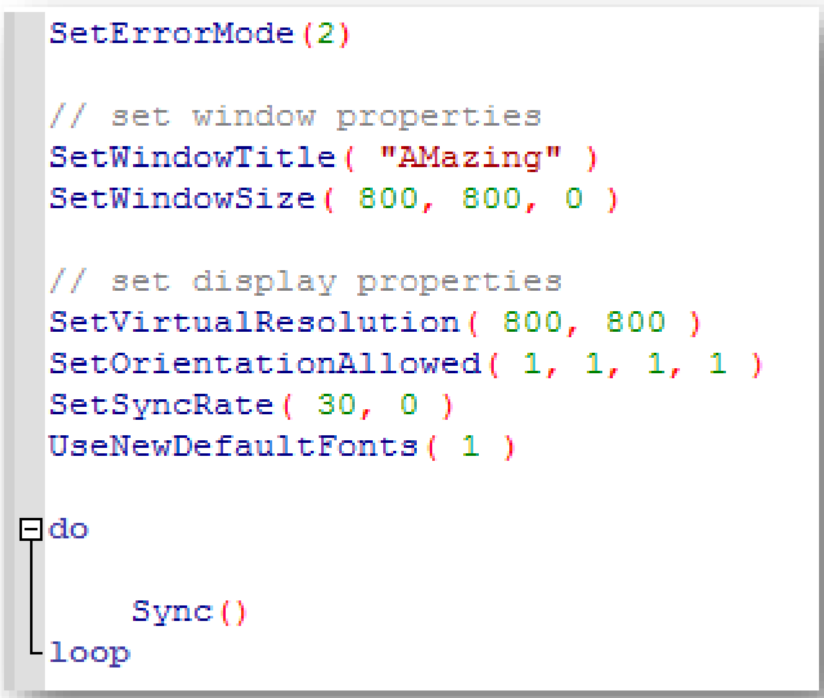
Making use of other Applications

Although we need game software to help makes games for modern devices, we should not exclude the use of other applications to aid the process as well. We can probably assume that most people are award of the importance of applications like Adobe Photoshop, but what about Microsoft Excel?

For this example we are going to do just that as an aid to developing a simple maze game. Two things we really need to consider at this point are the overall size and the size of individual cells within the maze. To keep it simple and have less data to work with, I'm going to keep my maze relatively simple and use large cell dimensions. The overall size will be 800 x 800 pixels with 50 x 50 pixel cells.

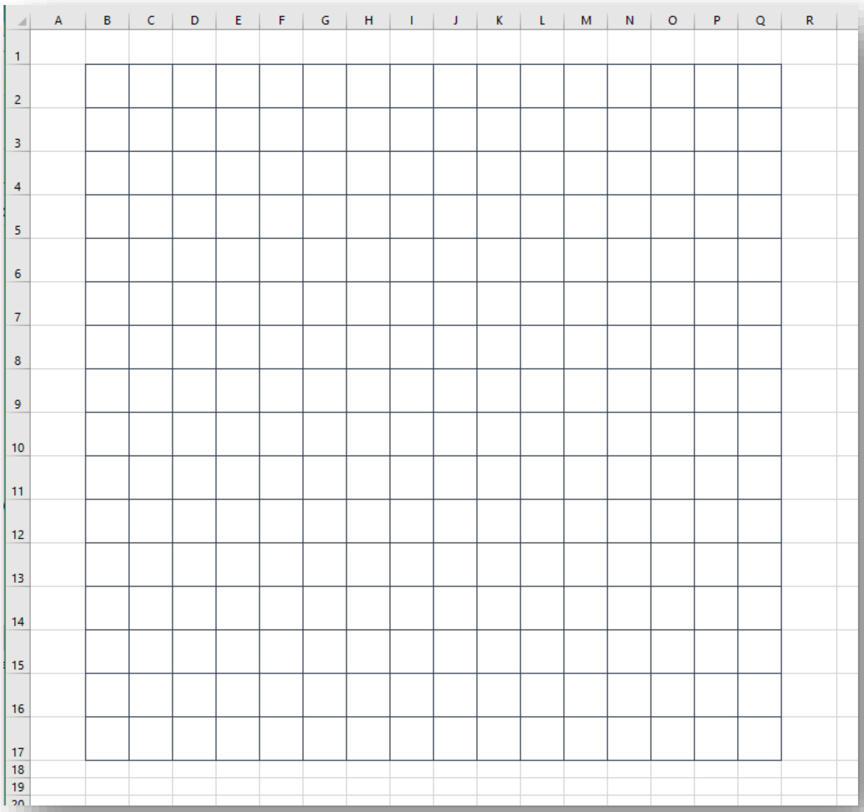
To begin with, we need a project. Create a folder called "AMazing" and a project of the same name. Save the project to that folder.



