**4. Ping-pong game for two**

Objective

Create a Ping Pong game for 2 players

Contents

[Introduction 2](#_Toc68290048)

[Index.html file 3](#_Toc68290049)

[Style.css file 4](#_Toc68290050)

[Script.js file 4](#_Toc68290051)

[Creating an SVG Tag and Learning to Read the Official Documentation 4](#_Toc68290052)

[Remembering the classes and finishing the look 5](#_Toc68290053)

[The ball and its movement 6](#_Toc68290054)

[Adding rackets 8](#_Toc68290055)

[Moving the paddles over the buttons 9](#_Toc68290056)

[Restricting the movement of rackets around the edges 12](#_Toc68290057)

[Introducing AddEventListener 13](#_Toc68290058)

[Bumping rackets 14](#_Toc68290059)

[Remembering the account 17](#_Toc68290060)

[The reset button 19](#_Toc68290061)

[Winning and revising browser events 20](#_Toc68290062)

[Ball movement and random 21](#_Toc68290063)

[**Hometasks** 22](#_Toc68290064)

[Homework 1 22](#_Toc68290065)

[Homework 2 22](#_Toc68290066)

[Homework 3 23](#_Toc68290067)

[Homework 4 24](#_Toc68290068)

What we repeat

1. SVG coordinates
2. Basic SVG tags
3. Variables and Functions
4. Timer operation
5. Working with classes
6. Pushing off edges and moving objects
7. Browser events

What's new

1. Using the SVG.js library
2. Constants
3. **addEventListener**

Links to materials and personal account

[Working materials](https://hwschool.bitrix24.ru/bitrix/tools/disk/focus.php?folderId=375969&action=openFolderList&ncc=1" \t "_blank)(for the teacher).

[Materials (edit)](https://hwschool.bitrix24.ru/~HBc5w" \t "_blank)(we send this link to the student at the beginning of the lesson).

[Video presentation of the finished project](https://youtu.be/09IFnUMHv6w" \t "_blank)

Methodical material

Introduction

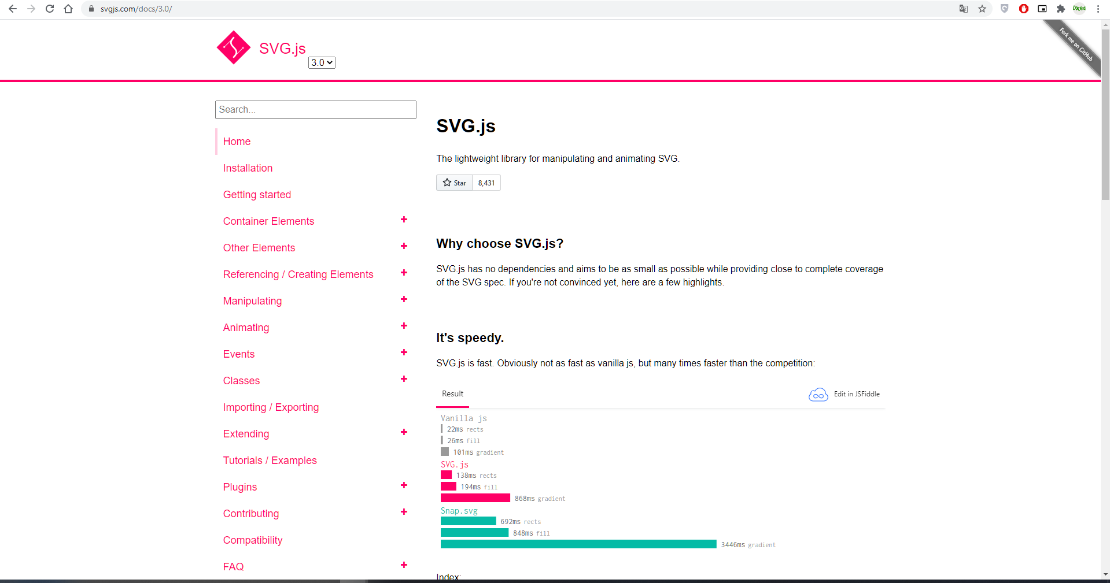
Attention. The student should be familiar with the "Chameleon Clock" project from the JavaScript section, because it covers the basics of working with classes.

Today we will create a browser game "ping pong", it will be possible to play it together with a friend on the same keyboard.

For this we will use the SVG.js library

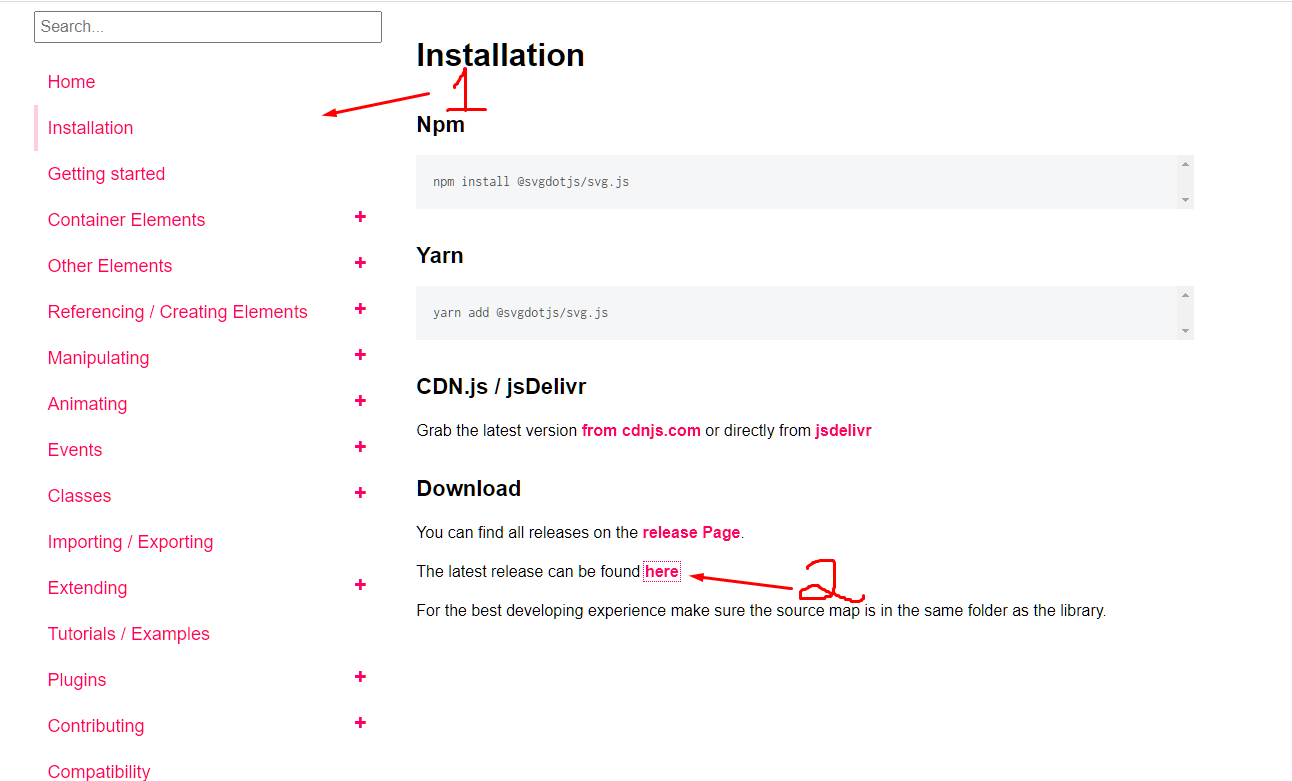
[https://svgjs.com/docs/3.0/](https://svgjs.com/docs/3.0/" \t "_blank)

