CHAPTER 13

Clever Computers: Playing Tic-Tac-Toe

In this chapter we are going to create a version of the game tic-tac-toe in which you are pitted against an intelligent computer opponent. This opponent must have a strategy that will regularly beat the player to keep it challenging, but the computer opponent must not be too strong; otherwise the player has no chance of winning, and will quickly become frustrated and give up. We will also show how the computer can adapt its play to the level of the player. The game will be almost completely written using the GML programming language, so make sure you read and understood Chapter 12 on GML before starting this chapter.

Designing the Game: Tic-Tac-Toe

I am sure you'll know how to play *Tic-Tac-Toe*, but even so, it is good to describe it carefully before we start to aid you in making the game.

The game of Tic-Tac-Toe is played on a 3×3 grid. The computer player uses red stones while the human player uses blue stones. The players take turns placing a stone of their color on an empty cell. When a player manages to create a horizontal, vertical, or diagonal row of three stones of his color, he wins the game. When all cells are filled and no row is created, the game ends in a draw.

The player uses the mouse to place the stones. The Esc key is used to end the game. The game consists of an arbitrary number of rounds. In each round the player who lost the previous round will start. The number of wins for each player and the number of draws are recorded, and displayed on the game interface for reference. Figure 13-1 shows the game in action.

The game requires just a few ingredients: the playing field, the stones, and a mechanism to show the number of wins. The most complicated part will be how to determine the moves for the computer player. All the resources can be found in the `Resources/Chapter13` folder on the CD.

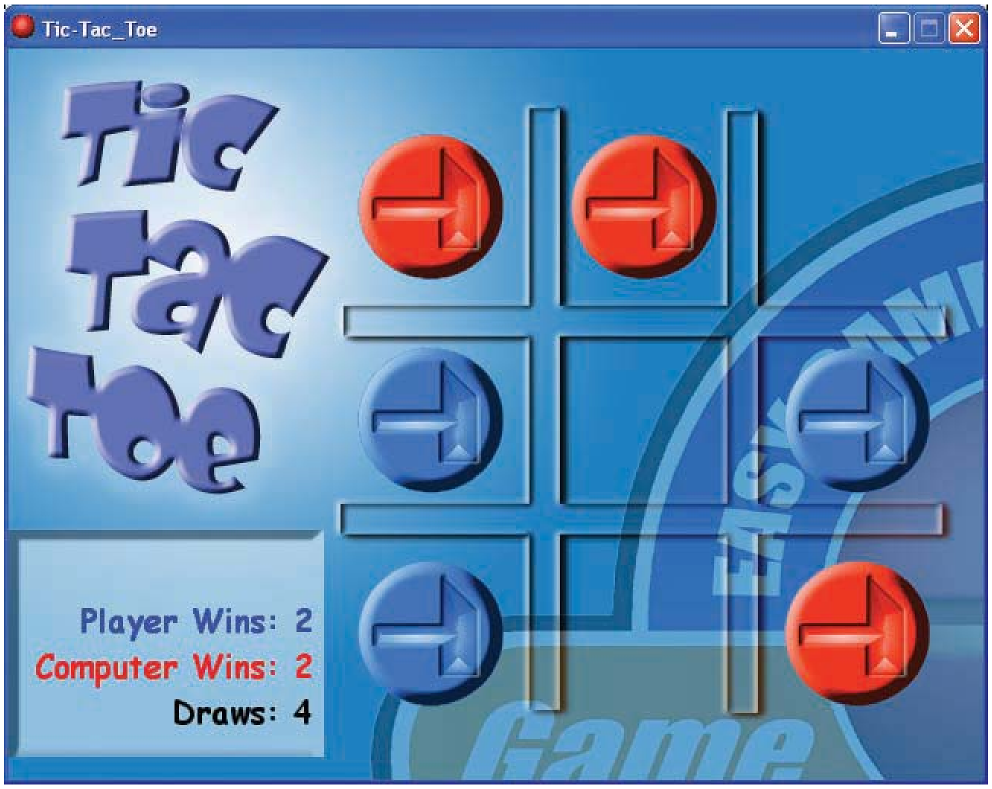

Figure 13-1. *The Tic-Tac-Toe game looks like this.*

The Playing Field

We will first need two sprites for the stones, a background to contain the playing field, and some sound effects. As this is a game in which the player is supposed to think a lot, background music is not really appropriate, so we won't use it.