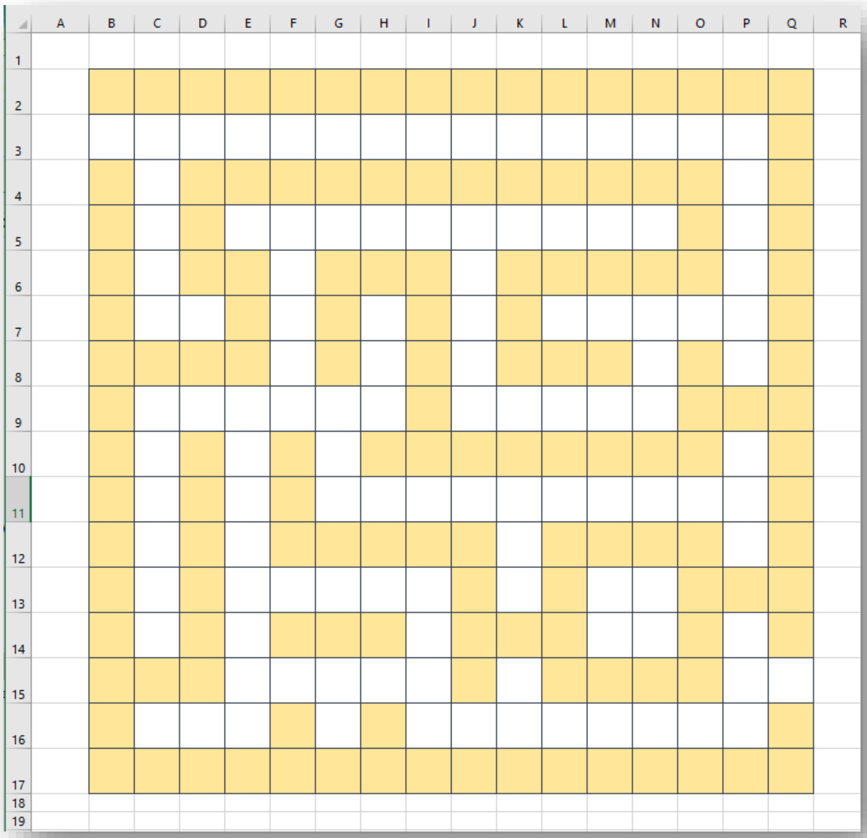
```
SetErrorMode(2)

// set window properties
SetWindowTitle( "AMazing" )
SetWindowSize( 800, 800, 0 )

// set display properties
SetVirtualResolution( 800, 800 )
SetOrientationAllowed( 1, 1, 1, 1 )
SetSyncRate( 30, 0 )
UseNewDefaultFonts( 1 )

do

    Sync()
loop
```

Set the resolution to 800 x x800 and run the program to produce the media folder, the location we will use for our sprites and background.

Now, launch Excel and create a new blank sheet. Our cell size allows for exactly 16 across and 16 down. Take 16 columns and 16 rows and change the cell height and width to 50 pixels. Put a border of any colour on those cells to make them stand out from the rest. You should have something like the image shown below.



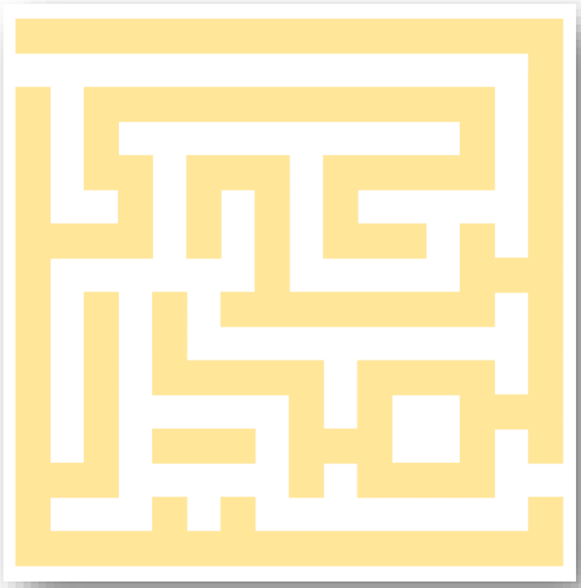
To create the maze, just use the cell fill colour tool. Choose whichever colour you want and make a maze similar to the one shown:



	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q
2	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨
3																🟨
4	🟨		🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨		🟨
5	🟨		🟨											🟨		🟨
6	🟨		🟨	🟨		🟨	🟨	🟨		🟨	🟨	🟨	🟨	🟨		🟨
7	🟨			🟨		🟨		🟨		🟨						🟨
8	🟨	🟨	🟨	🟨		🟨		🟨		🟨	🟨	🟨		🟨		🟨
9	🟨							🟨						🟨	🟨	🟨
10	🟨		🟨		🟨		🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨		🟨
11	🟨		🟨		🟨											🟨
12	🟨		🟨		🟨	🟨	🟨	🟨	🟨		🟨	🟨	🟨	🟨		🟨
13	🟨		🟨						🟨		🟨			🟨	🟨	🟨
14	🟨		🟨		🟨	🟨	🟨		🟨	🟨	🟨			🟨		🟨
15	🟨	🟨	🟨						🟨		🟨	🟨	🟨	🟨		
16	🟨				🟨		🟨									🟨
17	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨	🟨

