**Disciplina: Pensamento Computacional**

**Programação: JavaScript – P5.js**

**AULA: 01**

**01 Cenário e bolinha**

Desenvolvemos nosso jogo e consolidamos a lógica de programação! Agora, vamos desenvolvê-lo em outra linguagem de programação, o JavaScript. Mas onde iremos fazer isso, e como iremos visualizá-lo?

Utilizaremos um serviço Web chamado p5.js, que exige cadastro de uma conta para podermos salvar os nossos projetos. Iremos manter o Scratch aberto, também, para fazermos algumas comparações. Nele, temos o ícone de bandeira verde, enquanto no p5 temos um botão de play que, ao ser pressionado, habilita um quadrado de "Preview"; na área de código, há algumas linhas preexistentes.

O fundo cinza em "Preview" é criado a partir da função createCanvas(), que possui dois valores, ambos 400. Vamos testar alterando somente o primeiro para 800. Isto fará com que a largura do retângulo cinza, que é onde será exibido nosso jogo, aumente. As alterações feitas no código se refletem ao seu lado conforme são feitas.

Também temos a função background(), cujo valor default é 220. Trocaremos para 260, e o fundo ficará todo branco, em vez de cinza. Quanto menor este valor, mais escuro fica o fundo do retângulo que antes era cinza. Por isto, deixaremos 0 para que o fundo do nosso jogo fique preto.

Há um problema: no Scratch, todo o código fica visível e é intuitivo montá-los, bastando buscá-los de acordo com o que queremos fazer dentre as opções disponibilizadas. Começaremos desenhando uma bolinha, tal como fizemos no Scratch. Simplesmente escrever circulo dentro do bloco de código de draw() não nos traz nenhum retorno, e ainda dá erro — "circulo is not defined", ou "circulo não está definido".

Para buscarmos pelos códigos necessários, clicaremos em "Help & Feedback" e em "Reference", o que abrirá uma nova aba. Em Shape, dentre as funções listadas, está circle() que, após clicarmos, exibirá um exemplo: circle(30, 30, 20). Descendo um pouco a página, temos a explicação da sua sintaxe, circle(x, y, d), cujos parâmetros são, respectivamente, a coordenada no eixo X, no eixo Y e o diâmetro do círculo.

Lembrando que o diâmetro é o dobro do raio de uma circunferência, que é a linha que liga o centro da mesma à sua borda.

O trecho de código ficará, portanto, assim:

function setup() {

createCanvas(600, 400);

}

function draw() {

background(0);

circle(0, 0, 50);

}

Ao pressionarmos o botão de play, teremos cerca de 1/4 do círculo posicionado no canto superior esquerdo da tela. Por quê será que isso acontece? No Scratch, definimos como posição inicial da bolinha a coordenada (0, 0), a partir do centro, enquanto aqui o plano cartesiano, isto é, a movimentação nos eixos X e Y será um pouco diferente: o (0, 0) da nossa tela passa a ser o extremo canto superior esquerdo, exatamente onde se encontra a bolinha no momento.

Será necessário, portanto, aumentar os valores deles para que a bolinha fique visível e localizada no centro da tela. E para que o código fique ainda mais claro e legível, armazenaremos tais valores em variáveis, usando a palavra let:

let xBolinha = 300;

let yBolinha = 200;

let diametro = 15;

function setup() {

createCanvas(600, 400);

}

function draw() {

background(0);

circle(xBolinha, yBolinha, diametro);

}

As nomenclaturas de variáveis seguem a convenção de terem a primeira letra da primeira palavra em minúscula, e as primeiras letras das demais palavras, se houver, em maiúscula. Esta convenção se denomina Camel Case.

Agora, precisaremos fazer com que ela se movimente, em ambos os eixos. Para isso, indicaremos na função draw() que xBolinha sempre terá acréscimo de 1, o que fará com que a bolinha se movimente para a direita, em linha reta.

function draw() {

background(0);

circle(xBolinha, yBolinha, diametro);

xBolinha = xBolinha + 1;

}

O 1, então, seria a velocidade com que a bolinha se movimenta, porém isto não fica claro em nosso código. Criaremos a variável velocidadeXBolinha, para a qual atribuiremos o valor 6, para testarmos, e teremos xBolinha = xBolinha + velocidadeXBolinha. Com isto, a bolinha se locomove para a direita até sumir da tela.

Uma forma mais elegante de escrevermos esta mesma linha é xBolinha += velocidadeXBolinha, ou seja, o X da bolinha será seu valor acrescido de sua velocidade. Como queremos que a bolinha se movimente para direções distintas, criaremos yBolinha, e a variável velocidadeYBolinha, com valor 6.

let xBolinha = 300;

let yBolinha = 200;

let diametro = 15;

let velocidadeXBolinha = 6;

let velocidadeYBolinha = 6;

function setup() {

createCanvas(600, 400);

}

function draw() {

background(0);

circle(xBolinha, yBolinha, diametro);

xBolinha += velocidadeXBolinha;

yBolinha += velocidadeYBolinha;

}

Assim, sempre que iniciarmos o jogo, a bolinha surgirá do centro, indo para baixo diagonalmente até desaparecer na tela. Não é bem isso que queremos, e sim que ela permaneça dentro das bordas da tela. Faremos isso a seguir!

Atividades

1-Nesta aula, aprendemos a trabalhar em um editor de texto chamado p5.js. Se quisermos alterar a cor do preview, qual função devemos usar?

 createCanvas()

Alternativa incorreta. A função createCanvas() muda as dimensões do preview.





circle()

 Alternativa correta



**background()**

Alternativa correta. Ao modificarmos os valores dessa função, teremos uma alteração na tonalidade do preview.





triangle()

2-Criamos 2 variáveis para movimentar os eixos x e y da bolinha, como ilustra o código abaixo:

let velocidadeXBolinha = 6

let velocidadeYBolinha = 6COPIAR CÓDIGO

O que acontecerá se alterarmos os valores numéricos para 3, ao invés de 6, como mostra o código abaixo?

let velocidadeXBolinha = 3

let velocidadeYBolinha = 3

**Parte superior do formulário**

* **Alternativa correta**

****

**A velocidade da bolinha diminuirá.**

Alternativa correta. Quanto maiores os valores das variáveis de velocidade para os eixos x e y da bolinha, maior será sua velocidade. Logo, ao diminuirmos esses valores, a velocidade também diminuirá.



A bolinha altera de tamanho nos eixos X e Y.



A bolinha irá quicar 3 vezes, ao invés de 6.



Os eixos serão alterados.

Parte inferior do formulário

**AULA: 02**

**Colisão com as bordas**

A bolinha está se movendo para além das bordas do nosso jogo, e não é este o comportamento que queremos. Queremos que, assim que ela toque uma das bordas, ela inverta a posição, assim como fizemos no Scratch. Para isto, precisamos verificar se ela está tocando a borda em algum momento, dentro da função que está desenhando o nosso jogo (draw()).