First of all, we see a page with information about our library. On the left, we have complete instructions for using shapes, animations, short examples of working code, etc.



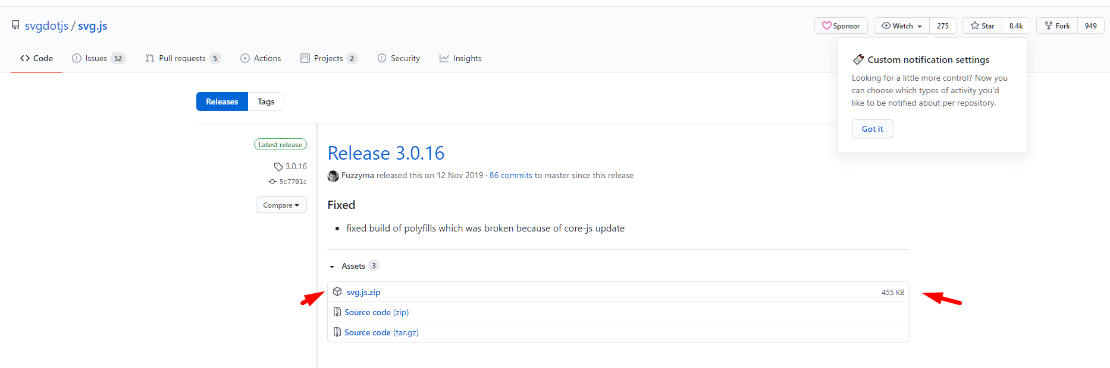
Here you can tell the student that it is faster than analogues and allows you to create animations much easier than if we wrote everything ourselves

We go to the Installation item and click on the link to download the latest version



Here we find GitHub, we are already familiar with it, because we have loaded our site there.

And download the archive with the library.



The archive contains a lot of files, we are interested in the files **svg.js** and **svg.min.js**. Let's extract them from the archive into our project folder and open them in a code editor.

Let's see how they differ.

In fact, they are almost the same, only the svg.min.js file is written in one line - because of this, it weighs almost 3 times less, and works faster. Many developers minimize their finished projects with special programs.

If the student is curious, then let's go over all the files in the archive.

1. Сhangelog.md - a list of all changes in the project in the journal
2. License.txt - license
3. Polyfills are files that provide compatibility with older browsers.
4. Readme.md - the file "read me", where the developer wants to tell something
5. Files with the .map extension - created by developers when minified to look for bugs in the future.

Finally, let's create 3 files that are familiar to us index.html, style.css, script.js

Index.html file

Let's write a standard template from the main tags, while not forgetting to include styles, scripts, it is very important to connect the library before our **JS** file.

**Inside the body of the site, we will create just one div tag and 2 buttons to stop and start our future game.**

**<! DOCTYPE html>**

**<html>**

**<head>**

**<title> SVG Lesson 2 </title>**

**<link rel = "stylesheet" type = "text / css" href = "style.css"> </head>**

**<body>**

**<div id = "game">**

**</div>**

**<script type = "text / javascript" src = "svg.min.js"> </script>**

**<script type = "text / javascript" src = "script.js"> </script>**

**</body>**

**</html>**

Style.css file

As for styles - I suggest styling the buttons a bit and centering all the content of the body

**body {**

**text-align: center;**

**}**

**button {**

**width: 350px;**

**margin: 0 auto;**

**display: block;**

**font-size: 28px;**

**border-radius: 6px;**

**outline: none;**

**}**

Script.js file

Creating an SVG Tag and Learning to Read the Official Documentation

At this point, I am considering 2 approaches:

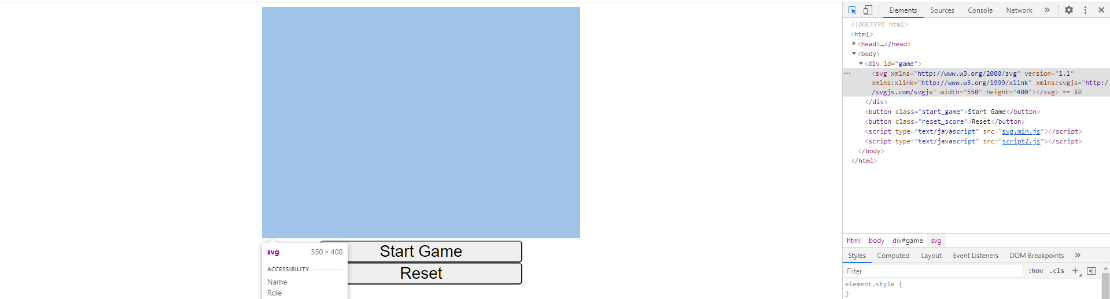
1. Our usual work is when we dictate to the student what to do and reinforce the material with repetitions.
2. We force the student to search for something on the official website svg.js.