Spruce it up in Photoshop

We have our basic maze pattern, so we can make it look a bit better in Photoshop. Go back to Excel and make a copy of Sheet1. On the copy sheet, remove the borders from the maze and the gridlines from the whole sheet. (found under the View tab)



Using the mouse, highlight all the cells belonging to the maze. Then on the home tab click on the little arrow head pointing down that is located beside the "Copy" command.

Choose "Copy as Picture", then check "Bitmap" and click "OK"

Launch Photoshop, create a new file size 800 x 800 pixels and paste in your maze.

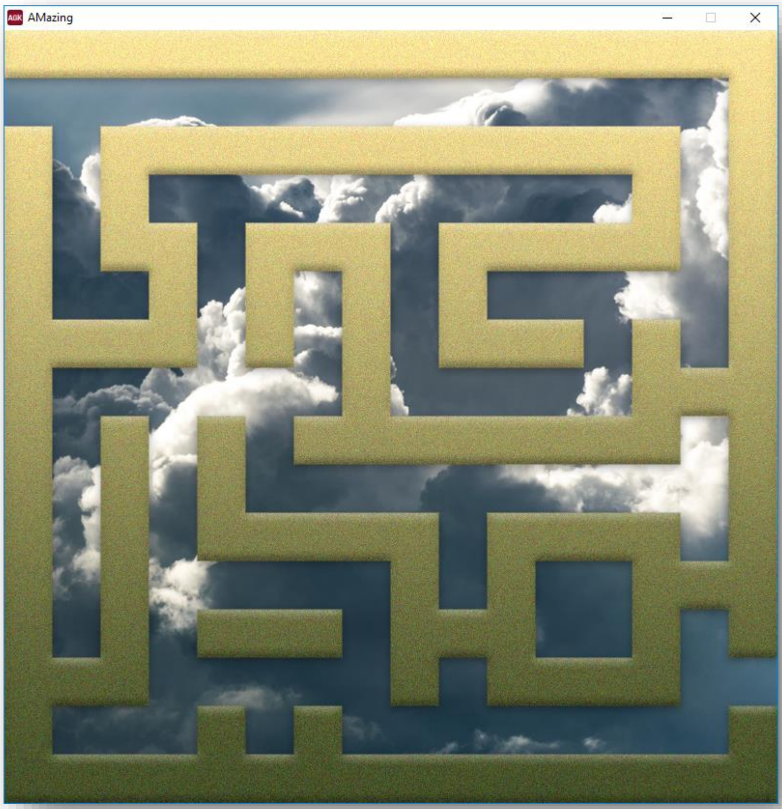
From here you can make your maze look as nice as you like, but I'll leave that part to you. When you're finished, save it as a .jpg called Maze, into the media folder created earlier.

Add the following two lines just before the "do" loop and run your program.

```
LoadImage(2001,"Maze.jpg")
CreateSprite ( 201, 2001 )
```

You should now see your completed maze on screen. Here's my version: which is nicer??

This tutorial is not about Photoshop or sprite creation. So I'll leave it to you to create your player.



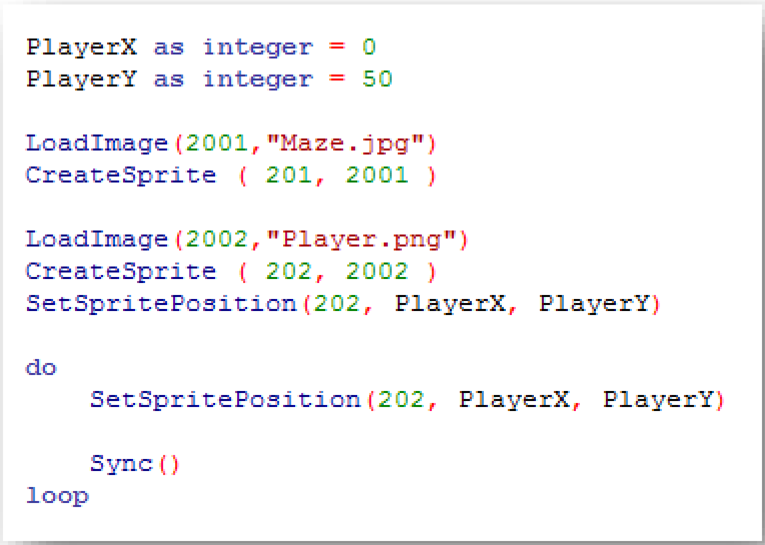
Just make sure it fits into a 50 x 50 cell size.