Na programação, esta verificação, o "se", é escrito com if, e usaremos uma variável que o próprio p5 disponibiliza. Além disso, comentaremos a linha contendo yBolinha. Se xBolinha for maior que a largura (width) da tela, queremos fazer algo, que por sua vez estará entre chaves (ou "bigodes"). No caso, iremos multiplicar velocidadeXBolinha por -1, para que ela se movimente no sentido oposto.

function draw() {

background(0);

circle(xBolinha, yBolinha, diametro);

xBolinha += velocidadeXBolinha;

//yBolinha += velocidadeYBolinha;

if (xBolinha > width) {

velocidadeXBolinha \*= -1;

}

}

Ao testarmos este código, a bolinha parte da área central da tela, colide com a lateral direita da tela, inverte o sentido, e desaparece quando ultrapassa a lateral esquerda da tela. Sabemos que esta lateral esquerda é o X = 0, portanto acrescentaremos outra condição no código, por meio de duas barras verticais, o que quer dizer "ou":

if (xBolinha > width || xBolinha < 0) {

velocidadeXBolinha \*= -1;

}

Desta vez, temos a bolinha reconhecendo tanto o limite lateral direito quanto o esquerdo. Para lidarmos com os movimentos verticais, descomentaremos a linha com yBolinha e comentaremos a linha com xBolinha, para melhor entendimento. Faremos algo bem similar ao que fizemos anteriormente, em relação a X, mas desta vez lidaremos com a altura (height) da tela.

function draw() {

background(0);

circle(xBolinha, yBolinha, diametro);

//xBolinha += velocidadeXBolinha;

yBolinha += velocidadeYBolinha;

if (xBolinha > width || xBolinha < 0) {

velocidadeXBolinha \*= -1;

}

if (yBolinha > height || yBolinha < 0) {

velocidadeYBolinha \*= -1;

}

}

Descomentaremos a linha com xBolinha e verificaremos que as bordas estão sendo reconhecidas, como gostaríamos.

**Atividades**

1-Usamos as condicionais para realizar uma verificação de algum aspecto do código e executar uma ação.

Qual é a sintaxe correta para escrevermos uma condicional, em programação na linguagem JavaScript, a fim de verificar se a posição da bolinha no eixo x é maior que a largura da tela?

**Parte superior do formulário**

* **Alternativa correta**

****

**if (xBolinha > width) { }**

Alternativa correta. Nessa opção, temos a condicional apresentada com if, seguido da variável em parênteses e, ao fim, o par de chaves para indicar a ação.



if {xBolinha > width}



if Bolinha > width. { }



if Bolinha =\* width. { }

Parte inferior do formulário

2-Durante a aula aprendemos como fazer comentários (ou anotações) no nosso código. Se escrevermos uma parte do nosso código da seguinte maneira, o que acontecerá com a segunda linha ao executarmos?

xBolinha += velocidadeXBolinha;

//yBolinha += velocidadeYBolinha;COPIAR CÓDIGO

Parte superior do formulário



A bolinha irá se movimentar apenas no eixo Y, o eixo X não será lido durante a execução, pois está com um comentário.



Não acontecerá nada, pois uma sinalização de “erro” irá aparecer.



A primeira linha será duplicada.

* Alternativa correta



A bolinha irá se mover apenas no eixo X, o eixo Y não será lido durante a execução, uma vez que está com um comentário.

Alternativa correta. As duas barras no início da linha indicam que há um comentário ou anotação e, portanto, a linha não será considerada na execução do código.

Parte inferior do formulário

**AULA: 03**

**Raio e diâmetro**

Alteraremos as velocidades relativas tanto ao eixo X quanto ao eixo Y para 2 para deixar a bolinha mais lenta e assim conseguirmos observar melhor estes movimentos. Uma parte da bolinha ainda ultrapassa os limites das bordas, e não queremos que isso aconteça. Vamos voltar às velocidades originais, 6, e pensar no porquê disso estar acontecendo.

Na documentação do circle(), é indicado que o X é o centro do círculo, o que será levado em consideração para que se reconheça que houve uma colisão da bolinha com alguma das bordas. No entanto, queremos que isto se dê a partir do raio, isto é, das extremidades da bolinha. Uma vez que o diâmetro é 2x o valor do raio, criaremos a variável raio, que receberá diametro / 2.

Com isso, diminuiremos as velocidades da bolinha novamente (para 2), para enxergarmos melhor os movimentos, e comentaremos a linha com yBolinha para testar primeiro no eixo X, em que somaremos o valor do raio para o lado direito, e subtrairemos o mesmo valor do lado esquerdo:

if (xBolinha + raio > width || xBolinha - raio < 0) {

velocidadeXBolinha \*= -1;

}

Em seguida, descomentaremos a linha com yBolinha e comentaremos a linha com xBolinha. Da mesma forma como fizemos em relação ao eixo X, para o eixo vertical teremos:

if (yBolinha + raio > height || yBolinha - raio < 0) {

velocidadeYBolinha \*= -1;

}

Voltaremos a velocidade da bolinha para 6 e testaremos mais uma vez, agora sem nenhum trecho comentado. Nossa bolinha está reconhecendo todas as bordas da tela do jogo!

**Refatoração e funções**

Em nosso jogo, criamos uma bolinha e verificamos suas colisões com as bordas e, para que pudéssemos ter este resultado, criamos variáveis para a bolinha, melhorando a legibilidade do código. Na função draw() fazemos várias ações: desenhamos e movimentamos a bolinha dentro de uma determinada área, e verificamos se a bolinha está de fato colidindo ou não com as bordas.

Será que existe uma maneira de deixarmos nosso código ainda melhor?

Poderemos fazê-lo sem alterar seu comportamento por meio da **refatoração** e o uso de funções para melhor identificarmos cada trecho de código. Fora do escopo de draw(), criaremos a função mostraBolinha(), mas isto não será o suficiente, pois é necessário chamá-la em draw(). Do mesmo modo, criaremos movimentaBolinha() e verificaColisaoBorda()

function draw() {

background(0);

mostraBolinha();

movimentaBolinha();

verificaColisaoBorda();

}

function mostraBolinha() {

circle(xBolinha, yBolinha, diametro)

}

function movimentaBolinha() {

xBolinha += velocidadeXBolinha;

yBolinha += velocidadeYBolinha;

}

function verificaColisaoBorda() {

if (xBolinha + raio > width || xBolinha - raio < 0) {

velocidadeXBolinha \*= -1;

}

if (yBolinha + raio > height || yBolinha - raio < 0) {

velocidadeYBolinha \*= -1;

}

}

Continuando, as variáveis xBolinha, yBolinha, diametro e raio se referem à bolinha, portanto poderemos adicionar um comentário //variáveis da bolinha, assim como um //variáveis da velocidade da bolinha logo acima das linhas que contém velocidadeXBolinha e velocidadeYBolinha.

//variáveis da bolinha

let xBolinha = 300;

let yBolinha = 200;

let diametro = 15;

let raio = diametro / 2;

//velocidade da bolinha

let velocidadeXBolinha = 6;

let velocidadeYBolinha = 6;

Esta é uma das formas de melhorarmos nosso código, deixando-o mais compreensível e organizado, sem modificar o comportamento final.