Even if we will not use the second approach, then in this section we will show that everything is on the official website.

Surprisingly, we didn't write the **SVG** tag. Correct, because we will create it via **JavaScript**. Let's create a variable **draw** and assign **SVG** to it. After another point, we indicate where the tag will be placed using the **addTo** command and the sizes through **size**

**let draw = SVG (). addTo ('# game'). size (550, 400);**

Let's immediately see what we have created through the code of the elements in the browser.



Everything is fine, we have an **SVG** tag, now let's go back to the official website of the library and look at the **Getting started** section. Everything is fine, we see very similar actions and a similar result



Remembering the classes and finishing the look

On the example from the site, we noticed that a square is still being created there. At the same time, we understand that **draw** is now an object of the **SVG** class and through a dot we can call different functions.

In the example, the **rect** function is called, we remember that this is a rectangle. Let's just create it, and make it the background for our game.

For filling, we will use the **fill** method, previously we used it as an attribute or **CSS property**

**let draw = SVG (). addTo ('# game'). size (550, 400);**

**let background = draw.rect (550,400) .fill ("# EEE154");**

We check it, everything works fine. But I propose to take out our sizes, so as not to write numbers every time. To do this, we will use constants, they differ from variables in that they cannot be overwritten, but at the same time they speed up our program and make it more logical.

If we are sure that we will not change the value in the variable in the future, then we use a constant.

**const WIDTH = 550;**

**const HEIGHT = 400;**

**let draw = SVG (). addTo ('# game'). size (550, 400);**

**let background = draw.rect (550,400) .fill ("# EEE154");**

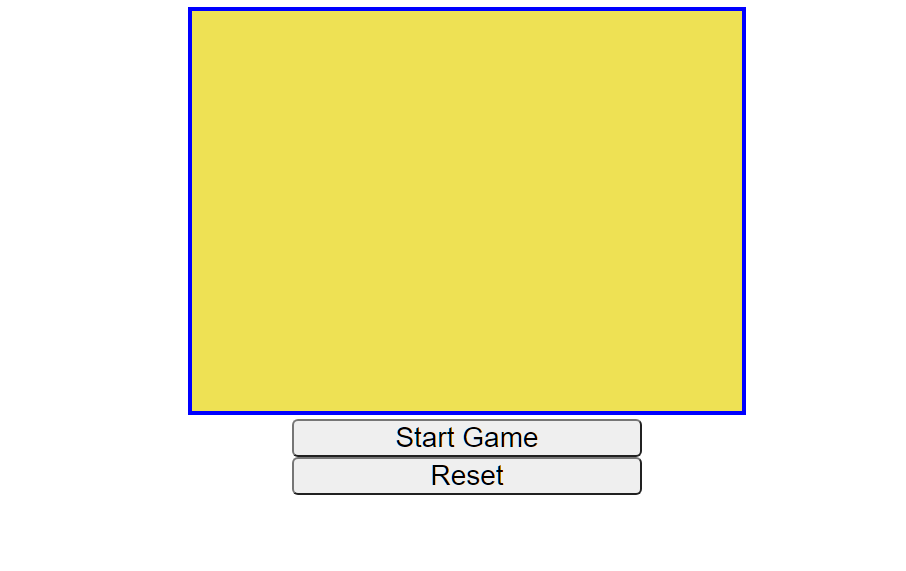
Everything is fine, now I suggest going back a bit to **CSS** and add a border for our **SVG** tag to visually see its borders

**svg {**

**border: 4px solid blue;**

**}**

As a result, we have a similar result



The ball and its movement

Let's create a circle, if you remember well how properties work in SVG, then this will not be difficult. Give the circle a radius, center coordinates, fill and outline.

Problems can only arise with the stroke property

**let**

**ball = draw.circle (). radius (20) .cx (WIDTH / 2) .cy (HEIGHT / 2) .fill ("# F97171"). stroke ({width: 1, color: "black"}) ;**

Now let's create a function for the continuous movement of the ball, we also need variables that will be responsible for its step and call the function on a timer. In general, the same as what we did in the last lesson with the DVD splash screen.

Perhaps the student will write everything himself, but if difficulties arise, then we go in order

Create a function and pull out the center coordinates and the radius, with the radius the easiest way is to subtract the center coordinates from the left corner coordinates.

**function move\_ball () {**

**let x = ball.cx ();**

**let y = ball.cy ();**

**let radius = ball.cx () - ball.x ();**

**}**

Now let's create global variables that will be responsible for the step of our ball and then we will add them to the coordinates of the center

**let step\_x = 2**

**let step\_y = 2**

**function move\_ball () {**

**let x = ball.cx ();**

**let y = ball.cy ();**

**let radius = ball.cx () - ball.x ();**

**x + = step\_x;**

**y + = step\_y;**

**}**

And at the very end, we will return the coordinates back to our ball

**let step\_x = 2**

**let step\_y = 2**

**function move\_ball () {**

**let x = ball.cx ();**

**let y = ball.cy ();**

**let radius = ball.cx () - ball.x ();**

**x + = step\_x;**

**y + = step\_y;**

**ball.cx (x);**

**ball.cy (y);**

**}**

Let's write comments in the code and don't forget to call the function

**let step\_x = 2**

**let step\_y = 2**

**const WIDTH = 550;**

**const HEIGHT = 400;**

**let draw = SVG (). addTo ('# game'). size (WIDTH, HEIGHT);**

**let background = draw.rect (WIDTH, HEIGHT) .fill ("# EEE154");**

**let ball = draw.circle (). radius (20) .cx (WIDTH / 2) .cy (HEIGHT / 2) .fill ("# F97171"). stroke ({width: 1, color: "black"}) ;**

**function move\_ball () {**

**// pull out the center coordinates and calculate the radius as the difference between the center point and the left edge point**

**let x = ball.cx ();**

**let y = ball.cy ();**

**let radius = ball.cx () - ball.x ();**

**// increase coordinates by a given step**

**x + = step\_x;**

**y + = step\_y;**

**// return values ​​back to html**

**ball.cx (x);**

**ball.cy (y);**

**}**

**// call the function by timer**

**setInterval (move\_ball, 10)**

I think the advantages of the library are already obvious, no need to fiddle with **set** and **getAttribute**. Let's add a reflection of the ball from the edges, again use the experience from the last lesson and repeat what we already know how.