I'm going with this basket ball:

Save it to the media folder and call it "Player" Make it a .jpg or a .png if you need to maintain transparency like my example.

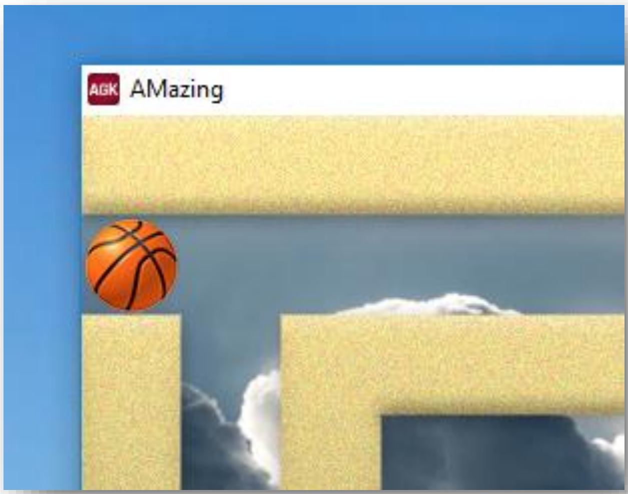
Load the image and create a sprite like before; also create two integer variables to store the X and

Y co-ordinates of the player when it is moving around the screen and draw it on screen so that it is constantly having its location updated – so inside the “do” loop. Your code should now look something look this:



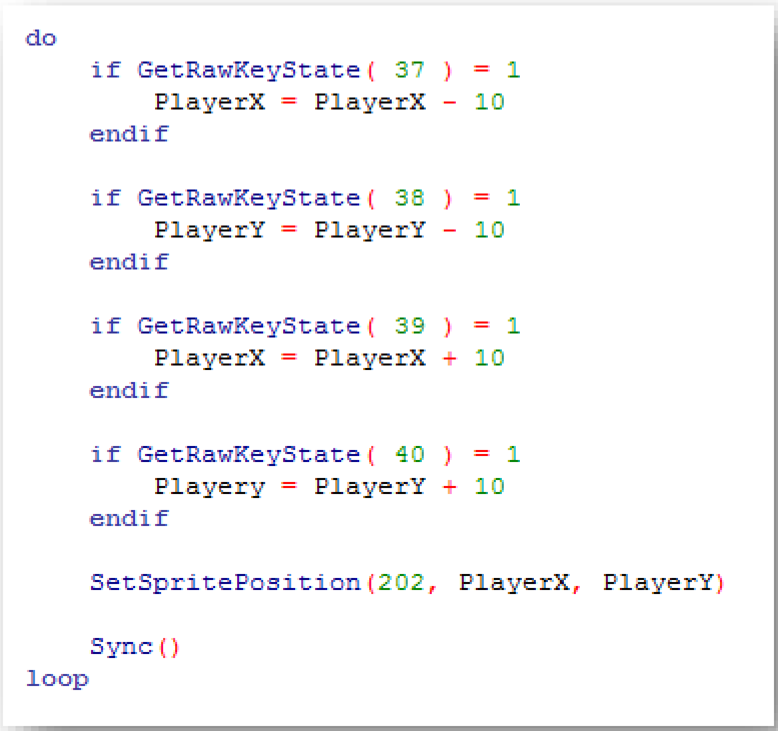
```
PlayerX as integer = 0
PlayerY as integer = 50

LoadImage(2001,"Maze.jpg")
CreateSprite ( 201, 2001 )

LoadImage(2002,"Player.png")
CreateSprite ( 202, 2002 )
SetSpritePosition(202, PlayerX, PlayerY)

do
    SetSpritePosition(202, PlayerX, PlayerY)

    Sync()
loop
```

The “Y” location of my player has been set to 50, which places the ball nicely at the start position of my maze. Check out the screenshot below:



We’ll now make the ball move up -> down -> left -> right by responding to key presses. You will have done this before in earlier tutorials, so go ahead and add the code to make the ball move by 10 pixels in all directions. Don’t worry about going through walls or off the screen at this point.