**Entendendo a biblioteca P5.js**

Ao longo do curso você já deve ter se perguntado “como a bolinha se movimenta na tela?”, ou mesmo “O que é essa tal de function draw() e por qual motivo preciso chamar as funções que construí na function draw()”, não é?!

Para solucionar essas dúvidas, devemos entender um pouquinho sobre a nossa ferramenta. A biblioteca p5.js. Vamos lá?

O p5.js é uma biblioteca JavaScript criada para tornar o desenvolvimento algo acessível e inclusivo para artistas, designers, educadores e iniciantes, ou seja, para qualquer pessoa! Por conta dessa característica, a biblioteca vem com inúmeras funções **”pré-fabricadas”** (algo pré-pronto apenas para você usar), que **facilita** o processo de escrita do código. Sendo assim, a função **draw()**, **setup()**, **preload()** são alguns exemplos dessas soluções que já estão incluídas na biblioteca para **facilitar a vida do desenvolvedor**.

Ok, mas você deve continuar se perguntando: **“e os desenhos?”**

A função própria da biblioteca p5.js responsável por mostrar os desenhos na tela de pré-visualização e atribuir comportamentos ao seu código é a **function draw()**. Além disso, o p5.js utiliza um **sistema de coordenadas** que atribui um valor em cada ponto da tela para que possamos identificar o local exato onde os desenhos aparecem.

Mas vamos entender como essa função funciona?

## function draw()

A **function draw()** é a função que faz a "mágica acontecer". Essa função “pré-pronta” do p5.js executa continuamente o seu código, linha por linha. Um aspecto interessante da função é que ela funciona como se possuísse um **loop** (um laço de repetição) , o que faz com que o código escrito em seu escopo seja executado continuamente e por isso que consegue **desenhar** os elementos no ambiente de visualização e fazê-los se movimentar.

Vamos conferir o código da função draw():

function draw() {

background(0); //1 - Desenha o background

mostraBolinha(); // 2 - Desenha a bolinha

movimentaBolinha(); // 3 - Movimenta a Bolinha

verificaColisaoBorda(); // 4 - Verifica Colisão da bolinha

// 5- Volta para o início da função draw()

}

É importante ter em mente que essa leitura é feita de forma muito rápida e por isso não "sentimos essa mudança

Vamos entender um pouco mais?

Logo abaixo você encontra algumas das características principais da function draw() e também formas de fazer um melhor uso dela, vamos conferir:

* Nós não precisamos construir e declarar a draw() porque ela já está contida na biblioteca do p5.js de forma padrão. :)
* A draw () é chamada automaticamente após a setup() e nunca deve ser chamada explicitamente;
* A draw () executa continuamente as linhas de código contidas em seu bloco até que o programa seja interrompido ou noLoop () seja chamada;
* Dentro da draw() as ações são controladas com noLoop(), redraw() e loop(). Depois que noLoop() interrompe a execução do código em draw(), a redraw() faz com que o código dentro de draw() seja executado uma vez e loop() fará com que o código dentro de draw() retome a execução continuamente;
* Podemos inserir diretamente o código na draw() ou chamar as funções que apresentam o comportamento desejado (por questões de boas práticas, o instrutor trabalha com funções no curso);
* Só pode haver uma função draw() para cada sketch, e draw() deve existir se você quiser que o código seja executado continuamente ou para processar eventos como mousePressed();
* Às vezes, você pode ter uma chamada vazia para draw() em seu programa, (...)”.

## Sistema de coordenadas

Além de todas essas informações, um elemento super importante para compreender o posicionamento dos objetos em tela é o **sistema de coordenadas do p5.js.** Para compreender melhor como funciona, vou deixar um link para o artigo sobre o tema:

* [**P5.JS: Plano cartesiano**](https://www.alura.com.br/artigos/p5-plano-cartesiano).

**Atividades**

1- Sem alterar o comportamento do jogo, movemos alguns códigos da função draw para funções que criamos:

* function mostraBolinha()
* function movimentaBolinha()
* function verificaColisaoBorda()

Sabendo disso, podemos definir que função é:

Parte superior do formulário



Um conceito que está relacionado com a ideia de classificar e abstrair coisas.



A união de vários códigos aleatórios, com comportamentos diferentes em um único lugar, visando tornar o código mais legível.

* Alternativa correta



Um conjunto de comandos ou códigos que realiza uma tarefa específica.

Certo! No nosso caso, cada função criada é responsável por um comportamento, tornando a leitura do código mais intuitiva e simples.

**AULA: 04**

**Criando minha raquete**

Em nosso jogo, atualmente temos a bolinha reconhecendo as bordas, mas ainda faltam componentes extremamente importantes: as raquetes. Para a criação da bolinha temos a função mostraBolinha() com circle(), uma palavra reservada do p5. Não adianta criarmos uma função raquete(), pois obteremos um erro indicando que este nome não está definido.

Vamos consultar a documentação acessando "Help & Feedback > Reference", no menu superior do p5. Dentre as funções listadas em Shape, está rect(), que se refere à forma retangular. Ao ser clicado, teremos alguns exemplos e os parâmetros necessários para seu uso. O X e o Y, que são os primeiros parâmetros, se relacionam à posição do retângulo, enquanto o terceiro e o quarto, W e H, respectivamente, são a largura e a altura.

rect(x, y, w, h, [tl], [tr], [br], [bl])COPIAR CÓDIGO

Acrescentaremos a função ao nosso código, passando parâmetros com valores arbitrários para testarmos:

function draw() {

background(0);

mostraBolinha();

movimentaBolinha();

verificaColisaoBorda();

rect(5, 150, 10, 90);

}

Já que neste momento iremos lidar apenas com a raquete, podemos comentar a linha movimentaBolinha() para testarmos melhor. E assim como fizemos para a bolinha, criaremos variáveis para armazenar os valores relacionados à raquete — não podemos nos esquecer de substituir os valores antes definidos na função rect() por estas variáveis!

//variáveis da raquete

let xRaquete = 5;

let yRaquete = 150;

let raqueteComprimento = 10;

let raqueteAltura = 90;

Por fim, colocaremos todo o código de rect() para uma função, visando melhor organização e legibilidade:

function mostraRaquete() {

rect(xRaquete, yRaquete, raqueteComprimento, raqueteAltura);

}

E então precisaremos executar esta função dentro do draw():

function draw() {

background(0);

mostraBolinha();

movimentaBolinha();

verificaColisaoBorda();

mostraRaquete();

}

Em seguida movimentaremos nossa raquete!

PRÓXIMA ATIVIDADEParte inferior do formulário

**Atividades**

**1-** Meu objetivo agora é desenhar uma raquete na tela com as funções pré-existentes na biblioteca do p5.js. O que acontece se eu apenas chamar uma função raquete() no código?

Parte superior do formulário



A função será automaticamente definida.