The only difference is that we are working with center coordinates and radius, so the circle is even more convenient in this regard than the **image** tag.

**function move\_ball () {**

**// check the y coordinates if (y + radius> = HEIGHT || y-radius <0) {**

**step\_y = -step\_y**

**}**

**// check the x coordinates if (x + radius> = WIDTH || x-radius <0) {**

**step\_x = -step\_x**

**}**

**// return values ​​back to html**

**ball.cx (x);**

**ball.cy (y);**

**}**

**// call the function by timer**

**setInterval (move\_ball, 10)**

**Please note that we perform all checks strictly before we return our numbers back to the circle.**

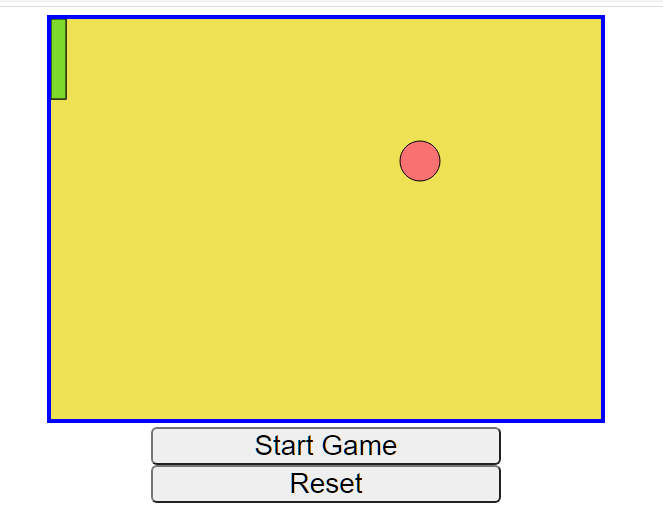
[Homework 1](https://hwschool.bitrix24.ru/knowledge/front-end/svgcanvas/43/?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y" \l "block131353" \t "_self)

Adding rackets

First of all, let's draw two rectangles around the edges using the rect method, as we already did with the background, only of course smaller.

**let racket1 = draw.rect (15,80) .fill ("# F6AC2B"). stroke ({width: 1, color: "black"})**

**let racket2 = draw.rect (15,80) .fill ("# 7ED92D"). stroke ({width: 1, color: "black"})**



The rackets appeared in the upper left corner. Place them along the **x** and **y** edges in the center.

**let racket1 = draw.rect (15,80) .fill ("# F6AC2B"). stroke ({width: 1, color: "black"}).move (0, HEIGHT / 2);**

**let racket2 = draw.rect (15,80) .fill ("# 7ED92D"). stroke ({width: 1, color: "black"}).move (WIDTH, HEIGHT / 2);**

The result is not very good, so we remember the SVG coordinates and where, what needs to be subtracted and added.

Do not forget that the origin is the upper left corner.

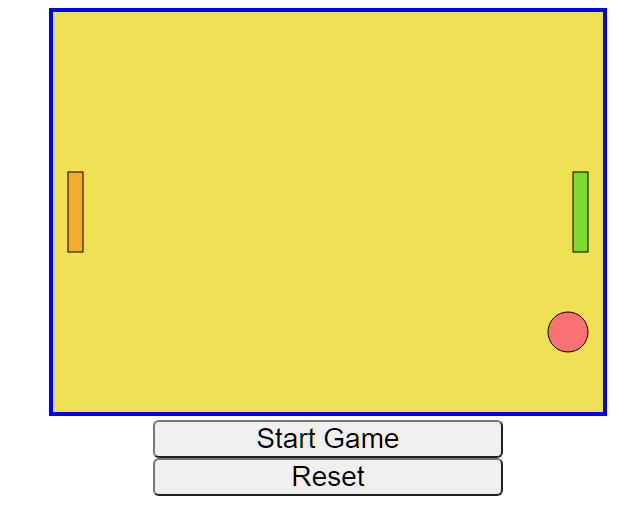
**let racket1 = draw.rect (15,80) .fill ("# F6AC2B"). stroke ({width: 1, color: "black"}).move (15, HEIGHT / 2-80 / 2);**

**let racket2 = draw.rect (15,80) .fill ("# 7ED92D"). stroke ({width: 1, color: "black"}).move (WIDTH-15-15, HEIGHT / 2-80 / 2);**

Let's analyze our actions point by point:

1. By **x** we ​​moved the paddles closer to the center, first subtracted or added their width and added another 15 pixels so that they would not stick to the edges.
2. In height, we took away half of our racket's height, so that the center of the racket was in the center of the field, and not the upper left edge.

Everything became as it should



Moving the paddles over the buttons

Now it's time to get the rackets to move. To do this, let's create variables that will be responsible for the speed of the rackets and we need to program their movement. Let's hang the button click on the entire document:

**let step\_left = 0;**

**let step\_right = 0;**

**/ \* rest of code \* /**

**document.onkeyd own = function () {**

**}**

Now let's find out the codes for the up and down arrow keys using the familiar **event**.

**document.onkeyd own = function () {**

**console.log (event)**

**}**

Various **keyboardEvent** browser events are returned to us, open any of them and find the **keyCode** property and will display it



**document.onkeyd own = function () {**

**console.log (event.keyCode)**

**}**

Great, we learned the key codes: 38 and 40. Now let's enter **event.keyCode** and, depending on what value it is equal to, we will increase or decrease **step\_right** and print it to the console

**document.onkeyd own = function () {**

**let key\_code = event.keyCode**

**if (key\_code == 38) {**

**step\_right--;**

**}**

**if (key\_code == 40) {**

**step\_right ++;**

**}**

**console.log (step\_right)**

**}**

Now we will return the value back to our racket.