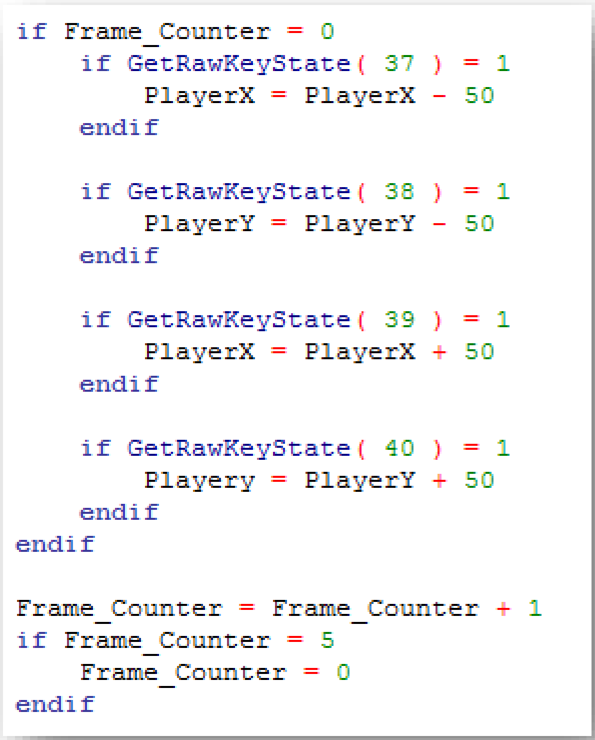
To refresh your memory, here is the code that will complete the movement described:



```
do
    if GetRawKeyState( 37 ) = 1
        PlayerX = PlayerX - 10
    endif

    if GetRawKeyState( 38 ) = 1
        PlayerY = PlayerY - 10
    endif

    if GetRawKeyState( 39 ) = 1
        PlayerX = PlayerX + 10
    endif

    if GetRawKeyState( 40 ) = 1
        Playery = PlayerY + 10
    endif

    SetSpritePosition(202, PlayerX, PlayerY)

    Sync()
loop
```

I said at the beginning that we would keep this simple. So, with that in mind change the code so that the ball moves 50 pixels at a time. That will make it easier for the player to line the ball up with the paths through the maze. However, the speed is now 5 times faster and needs to be slowed down. We could change the frame rate, but that would ultimately affect any other sprites we might create. We will just allow the ball to move once every 5 frames rather than every frame by including our own delay code.

Create a variable called Frame_Counter and set it to zero. Amend the code so that it now looks like this:



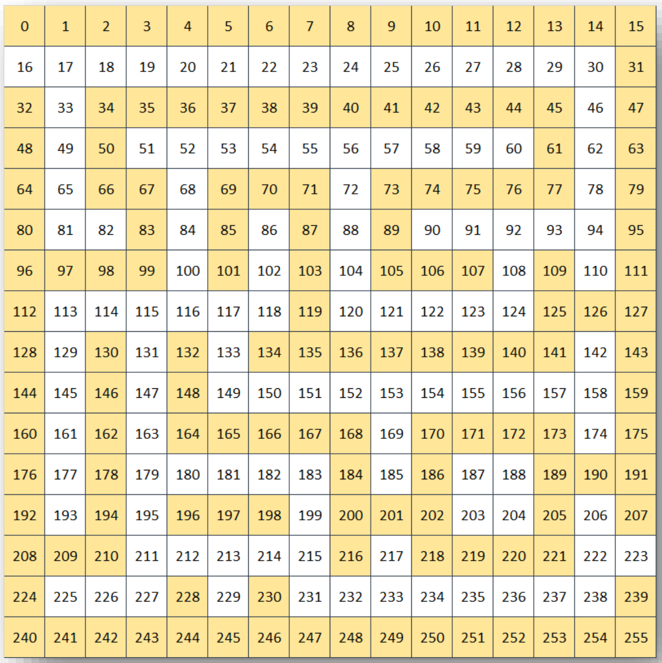
```
if Frame_Counter = 0
    if GetRawKeyState( 37 ) = 1
        PlayerX = PlayerX - 50
    endif

    if GetRawKeyState( 38 ) = 1
        PlayerY = PlayerY - 50
    endif

    if GetRawKeyState( 39 ) = 1
        PlayerX = PlayerX + 50
    endif

    if GetRawKeyState( 40 ) = 1
        Playery = PlayerY + 50
    endif
endif

Frame_Counter = Frame_Counter + 1
if Frame_Counter = 5
    Frame_Counter = 0
endif
```

Test your program.

We are now ready to implement the actual collision testing using an array.

The Array

It will be helpful to return to the Excel grid again for this part as it will help us to visualise the relationship between our array and the maze. Our maze contains 16 cells horizontally and 16 cells vertically – giving a total of 256 cell locations. So that will be the size of our array. As arrays start with an index value of zero, we can visualise it like this in the spreadsheet.