* **Alternativa correta**

****

**Uma mensagem de erro aparecerá, avisando que o nome não está definido.**

Alternativa correta. Nossa raquete será um retângulo. No p5, já existe uma função predefinida para desenhá-lo, esta chamada de rect(). É possível criar uma função chamada raquete() e obter o mesmo resultado na tela. Mas para isso, devemos inserir a função rect() dentro da função raquete() e chamá-la na função draw(). Para facilitar, vamos usar somente a função rect() e criar nossa raquete através dela.



a) Será criada uma raquete, que será chamada na função draw().



Uma nova biblioteca será importada para o p5.js.

Parte inferior do formulário

**AULA: 05**

**Movimento minha raquete**

Nossa raquete ficou bem legal no nosso jogo — seu posicionamento e tamanhos estão da maneira como gostaríamos. No entanto, ela ainda não se movimenta como no Scratch, em que verificávamos se a seta para cima ou para baixo estava pressionada, e para cada um destes casos executávamos uma ação correspondente.

O JavaScript está em inglês, e não em português como no Scratch; ao modificarmos o idioma no Scratch, verificamos como estre trecho fica após a tradução. Será que se escrevermos desta forma no p5 conseguimos o resultado desejado?

Para garantirmos isto, vamos acessar a documentação mais uma vez e buscar por "events". Estão listados eventos de aceleração, teclado, mouse, toque, dentre os quais optaremos por keyPressed(). No exemplo dado pela documentação, a cor do retângulo é alterada depois de se pressionar a tecla.

Porém, queremos criar um movimento por meio do uso das setas do teclado, e não mudar a cor da raquete. Voltando à documentação, poderíamos testar cada uma das funções relativas ao Keyboard, mas como não há tempo, ficaremos com keyIsDown(). Vamos criar uma função para de fato movimentarmos a raquete.

Assim, se a tecla da seta para cima for pressionada, queremos que a raquete se movimente para cima, e para isto é necessário subtrairmos a posição de Y. Faremos algo muito similar para quando a tecla da seta para baixo for pressionada, e teremos o seguinte código:

function movimentaMinhaRaquete() {

if (keyIsDown(UP\_ARROW)) {

yRaquete -= 10;

}

if (keyIsDown(DOWN\_ARROW)) {

yRaquete += 10;

}

}

Para que esta função seja executada no bloco draw(), incluiremos-na junto às demais:

function draw() {

background(0);

mostraBolinha();

//movimentaBolinha();

verificaColisaoBorda();

mostraRaquete();

movimentaMinhaRaquete();

}

Testaremos pressionando as teclas de seta para cima e para baixo, e repararemos que há um som vinculado a estes movimentos, colocado pelo p5 por questões de **acessibilidade**. Podemos ativá-lo ou desativá-lo clicando no ícone de engrenagem no canto superior direito da tela, na aba "Accessibility". Por ora, desmarcaremos as checkboxes de "Plain-text", "Table-text" e "Sound".

Para que possamos testar o jogo no p5, é necessário dar ênfase à tela do jogo, isto é, clicar nela após pressionarmos o ícone de play.

**AULA: 06**

**Colisão com a raquete**

Já fizemos com que a raquete se movimente por meio das setas do teclado! E para que o jogo funcione, descomentaremos movimentaBolinha() e testaremos novamente. A bolinha parece ultrapassar a raquete, ignorando sua existência, o que pode ser percebido com maior clareza se diminuirmos a velocidade da bolinha. Além disso, poderemos comentar a linha yBolinha += velocidadeYBolinha, para que o movimento só aconteça no eixo X.

Assim como no Scratch, queremos criar uma colisão entre a raquete e a bolinha, portanto criaremos a função verificaColisaoRaquete(), que chamaremos dentro de draw() e será bem parecida com o que temos em verificaColisaoBorda().

function verificaColisaoRaquete() {

if (xBolinha - raio < xRaquete + raqueteComprimento) {

velocidadeXBolinha \*= -1;

}

}

A colisão funciona bem, então testaremos posicionar a raquete para cima do nível em que a bolinha se locomove horizontalmente. Mesmo sem tocar a raquete, ela deixa de tocar a borda lateral esquerda. O mesmo acontece se mantivermos a raquete abaixo do nível de locomoção horizontal da bolinha.

Isto acontece pois definimos que se xBolinha - raio < xRaquete + raqueteComprimento, a bolinha deverá mudar de direção. Não especificamos a posição de Y da bolinha em relação à posição de Y da nossa raquete. Para isso, não precisaremos criar um if para verificar se a bolinha está acima, ou abaixo da raquete.

Queremos inverter a velocidade de X caso haja colisão, portanto incluiremos mais duas verificações:

function verificaColisaoRaquete() {

if (xBolinha - raio < xRaquete + raqueteComprimento

&& yBolinha - raio < yRaquete + raqueteAltura

&& yBolinha + raio > yRaquete) {

velocidadeXBolinha \*= -1;

}

}

Agora que conseguimos corrigir isso, voltaremos a velocidade da bolinha para 6 e descomentaremos todos os trechos de código que estavam comentados.

**Atividades**

**1-** Na função para verificar a colisão com a raquete, criamos a seguinte condicional:

function verificaColisaoRaquete(){

if (xBolinha - raio < xRaquete + raqueteComprimento

&& yBolinha - raio < yRaquete + raqueteAltura

&& yBolinha + raio > yRaquete){

velocidadeXBolinha \*= -1;

}

}

Ao utilizarmos a sintaxe &&, estamos nos referindo a qual operador lógico em JavaScript?

Parte superior do formulário

* Alternativa correta



E

Alternativa correta. Com o operador AND (e em inglês), a condicional SE só será executada quando **todas** as operações matemáticas forem verdadeiras.



OU



Operador de negação.



A sintaxe está incorreta.

Parte inferior do formulário

**AULA: 07**

**Importando outra biblioteca**

Implementamos a colisão da bolinha com a raquete, e obtivemos um comportamento esperado, a mudança de direção da bolinha sempre que isto acontece. Será que outras pessoas já não passaram por este mesmo problema utilizando o p5 e o JavaScript? Será que elas não compartilharam as soluções encontradas para que outras pessoas pudessem usá-las também? A resposta é sim!

Na documentação do p5, há em Libraries algumas bibliotecas contendo soluções ou implementações que podemos adicionar em nosso projeto, feitas por outras pessoas. Escolheremos, por exemplo, "p5.collide2D", que ao ser clicado abrirá uma página do GitHub, plataforma que serve para hospedar código, seja de projetos pessoais ou profissionais.

Neste caso, a página exibe o funcionamento das funções contidas nesta biblioteca. Em nosso projeto, na função mostraRaquete() temos o rect(), e em mostraBolinha() temos circle(). Similarmente, na biblioteca existe a função collideRectCircle(), exatamente o que precisaríamos para o nosso projeto.

Na explicação sobre como ela funciona existe um trecho de código. Será que basta adicioná-la em nosso projeto para que funcione conforme gostaríamos?