Creating sprites, a background, and sound effects:

1. Start a new game.
2. Create a new sprite using the file `Stone1.gif` from the `Resources/Chapter13` folder on the CD.
3. Create another sprite using the file `Stone2.gif`.
4. Create a background using the file `Background.bmp`.
5. Finally create sound effects using the files `Place.wav`, `Win.wav`, `Lose.wav`, and `Draw.wav`.

We will also need a font for our game. We will use this to draw the score, that is, how many games are won by the player and the computer.

Creating a font:

1. Create a new font for the game. We named ours `fnt_score`. Select a nice font; for example, **Comic Sans MS**, give it a size of `16`, and select **Bold**.

The playing field is where all the action happens. We will create just one object in the game, which represents the playing field, and there will be just one instance of this object in the game. This object does not need a sprite, as the field is already drawn on the background.

Creating the field object and the room:

1. Create a new object. Give it the name `obj_field`. No sprite is required.
2. Create a new room. In the **backgrounds** tab, assign to it the background.
3. In the **settings** tab, give the room an appropriate caption.
4. In the **objects** tab, add one instance of the field object at an arbitrary place.

You might want to run the game just to test that the playing field is indeed there. Obviously, nothing can be done at this stage, as we are yet to specify the behavior for the field object. We will only use scripts for this.

Internally we represent the playing field with a variable `field` that will be a two-dimensional array. This variable represents the cells in the field, as shown in Figure 13-2. Each entry can have three values: `0` means that the cell is empty, `1` means that the human player placed a stone there, and `2` means that the computer player placed a stone there.

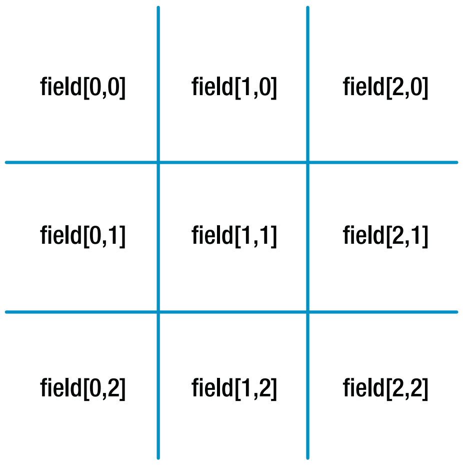

Figure 13-2. *The playing field is represented by a two-dimensional array called field.*

Let's start by creating a script to initialize the field. Call this script `scr_field_init`. This script must set all the field entries to 0. We will use two local variables for this, and then use a double loop to fill in the entries, as shown in Listing 13-1.

Listing 13-1. *The Script scr_field_init*

```
{
    var i,j;
    // clear the field
    for (i=0; i<=2; i+=1)
        for (j=0; j<=2; j+=1)
            field[i,j] = 0;
}
```

Note Notice the line that starts with `//`. This line is a comment, so it is not really part of the program. Comments are ignored by Game Maker, and exist purely to help you, or someone else, know what is going on in the code—would you remember what all your variables and loops do when coming back to a piece of code after six months?

The game must store the number of wins by the two players, as well as the number of draws. For this we will use three variables: `score_player`, `score_computer`, and `score_draw`. To initialize the game, we must initialize these variables to 0 and we must initialize the playing field, as shown in Listing 13-2.

Listing 13-2. *The Script scr_game_init*

```
 {
    // initialize the score
    score_player = 0;
    score_computer = 0;
    score_draw = 0;
    // initialize the field
    scr_field_init();
}
```

As you can see, we call the first script (`scr_field_init`) from within this script. Scripts can be used as functions that can be called from other scripts—we will use this technique a lot in our game. The `scr_game_init` script is executed from the **Create** event of the field object.

Creating and executing the scripts:

1. Create the two scripts, `scr_field_init` and `scr_game_init`, as described earlier.
2. Reopen the properties form of the field object by double-clicking on it in the resource list.
3. Add a **Create** event. In it include the **Execute Script** action and indicate the script `scr_game_init`.

