case 78 :
            // N = next
            currentLevel=(currentLevel+1)%levels.length;
            removeChild(level_container);
            setupLevels();
            drawLevel(currentLevel);
            break;
        case 80 :
            // P = prev
            currentLevel--;
            if (currentLevel<0) {
                currentLevel=levels.length-1;
            }
            removeChild(level_container);
            setupLevels();
            drawLevel(currentLevel);
            break;
        case 82 :
            // R = restart
            removeChild(level_container);
            setupLevels();
            drawLevel(currentLevel);
            break;
    }
}
```

As you can see, all cases are almost identical.

## Next level

As said, when skipping to next level, we must remove the current level and draw next one. First, we need to update currentLevel variable with the new value of the level to be shown:

```
currentLevel=(currentLevel+1)%levels.length;
```

Nothing new, this line just adds 1 to currentLevel and sets it as the remainder of a division by the number of levels. This way, when the player is on the last level (level 2) and presses Next, he's brought to the first level (level 0).

```
removeChild(level_container);
```

Dismisses the entire level by removing `level_container` `DisplayObjects`.

```
setupLevels();
```

Calls `setupLevels` function to bring `levels` array back to its initial value and have all levels ready to be played.

```
drawLevel(currentLevel);
```

Draws the level itself.

## Previous level

Going to previous level is very similar to what you have just seen. The only difference is we need to subtract 1 from `currentLevel` and set it to highest level possible if it's less than zero.

So the only difference is how we update `currentLevel` variable:

```
currentLevel--;  
if (currentLevel<0) {  
    currentLevel=levels.length-1;  
}
```

As you can see, subtracting 1 from `currentLevel` when its value is 0 will give `currentLevel` a negative value, so to prevent strange behaviors we should declare `currentLevel` as `int` in class level variables.

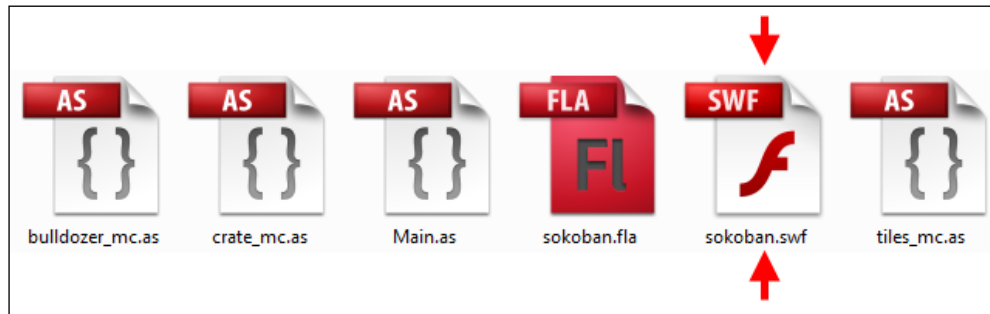
```
public var levels:Array=new Array();  
private var currentLevel:int=0;  
private var tile:tiles_mc;  
public var level_container:Sprite;  
private var crate:crate_mc;  
private var bulldozer:bulldozer_mc;  
private var floor_container:Sprite;
```

This way we can work with negative values too.

## Restart level

This is the easiest case because you don't need to change `currentLevel` value. Test the game, try to press N, P, or R key but nothing will happen.

Did you do something wrong? No, it's just when testing the game in the Flash IDE, most keys are already bound to keyboard shortcuts and you won't be able to use them in your Flash movie. For this reason, to test this game from now on, you will have to directly launch the `swf` file. You can find it in the same path of `sokoban.fl` and it's called `sokoban.swf`.



Double-click on it, and you'll be able to play with Restart, Next, and Previous level options.



When testing your games in Flash IDE, most keys won't work as they are bound to environment's keyboard shortcuts.

And now, the most complicated feature.

## Managing the Undo

In puzzle games sometimes it's frustrating to have to restart a level just because you made a wrong move, maybe due to pressing a wrong key.

That's why an Undo option can prevent frustration and make people play longer.

While dealing with Undo can lead to very complex algorithms with multiple undo levels and history management as in Photoshop, in this game we'll just cover the most basic way, with only one Undo level. Pressing the "U" key, the player will undo the latest move. Pressing it again, will do nothing.

If you are fascinated by Undo, Google for "command pattern" and "memento pattern".

**The idea:** Before any move, we need to save the current level and make it available when "U" key is pressed.



**The development:** The first thing that comes to mind is the creation of an undo array where you can save the latest level situation available.

Before you start coding, I want you to test this little script (you can create a new project or just replace Main content with this one):