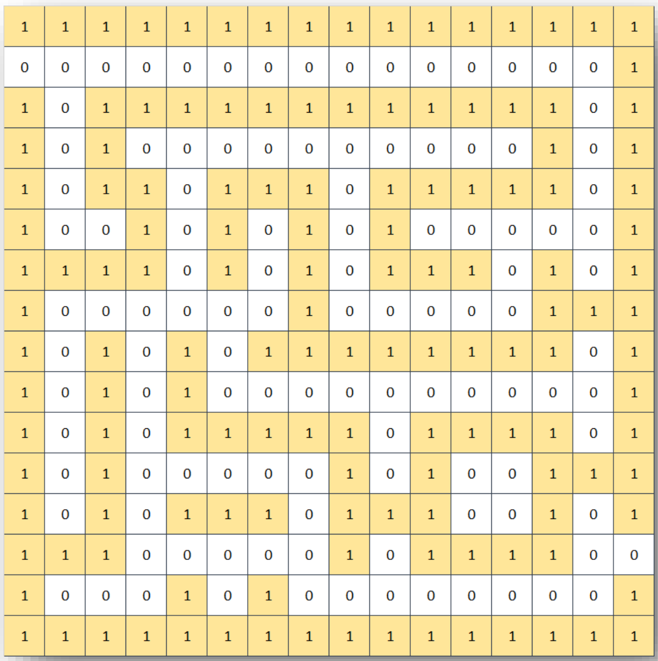


0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

It will help if you put these values into a copy of your spreadsheet.

Our array will just contain one of two values to represent places where the player can walk and walls. So, the values 0 and 1 will suffice. Here is another copy of the maze in the spreadsheet with 0's and 1's marked.

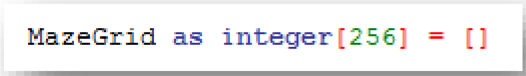
Do that to a version of yours as well.



1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1
1	0	1	1	1	1	1	1	1	1	1	1	1	1	0	1
1	0	1	0	0	0	0	0	0	0	0	0	0	1	0	1
1	0	1	1	0	1	1	1	0	1	1	1	1	1	0	1
1	0	0	1	0	1	0	1	0	1	0	0	0	0	0	1
1	1	1	1	0	1	0	1	0	1	1	1	0	1	0	1
1	0	0	0	0	0	0	1	0	0	0	0	0	1	1	1
1	0	1	0	1	0	1	1	1	1	1	1	1	1	0	1
1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	1
1	0	1	0	1	1	1	1	1	0	1	1	1	1	0	1
1	0	1	0	0	0	0	0	1	0	1	0	0	1	1	1
1	0	1	0	1	1	1	0	1	1	1	0	0	1	0	1
1	1	1	0	0	0	0	0	1	0	1	1	1	1	0	0
1	0	0	0	1	0	1	0	0	0	0	0	0	0	0	1
1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1

We need to create the array now in our game code and populate it with 0's and 1's like in my image.

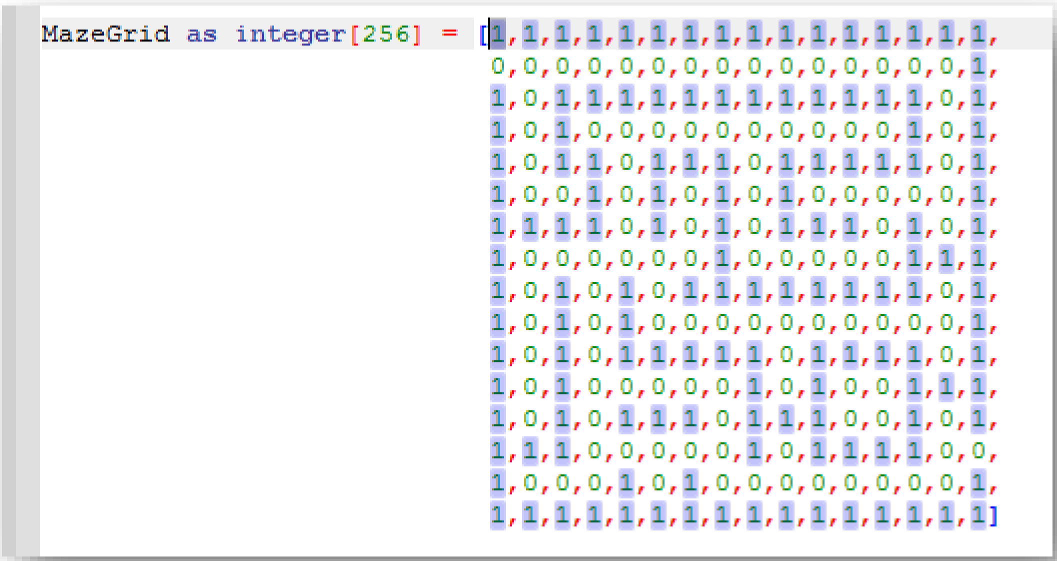
Fortunately, as we have used a spreadsheet, we can save it as a CSV file (Comma Separated Values) and then just copy and paste them into our code.



```
MazeGrid as integer[256] = []
```

The above line of code shows how a single dimension array is created in AppGameKit 2 where we can specify initial values within the square brackets.

