Simple Tetris Tutorial

Prolog

I am a Computer Science student and the Canterbury University, New Zealand. I have been programming for roughly 5 years and participated in the creation of commercial software. I use Game Maker for a lot of proof-of-concept work and a few small games. This is my first tutorial, but as I make more I will be posting them here. Check back for updates.

This tutorial does not assume you know anything past the basics of use, it is primarily for new users to learn how some gml and some basic game design features. It uses code, rather than drag and drop because I believe this to be a cleaner, better way to work. Before you get scared however, the code is all explained.

I have an idea, let’s go!

As the title suggests, this is a tutorial for making a Tetris game. This must be nice and simple, right? I mean, you just need some blocks to fall. Oh, and make them disappear when a line is made. Well, that is true. The system for Tetris is pretty simple, but there’s a little more to it than just that. You need to know what blocks to fall. What happens when they rotate or are sped up? How is the score calculated? Do we want a score table and menus?

Even though the idea of Tetris is simple there are a lot of things to handle when actually making it, that’s why commercial games are made by a team of people and can take years to finish. Fortunately for you, we will be able to make the core of the game in a few hours. Now we can actually begin.
The blocks

The main feature of Tetris is its falling blocks. These are formed in groups of four, as shown below

![blocks](image)

The Block Object

We will make an object for each of the small squares that make up a block. You may be thinking “Why can’t we just make an image for each block and use that?” Well, we could do that, which would make rotating them and dropping them nice and simple, but how are we going to deal with removing lines? It is possible to do it that way still, but it is far too complicated, if we model each group as a collection of squares, we can remove squares as we see fit.

To make our square, we will start with the object, and then make the sprite. You could do it the other way around, but for more complex games you do not always know how the objects are going to work until you make the actual object. If you have to compromise your code to fit the images, I believe you are going about it the wrong way.

Ok, make a new object and give that object a name; I called mine ‘Block.’

Block is a nice simple object, all it has to do is draw itself in the right position, and this is done automatically if we do not put anything in the draw event. All we are going to do is make sure that the visible tick box is ticked and that the solid tick box is un-ticked. A good rule of thumb here is that if an object moves; it is not solid, if it is stationary; make it solid.

The Block Sprite

Now the block needs a sprite. A sprite is the image that represents an object. The size of the sprite is dependent on your grid size. To work out your grid size, you need to think about how big the room
is, how many rows you want to have in the game and how big the borders around the game are. I will go into more detail about the room size a little later, so for now I will just state it is 800 by 600 pixels. That is, the window our game is in will draw 800 pixels wide and 600 pixels high. We will keep a frame around the playing area that is about 40 pixels wide. To get the height of the area we subtract 40 from 800 twice, once for the top and once for the bottom. That leaves us with 520 pixels reserved for blocks to fall in. I want to be able to handle 20 rows high, so if we divide the height by 20 we get the block height. That is 26 pixels. Because the blocks are square, we will make them 26 by 26 pixels.

Now that we know the dimensions, we can make a square. Create a sprite that is 26 pixels wide and 26 pixels high, make it a square and save it. We also want to consider its origin. This determines where the sprite is drawn in relation to the object’s x and y positions. Making it 0 for both is suitable for these blocks. I gave my block the name ‘spr_block.’ I prefixed it with ‘spr_’ so that I would always know it is a sprite.

Finally, you can set the sprite to the block. Do this by double clicking on the block, clicking its sprite section and selecting ‘spr_block.’ Now that our block is finished, we can make a group of blocks.

**A simple line group**

The first group we will make is the line; made from four blocks stacked on top of each other. The methods used for this group will be used for the other ones, this is just the easiest to visualise.

A group of blocks needs to be able to drop down, rotate, stop when one of the blocks collides with another block and not move outside of the usable grid.

**The group of blocks**

The first thing we will do is create our group of blocks. If we think of our group as being one main object, there is two blocks above its centre and two below. We know the dimensions of the blocks so we can create them all at the relative positions to the centre. Create a new object and name it “LineGroup.”

We will be storing the blocks in an array; which is sort of like a list of objects, numbers, strings or anything else. Arrays have a name and use a number to identify which item in the list you are dealing with, starting from zero.

Our array will be called ‘blocks’ and will have four items, so will go from zero to three. In the create event of LineGroup add a new piece of code. Do this by going to the control tab and selecting the ‘Execute Code’ tool. Here is the first piece of code we will add:

```javascript
blocks[0] = instance_create(x, y - (26 * 2), Block).id;
```

Before adding the other blocks, I will explain this code. First part is the variable name. Blocks[0] indicates that we want to use the very first item of the blocks array. This can then be used like any other variable. The next thing we do is set that variable to a value. Our value for this is the id of the
Block object that is at the top of our line group. We give it the group’s x value for its own, because it is in the same place. The next thing we do is give it the y value minus some number. We subtract a number to signal that we want it to be above the line (the y values increase going down the window and the x values increase going to the right). We subtract $2 \times 26$ because the sprites are 26 pixels high and the Block is two blocks above. Finally we get the id of the object. This is the unique identifier that lets us use this object in the future.