Na verdade, uma biblioteca é composta por uma série de códigos, portanto, inicialmente iremos baixá-los, clicando no botão "Clone or download > Download ZIP", no GitHub. O arquivo que iremos utilizar é o p5.collide2d.js; no p5, do lado esquerdo do painel que contém nosso código, existe um > que, ao ser clicado, exibe todas as pastas e arquivos que compõem o projeto: index.html, style.css e sketch.js.

Clicaremos no símbolo v ao lado de "project-folder", e em "Add file" para adicionar o arquivo recém baixado. Segundo a documentação, a função collideRectCircle() cria uma variável hit para verificar se há colisão ou não, com var hit = false. Em nosso projeto, o var corresponde a let. Vamos criar uma variável similar:

let colidiu = false;COPIAR CÓDIGO

E então, todo o código referente à colisão é inserida na função draw(), que em nosso código é bem específica e contém uma função para cada ação tomada. Vamos acrescentar a função colisaoMinhaRaqueteBiblioteca() e comentar a linha que vem acima, com verificaColisaoRaquete(), uma vez que queremos utilizar a solução encontrada por outra pessoa, isto é, colisaoMinhaRaqueteBiblioteca(), que criaremos no fim do nosso código:

function colisaoMinhaRaqueteBiblioteca() {

collideRectCircle(200, 200, 100, 150, mouseX, mouseY, 100);

}

O que significam estes parâmetros?

Os quatro parâmetros iniciais se referem ao nosso retângulo, que é a raquete, os quais podem ser substituídos por xRaquete, yRaquete, raqueteComprimento e raqueteAltura, respectivamente. Os demais parâmetros têm a ver com o círculo, e os trocaremos por xBolinha, yBolinha e raio.

Para identificarmos se de fato há colisão, ou não, atribuiremos o resultado desta função à variável colidiu, e implementaremos uma condição para quando esta variável for true, a direção do movimento da bolinha se inverta.

function colisaoMinhaRaqueteBiblioteca() {

colidiu = collideRectCircle(xRaquete, yRaquete, raqueteComprimento, raqueteAltura, xBolinha, yBolinha, raio);

if (colidiu) {

velocidadeXBolinha \*= -1;

}

}

Ao testarmos, obteremos um erro, e de acordo com o console, collideRectCircle não está definido. Acontece que não referenciamos o novo arquivo em nosso código! Faremos isto em index.html, entre as tags de abertura e fechamento de <body>:

<body>

<script src="sketch.js"></script>

<script src="p5.collide2d.js"></script>

</body>

Desta vez, ao testarmos, teremos o resultado que gostaríamos. Então, quando precisamos de uma solução, poderemos nos perguntar se ela não existe, se alguém já não passou por isso. Muito provavelmente a resposta será sim, e podemos pesquisar por muitas alternativas na internet.

Estamos utilizando a solução de outra pessoa, e quando pressionamos o botão de play, o console exibe ### p5.collide ###. Isto não foi escrito por nós, sendo proveniente do código que baixamos, em console.log("### p5.collide ###"). Para que nada seja exibido no console, basta deletarmos toda esta linha. Isto nos mostra que, mesmo quando pegamos uma possível solução alheia, é possível alterá-la.

**Atividades**

**1-** Criamos o código que verifica a colisão da bolinha com a nossa raquete verificando a posição x e y de cada um deles. Porém, acessando a documentação do P5 Js, vimos que existe [uma solução disponível no GitHub](https://github.com/bmoren/p5.collide2D), para reconhecer a colisão entre objetos 2D.

Sabendo disso, analise as afirmações abaixo e marque as verdadeiras.

Parte superior do formulário



O GitHub é o único lugar da web onde é possível hospedar código.



O GitHub possui apenas soluções para projetos feitos em Javascript.

O GitHub hospeda o código de pessoas que utilizam diferentes linguagens de programação, como Java, Javascript, Ruby, entre outras.

* Alternativa correta



É possível compartilhar um código com a solução desenvolvida por nós para que outras pessoas possam usar.

Certo! Podemos hospedar nosso código no GitHub, compartilhando uma solução, ajudando assim outras pessoas da comunidade.



O GitHub permite a socialização com pessoas que possuem os mesmos interesses que você.

**AULA: 07**

**Raquete do oponenteParte inferior do formulário**

Atualmente, temos a bolinha se movimentando para várias direções, a raquete se movimentando por meio das teclas de setas, a colisão entre elas acontecendo... Mas não estamos jogando contra ninguém. Assim como fizemos no Scratch, poderemos criar um oponente, isto é, outra raquete. Em nosso código, acrescentaremos:

//variáveis do oponente

let xRaqueteOponente = 585;

let yRaqueteOponente = 150;

Também criaremos uma função similar à mostraRaquete():

function mostraRaqueteOponente() {

rect(xRaqueteOponente, yRaqueteOponente, raqueteComprimento, raqueteAltura);

}

Não podemos nos esquecer de inclui-la no draw(), em que comentaremos a linha movimentaBolinha() para deixar o jogo pausado. Reparem, porém, que as funções mostraRaqueteOponente() e mostraRaquete() são iguais, exceto pelos dois primeiros parâmetros de rect(). Será que de fato é necessário criarmos uma função para cada caso?

Vamos deletar a função recém criada, então, e alterar a mostraRaquete(), que passará a necessitar dois parâmetros. Do mesmo modo, xRaquete e yRaquete serão, simplesmente, x e y, respectivamente:

function mostraRaquete(x,y) {

rect(x, y, raqueteComprimento, raqueteAltura);

}

Então, a função draw() ficará da seguinte forma:

function draw() {

background(0);

mostraBolinha();

//movimentaBolinha();

verificaColisaoBorda();

mostraRaquete(xRaquete, yRaquete);

movimentaMinhaRaquete();

//verificaColisaoRaquete();

colisaoMinhaRaqueteBiblioteca();

mostraRaquete(xRaqueteOponente, yRaqueteOponente);

}

Assim, **reutilizamos as funções que criamos**, e temos uma função que mostra a raquete, não importa se a nossa ou a do oponente. A única diferença são os parâmetros passados a ela. Nossa raquete se movimenta verticalmente, porém não queremos que a raquete do oponente permaneça imóvel, sendo assim chamaremos a função movimentaRaqueteOponente() dentro do bloco de draw(), e a criaremos no fim do nosso código:

function movimentaRaqueteOponente() {

}

No Scratch movimentamos a raquete adversária passando a posição Y da bolinha, para que ela fosse seguida, e alterávamos esta posição. Vamos fazer o mesmo no p5: criaremos uma função para alterar a velocidade Y relacionada ao oponente, similarmente ao que temos para a movimentação da bolinha, velocidadeXBolinha e velocidadeYBolinha.

//variáveis do oponente

let xRaqueteOponente = 585;

let yRaqueteOponente = 150;

let velocidadeYOponente;

Manipularemos o valor de velocidadeYOponente em movimentaRaqueteOponente(), que ficará da seguinte forma:

function movimentaRaqueteOponente() {

velocidadeYOponente = yBolinha - yRaqueteOponente - raqueteComprimento / 2 - 30;

yRaqueteOponente += velocidadeYOponente

}

Deste modo, indicaremos que velocidadeYOponente receberá a posição Y da bolinha, subtraída da posição Y da raquete do oponente e a altura da raquete, para que a bolinha sempre toque em algum ponto dela. Este valor será dividido por 2 e, ainda, subtrairemos 30, mesma margem utilizada no Scratch, em que deixamos 30. Vamos testando e entendendo o que ficará melhor.