The next step is to make it possible for the player to place stones. When the player clicks the left mouse button on the screen, we must detect which cell the click occurs in. If the click is outside the playing field or on a cell that is already filled, there will be no resulting action.

To determine the cell that has been clicked, we consider the current position of the mouse, which is indicated by the global variables `mouse_x` and `mouse_y`. The cells each have a size of 140×140, so to get the correct cell index (0, 1, or 2), we divide the mouse position by 140 and then round it down to the nearest whole number using the `floor()` function. This would give the correct cell index if our playing field were in the top-left corner of the screen. Because the top-left corner of the field is at position (208,32), we must subtract this offset from the mouse position, as we want the position relative to the top-left corner of the playing field, not the top-left corner of the screen.

So, for example, say the human player clicks at `x` = 350. The sum we do is (350-208)/140 = 1.01. Rounded down, the result is 1, which tells us that the cursor has been clicked inside one of the middle columns of cells (remember that the array starts at 0, not 1).

If the results of the calculations for `x` and `y` are less than 0 or more than 2, we ignore the click. If they *are* in this range, we check whether the corresponding cell is empty. If the cell is empty, we change its value to 1 to place the stone and play the sound effect.

The script looks like the one shown in Listing 13-3.

Listing 13-3. *The Script scr_field_click*

```
{
    var i,j;
    // find the position that is clicked
    i = floor((mouse_x-208)/140);
    j = floor((mouse_y-32)/140);
    // check whether it exists and is empty
    if (i<0 || i>2 || j<0 || j>2) exit;
    if (field[i,j] != 0) exit;
    // set the stone
    field[i,j] = 1;
    sound_play(snd_place);
}
```

Note that we use a new statement here: `exit`. The `exit` statement ends the execution of the script. We need to call this script in the **Global left pressed** event. This event is called when the left mouse button is pressed anywhere on the screen (not necessarily in the field object).

Creating the mouse click script:

1. Create the script `scr_field_click`, as shown earlier.

2. Reopen the properties form of the field object by double-clicking on it in the resource list.

3. Add a **Mouse**, **Global mouse**, **Global left pressed** event. In it include the **Execute Script** action and indicate the script `scr_field_click`.

If you run the game now, you will notice that you do hear the sound effect when you click on a cell but that no stones appear. This makes sense, as we have not yet added any code to draw the stones. Rather than using stone objects, we will create a script that draws the stones. Actually, this script will draw everything that is required: the stones and the current score (remember that the field does not need to be drawn as it is on the background image).

The script (shown in Listing 13-4) consists of two parts. First, all cells that are nonempty are drawn. We use a double loop for this. Depending on the value of the field cell at that position, the red or blue stone sprite is drawn. Second, the score is drawn. For this we set the correct font and position, and for each line we set a different color. (Note that the function `string()` turns a number into a string.)

Listing 13-4. *The Script scr_field_draw*

```
{
    var i,j;
    // draw the correct sprites
    for (i=0; i<=2; i+=1)
        for (j=0; j<=2; j+=1)
        {
            if (field[i,j] == 1)
                draw_sprite(spr_stone1,0,208+140*i,32+140*j);
            if (field[i,j] == 2)
                draw_sprite(spr_stone2,0,208+140*i,32+140*j);
        }
    // draw the score
    draw_set_font(fnt_score);
    draw_set_halign(fa_right);
    draw_set_color(c_blue);
    draw_text(200,340,'Player Wins: ' + string(score_player));
    draw_set_color(c_red);
    draw_text(200,375,'Computer Wins: ' + string(score_computer));
    draw_set_color(c_black);
    draw_text(200,410,'Draws: ' + string(score_draw));
}
```

We must call this script in the **Draw** event of the field object.

Drawing the field:

1. Create the script `scr_field_draw` as described earlier.
2. Add the **Draw** event to the field object. In it include the **Execute Script** action and indicate the script `scr_field_draw`.

Now when you test the game, you should be able to place stones. The computer opponent is not yet doing anything, so only your own stones exist. In the next section we will create some simple opponent behavior. The current version of the game can be found in the file `Games/Chapter13/tic_tac_toe1.gm6` on the CD.