**console.log (step\_right)**

**racket2.y (step\_right)**

The racket moves, but jumps up rather slowly and sharply. To fix this, replace **y** with **dy**. This function will add our step to the current value, and not set it to a specific number, as if + = and let's add 5 at a time

**document.onkeyd own = function () {**

**let key\_code = event.keyCode**

**if (key\_code == 38) {**

**step\_right+ = 5;**

**}**

**if (key\_code == 40) {**

**step\_right+ = 5;**

**}**

**console.log (step\_right)**

**racket2.dy (step\_right)**

**}**

Things got even weirder, so let's do dy in each of the conditions

**document.onkeyd own = function () {**

**let key\_code = event.keyCode**

**if (key\_code == 38) {**

**step\_right + = 5;**

**racket2.dy (step\_right)**

**}**

**if (key\_code == 40) {**

**step\_right + = 5;**

**racket2.dy (step\_right)**

**}**

**console.log (step\_right)**

**}**

What happened with us? The racket moves faster and faster. Let's think about why this is happening. To do this, let's take a close look at what is written in the console. If you haven't guessed, then everything is very simple - when we release the button, the variable should be zeroed, but this does not happen for us. To fix the situation, we will write a function to release the keys.

**document.onke yup = function () {**

**}**

In it, we will perform similar actions, only we will set the step value 0

**document.onke yup = function () {**

**let key\_code = event.keyCode**

**if (key\_code == 38) {**

**step\_right = 0**

**}**

**if (key\_code == 40) {**

**step\_right = 0;**

**}**

**console.log (step\_right)**

**}**

Everything works, also note that the results of **if** are the same, so we can write it in one statement. Let's think about what kind of logical connectivity we need. The correct answer is **either (||)**

**let key\_code = event.keyCode**

**if (key\_code == 38 || key\_code == 40) {**

**step\_right = 0**

**}**

Restricting the movement of rackets around the edges

Everything works, but the racket crawls out of the edges - let's think about how to fix it. To do this, we need additional conditions in which we write that if the y of the paddle is greater than the height of the entire **SVG** tag, then set the y of the paddle to the height position.

Note that in this case y is a method that pulls out the current coordinate from the object, so after it you need to put empty parentheses if you want to take a value or write a value in these brackets when doing the opposite. Programmers call this "getter" and "setter" by analogy with the already familiar **getAttribute** and **setAttribute**.

If a student has forgotten about the peculiarities of SVG coordinates, then do not rush to prompt, let him try to fix it himself. As a result, such a code comes out by clicking on the "down arrow"

**if (key\_code == 40) {**

**step\_right + = 5;**

**racket2.dy (step\_right)**

**if (racket2.y ()> HEIGHT-racket2.height ()) {**

**racket2.y (HEIGHT-racket2.height ())**

**}**

**}**

We will write a similar one for the top edge, but here it will be easier.

**if (key\_code == 38) {**

**step\_right- = 5;**

**racket2.dy (step\_right)**

**if (racket2.y () <0) {**

**racket2.y (0)**

**}**

**}**

[Homework 2](https://hwschool.bitrix24.ru/knowledge/front-end/svgcanvas/43/?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y" \l "block131355" \t "_self)

Attention, if there is still time in the lesson, then we start working with the second racket right in the lesson. Let the student try on the analogy to make it on the **w** and **s** keys.

Introducing AddEventListener

Everything works well, but the movements are very sharp, it is inconvenient to play. To fix the situation, let's move **dy** for both rackets into the ball movement function, since it works on a timer. Now, if you try hard, the racket will fly over the map, but immediately return.

**function move\_ball () {**

**racket2.dy (step\_right)**

**racket1.dy (step\_left)**

**// rest of the code**

**}**

After we have the movement of two rackets, let's try to divide it into 2 separate functions, it will come out something like this

**// left paddle**

**document.onkeyd own = function () {**

**// two if when when the keycode is 87 and 83**

**}**

**// right racket**

**document.onkeyd own = function () {**

**// two if when when the key code is 38 and 40**

**}**

Now let's check how everything works. Alas, only one button works. This happens because a document, and indeed any node (tag, class, id), can have only one event handler. But the **addEventListener** method will help to solve this situation. Its syntax is as follows:

**addEventListener (“browser event”, function {})**

It allows you to add handlers rather than replace them. Let's use it and rewrite the code