Now we create the other three blocks as follows:

```
blocks[1] = instance_create(x, y - (26 * 1), Block).id;
blocks[2] = instance_create(x, y, Block).id;
blocks[3] = instance_create(x, y + (26 * 1), Block).id;
```

Now we have a list of blocks for our group and we can think about moving them down the page.

### Moving Down in the World

At regular intervals, we want all of the blocks that make this shape to drop down one value in the grid. For this to happen we need to know two things. One is the grid size, which we know. The other is the time between movements. We will make a variable in the create event to hold these values so we can use them wherever we want and if we want to change them at any stage we only have to change them at one place.

Below the code for creating blocks, add in the following two lines

```
grid_size    = 26;
falling_rate = room_speed;
```

Setting the grid size is simple enough, we just state that the grid is 26 pixels wide and high for each block.

For the falling_rate, we have introduced an important new variable; the room_speed. This variable is set in the settings for rooms and tells you how many times the game us updated (or calls its step event) every second. By setting our falling rate to room_speed we know that for any speed the room is set to, our blocks will drop once every second. This way if we want to change the room speed we don’t need to change any values that handle speeds.

Now we can actually tell the blocks to fall down. The easiest way to do this is to use alarms. This is also the most reliable way to do it. However there is a limited number of alarms and this is a tutorial to teach you new things, so we will make our own alarm. Create a new variable in the create event, after the above code, and name it ‘time_to_next_drop.’ Set this variable to falling_rate as follows.

```
time_to_next_drop = falling_rate;
```

Now we need to create some code in the step event. To do this, we will make a script that depicts how the group will move, and then we will put that in the step event. We are doing this because all of the groups will move in the same way and we can save writing out the same code multiple times by using a script.

Create a new script and name it ‘shift_down.’
The code for this is simple, we check to see if the time to the next block is zero. If it is, shift all the blocks down and reset the timer. After this check, we also want to decrease the timer by one, to signal that we have completed a step. We decrease the timer immediately after we have finished doing things that rely on it. The code for this is as follows:

```c
if ( time_to_next_drop <= 0) {
    for(i = 0; i < 4; i += 1) {
        blocks[i].y += grid_size;
    }
    time_to_next_drop = falling_rate;
    y += grid_size;
}
time_to_next_drop -= 1;
```

There are three important things in this code.

The first is the if-then-else statement. We check if the time to the next block is less than or equal to zero by putting it in a pair of parenthesis and placing the word ‘if’ before it. This will check that statement, if it is true it will do the things inside the {} section, if it is false then it will see if there is an else statement and perform its commands. We do not need an else statement because nothing happens if the check is false.

The second thing is the ‘for loop’. This loops through a block of code while the conditions are met. The first condition is that for the first loop i is zero. The second condition is that we only start the block of code if i is still less than four. The third condition is that at the end of the loop i is increased by one. Using this we are able to loop through our four blocks and move them all down by one grid position.

The third is how we change the y value of each block, as well as for the controller. If you remember, we set the value in the blocks array to the id of each block. When we want to use a block’s variables, the code ‘id.variable’ will return the variable and let us manipulate it. The ‘+=’ part is just saying “add the following from our variable.” You can also use ‘-=’, ‘*=’ and ‘/=’ to subtract, multiply or divide the variable by a number.

Now moving to the step event for our LineGroup, add in a code segment like you did in the create event and put the following line

```c
shift_down();
```

This will run through all the code in the shift_down script and let it use all of the LineGroup’s variables.

**Moving Left to Right**

Ok, now that our blocks can fall, we should probably let them move left to right. It’s not really a Tetris game if all you do is watch blocks fall down from one place. To do this, we need to do four things. Check for keyboard input, check for availability in that direction and then move only if we haven’t moved recently. We do this last check to make sure the blocks don’t move too fast.
Before we actually get the input, we will set up the variables for restricting movement speed. In the `create` event for the LineGroup object, put the following code:

```plaintext
shift_rate = room_speed / 4;
time_to_next_shift = 0;
border_size = 40;
```

What we have done here is set the number of times we can shift left or right in a second to 4 and we have also set the time to next shift to zero. This is because we can move left or right immediately. In the `step` event for the Line, add in a line to decrease the time to the next shift by one. Do this as follows:

```plaintext
time_to_next_shift -= 1;
```

Now when we get the input we can check to see if the time to next shift is less than or equal to zero, if it is then go and shift.