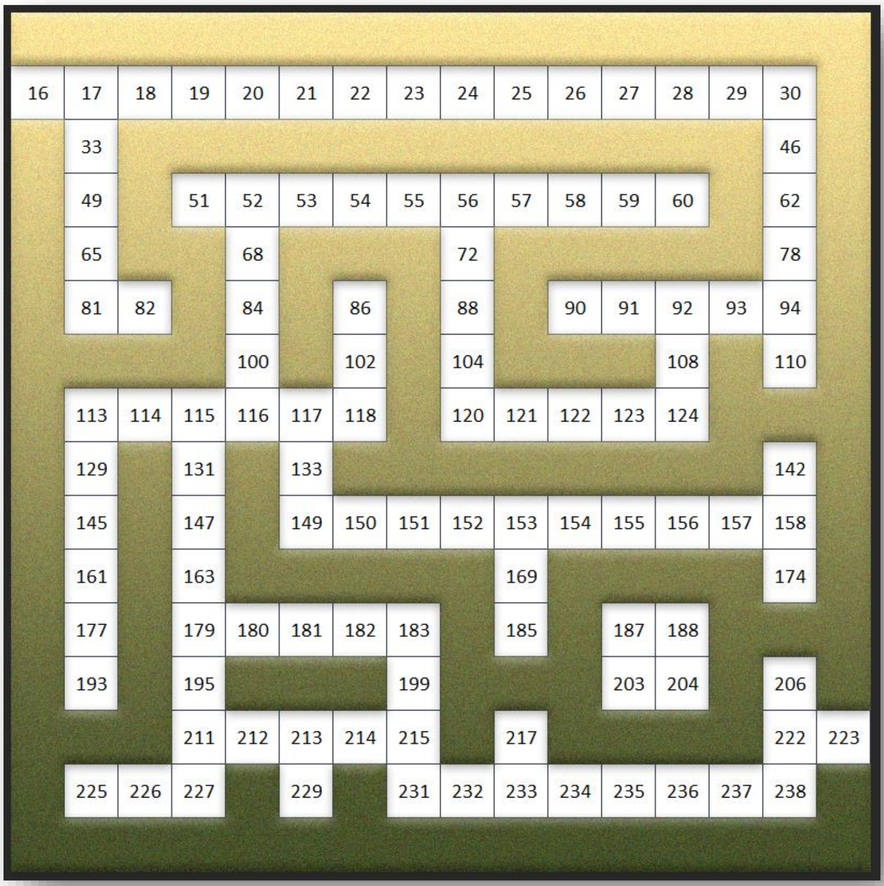
To get the values, save the sheet as a CSV file, open the file using Notepad, copy the values and paste them into your code. Make sure you have a comma separating each of the values and you should end up with something like this:



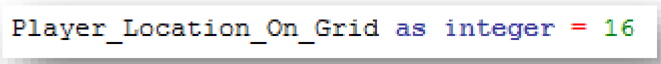
```
MazeGrid as integer[256] = [1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
                            0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,
                            1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,
                            1,0,1,0,0,0,0,0,0,0,0,0,0,1,0,1,
                            1,0,1,1,0,1,1,1,0,1,1,1,1,1,0,1,
                            1,0,0,1,0,1,0,1,0,1,0,0,0,0,0,1,
                            1,1,1,1,0,1,0,1,0,1,1,1,0,1,0,1,
                            1,0,0,0,0,0,0,1,0,0,0,0,0,1,1,1,
                            1,0,1,0,1,0,1,1,1,1,1,1,1,1,0,1,
                            1,0,1,0,1,0,0,0,0,0,0,0,0,0,0,1,
                            1,0,1,0,1,1,1,1,1,0,1,1,1,1,0,1,
                            1,0,1,0,0,0,0,0,1,0,1,0,0,1,1,1,
                            1,0,1,0,1,1,1,0,1,1,1,0,0,1,0,1,
                            1,1,1,0,0,0,0,0,1,0,1,1,1,1,0,0,
                            1,0,0,0,1,0,1,0,0,0,0,0,0,0,0,1,
                            1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1]
```

I've highlighted the 1's so that you can clearly see the pattern of the maze. Although it is a single dimension array, it is helpful to lay it out like this. All we have to do now is navigate through the maze in our array and update the X and Y location of the ball on screen accordingly.

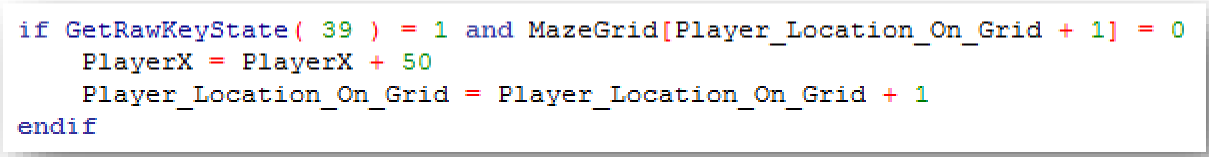
Right, that might have been over simplifying things, so let's break it down a bit. Here's my maze in Photoshop with the array locations of the places we can walk on overlaid.