Movimentaremos a bolinha descomentando a linha movimentaBolinha() de draw(), entretanto o faremos apenas no eixo Y, ou seja, comentaremos xBolinha += velocidadeXBolinha em movimentaBolinha(). Ao pressionarmos o play, teremos que a raquete do oponente segue a bolinha.

Descomentaremos xBolinha += velocidadeXBolinha, mas, assim como já aconteceu anteriormente, a impressão é a de que a bolinha ultrapassa a raquete. Precisaremos corrigir isto, mas, antes, diminuiremos a velocidade da bolinha para 2 e movimentaremos a bolinha apenas no eixo horizontal comentando a linha yBolinha += velocidadeYBolinha, para comprovarmos que isto realmente acontece.

Significa que não temos uma colisão da bolinha com a raquete do oponente. Lidaremos com isto a seguir!

**Atividades**

**1-** Durante a construção da raquete do oponente, percebemos que não precisaríamos criar a função mostraRaqueteOponente(), pois a função mostraRaquete() já continha os parâmetros necessários. Selecione a alternativa que explica por que decidimos usar a função existente em nosso código em vez de criar uma nova.

Parte superior do formulário

* Alternativa correta



Pois assim usamos menos linhas de código, já que nossos parâmetros são muito semelhantes.

Alternativa correta. Utilizamos a mesma função. O conteúdo das funções mostraRaqueteOponente() e mostraRaquete() são iguais porque executam a mesma ação, que é desenhar um ator na tela. No entanto, os dois primeiros parâmetros de rect() são diferentes, isso ocorre para identificar o posicionamento das raquetes na tela



Decidimos utilizar uma única função para mostrar apenas uma raquete.



Porque o p5.js. possui um número limitado de funções por código.



Não é possível utilizar a mesma função para duas raquetes.

Parte inferior do formulário

**AULA: 08**

Criamos e movimentamos a raquete do oponente, porém ainda não existe uma colisão entre ela e a bolinha. No momento, temos colisaoMinhaRaqueteBiblioteca(), que duplicaremos e alteraremos:

function colisaoRaqueteOponenteBiblioteca() {

colidiu = collideRectCircle(xRaqueteOponente, yRaqueteOponente, raqueteComprimento, raqueteAltura, xBolinha, yBolinha, raio);

if (colidiu){

velocidadeXBolinha \*= -1;

}

}

Chamaremos esta função em draw(), colocando-a junto às demais, e ao pressionarmos play, agora, sim, teremos a colisão, comportamento que queremos, mas será que nosso código está escrito da melhor forma possível?

Esperamos o mesmo resultado de colisaoMinhaRaqueteBiblioteca() e colisaoRaqueteOponenteBiblioteca(), e ambas as funções são idênticas entre si. Será que precisamos de mais nove linhas de código para mudarmos dois parâmetros? Como vimos anteriormente, poderemos dar um nome genérico para a função que unirá estas duas, como verificaColisaoRaquete().

No lugar de xRaquete, passaremos simplesmente x, e da mesma forma, yRaquete ficará com y, os quais receberemos como parâmetro da função. Lembrando que x e y podem ser nomeados como preferirmos.

function verificaColisaoRaquete(x, y) {

colidiu = collideRectCircle(x, y, raqueteComprimento, raqueteAltura, xBolinha, yBolinha, raio);

if (colidiu){

velocidadeXBolinha \*= -1;

}

}

Em draw(), substituiremos as linhas colisaoRaqueteOponenteBiblioteca() por verificaColisaoRaquete(xRaqueteOponente, yRaqueteOponente) e colisaoMinhaRaqueteBiblioteca() por verificaColisaoRaquete(xRaquete, yRaquete).

Quando verificaColisaoRaquete() for executada, com os parâmetros correspondentes, teremos o mesmo comportamento de antes, o reconhecimento da colisão em ambas as raquetes. Vamos, então, voltar a velocidade da bolinha para 6, descomentar a linha verificaColisaoRaquete() em draw(), e yBolinha += velocidadeYBolinha em movimentaBolinha(), e testar o jogo. Tudo funciona conforme gostaríamos!

Atividades

1- Até o momento no jogo Pong, podemos visualizar o código abaixo dentro da função draw:

function draw() {

background(0);

mostraBolinha();

movimentaBolinha();

verificaColisaoBorda();

mostraRaquete(xRaquete, yRaquete);

movimentaMinhaRaquete();

verificaColisaoRaquete(xRaquete, yRaquete);

mostraRaquete(xRaqueteOponente, yRaqueteOponente);

movimentaRaqueteOponente();

verificaColisaoRaquete(xRaqueteOponente, yRaqueteOponente);

incluiPlacar();

marcaPonto();

}

Sabendo disso, analise as alternativas abaixo e marque aquela que mostra a função correta que está relacionada à colisão da raquete do oponente.

**Parte superior do formulário**

* **Alternativa correta**

****

**verificaColisaoRaquete(xRaqueteOponente, yRaqueteOponente)**

Alternativa correta. A função verificaColisaoRaquete(xRaqueteOponente, yRaqueteOponente) verifica se há colisão nos eixos X e Y da raquete do oponente.



movimentaRaqueteOponente()



verificaColisaoRaquete (xRaquete, yRaquete)



marcaPonto()

2- Um aluno ingressou na plataforma da Alura e foi muito bem na construção de seu jogo com Scratch, mas ainda está meio perdido durante a escrita e leitura de algumas linhas de códigos do jogo com a linguagem JavaScript. Em especial com as barras // que aparecem antes de algumas frases ou blocos de códigos.

//variáveis da raquete

let xRaquete = 5;

let yRaquete = 150;

let comprimento = 10;

let altura = 90;

//variáveis da raquete oponente

let xRaqueteOponente;

let yRaqueteOponente ;

O que será executado quando as linhas //variáveis da raquete e //variáveis da raquete oponente do oponente forem lidas?

Parte superior do formulário



Serão executadas as funções referentes às variáveis.



//variáveis da raquete irá definir os valores de X e Y.



O código irá apontar um erro pois não devemos escrever com a sintaxe incorreta.

* Alternativa correta



As primeiras linhas apontadas estão comentadas, ou seja, não são consideradas na execução do código.

Alternativa correta. Todo código que possua // em seu início, **não será executado** pelo Javascript.

Parte inferior do formulário

**AULA: 09**

**Placar do jogo**

Nosso jogo está bem interessante, mas está faltando algo — saber quem venceu o jogo. No Scratch, inserimos um placar para quando a bolinha tocar na lateral da tela, sem o intermédio da raquete. Além disso, criamos duas variáveis, meus pontos e pontos do oponente, em que armazenamos os pontos de cada jogador, e queremos fazer algo similar no p5.