```
package {
    import flash.display.Sprite;
    public class Main extends Sprite {
        public function Main() {
            var a:Array=[0,1];
            trace("my A array: "+a);
            var b:Array=a;
            b[0]=1;
            trace("my B array: "+b);
            trace("my A array: "+a);
        }
    }
}
```

In the output window you will see:

```
my A array: 0,1
my B array: 1,1
my A array: 1,1
```

This seems absolutely wrong because you never changed anything in a array, but at the end of the script, it seems `b[0]=1` affected a array too. A bug?

## Assigning variables by value or by reference

AS3 does not handle all datatypes in the same way. Primitive datatypes like numbers, strings and Booleans assign their value by value. This means when you copy an integer you copy the value of the first variable into the second one, leaving the first variable intact no matter what you do with the second one.

Complex datatypes like arrays assign their value by reference, that means when you copy an array into another array you aren't physically duplicating it but you are using the same array within two variables. This way, changing the first variable will change the second one too, and changing the second variable will change the first one, like in the previous example.

To clarify the concept of assigning by reference, open a text file and create a shortcut if you use Windows or an alias if you use Max OS. From now on, no matter if you edit the file opening it from the original file itself or from the shortcut/alias; you are working on the same file despite the way you are accessing it.

The same thing happened to our array. Knowing this, we still can copy an array to another array by "manually" copying all items with a `for` loop this way:

```
package {
  import flash.display.Sprite;
  public class Main extends Sprite {
    public function Main() {
      var a:Array=[0,1];
      trace("my A array: "+a);
      var b:Array=new Array();
      for (var i:uint=0; i<a.length; i++) {
        b.push(a[i]);
      }
      b[0]=1;
      trace("my B array: "+b);
      trace("my A array: "+a);
    }
  }
}
```

which outputs:

```
my A array: 0,1
my B array: 1,1
my A array: 0,1
```

that's just what we expect, anyway we'll use a different approach, just to see something new.

## Storing arrays into strings

We'll store level array into a string that we'll recall once an Undo is needed. We need two more class level variables. A string, that will contain the saved array, and a Boolean to let us know if we have to recall the Undo level. Add them to class level variables in Main class this way:

```
public var levels:Array=new Array();
private var currentLevel:int=0;
private var tile:tiles_mc;
public var level_container:Sprite;
private var crate:crate_mc;
private var bulldozer:bulldozer_mc;
private var floor_container:Sprite;
private var undoString:String;
private var isUndo:Boolean=false;
```

Again, toString() method will speed up the process. Add this function to Main class:

```
public function saveUndo():void {
    undoString=levels[currentLevel].toString();
}
```

This just assigns to undoString variable a string representation of the current level.

Now the question is: when should you save the current level? Obviously before the bulldozer moves, changing level status. Change moveBulldozer function adding this line at the very beginning:

```
private function moveBulldozer(level:uint,row:int,col:int):void {
    parentMovieClip.saveUndo();
    ...
}
```

Notice I'm saving the level only when the bulldozer is about to move for real, not in moveBy function when it's just checking whether it can move or not.

And now, to prevent people from messing up everything by pressing the Undo key before they move (yes, it's a nonsense, but players can do even worse), let's save the level as soon as we create it. Change drawLevel function just as you did with moveBulldozer:

```
private function drawLevel(level:uint):void {
    saveUndo();
    ...
}
```

Now we are sure the player will always find a saved level, no matter when he presses the Undo key, although in some cases the saved level will be the same as the one he's playing.

## The Undo itself

The concept behind the Undo is the same we used to restart and move to previous or next level. We have to remove the current level, reset `levels` array with `setupLevels` function and draw the current level.

The only difference is we need to keep in mind during this process that we are doing an Undo. That's when `isUndo` variable comes into play. Add this new case to switch statement in `onKeyD` function:

```
private function onKeyD(e:KeyboardEvent):void {
    switch (e.keyCode) {
        ...
        case 85 :
            // U = undo
            removeChild(level_container);
            isUndo=true;
            setupLevels();
            drawLevel(currentLevel);
            isUndo=false;
            break;
    }
}
```

As you can see, the code is almost the same as the ones you used to manage restart, previous, and next level. We just keep `isUndo` variable to `true` during the entire setup and drawing process, and this will affect the behavior of the `setupLevels` function itself.

When `isUndo` is `true`, it means we aren't doing an Undo, so `setupLevels` just acts as before, populating `levels` array with levels data.

The big change happens when `isUndo` variable is `false`, because we need to populate `levels[currentLevel]` with the data we previously stored in `undoString` string.

Change `setupLevels` function this way:

```
private function setupLevels():void {
    if (! isUndo) {
        levels[0]=[...];
        levels[1]=[...];
    }
```

---

```

        levels[2]=[...];
    } else {
        var valuePosition:uint;
        var recovered:uint;
        for (var i:uint=0; i<levels[currentLevel].length; i++) {
            for (var j:uint=0; j<levels[currentLevel][i].length; j++) {
                valuePosition = (j+i*levels[currentLevel][i].length)*2;
                recovered=int(undoString.charAt(valuePosition));
                levels[currentLevel][i][j]=recovered;
            }
        }
    }
}

```

Test the movie and you'll be able to play with Undo. Try it in all circumstances, and everything should work fine.

Let's see in detail what happened:

```
if (! isUndo) { ... }
```

This is the old code now it's executed if we aren't performing an Undo. Nothing new.