From the image we can see that our player (the ball) starts off located at index value 16 in the array. So we should create a variable in the code to reflect that.


```
Player_Location_On_Grid as integer = 16
```

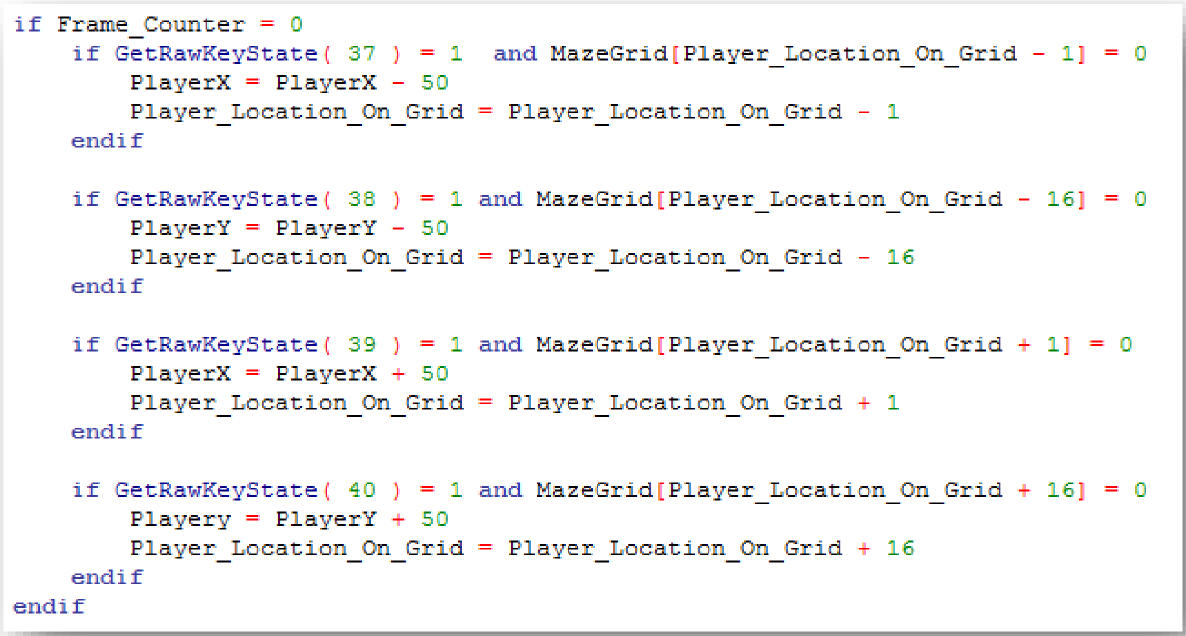
We can also see the locations where we can walk and those that are walls. If we go back to our code for moving to the right and amend it, we just have to ensure that the contents of the array location to the right (or one place greater than 16 contains a zero and not a one. Here's the amended code:


```
if GetRawKeyState( 39 ) = 1 and MazeGrid[Player_Location_On_Grid + 1] = 0
    PlayerX = PlayerX + 50
    Player_Location_On_Grid = Player_Location_On_Grid + 1
endif
```

As before, 39 represents the "Right Arrow" key. Added to that is a test to see if the array "MazeGrid" contains a zero at the current location of the player, plus one, location 17. Only if both conditions are true do we move the ball by adding 50 to its X location. We must also change the location of our player in the array, and that is achieved by increasing the relevant variable by one.

You should be able to determine how to amend the code for moving left! It's the same as above but we subtract one rather than adding one when checking if it is okay to move.

Moving up and down is similar, but because there are 16 cells in each row we have to either add or subtract 16 instead of one to verify if it is okay to move.


```
if Frame_Counter = 0
    if GetRawKeyState( 37 ) = 1  and MazeGrid[Player_Location_On_Grid - 1] = 0
        PlayerX = PlayerX - 50
        Player_Location_On_Grid = Player_Location_On_Grid - 1
    endif

    if GetRawKeyState( 38 ) = 1 and MazeGrid[Player_Location_On_Grid - 16] = 0
        PlayerY = PlayerY - 50
        Player_Location_On_Grid = Player_Location_On_Grid - 16
    endif

    if GetRawKeyState( 39 ) = 1 and MazeGrid[Player_Location_On_Grid + 1] = 0
        PlayerX = PlayerX + 50
        Player_Location_On_Grid = Player_Location_On_Grid + 1
    endif

    if GetRawKeyState( 40 ) = 1 and MazeGrid[Player_Location_On_Grid + 16] = 0
        Playery = PlayerY + 50
        Player_Location_On_Grid = Player_Location_On_Grid + 16
    endif
endif
```

The code shown completes the requirements for all four directions. Make the amendments and test your maze game. If all is well you will not be able to walk through the walls anymore. So you should be able to see that, although it looks like the ball is colliding with the walls, it is, in fact, just following an instruction in a line of code that checks a cell ahead of where you want to go and doesn't do anything if it's not free.