Let the Computer Play

In this section, we are mainly going to concentrate on completing the first version of the game by adding the logic for a simple computer opponent. But first we need some scripts to test whether the player or the computer won the last game, or if it was a draw. We start with a script to check whether the player did win. There are eight different lines of three stones that can be filled to win: three horizontal ones, three vertical ones, and two diagonal ones. In the script (shown in Listing 13-5), we simply test all of these to see whether the cells contain the correct value. The function will return either the value `true` indicating that the player did win, or `false`, indicating that the player did not yet win. The value returned by the script can then be used later as a condition in other scripts. By this point you should know how to create a script, so we will just show the code from here on out. If you are confused at any point, remember that the game is available on the CD in `Games/Chapter13/tic_tac_toe2.gm6`, so feel free to open it up and have a look.

Listing 13-5. *The Script scr_check_player_win*

```
{
    if (field[0,0]==1 && field[0,1]==1 && field[0,2]==1) return true;
    if (field[1,0]==1 && field[1,1]==1 && field[1,2]==1) return true;
    if (field[2,0]==1 && field[2,1]==1 && field[2,2]==1) return true;
    if (field[0,0]==1 && field[1,0]==1 && field[2,0]==1) return true;
    if (field[0,1]==1 && field[1,1]==1 && field[2,1]==1) return true;
    if (field[0,2]==1 && field[1,2]==1 && field[2,2]==1) return true;
    if (field[0,0]==1 && field[1,1]==1 && field[2,2]==1) return true;
    if (field[0,2]==1 && field[1,1]==1 && field[2,0]==1) return true;
    return false;
}
```

■ **Note** To check whether two values are equal, you must use `==`, not `=`, as a single `=` is the assignment operator. Also remember that once a `return` statement is reached, the rest of the script is not executed.

Checking whether the computer wins is exactly the same, except that this time, we are testing whether the cells contain values of 2, not 1, as shown in Listing 13-6.

Listing 13-6. *The Script scr_check_computer_win*

```
{
    if (field[0,0]==2 && field[0,1]==2 && field[0,2]==2) return true;
    if (field[1,0]==2 && field[1,1]==2 && field[1,2]==2) return true;
    if (field[2,0]==2 && field[2,1]==2 && field[2,2]==2) return true;
    if (field[0,0]==2 && field[1,0]==2 && field[2,0]==2) return true;
    if (field[0,1]==2 && field[1,1]==2 && field[2,1]==2) return true;
    if (field[0,2]==2 && field[1,2]==2 && field[2,2]==2) return true;
    if (field[0,0]==2 && field[1,1]==2 && field[2,2]==2) return true;
    if (field[0,2]==2 && field[1,1]==2 && field[2,0]==2) return true;
    return false;
}
```

Checking for a draw is even simpler (see Listing 13-7)—we check all cells; if one is empty we return `false` as there is still a move possible. Only when all cells are filled do we return `true`.

Listing 13-7. *The Script scr_check_draw*

```
{
    var i,j;
    for (i=0; i<=2; i+=1)
        for (j=0; j<=2; j+=1)
        {
            if (field[i,j] == 0) return false;
        }
    return true;
}
```

To act on the outcome of these three possibilities, we will use another script, as shown in Listing 13-8. For each possible outcome, the correct score variable is increased; a sound is played; we redraw the screen to actually show the last move and the new score; wait for a second; show a message; and initialize the field again.

Listing 13-8. *The Script scr_check_end*

```
{
    // check whether the player did win
    if (scr_check_player_win())
    {
        score_player += 1;
        sound_play(snd_win);
        screen_redraw();
        sleep(1000);
        show_message('YOU WIN');
        scr_field_init();
    }
    // check whether the computer did win
    if (scr_check_computer_win())
    {
        score_computer += 1;
        sound_play(snd_lose);
        screen_redraw();
        sleep(1000);
        show_message('YOU LOSE');
        scr_field_init();
    }
    // check whether there is a draw
    if (scr_check_draw())
    {
        score_draw += 1;
        sound_play(snd_draw);
        screen_redraw();
        sleep(1000);
        show_message("IT'S A DRAW");
        scr_field_init();
    }
}
```

We must call this script after each move by either the player or the computer.