To get keyboard input, we will use the keyboard events. We could create our own in the step event but that is rarely needed.

Because moving left and right is done the same way for each of the groups, we can use another script. Make a script called `shift_left` and put the following code in it:

```plaintext
if (time_to_next_shift <= 0) {
    can_move = true;
    for(i = 0; i < 4; i += 1) {
        if(blocks[i].x <= border_size) {
            can_move = false;
        }
    }
    if(can_move) {
        for(i = 0; i < 4; i += 1) {
            blocks[i].x -= grid_size;
        }
        x -= grid_size;
        time_to_next_shift = shift_rate;
    }
}
```

One major thing here is the ‘can_move’ variable. We use this to check if we are able to move to the left any more. We check all the blocks to make sure that they aren’t already at the left border. If one of them is, it sets can_move to false and then none of them will move.

Put a call in the event for the right keyboard arrow to this script as follows

```plaintext
Shift_left();
```

We will edit that script later to check for other blocks that we might collide with but for now it is complete.
We do a very similar thing for the right arrow event, but we need to add in another value. We need to know the number of columns in the game. We will use 10, so add in a variable in the create event again.

```javascript
number_of_columns = 10;
```

We can now work out the width of the playing area and use that like we did the border for shifting left.

Create a script called `shift_right` and put a call to it in the right arrow event just like you did with `shift_left` in the left arrow event.

```javascript
if (time_to_next_shift <= 0) {
    can_move = true;
    for(i = 0; i < 4; i += 1) {
        if(blocks[i].x + grid_size < border_size + (grid_size * 10)) {
            can_move = false;
        }
    }
    if(can_move) {
        for(i = 0; i < 4; i += 1) {
            blocks[i].x += grid_size;
        }
        x += grid_size;
        time_to_next_shift = shift_rate;
    }
}
```

**Rotation**

Ok, we’re starting to get a system that seems kind of like Tetris. We have a group of blocks that will move down, left and right while staying within our borders. Now we need to get those blocks to rotate. After that we can think about having them build up.

Rotating the blocks can be done in two ways; mathematically or using pre-defined rotations. We are going to use pre-defined rotations because they are much easier, but I will give you a basic idea of how it could be done mathematically.

We know that the distance from the centre of the group is a fixed value and that the blocks are rotated at 90 degree angles. Using both of these we can use Pythagoras or Matrix Rotation to rotate them. Pythagoras is basically a method for calculating lengths of sides on a triangle from their angles and vice-versa and Matrix Rotation changes the position of an item in an array. I might go into a little more information on that in a later tutorial, but it is not necessary at the moment so we will leave it there.
We will pre-determine the position of the blocks and then change that configuration when they're rotated. To do this, we will use another type of array; the 3D array. You can think of this as an array of arrays; for every item in the array there is a list of values.

We will have a separate array for the x and y values. The first value in the array will be the rotation number and the second will contain the list of x or y coordinates for each object (there will be four here.) The LineGroup will require two sets of rotation values, one for when it’s upright, and the other for sideways. Different group types will have different numbers of rotations available so we will also store the number of rotations and the current rotation. Put the following in the create event for your LineGroup:

```csharp
x_offset[0,0] = 0;
x_offset[0,1] = 0;
x_offset[0,2] = 0;
x_offset[0,3] = 0;
x_offset[1,0] = -1 * (2 * grid_size);
x_offset[1,1] = -1 * (1 * grid_size);
x_offset[1,2] = 0;
x_offset[1,3] = 1 * (1 * grid_size);
y_offset[0,0] = -1 * (2 * grid_size);
y_offset[0,1] = -1 * (1 * grid_size);
y_offset[0,2] = 0;
y_offset[0,3] = 1 * (1 * grid_size);
y_offset[1,0] = 0;
y_offset[1,1] = 0;
y_offset[1,2] = 0;
y_offset[1,3] = 0;
current_rotation = 0;
number_of_rotations = 2;
```

The first value for each array is the index, which is the rotation number; the second value is the block it refers to. This way, when we want to rotate each block we just need to increase the value of the current rotation, make sure we keep it under the number of rotations and finally set the position of each block relative to the group positions.

To do this, create a script called `rotate_blocks` and put the following code:

```csharp
current_rotation += 1;
if(current_rotation >= number_of_rotations) {
    current_rotation = 0;
}
```
can_move = true;
for(i = 0; i < 4; i += 1) {
    new_x = x_offset[current_rotation,i] + x;
    if((new_x >= border_size + (grid_size * 10)) or
        new_x < border_size) {
        can_move = false;
    }
}
}
if(can_move) {
    for(i = 0; i < 4; i += 1) {
        blocks[i].x = x_offset[current_rotation,i] + x;
        blocks[i].y = y_offset[current_rotation,i] + y;
    }
}
We first work out what the new x position for each block will be and make sure that we can move there, once we have check all of those positions and have determined that we can move, we do.