//placar do jogo

let meusPontos = 0;

let pontosDoOponente = 0;

Então, precisaremos desenhar e exibir estes valores na tela. Para deixarmos nosso projeto organizado, assim como temos feito, criaremos a função incluiPlacar(), que também chamaremos em draw(). Nela, incluiremos text(), que exige como parâmetros o conteúdo da variável meusPontos, e as posições X e Y de onde esta pontuação será exibida na tela.

Para ganharmos tempo, poderemos colocar uma posição já testada anteriormente, 218 no eixo horizontal e 26 no eixo vertical. Vamos pausar o movimento da bolinha comentando em movimentaBolinha() para testarmos. Os pontos ainda não estão sendo exibidos...

Na verdade, eles estão sendo exibidos, porém na cor preta, mesma cor do nosso fundo. Precisaremos pintar a pontuação, por meio de fill().

function incluiPlacar() {

fill(255);

text(meusPontos, 278, 26);

text(pontosDoOponente, 321, 26);

}

Para testarmos a implementação, comentaremos o movimento da raquete do oponente na função draw(), movimentaRaqueteOponente(), e descomentaremos movimentaBolinha(). Os pontos não estão sendo contabilizados, e queremos que quando a bolinha tocar a borda direita, nós marquemos um ponto, e quando ela tocar a borda esquerda, o oponente marque um ponto.

No Scratch lidamos com a posição que corresponde quase ao limite da tela e, ao alcançarmos um número maior que ele, marcamos um ponto. Sabendo que a largura total da nossa tela é de 600, quando o jogo se inicia, a extrema borda lateral esquerda corresponde a 0, enquanto a extrema borda lateral direita corresponde a 600, nosso limite.

Tiraremos um pouco deste valor, por exemplo, 590. Assim, se o X da bolinha for maior que este valor, marcaremos um ponto. Para isso, criaremos a função marcaPonto(), que não podemos deixar de chamar em draw().

function marcaPonto() {

if (xBolinha > 590) {

meusPontos += 1;

}

if (xBolinha < 10) {

pontosDoOponente += 1;

}

}

Em draw(), manteremos apenas verificaColisaoRaquete() comentado e testaremos o jogo, que funciona corretamente.

No momento, temos a função incluiPlacar(), que contém dois text() preenchidos na cor branca. Pensando em alguma forma de reutilizarmos o código, uma opção seria incluir pontos, x e y como parâmetros de incluiPlacar(), e em draw(), quando ela é chamada, passarmos os parâmetros necessários.

Entretanto, será que de fato esta refatoração é imprescindível? Da maneira como está, o código não utiliza poucas linhas para fazer uma determinada ação de forma mais clara possível?

Estes pontos são sempre importantes de serem analisados e discutidos durante a criação de um projeto, **ficando a critério do seu time ou empresa**. O importante é levar em consideração o que pode ser feito com o código para melhorá-lo e torná-lo cada vez mais legível.

**Atividades**

**1-** Durante o curso, após movimentar a raquete do oponente de forma automática, suponhamos que um aluno ou uma aluna resolveu criar um placar do jogo, mostrando os pontos ganhos por cada raquete. Para isso, desenvolveu a seguinte função:

function marcaPonto(){

if (xBolinha > 590){

pontosDoOponente += 1;

}

if (xBolinha < 10){

meusPontos += 1;

}

}

Porém, ao executar o programa, percebeu que algo não funcionou como esperado.

Com base no código, podemos afirmar que:

Parte superior do formulário



Existe um erro de sintaxe na linha de cada código if.

* Alternativa correta



Quando eu fizer um ponto, este ponto será atribuído ao oponente.

Certo! A parede do lado do opoente tem o valor x igual a 600. Sendo assim, caso a primeira condição seja verdadeira, é somado a variável pontosDoOponente 1.



Os pontos não sofrerão alteração quando a bolinha tocar a parede da minha raquete ou do oponente

Parte inferior do formulário

**AULA: 10**

**Editando o placar do jogo**

O placar do nosso jogo não está visualmente agradável, pequeno demais. Para editá-lo sem nos distrairmos, pausaremos o movimento da bolinha comentando movimentaBolinha(), definiremos um tamanho por meio de textSize(), centralizaremos os textos com textAlign().

No Scratch, as pontuações estão envoltas em retângulos laranjas com cantos arredondados e borda branca, além de estarem um pouco mais afastados uma da outra. Para fazermos isto, alteraremos o X do text() de ambos, assim, eles ficarão a 150 de distância das bordas. E para criarmos a caixinha laranja que envolve os números, utilizaremos a função rect() para a nossa pontuação e para a do oponente.

function incluiPlacar() {

textAlign(CENTER);

textSize(16);

fill(255);

rect(150, 10, 40, 20);

text(meusPontos, 150, 26);

rect(450, 10, 40, 20);

text(pontosDoOponente, 450, 26);

}

As caixinhas estão brancas, portanto teremos que incluir a função fill(color()) para cada rect(). Em seguida, será necessário ajustar os posicionamentos delas em relação ao eixo X, para que fiquem alinhadas com os textos contidos nelas. Por fim, no Scratch, temos uma borda branca contornando as caixinhas com as pontuações. Para aplicá-la utilizamos stroke(), na cor branca.

Conseguimos encontrar facilmente na internet os números correspondentes a cada cor, pesquisando por tabelas de cores HTML, RGB e hexadecimais.

function incluiPlacar() {

stroke(255);

textAlign(CENTER);

textSize(16);

fill(color(255, 140, 0));

rect(150, 10, 40, 20);

fill(255);

text(meusPontos, 170, 26);

fill(color(255, 140, 0));

rect(450, 10, 40, 20);

fill(255);

text(pontosDoOponente, 470, 26);

}

Porém, para que esta função não aumente o contorno de todos os componentes do jogo, isto é, da bolinha e das raquetes, diminuiremos o valor de diametro para 13. Vamos verificar o funcionamento do jogo descomentando movimentaBolinha() e comentando movimentaRaqueteOponente(). Feito o teste, descomentaremos esta função novamente.

Pronto! Agora temos o mesmo jogo feito em outra linguagem de programação!

**Adicionando sons**

O último detalhe que falta inserirmos no jogo para que ele fique de fato completo é o som! No Scratch, temos três sons principais: o da trilha sonora, das raquetadas, e o dos pontos sendo marcados. Sendo assim, vamos inseri-los no p5; na atividade anterior, estão disponíveis para download três sons que utilizaremos no projeto.

Carregaremos estes áudios da mesma forma como fizemos com p5.collide2d.js, clicando na seta para baixo ao lado de "project-folder" e em "Add file". No Scratch, os sons são acessados em uma aba específica para isso, e já se encontram na memória do nosso jogo. No p5, por enquanto apenas os carregamos, e para armazená-los criaremos algumas variáveis.

Então, precisaremos de uma função que os carregue antes do nosso jogo ser inciado. Usaremos a função preload(), dentro do qual chamaremos as variáveis, e iremos atribuir um pré-carregamento de cada som, por meio de loadSound(), que exige como parâmetro o caminho do local onde está o áudio.