But we still need to give the computer the power to make a move. Let's create a very simple mechanism here; in the next section we'll create a much more intelligent opponent. Our simple mechanism makes a random move. We do this as follows. We select a random cell, test whether it is empty, and if so, place the stone there. If the selected cell is not empty, we repeat the search until we find one that is. Finding a random position works like this—we use the function `random(3)` to obtain a random real number below 3. Using the `floor()` function, we round this down, obtaining 0, 1, or 2. The script is shown in Listing 13-9.

Listing 13-9. *The Script scr_find_move*

```
{
    var i,j;
    while (true)
    {
        i = floor(random(3));
        j = floor(random(3));
        if (field[i,j] == 0)
        {
            field[i,j] = 2;
            exit;
        }
    }
}
```

This script makes use of a `while` loop to find a random free cell. `while(true)` can be dangerous, because as the expression is always true, the loop never exits by itself. In this case, however, we are exiting in our own code as soon as an empty cell is detected, so it is safe as long as an empty cell exists. If there are no empty cells, we already know it is a draw, so we need not worry about that case.

We are going to use this script in an updated version of the script `scr_field_click`, which we created earlier. When the player has made a valid move, there are three things we must do. First, we check whether the player won or whether there is a draw, in which case the field is initialized again, ready for the next game. Next, we let the computer make a move. Finally, we check whether the computer won or whether there is a draw.

We adapt the scr_field_click script by adding a few lines, as shown in Listing 13-10.

Listing 13-10. *The Adapted Script scr_field_click*

```
{
    var i,j;
    // find the position that is clicked
    i = floor((mouse_x-208)/140);
    j = floor((mouse_y-32)/140);
    // check whether it exists and is empty
    if (i<0 || i>2 || j<0 || j>2) exit;
    if (field[i,j] != 0) exit;
    // set the stone
    field[i,j] = 1;
    sound_play(snd_place);
    scr_check_end();
    // let the computer make a move
    scr_find_move();
    scr_check_end();
}
```

Once you have added the new scripts and made the change to scr_field_click, test the game, and you should find it is now fully operational. You can find it in the file Games/Chapter13/tic_tac_toe2.gm6 on the CD.

The game as it stands is fine, but it is extremely easy to win as the computer plays random moves. In the next section, we will make the computer a bit more intelligent.

A Clever Computer Opponent

To be able to make a clever computer opponent we must first be clever ourselves. How would you play the game? What would your strategy be? If you have played the game often, here is a strategy you might come up with:

- If there is a move available that will make you win, then play it.
- If there is no winning move available for you, but there is one available for the opponent, you'd better play that move to block them; otherwise you will lose.
- If neither is the case but the center cell is free, then play the center.
- If none of the above is true, play a random move.

This is not the best strategy possible but it is pretty good, and still leaves the player with a chance to win, so let's implement it. To give the game a bit more variation, we will program the computer opponent to only do the third step half the times it is presented. We are going to create four scripts, one for each of the four cases. The last one we already have—all we have to do is rename it from scr_find_move to scr_find_random. The other scripts still have to be constructed.

We will start with the script that tests for the existence of a winning move for the computer. If such a move exists, it is made, and `true` is returned. Otherwise `false` is returned. The script works as follows—we consider every empty cell. We place a stone there and test whether we won. If so, we return `true`. If not, we make the cell empty again and proceed with the next empty cell. The script is shown in Listing 13-11.

Listing 13-11. *The Script scr_find_win, Which Tries to Find a Winning Move*

```
{
    var i,j;
    for (i=0; i<=2; i+=1)
        for (j=0; j<=2; j+=1)
            if (field[i,j] == 0)
            {
                field[i,j] = 2;
                if scr_check_computer_win() return true;
                field[i,j] = 0;
            }
    return false;
}
```

The next script tries to find a potential winning move for the human player. If such a position exists, the computer places a stone there. It largely works the same, except that we are testing for a potential row of three human player stones, not three computer player stones. The cell is then given a value of `2` to place a computer stone there, to block the human player's winning move, as shown in Listing 13-12.