Now, for every block in blocks, set that block’s position to the x and y offset value for that block that we have made.

All that is left for us now is to call that code from the ‘up arrow’ key pressed event as follows:
rotate_blocks();
And just like that, every time the user presses the up arrow the blocks will rotate.

**Collisions**

Excellent, we now have falling blocks that can be moved and rotated! Now what we need to do is get them to both stop at the bottom and stop when they hit other blocks. To do this we are going to have to change our code a little bit for all the moving. Not only do we need to check the positions left and right for boundaries, we also need to check for blocks there. The same must be done for moving down. First we will check for the bottom edge of the area.

This is nice and easy, we just need to check if we are at the bottom. If our next move is going to take us past the bottom then we shouldn’t move there. At that point, we will also do two other things. We will make the blocks solid and we will destroy our group. If you remember correctly, I stated that static objects should be solid and moving ones should not. While the blocks are moving down, they are not solid because they are moving, but once they hit the bottom they stop moving, so we can safely make them solid again. What’s more is that we can then check for solid objects below us or to the side when we are moving them.

Now open up the shift_down script. We need to modify this to check for objects below. There is a pretty major new piece of code in here that I will explain following the code. The rest of it is similar to what you have seen already.

Here is the new code for shift_down:

```cpp
if ( time_to_next_drop <= 0) {
```
can_move = true;
for(i = 0; i < 4; i += 1) {
    if((blocks[i].y + grid_size) >= (border_size + (20 * grid_size))) {
        can_move = false;
    }
    with(blocks[i]) {
        if(!place_free(x, y + other.grid_size)) {
            other.can_move = false;
        }
    }
}
if(can_move) {
    for(i = 0; i < 4; i += 1) {
        blocks[i].y += grid_size;
    }
    time_to_next_drop = falling_rate;
    y += grid_size;
}
else {
    for(i = 0; i < 4; i += 1) {
        blocks[i].solid = true;
    }
    instance_destroy();
}
time_to_next_drop -= 1;

As I said, there is a major new piece of code in here. There is actually two. The first is the ‘with(blocks[i])’ and the second is the ‘position_empty.’

First, the with statement. This is kind of like doing the following “run the following code for an object and treat me as other.” So we give it ‘blocks[i]’ to use and then we can check the y value without calling blocks[i] and we have to get the grid size by calling other.grid_size. The reason we use this with statement is because we are using position_empty.

First, the ‘!’ symbol is saying “Consider the opposite of the next statement.” So if the next statement is true, we think of it as false. For example, “!true” is false and “!false” is true. Next we have the actual position_empty part. This will check to see if there is a collision with any solid object at the given x,y coordinate pair if we place our sprite in that position; in this case the position we are wanting to move to. Because the falling blocks are not solid, it will not return true when it collides with them, which is why we make them solid when we ‘land.’ The reason we put this in the with statement is because if we didn’t, we would be checking the groupLine object at that position, and the groupLine object doesn’t have a sprite to check with.
We will also put the line “time_to_next_drop = 1;” in the keyboard Down event. This way when the Down key is held, the block with move down twice as fast.

Then we have the check to see if we can move. If we can, we do. If we cannot, we set the ‘solid’ value of each block to true and destroy our self. We destroy our self because it no longer needs to be treated as a group of blocks, now each block is an individual.

Finally, we need to check the positions left and right to make sure we can move there. The way we do that is just the same as checking the bottom, so I will simply paste the code and you can see what has changed.

**Code for shift_left:**

```java
if (time_to_next_shift <= 0) {
    can_move = true;
    for(i = 0; i < 4; i += 1) {
        if(blocks[i].x <= border_size) {
            can_move = false;
        }
        with(blocks[i]) {
            if(!place_free(x - other.grid_size,y)) {
                other.can_move = false;
            }
        }
    }
    if(can_move) {
        for(i = 0; i < 4; i += 1) {
            blocks[i].x -= grid_size;
        }
        x -= grid_size;
        time_to_next_shift = shift_rate;
    }
}
```
Code for shift_right:

```java
if (time_to_next_shift <= 0) {
    can_move = true;
    for(i = 0; i < 4; i += 1) {
        if(blocks[i].x + grid_size < border_size + (grid_size * 10)) {
            can_move = false;
        }
        with(blocks[i]) {
            if(!place_free(x + other.grid_size, y)) {
                other.can_move = false;
            }
        }
    }
    if(can_move) {
        for(i = 0; i < 4; i += 1) {
            blocks[i].x += grid_size;
        }
        x += grid_size;
        time_to_next_shift = shift_rate;
    }
}
```