//sons do jogo

let raquetada;

let ponto;

let trilha;

function preload() {

trilha = loadSound("trilha.mp3");

ponto = loadSound("ponto.mp3");

raquetada = loadSound("raquetada.mp3");

}

O som da trilha poderá ser incluído na função setup(), em que ocorre a preparação de todo o jogo:

function setup() {

createCanvas(600, 400);

trilha.play();

}

Vamos testar para verificar se tudo funciona conforme esperado. Após um período de tempo, o som para de tocar, já que solicitamos sua execução apenas uma única vez. Sendo assim, precisaremos substituir play() por loop():

function setup() {

createCanvas(600, 400);

trilha.loop();

}

Incluiremos o som da raquetada no if quando a bolinha colide com alguma das raquetes e muda a sua direção:

function verificaColisaoRaquete() {

if (xBolinha - raio < xRaquete + raqueteComprimento && yBolinha - raio < yRaquete + raqueteAltura && yBolinha + raio > yRaquete) {

velocidadeXBolinha \*= -1;

raquetada.play();

}

}

function verificaColisaoRaquete(x, y) {

colidiu = collideRectCircle(x, y, raqueteComprimento, raqueteAltura, xBolinha, yBolinha, raio);

if (colidiu) {

velocidadeXBolinha \*= -1;

raquetada.play();

}

}

Falta o som de quando marcamos ponto:

function marcaPonto() {

if (xBolinha > 590) {

meusPontos += 1;

ponto.play();

}

if (xBolinha < 10) {

pontosDoOponente += 1;

ponto.play();

}

}

Com isso, temos o jogo completo.

**Atividades**

**1-** Para deixar o jogo mais emocionante e criarmos uma atmosfera de ação para ele, escolhemos colocar uma trilha sonora, que é executada quando o jogo é iniciado.

Dentro de um jogo, são necessárias algumas ações para reproduzir um áudio ou uma trilha em formato de loop (repetidamente). Selecione as alternativas que apresentam partes necessárias nesse processo:

Parte superior do formulário

* Alternativa correta



Carregar o áudio com a trilha que queremos tocar.

Certo! Precisamos adicionar o arquivo com a música que queremos executar. Utilizando o P5 js, podemos fazer isso escolhendo a opção add file e selecionar o arquivo necessário.

* Alternativa correta



Criar a função preload() e atribuir a variável o resultado da função loadSound() com o caminho da trilha e sua extensão.

Certo! Sem essa função, não é possível carregar a trilha do jogo nem executá-la.

* Alternativa correta



Criar uma variável para armazenar a trilha em tempo de execução do nosso jogo.

Certo! Podemos usar a palavra let para criar a variável.

* **Alternativa (ERRADA)**

****

**Executar a trilha na função setup() com o comando trilha.play()**

**MultiplayerParte inferior do formulário**

Vamos conferir como está o nosso jogo?

Quando executamos, temos nossa aplicação funcionando. A nossa raquete está mexendo como esperado, mas a raquete do oponente sempre está seguindo a bolinha, porém não é o que pretendemos.

Queremos que a raquete do oponente seja controlada por outra pessoa, ou seja, que a nossa raquete seja movimentada com as setas **"↑"** e **"↓"**, como já acontece, mas a do oponente seja movimentada com as teclas **"W** e **"S"**. Descobriremos como fazer isso, porque assim conseguiremos jogar com mais de uma pessoa, que é o nosso propósito.

Analisando nosso código, encontraremos a function movimentaRaqueteOponente() . No nosso cálculo, ela está sempre seguindo a bolinha, mas não é o que queremos. Apagaremos o código dentro dessa função e codar um código semelhante ao movimento da nossa raquete.

Na function movimentaMinhaRaquete() , escrevemos a propriedade keyIsDown(UP\_ARROW) , informando que a seta "↑" move a nossa raquete para cima no eixo y, e keyIsDown(DOWN\_ARROW) , que move a nossa raquete para baixo quando apertamos a seta "↓". Copiaremos esse trecho do código e colar na function movimentaRaqueteOponente() para fazermos algumas alterações.

function movimentaRaqueteOponente(){

if (keyIsDown(UP\_ARROW)){

yRaqueteOponente -= 10;

}

if (keyIsDown(DOWN\_ARROW)){

yRaqueteOponente += 10;

}

}

Entretanto, não queremos movimentar o yRaquete , porque essa é a nossa raquete. Queremos movimentar a YRaqueteOponente . Agora nosso problema é descobrir o nome da tecla. Se, por exemplo, colocarmos "W" e executarmos, perceberemos que, quando apertamos a tecla "W", a raquete não se movimenta para cima, ou seja, não muda nada. Portanto, precisamos passar algum código para função keyIsDown() funcionar.

Para isso, pesquisaremos por "keydown p5 js" no navegador e clicaremos no [primeiro link](https://p5js.org/reference/#/p5/keyIsDown). Nesta página temos a documentação e um exemplo. Quando descemos a página, temos uma descrição que informa que “podemos utilizar o código da tecla que queremos alterar para verificarmos se está alterando”, informando um link para [key code](https://p5js.org/reference/" \l "/p5/keyCode" \t "_blank).

Na nova página, observamos exemplos do código que estamos utilizando, ou seja, UP\_ARROW e DOWN\_ARROW. Contudo, o que queremos descobrir é o código da tecla "W" e da tecla "S". Descendo esta página, encontramos outro link chamado [keycode.info](https://keycode.info/). Ao clicarmos, somos levados para uma página com a mensagem "Press any key to get the JavaScript event keycode", ou seja, pressione qualquer tecla para ter o código em JavaScript. Quando apertamos a tecla "W", aparece o número "87".

Vamos voltar para o nosso código e, ao invés de codarmos "W", escreveremos keyIsDown(87) . Ao executarmos o código, poderemos reparar que agora a raquete do oponente sobe ao apertarmos “W”, mas ainda não desce com a tecla "S". Então voltaremos naquela página que informa o código e apertaremos "S".

O site nos informa o número "83" e, no canto inferior direito, informa que o código se refere à "keyS", ou seja, à tecla "S". Portanto escreveremos keyIsDown(83) , ou seja, se a tecla "S" estiver pressionada para baixo, a raquete do oponente irá para baixo.

Executaremos o código e, a princípio, movimentaremos apenas a raquete do oponente para baixo e para cima, apertando **"W"** e **"S"**. Podemos perceber que o código funcionou. Ao testarmos os movimentos da raquete do jogador com as setas **"↑"** e **"↓"**, notamos que as duas raquetes estão funcionando.

Claramente não jogaremos movimentando as duas raquetes, porque não dará certo. Nem sempre teremos coordenação suficiente para isso. Contudo, dessa forma poderemos jogar com mais de uma pessoa, não deixando apenas a máquina rodando.

Se quisermos deixar a máquina rodando, basta deixarmos o código como estava, ou podemos manter a alteração e chamar alguém da família, algum amigo ou alguma amiga para jogar com a gente.

**Parte inferior do formulário**