Listing 13-12. *The Script scr_find_lose, Which Tries to Block a Winning Move of the Player*

```
{
    var i,j;
    for (i=0; i<=2; i+=1)
        for (j=0; j<=2; j+=1)
            if (field[i,j] == 0)
            {
                field[i,j] = 1;
                if scr_check_player_win()
                    { field[i,j] = 2; return true; }
                field[i,j] = 0;
            }
    return false;
}
```

Finally, we need the script that tries the center position (Listing 13-13.) It will only try it once out of every two times.

Listing 13-13. *The Script scr_find_center, Which Tries to Place a Stone in the Center*

```
{
    if (random(2) < 1 && field[1,1] == 0)
        { field[1,1] = 2; return true; }
    return false;
}
```

With all these scripts in place, we have to remake the script `scr_find_move` that determines the next move of the computer. This script calls the four scripts in order and, whenever one succeeds, it stops further processing because the opponent has made a move. This script appears in Listing 13-14.

Listing 13-14. *The New Script scr_find_move*

```
{
    if scr_find_win() exit;
    if scr_find_lose() exit;
    if scr_find_center() exit;
    scr_find_random();
}
```

That's the whole game. You can find this finished version in the file `Games/Chapter13/tic_tac_toe3.gm6` on the CD. It will be quite a bit harder to beat this opponent, and if you are not very good at the game you will most likely lose a few times! In particular, the game might be too difficult for young children. In the next section, we will see how we can automatically adapt the game to the level of the player.

Adaptive Gameplay

When the player is good, we should confront him with a strong computer opponent. But when the player is a novice, the computer opponent should be weaker. Many games achieve this by letting the user manually select the level of the game (for example, easy, normal, or hard). But it can be more desirable when the game automatically adapts to the level of the player. For our game this can easily be achieved—we maintain the score of the player and the computer, so we know who is winning. When the player is winning a lot, we can make the computer play better, and when the player is losing a lot, we can make the computer play down its abilities.

As an indication of how good the player is, we use `(score_player+1) / (score_computer+1)`. The reason for adding 1 to both values is that we do not want to divide by 0, as this would cause an error. When this value is larger than 1, the player is better than the computer. When it is smaller than 1, the computer is better. In the `scr_find_move` script where we decide on the computer move, we will also compute this value, and based on the result, we decide which moves we check. When the value is larger than 1.2, we try all moves. When it is smaller than 0.5, we only do a random move. We add in the other two moves as the value increases. The updated version of `scr_find_move` is shown in Listing 13-15.

Listing 13-15. *Further Adapting the Script scr_find_move*

```
{
    var level;
    level = (score_player+1) / (score_computer+1);
    if (level > 0.5)
        { if scr_find_win() exit; }
    if (level > 0.8)
        { if scr_find_lose() exit; }
    if (level > 1.2)
        { if scr_find_center() exit; }
    scr_find_random();
}
```

You should now let a number of people play the game and see whether the game indeed adapts to their level of play. You might want to vary the numbers in the script to make the game easier or harder for the players.

Congratulations

You have just created your first intelligent computer opponent. Also, you have created some adaptive gameplay, something that is very important for good games. You will find the last version of the game in the file `Games/Chapter13/tic_tac_toe4.gm6` on the CD.

You might want to extend the game even further. First, you can think a bit more about the strategy. A very good move in the game is a move where you create two winning positions for yourself in the next move. As the opponent can only play one of the two, you are then assured of winning the game in the next move. You might want to add this to the strategy. Another thing you could do is add a two-player mode in which two people can play against each other. This is relatively easy, as there is no need for an intelligent computer opponent anymore—here you only need to remember and indicate who is to play in each move.

The registered version of Game Maker makes available functions that make it possible to play such a game over a network with two computers, but that is something really advanced.

In this chapter you saw how useful GML code is—we put everything in scripts. In fact, a game like this would have been almost impossible to create without GML. And it wasn't really all that much work. Once you get accustomed to using GML code, you will probably start using actions much less.

Intelligent opponents make games more interesting. In this chapter we had just one intelligent opponent. In the next chapter, we will create a whole collection of intelligent enemies.