Finally, we also need to check with rotating the blocks. Change the rotate_blocks() to the following, which uses all the same methods explained above.
current_rotation += 1;
if(current_rotation >= number_of_rotations) {
    current_rotation = 0;
}

can_move = true;
for(i = 0; i < 4; i += 1) {
    new_x = x_offset[current_rotation,i] + x;
    if((new_x >= border_size + (grid_size * 10)) or
       new_x < border_size) {
        can_move = false;
    }
    with(blocks[i]) {
        if(!place_free(other.new_x,y)) {
            other.can_move = false;
        }
    }
}

if(can_move) {
    for(i = 0; i < 4; i += 1) {
        blocks[i].x = x_offset[current_rotation,i] + x;
        blocks[i].y = y_offset[current_rotation,i] + y;
    }
}

With that, we have completed the code for our block groups. They will drop at a fixed rate, move left to right and stop when there is a collision. Now we just need to make sure the next one comes down when this one is finished, check for completed rows and add in our other block types. All of these things can be done in a game controller.
The Game Controller

The game controller is an object that controls how our game operates. It will be responsible for checking for completed rows, dropping the next block group, scoring, speed and checking for a finished game.

Creating New Groups

The first thing we will do is make it so that when one group of blocks has stopped, the next group will start. To do this, we first need to make a game controller object and put some variables in its create event. My game controller will be called ‘GameController.’

We will put a few things in here, preparing it for later events as well as current. We will store an array of all the block groups, even though we only have one at the moment. We will also store the number of block groups and the index of the next one, the same grid size and border size as the block groups, the starting x and y positions for the groups and the current level and size the score increases each time you increase a row. Finally, we will store the id of the current group that is falling.

Put all that together and you have the following code:

```plaintext
border_size = 40;
grid_size = 26;

number_of_rows = 20;
number_of_columns = 10;

score_value = 10;
current_level = 0;

group[0] = LineGroup;

number_of_groups = 1;
next_group = 0;

group_start_x = border_size + ((number_of_columns / 2) * grid_size);
group_start_y = border_size;

current_group = instance_create(group_start_x, group_start_y, group[next_group]).id;
next_group = floor(random(number_of_groups));
```

The three things I should need to explain here are the starting positions, what we store in the array and how we find the next group.
The starting positions are calculated by finding the middle of the area for the blocks as the x and the border of the area for blocks as the y. This gives you the middle of the top of the blocks.

The array stores the type of objects that we are making. So when we create a new group of blocks we can take a number and find out what object we are meant to create by looking in that array.

We get the next group by choosing a random number between zero and the number of groups. When we call random(x) we get a value between 0 and x but this value can have a decimal part as well. All we want is a whole number, known as an integer. To get that, we call floor(x) and treat x as the random number we get. This rounds the number down to the next integer. The other two rounding functions are ceil(x) which rounds up to the next integer and round(x) which rounds to the closest integer. Now that we have that sorted, we can look in the step event for this object.

In the step event, we want to check if the current group of objects still exists. If it does not, we make a new one; if it does, we do nothing. To do this we call instance_exists(obj) where ‘obj’ will be the id of our group of objects. This will return true if our object still exists in the room. ‘obj’ can be the id of any instance or the name of the type of instance. So we could check if any instance of LineGroup exists, but I like to check for individual ids instead. This way if I have a different game where different objects hold instances of the same type, I can check for the exact one I want.
The code looks like this:

```javascript
if(!instance_exists(current_group)) {
    current_group = instance_create(group_start_x, group_start_y, group[next_group]).id;
    next_group = floor(random(number_of_groups));
}
```

Now whenever the current group doesn’t exist we make a new one. We’re getting close to a completed game as well, the major thing left is checking for completed rows.

**Checking for Completed Rows**

The way we do this is similar to how we check for collisions. We need to check every column in a row, if there is an object in all of them, that row can be completed. We then need to move all the rows above that one down and continue. We also only want to do this when we are about to create a new group, so we will put the code for it in the if statement above, but we will put it before we create the new object.
Here is the code for the new step event:

```javascript
if(!instance_exists(current_group)) {
    for(row = 0; row < number_of_rows; row += 1) {
        complete = true;
        yPos = border_size + (row * grid_size);
        for(column = 0; column < number_of_columns; column += 1) {
            xPos = border_size + (column * grid_size);
            if(!collision_point(xPos,yPos,Block,false,true)) {
                complete = false;
                break;
            }
        }
        if(complete) {
            with(Block) {
                if(y == other.yPos) {
                    instance_destroy();
                } else if(y < other.yPos) {
                    y += other.grid_size;
                }
            }
            score += score_value;
        }
    }
    current_group = instance_create(group_start_x, group_start_y, group[next_group]).id;
    next_group = floor(random(number_of_groups));
}
```