**// right racket**

**document.addEventListener ("keydown", function () {**

**let key\_code = event.keyCode**

**console.log (key\_code)**

**if (key\_code == 38) {**

**step\_right- = 5;**

**racket2.dy (step\_right)**

**if (racket2.y () <0) {**

**racket2.y (0)**

**}**

**}**

**if (key\_code == 40) {**

**step\_right + = 5;**

**racket2.dy (step\_right)**

**if (racket2.y ()> HEIGHT-racket2.height ()) {**

**racket2.y (HEIGHT-racket2.height ())**

**}**

**}**

**})**

**// left paddle**

**document.addEventListener ("keydown", function () {**

**let key\_code = event.keyCode**

**if (key\_code == 87) {**

**step\_left- = 5;**

**racket1.dy (step\_left)**

**if (racket1.y () <0) {**

**racket1.y (0)**

**}**

**}**

**if (key\_code == 83) {**

**step\_left + = 5;**

**racket1.dy (step\_left)**

**if (racket1.y ()> HEIGHT-racket1.height ()) {**

**racket1.y (HEIGHT-racket1.height ())**

**}**

**}**

**})**

Now everything works, let's rewrite the release of the keys in the same way

**// left paddle**

**document.addEventListener ("keyup", function () {**

**let key\_code = event.keyCode**

**if (key\_code == 38 || key\_code == 40) {**

**step\_right = 0**

**}**

**})**

**// right paddle document.addEventListener ("keyup", function () {**

**let key\_code = event.keyCode**

**if (key\_code == 87 || key\_code == 83) {**

**step\_left = 0**

**}**

**})**

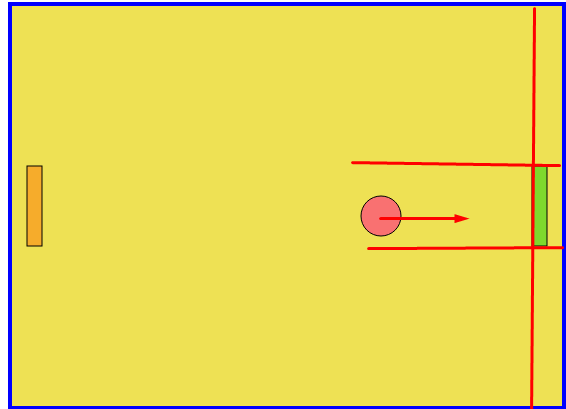
Everything works as it should. I advise in the future to add handlers in this way, if there may be several of them.

Bumping rackets

Now let's talk about pushing the ball off the racket. This will already be more difficult to do than pushing off the edges. We will need to consider several points.

1. The bottom of the ball must be higher than the bottom of the racket
2. The top of the ball must be lower than the top of the racket
3. The right edge of the ball must be greater than the left edge of the racket

It sounds complicated, but in the picture everything is much clearer. Let's think about these 3 conditions, do they have to fulfill at the same time, or is it enough so that only one is fulfilled? The correct answer is simultaneously, so we will combine them with a logical link **AND (&&)**



Let's start with condition 3, which is the simplest. Let us write in the ball motion function that if its center plus radius is greater than the x coordinate of the racket, then let its step change sign

**function move\_ball () {**

**// all the previous code**

**if (x + radius> racket2.x ()) {**

**step\_x = -step\_x**

**}**

**// return values ​​back to html**

**ball.cx (x);**

**ball.cy (y);**

**}**

Now we see that the ball does not reach the wall, but is immediately pushed off. This is normal because we have only met one of 3 conditions so far. Add **&&** and write condition 2. In it, we say that the y-coordinate of the ball should be less than the bottom edge of the paddle. Don't forget about the **SVG** coordinate system.

**This stage is often given to students to “think”, as it develops logic perfectly, if it doesn’t work, then we write the code together.**

**if (x + radius> racket2.x () &&y + radius> racket2.y ()) {**

**step\_x = -step\_x**

**}**

It seems that nothing has changed, the ball bounces again even when it does not hit the racket, but let's launch it into the upper corner by changing its y step to negative

**let step\_x = 2;**

**let step\_y = -2;**

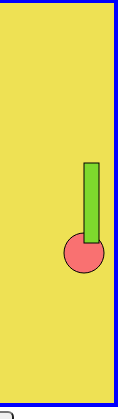
Now we see that the ball flies from above, but from below it continues to fight back even without a racket, before we write one more restriction.

**if (x + radius> racket2.x () && y + radius> racket2.y () &&y + radius <racket2.y () + racket2.height ()) {**

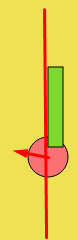
**step\_x = -step\_x**

**}**

During the checks, it is easy to notice that if you hit the ball with the edge of the racket, it will get stuck in it and hit the inner walls several times. We simulate the situation



To avoid this in a collision, in addition to changing the step, we will move the center of the ball to the edge of the racket.



**if (collision\_racket2) {**

**step\_x = -step\_x**

**x = racket2.x ()**

**}**

The result is even stranger, the ball gets stuck in the middle of the racket. Let's think about why this is happening. The answer is simple - we press the center of the ball against the edge of the racket, not the edge. To correct this situation, subtract the radius from the edge of the racket, thereby shifting the ball to the left.

**if (collision\_racket2) {**

**step\_x = -step\_x**

**x = racket2.x ()-radius ()**

**}**

Remembering the account

We check the first homework in the project. If the student did it, then great. If not, then we will spend 10-15 minutes on its implementation.

We have variables for counting, and we increment them and output them to the console when we touch the left or right edge. It's time to make the player see these numbers. Let's create some text using SVG. Let's write somewhere at the beginning of the file that we want to create a text that initially contains zero.

**// create all the game objects**

**let draw = SVG (). addTo ('# game'). size (WIDTH, HEIGHT);**

**let background = draw.rect (WIDTH, HEIGHT) .fill ("# EEE154");**

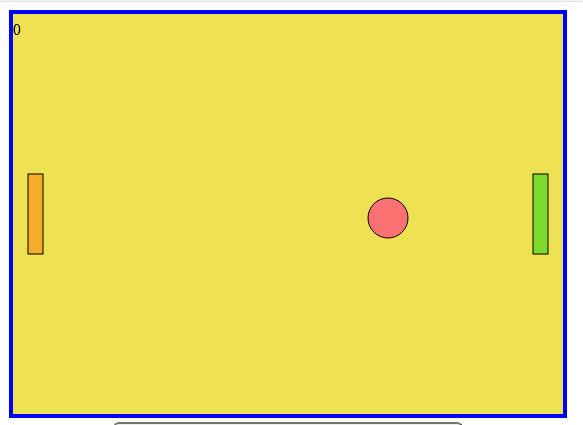
**let ball = draw.circle (). radius (20) .cx (WIDTH / 2) .cy (HEIGHT / 2) .fill ("# F97171"). stroke ({width: 1, color: "black"}) ;**

**let racket1 = draw.rect (15,80) .fill ("# F6AC2B"). stroke ({width: 1, color: "black"}). move (15, HEIGHT / 2-80 / 2);**

**let racket2 = draw.rect (15,80) .fill ("# 7ED92D"). stroke ({width: 1, color: "black"}). move (WIDTH-15-15, HEIGHT / 2-80 / 2 );**