```
else { ... }
```

This is the interesting part. We are entering in Undo world.

```
var valuePosition:uint;
var recovered:uint;
```

These are two dummy variables used to store temporary data. `valuePosition` will contain the position of the character we need to retrieve in `undoString`, and `recovered` is the numeric representation of the character itself.

Look how we are using them:

```

for (var i:uint=0; i<levels[currentLevel].length; i++) {
    for (var j:uint=0; j<levels[currentLevel][i].length; j++) {
        ...
    }
}

```

These two for loops scan through all `levels[currentLevel]` array, to populate it with `undoString` content.

```
valuePosition = (j+i*levels[currentLevel][i].length)*2;
```

This is how we retrieve the position of the appropriate character in `undoString` according to `i` and `j` values.

```
recovered=int(undoString.charAt(valuePosition));
```

`charAt(i)` method returns the character in the `i`-th position of a string (remember the first character is in position zero) and `int` method converts a numeric value to an integer value. In the end this line retrieves the `valuePosition`-th character in `undoString` string and converts it to an integer, finally assigning its value to `recovered` variable.

```
levels[currentLevel][i][j]=recovered;
```

At this time, just assign `recovered` value to the currently scanned array item and the job is done.

The only issue is the Undo does not remember the bulldozer orientation, but it's easy to store it in another variable and retrieve it when needed. Unfortunately this chapter has been far too long and I must leave this feature to you.



Assigning values by reference does not create a physical copy of such variables but only creates more variables referring to the same variable.  
`charAt(i)` method returns the character at `i` position in a string. `i` starts from zero and will retrieve the first character.  
`int` method converts a numeric value to an integer value.

You are just one step from the completed game.

## Playing with WASD keys

Do you remember we wanted to make everyone happy and let people play both with arrow and WASD keys? One last effort and the game will be completed.

**The idea:** There's nothing new to develop, as W, A, S, and D keys must do the same operations as UP, LEFT, DOWN, and RIGHT arrows respectively. So we just need to see if the player pressed one of these keys, and move the bulldozer in the appropriate direction.

**The development:** The most intuitive way would be to create four new entries (one for each WASD key) to the `switch` statement in `onKeyDown` function and copy/paste the code of the corresponding arrow key.

Anyway, you know you have to end each case with a break to prevent the execution of the code to the next case. This time, we'll use this feature to our advantage by placing the WASD cases before their corresponding arrow key cases, without any break.

Change onKeyD function this way:

```
private function onKeyD(e:KeyboardEvent):void {
    switch (e.keyCode) {
        case 65 :// A
        case 37 :
            bulldozer.moveBy(currentLevel,0,-1);
            break;
        case 87 :// W
        case 38 :
            bulldozer.moveBy(currentLevel,-1,0);
            break;
        case 68 :// D
        case 39 :
            bulldozer.moveBy(currentLevel,0,1);
            break;
        case 83 :// S
        case 40 :
            bulldozer.moveBy(currentLevel,1,0);
            break;
        ...
    }
}
```

Let's see what happens when the player presses "A" or "a" (keyCode 65):

```
case 65 :// A
```

The script enters from this line and does nothing as there's just a comment. Then, without any break to force the exit from switch statement the script continues here:

```
case 37:
    bulldozer.moveBy(currentLevel,0,-1);
    break;
```

That is the code to manage the left arrow key. The same concept is applied to other WASD keys and the job is done.



You can execute more cases in a `switch` statement by placing them in cascade, without any `break`.

And finally the game is completed and ready to be played. Again, remember most alphabet keys don't work when testing the game in the Flash IDE.

## Summary

Be proud of yourself because this has been a very long game to create, but you managed to do it by learning how to interact with keyboard, and how to pass variables by reference or by value. But most of all you made your way through the mechanics of a tile-based puzzle game with the complete set of options and features modern puzzle games provide.

## Where to go now

Since the prototype can be considered complete, I won't ask you to add new features, although you are free to add more levels to the game.

What you should do is remove the `ENTER_FRAME` listener when you change or restart a level. If you noticed, such events are always added but never removed. You can add a function to `bulldozer_mc` and `crate_mc` classes and remove the listeners inside that function. Then you can call these functions before redrawing the level.