Ok, that’s a lot to take in, right? Let me break it down for you. The first for loop will move down each row from the top, letting us check if that row is complete. Inside that for loop we start by assuming the row is complete and getting the y position of the row. Next we have a for loop that checks each block going through every column in that group. The collision_point command will check if there is an object of type Block at the x and y positions. The last two values are for precision checking, which we don’t need because the whole sprite is our block; and whether we should check if we are at that place too. We only want to know if the block is at that place, so we set this to false.

If there is not a Block at that position, we set complete to false, because this row is not yet complete. We also use the ‘break’ command to exit this for loop, because there is no need to move on to the next blocks when we know it’s incomplete.

That ends that second for loop. Now we want to check if the row is complete. If it is, we want all the blocks in that y value to be destroyed and all the blocks above it to move down one level. We do this
in another ‘with’ statement. In that, we check what the block’s y value is in relation to the row that was just completed. If it is the same, destroy the block. If it is less, i.e. it is above it, move it down one block.

Finally, we increase the score after that with statement. ‘score’ is an inbuilt variable in Game Maker and we can add to or remove from it without having to set it up first.

**Scoring and Levels**

Well, we’ve already set where we increase the score, but we also want to increase the amount that score is increased each time, dependent on the level you are up to. We also want to increase the speed of the blocks falling each level and increase the level when you have removed a certain number of rows.

To do this, we will add in a variable in the create event called ‘completed_rows’ and set it to zero. We also want to set the speed of the blocks here, too. We will use this to set the speed of the falling blocks when we create them.

```plaintext
completed_rows = 0;
speed_of_groups = room_speed / 2;
```

We then want to increase that by one whenever we remove a row and then check if we have destroyed enough to increase the level. We do this at the same time we increase the score. Just after increasing it, but before we leave the for loop, put the following code:

```plaintext
completed_rows += 1;
if((number_of_rows mod 10) == 0) {
    speed_of_groups += room_speed / 10;
    score_value += 5;
    score += 100 * level;
    level += 1;
}
```

We’ve introduced another feature of Game Maker here, ‘mod.’ This will divide the number to the left of it by the number on its right and return whatever is left over. So if you do (10 mod 10) you will get 0 because there is nothing left when you divide ten by itself. But if you did (12 mod 10) you would get a result of 2 because when you divide 12 by 10 you have 2 left over. This way we can increase the level and everything every time we remove ten more rows. We increase the speed by a tenth of a second; using the same method we set the speed in the first place. We also increase the score you get each time you remove a row by 5 and give the player a bonus score of 100 times the level they have got to.

We also need to check if any blocks lie above our set maximum. That means, if there is any block above the border, the game needs to be over. For this, we will get the name of the player, show the high score and then restart the game. We will also do this after the rows have been checked for completion so the player has a chance to carry on playing. We will use very similar code to above, so I won’t explain it. Just have a look at it and try and follow what is happening.
Here it is:

```c
game_over = false;
for(i = 0; i < number_of_columns; i += 1) {
    xPos = border_size + (i * grid_size);
    if(collision_point(xPos, border_size - grid_size, Block, false, true)) {
        game_over = true;
    }
}
if(game_over) {
    highscore_show(score);
    game_restart();
}
```

The highscore_show(score) will check if the value of score is big enough to go on the highscore table. If it is, the player will be asked for their name and the table will be shown. If it is not, the table will just be shown.

Now all we need to do is increase the speed the blocks fall when we make them. We do this just after we create them. At this point they have finished doing all the things in their create event and we can modify their variables. So just after the point where we make a new group, put the following code:

```c
current_group.falling_rate = speed_of_groups;
```
After putting all of these things together, our step event code looks like this:

Now we’re almost done. We just need to make the rest of our blocks.

```java
if(!instance_exists(current_group)) {
    for(row = 0; row < number_of_rows; row += 1) {
        complete = true;
        yPos = border_size + (row * grid_size);
        for(column = 0; column < number_of_columns; column += 1) {
            xPos = border_size + (column * grid_size);
            if(!collision_point(xPos,yPos,Block,false,true)) {
                complete = false;
                break;
            }
        }
        if(complete) {
            with(Block) {
                if(y == other.yPos) {
                    instance_destroy();
                } else if(y < other.yPos) {
                    y += other.grid_size;
                }
            }
            score += score_value;
            number_of_rows += 1;
            if((number_of_rows mod 10) == 0) {
                level += 1;
                speed_of_groups += room_speed / 10;
                score_value += 5;
                score += 100;
            }
        }
    }
}

game_over = false;
for(i = 0; i < number_of_columns; i += 1) {
    xPos = border_size + (i * grid_size);
    if(collision_point(cPos,border_size - grid_size,Block,false,true))
    {
        game_over = true;
    }
}
```
if(game_over) {
    highscore_show(score);
    game_restart();
}

current_group = instance_create(group_start_x, group_start_y,
group[next_group]).id;
current_group.falling_rate = speed_of_groups;
next_group = floor(random(number_of_groups));
}