**let racket1\_score = draw.text ("0")**

**let racket2\_score = draw.text ("0")**

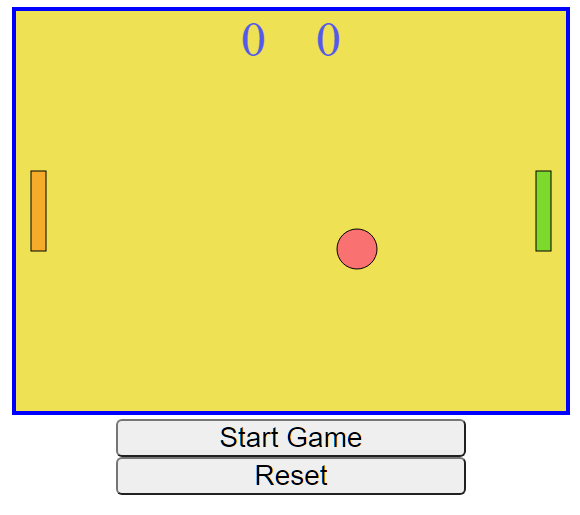


And now we will use the move function, plus fill our numbers with some other color and increase the font size

**let racket1\_score = draw.text ("0").fill ("# 5559EA"). font ({size: 50}). move (225,0);**

**let racket2\_score = draw.text ("0").fill ("# 5559EA"). font ({size: 50}). move (300,0);**

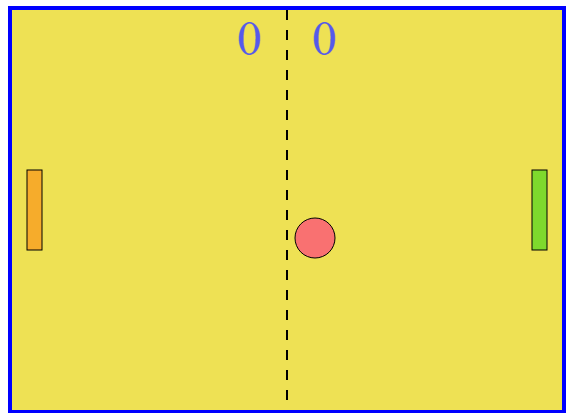
This already looks nicer



And I also propose to make a dividing line in the center. The calculation of the coordinates of the beginning and end is left to the student. Let's also make it dashed using the **dasharray** property of the **stroke**.

**let line = draw.line (275,0,275, HEIGHT) .stroke ({color: "black", width: 2, dasharray: "10"});**

In addition to 10, we will try other values ​​and find a pattern that the smaller the number, the less the strokes of our lines come out. If you are interested, then after the lesson you can search the Internet for what else you can customize for stroke and fill in **SVG**



Great, now let's say that the values ​​of our numerical variables are entered into the text. To do this, we return to the ball motion function and find the conditions responsible for collisions with the side edges and add

**if (x + radius> = WIDTH) {**

**x = WIDTH / 2**

**score\_left ++**

**racket1\_score.text (score\_left)**

**}**

**if (x-radius <= 0) {**

**x = WIDTH / 2**

**score\_right ++**

**racket2\_score.text (score\_right)**

**}**

We get an error in the console, the reason is that we are trying to put a number in a string, and SVG cannot do that. This problem often occurs in other programming languages. To solve it, we simply convert the number to a string. If you remember, we already know how to convert a string to a number - this is the **int()** function, where the argument is a string or a variable.

The reverse action requires the **.toString()** method, it is called on the desired string and has no arguments.

**racket1\_score.text (score\_left.toString ())**

We do the same for the second variable and enjoy the result.

[Homework 3](https://hwschool.bitrix24.ru/knowledge/front-end/svgcanvas/43/?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y" \l "block131361" \t "_self)

The reset button

We are already at the home stretch. Let's make a function that will reset the score, return the ball and rackets to their original position. Let's go in order.

Let's write a handler for pressing the **reset** button

**document.querySelector (". reset\_score"). oncl ick = function () {**

**}**

Next, we will zero out the variables responsible for the account and add them to the text.

**document.querySelector (". reset\_score"). oncl ick = function () {**

**score\_left = 0;**

**score\_right = 0;**

**racket1\_score.text (score\_left.toString ());**

**racket2\_score.text (score\_right.toString ());**

**}**

We will do the same with the rackets, so as not to remember which coordinates we should specify - just look at the variables when creating our rackets.

**document.querySelector (". reset\_score"). oncl ick = function () {**

**score\_left = 0;**

**score\_right = 0;**

**racket1\_score.text (score\_left.toString ());**

**racket2\_score.text (score\_right.toString ());**

**racket1.move (15, HEIGHT / 2-80 / 2);**

**racket2.move (WIDTH-15-15, HEIGHT / 2-80 / 2);**

**}**

Winning and revising browser events

Now let's make it so that the reset occurs when one of the sides reaches 10 points. Let's think about how to do it. The answer is very simple - let's move the code into a function with a name and call it provided that **score\_left>10 or score\_right>10**.

**function reset (){**

**score\_left = 0;**

**score\_right = 0;**

**racket1\_score.text (score\_left.toString ());**

**racket2\_score.text (score\_right.toString ());**

**racket1.move (15, HEIGHT / 2-80 / 2);**

**racket2.move (WIDTH-15-15, HEIGHT / 2-80 / 2);**

**}**

**function move\_ball () {**

**// previous code**

**if (score\_left> = 10 || score\_right> = 10) {**

**alert ("Victory")**

**reset ()**

**}**

**// return values ​​back to html**

**ball.cx (x);**

**ball.cy (y);**

**}**

**// call the function by timer**

**setInterval (move\_ball, 10)**

Works as expected, but it is not clear which player won. To do this, we divide the solution into two conditional operators

**function move\_ball () {**

**// previous code**

**if (score\_left> = 10) {**

**alert ("Left player wins")**

**reset ()**

**}**

**if (score\_right> = 10) {**

**alert ("Right player wins")**

**reset ()**

**}**

**// return values ​​back to html**

**ball.cx (x);**

**ball.cy (y);**

**}**

**// call the function by timer**

**setInterval (move\_ball, 10)**

Now let's remember our button and let's bind a function to the browser's **onclick** event.

