There's always a room for improvements in any game we write. For example, we can offer more than one life to the player. They could make a mistake or two and still be able to finish the game. We could also improve our code rendering.

**Giving the player some lives**

Implementing lives is quite straightforward. Let's first add a variable to store the number of lives, in the same place where we declared our other variables:

var lives = 3;

Drawing the life counter looks almost the same as drawing the score counter — add the following function to your code, below the drawScore() function:

function drawLives() {

ctx.font = "16px Arial";

ctx.fillStyle = "#0095DD";

ctx.fillText("Lives: "+lives, canvas.width-65, 20);

}

Instead of ending the game immediately, We will decrease the number of lives until they are no longer available. We can also reset the ball and the paddle positions when the player begins with their next life. So, in the draw() function replace the following two lines:

alert("GAME OVER");

document.location.reload();

With this,we add slightly more complex logic to it as given below:

lives--;

if(!lives) {

alert("GAME OVER");

document.location.reload();

}

else {

x = canvas.width/2;

y = canvas.height-30;

dx = 2;

dy = -2;

paddleX = (canvas.width-paddleWidth)/2;

}

Now, when the ball hits the bottom edge of the screen we're subtracting one life from the livesvariable. If there are no lives left, the game is lost; if there are still some lives left, then the position of the ball and the paddle are reset, along with the movement of the ball.

Rendering the lives display

Now you need to add a call to drawLives() inside the draw() function  add it below thedrawScore() call.

drawLives();

**Improving rendering with requestAnimationFrame()**

Now let's work on something that is  not connected to the game mechanics, but to the way it is being rendered. [requestAnimationFrame](https://developer.mozilla.org/en-US/docs/Web/API/Window/requestAnimationFrame) helps the browser render the game better than the fixed framerate we currently have implemented using [setInterval()](https://developer.mozilla.org/en-US/docs/Web/API/WindowTimers/setInterval). Replace the following line:

setInterval(draw, 10);

with simply:

draw();

Then, at the very bottom of the draw() function (just before the closing curly brace), add in the following line, which causes the draw() function to call itself over and over again:

requestAnimationFrame(draw);

The draw() function is now getting executed again and again within arequestAnimationFrame() loop, but instead of the fixed 10 milliseconds frame rate, we are giving control of the framerate back to the browser. It will sync the framerate accordingly and render the shapes only when needed. This produces a more efficient, smoother animation loop than the older setInterval() method.

**Compare your code**

That's all — the final version of the game is ready and set to go !

**Exercise**: change the number of lives and the angle the ball bounces off the paddle.

**Game over - for now!**

You've finished all the lessons - congratulations! By this point you would have learnt the  basics of canvas manipulation and the logic behind simple 2D games. Now it's a good time to learn some frameworks and continue game development. You can check out this series' counterpart,[2D breakout game using Phaser](https://developer.mozilla.org/en-US/docs/Games/Workflows/2D_breakout_game_Phaser) or the [Cyber Orb built in Phaser](https://developer.mozilla.org/en-US/docs/Games/Workflows/HTML5_Gamedev_Phaser_Device_Orientation) tutorial, you can also look through the [Games section on MDN](https://developer.mozilla.org/en/docs/Games) for inspiration and more knowledge.

You could also go back to [this tutorial series' index page](https://developer.mozilla.org/en-US/docs/Games/Workflows/Breakout_game_from_scratch).