**Final Touches**

All that is left to do now is make the other block groups and maybe do a little extra drawing. The other block groups will have the exact same code as the line one, but they will have slightly different create events. The code for them is in Appendix A. Don’t forget to add each of those objects to the array of groups in the game controller and increase the number of groups there are.

It would also be nice to draw a line around the border of the playing area and show the score while the player is playing. To do that, we add some code into the Draw event for the game controller. Also take note that when you add anything into the draw event of an object, the sprite for that object will not be drawn automatically. To have it drawn you need to use the draw_sprite function, which is explained in the help file of Game Maker.

To draw the rectangle and the score, do the following:

draw_rectangle(border_size,border_size,border_size + (number_of_columns * grid_size), border_size + (number_of_rows * grid_size), true);
draw_text(600,border_size + 50, "Score: " + string(score));

That completes the code for the game. Create a new room with width of 800 and height of 600 and place the GameController object in it. Now you can run it and enjoy your fun new game.
Appendix A

Square Group

```
blocks[0] = instance_create(x - (26 * 1), y - (26 * 1), Block).id;
blocks[1] = instance_create(x, y - (26 * 1), Block).id;
blocks[2] = instance_create(x - (26 * 1), y, Block).id;
blocks[3] = instance_create(x, y, Block).id;

grid_size = 26;
falling_rate = room_speed;

time_to_next_drop = falling_rate;

shift_rate = room_speed / 4;
time_to_next_shift = 0;
border_size = 40;

number_of_columns = 10;

x_offset[0,0] = - (26 * 1);
x_offset[0,1] = - (26 * 1);
x_offset[0,2] = 0;
x_offset[0,3] = 0;

y_offset[0,0] = - (26 * 1);
y_offset[0,1] = - (26 * 1);
y_offset[0,2] = 0;
y_offset[0,3] = 0;

current_rotation = 0;
number_of_rotations = 0;
```
**Right Stair Group**

```
blocks[0] = instance_create(x, y - (26 * 1), Block).id;
blocks[1] = instance_create(x + (26 * 1), y - (26 * 1), Block).id;
blocks[2] = instance_create(x, y, Block).id;
blocks[3] = instance_create(x - (26 * 1), y, Block).id;

grid_size = 26;
falling_rate = room_speed;

time_to_next_drop = falling_rate;

shift_rate = room_speed / 4;
time_to_next_shift = 0;
border_size = 40;

number_of_columns = 10;

x_offset[0,0] = 0;
x_offset[0,1] = (26 * 1);
x_offset[0,2] = 0;
x_offset[0,3] = -(26 * 1);
y_offset[0,0] = -(26 * 1);
y_offset[0,1] = -(26 * 1);
y_offset[0,2] = 0;
y_offset[0,3] = 0;

x_offset[1,0] = 0;
x_offset[1,1] = 0;
x_offset[1,2] = (26 * 1);
x_offset[1,3] = (26 * 1);
y_offset[1,0] = -(26 * 1);
y_offset[1,1] = -(26 * 2);
y_offset[1,2] = 0;
y_offset[1,3] = -(26 * 1);

current_rotation = 0;
number_of_rotations = 2;
```
**Left Stair Group**

```javascript
blocks[0] = instance_create(x, y - (26 * 1), Block).id;
blocks[1] = instance_create(x - (26 * 1), y - (26 * 1), Block).id;
blocks[2] = instance_create(x, y, Block).id;
blocks[3] = instance_create(x + (26 * 1), y, Block).id;

grid_size = 26;
falling_rate = room_speed;

time_to_next_drop = falling_rate;

shift_rate = room_speed / 4;
time_to_next_shift = 0;
border_size = 40;

number_of_columns = 10;

x_offset[0,0] = 0;
x_offset[0,1] = -(26 * 1);
x_offset[0,2] = 0;
x_offset[0,3] = (26 * 1);

y_offset[0,0] = -(26 * 1);
y_offset[0,1] = -(26 * 1);
y_offset[0,2] = 0;
y_offset[0,3] = 0;

x_offset[1,0] = 0;
x_offset[1,1] = 0;
x_offset[1,2] = -(26 * 1);
x_offset[1,3] = -(26 * 1);

y_offset[1,0] = -(26 * 1);
y_offset[1,1] = -(26 * 2);
y_offset[1,2] = 0;
y_offset[1,3] = -(26 * 1);

current_rotation = 0;
number_of_rotations = 2;
```
L Group