**document.querySelector (". reset\_score"). oncl ick = reset**

**Note that we do not call the function here, but we bind it, as is the case with setInterval. If the student puts the reset() parentheses, then the console will not generate an error, but the function will not be triggered either.**

You can also write this through the **addEventListener** method already familiar to us.

**document.querySelector (". reset\_score"). addEventListener ("click", reset)**

It works both ways, let's try to remember the difference.

The answer is that with the second method, we can bind several handlers to the **click** event.

Ball movement and random

Now the ball always flies to the lower left corner at the beginning of the game and after the reset. Let's create a function to generate random numbers. If you remember, the standard **Math.random ()** returns a fractional random number from 0 to 1 and we cannot adjust the range.

**function getRandomInt () {**

**}**

The function will take 2 arguments, the minimum and maximum value, and return our "pseudo-random" number.

**Formula**

**Math.floor (Math.random () \* (max - min + 1)) + min;**

**can be easily found on the Internet, so we will not complicate the task and send it to the student**

**function getRandomInt (min, max) {**

**return Math.floor (Math.random () \* (max - min + 1)) + min;**

**}**

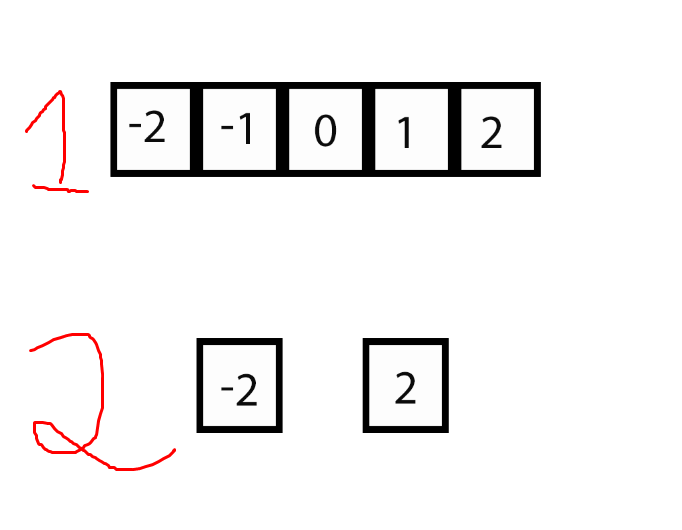
We should have a random step at the beginning of the game

**let step\_x =getRandomInt (-2,2);**

**let step\_y =getRandomInt (-2,2);**

Everything works, but now the ball sometimes flies up and it is impossible to hit it back. This is because **getRandomInt** returns a number in the range, so it can return 0 or 1.

I propose to create an array and select a number from its elements.**Note that we make a random choice not between numbers, but between their indices (numbers).**



**let array = [- 2,2]**

**let step\_x= array [getRandomInt (0,1)];**

**let step\_y= array [getRandomInt (0,1)];**

[Homework 4](https://hwschool.bitrix24.ru/knowledge/front-end/svgcanvas/43/?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y?IFRAME=Y#block131367)

**Hometasks**

Homework 1

Make it so that when the ball returns to the center touching the right or left edge of the playing field , it does not push off.

Create two variables responsible for the score and change them depending on the edge we touched. Output them to the console for now.

Teacher code:

**let score\_left = 0;**

**let score\_right = 0;**

**function move\_ball () {**

**// previous code**

**if (y + radius> = HEIGHT || y-radius <0) {**

**step\_y = -step\_y**

**}**

**if (x + radius> = WIDTH) {**

**x = WIDTH / 2**

**score\_left ++**

**console.log (score\_left)**

**}**

**if (x-radius <= 0) {**

**x = WIDTH / 2**

**score\_right ++**

**console.log (score\_right)**

**}**

**ball.cx (x);**

**ball.cy (y);**

**}**

Homework 2

Move the second (left) racket using the W and S keys

Teacher code

**document.onkeyd own = function () {**

**let key\_code = event.keyCode**

**if (key\_code == 38) {**

**step\_right- = 5;**

**racket2.dy (step\_right)**

**if (racket2.y () <0) {**

**racket2.y (0)**

**}**

**}**

**if (key\_code == 40) {**

**step\_right + = 5;**

**racket2.dy (step\_right)**

**if (racket2.y ()> HEIGHT-racket2.height ()) {**

**racket2.y (HEIGHT-racket2.height ())**

**}**

**}**

**if (key\_code == 87) {**

**step\_left- = 5;**

**racket1.dy (step\_left)**

**if (racket1.y () <0) {racket1.y (0)**

**}**

**}**

**if (key\_code == 83) {**

**step\_left + = 5;**

**racket1.dy (step\_left)**

**if (racket1.y ()> HEIGHT-racket1.height ()) {**

**racket1.y (HEIGHT-racket1.height ())**

**}**

**}**

**}**

**// release buttons**

**document.onke yup = function () {**

**let key\_code = event.keyCode**

**if (key\_code == 38 || key\_code == 40) {**

**step\_right = 0**

**}**

**if (key\_code == 87 || key\_code == 83) {**

**step\_left = 0**

**}**

**}**

Homework 3

Make a collision with the second paddle on the example of the first one yourself. In the process, keep the SVG coordinate system in memory.

Teacher code

**let collision\_racket1 = x-radius <racket1.x () + racket1.width () && y> = racket1.y () && y <= racket1.y () + racket1.height ()**

**if (collision\_racket1) {**

**step\_x = -step\_x; x = racket1.x () + radius + racket1.width ()**

**}**

Homework 4

We completely forgot about the “Start Game” button.

1. Make it using a boolean variable so that at the beginning the game does not work, and when the button is pressed, the ball and rackets begin to move
2. When you press the Reset button, the game also stops

Teacher code

**let start = false;**

**function move\_ball () {**

**if (start) {**

**// all the function code**

**}**

**}**

**function reset () {**

**start = false;**

**// rest of the function code**

**}**

**document.querySelector (". start\_game"). oncl ick = function () {**

**start = true;**

**}**

**document.querySelector (". reset\_score"). oncl ick = reset**