blocks[0] = instance_create(x , y - (26 * 2), Block).id;
blocks[1] = instance_create(x, y - (26 * 1), Block).id;
blocks[2] = instance_create(x, y, Block).id;
blocks[3] = instance_create(x + (26 * 1), y, Block).id;

grid_size= 26;
falling_rate= room_speed;

time_to_next_drop = falling_rate;

shift_rate = room_speed / 4;
time_to_next_shift = 0;
border_size= 40;

number_of_columns= 10;

x_offset[0,0] = 0;
x_offset[0,1] = 0;
x_offset[0,2] = 0;
x_offset[0,3] = (26 * 1);

y_offset[0,0] = - (26 * 1);
y_offset[0,1] = - (26 * 1);
y_offset[0,2] = 0;
y_offset[0,3] = 0;

x_offset[1,0] = - (26 * 1);
x_offset[1,1] = 0;
x_offset[1,2] = (26 * 1);
x_offset[1,3] = (26 * 1);

y_offset[1,0] = 0;
y_offset[1,1] = 0;
y_offset[1,2] = 0;
y_offset[1,3] = - (26 * 1);

current_rotation  = 0;
number_of_rotations = 2;
**Reverse L Group**

```python
blocks[0] = instance_create(x, y - (26 * 2), Block).id;
blocks[1] = instance_create(x, y - (26 * 1), Block).id;
blocks[2] = instance_create(x, y, Block).id;
blocks[3] = instance_create(x - (26 * 1), y, Block).id;

grid_size = 26;
falling_rate = room_speed;

time_to_next_drop = falling_rate;

shift_rate = room_speed / 4;
time_to_next_shift = 0;
border_size = 40;

number_of_columns = 10;

x_offset[0,0] = 0;
x_offset[0,1] = 0;
x_offset[0,2] = 0;
x_offset[0,3] = -(26 * 1);

y_offset[0,0] = -(26 * 2);
y_offset[0,1] = -(26 * 1);
y_offset[0,2] = 0;
y_offset[0,3] = 0;

x_offset[1,0] = -(26 * 1);
x_offset[1,1] = 0;
x_offset[1,2] = (26 * 1);
x_offset[1,3] = -(26 * 1);

y_offset[1,0] = 0;
y_offset[1,1] = 0;
y_offset[1,2] = 0;
y_offset[1,3] = -(26 * 1);

current_rotation = 0;
number_of_rotations = 2;
```
**T Block**

```c
blocks[0] = instance_create(x, y - (26 * 1), Block).id;
bblocks[1] = instance_create(x - (26 * 1), y, Block).id;
bblocks[2] = instance_create(x, y, Block).id;
bblocks[3] = instance_create(x + (26 * 1), y, Block).id;

ggrid_size = 26;
f falling_rate = room_speed;

time_to_next_drop = falling_rate;

shift_rate = room_speed / 4;
time_to_next_shift = 0;
border_size = 40;

number_of_columns = 10;

x_offset[0, 0] = 0;
x_offset[0, 1] = -(26 * 1);
x_offset[0, 2] = 0;
x_offset[0, 3] = (26 * 1);
y_offset[0, 0] = -(26 * 1);
y_offset[0, 1] = 0;
y_offset[0, 2] = 0;
y_offset[0, 3] = 0;

x_offset[1, 0] = -(26 * 1);
x_offset[1, 1] = 0;
x_offset[1, 2] = 0;
x_offset[1, 3] = 0;
y_offset[1, 0] = -(26 * 1);
y_offset[1, 1] = 0;
y_offset[1, 2] = -(26 * 2);
y_offset[1, 3] = -(26 * 2);

x_offset[2, 0] = 0;
x_offset[2, 1] = -(26 * 1);
x_offset[2, 2] = 0;
x_offset[2, 3] = (26 * 1);
y_offset[2, 0] = 0;
y_offset[2, 1] = -(26 * 1);
y_offset[2, 2] = -(26 * 1);
```
y_offset[2,3] = -(26 * 1);

x_offset[3,0] = (26 * 1);
x_offset[3,1] = 0;
x_offset[3,2] = 0;
x_offset[3,3] = 0;
y_offset[3,0] = -(26 * 1);
y_offset[3,1] = 0;
y_offset[3,2] = -(26 * 1);
y_offset[3,3] = -(26 * 2);

current_rotation = 0;
number_of_rotations = 4;