Curso: Lógica de programação: comece em logica com o jogo Pong e Javascript

* Funções – function
  + “function setup() {
  + createCanvas(x, y);”

Se trata da configurações (“function setup() {“) como a configuração da dimensão do background – “createCanvas(x, y);” – onde “x” e “y” representam vetores do plano cartesiano.

* + “function draw() {

background(0);”

Se trata do design, como a cor do fundo “background(0);” – onde “0” representa uma cor, sendo 0 a cor mais escura (preto) e 255 a cor mais clara (branco).

* Criando uma variável
  + “let” identifica que se trata de uma variável;
  + “x” ou “y” se trata da direção do vetor no plano cartesiano – para escrever deve-se sempre utilizar a referência com letra minúscula em sequência escrever a palavra (nome da variável) iniciando com letra maiúscula. Exemplo: “xNome” ou “yNome”.
  + Também é possível utilizar somente o nome, assim não referenciando uma direção a variável. Exemplo: “nome” (não acentue as palavras).
  + A variável (“let) é utilizada para referenciar e simplificar um código.

Exemplo: “circle(300, 200, 20);” – Onde 300 se trata da posição x, 200 da posição y e 20 o diâmetro do círculo, entretanto, essas referências podem ser de difícil entendimento para situações de edição ou de diferentes usuários trabalhando em um mesmo projeto, para simplificar a interpretação utilizamos a variável – Para utilizar a variável deve-se escrever cada regra em uma linha: “let xCirculo = 300;” – “let yCirculo = 200;” – “let diametro = 20;”. Com isso a escrita do código se torna mais simples e intuitiva, resultando em: “circle(xCirculo, yCirculo, diametro)”.

* Criando movimento
  + Nova variável – “let velocidadeXCirculo = 1;” – Para simplificar a escrita cria-se uma nova variável, assim o código que seria “xCirculo + xCirculo = 1” se torna “xCirculo + xCirculo = velocidadeXCirculo;” ou para facilitar ainda mais escrevemos “xCirculo += velocidadeXCirculo;”
* Reconhecendo as bordas
  + **“if” = se**
  + Como aprendido no Scratch, os controles são feitos com funções como “se”, “sempre”, “espere até que”, etc...

Na linguagem java não é diferente, entretanto utilizamos termos em inglês como “if”, sendo assim, como queremos que a borda seja reconhecida como uma barreira devemos dizer que o círculo não deve ultrapassar as maiores distâncias do background, comprimento (x) e altura (y), ou em inglês “width” (x) e “height” (y).

Sabendo que o círculo se movimenta entre as direções x e y, e que ele não deve ultrapassar o limite interno do background de width e height, utilizamos os símbolos matemáticos de grandeza maior que “>” e menor que “<” para referenciar o limite do background.

Então utilizamos a variável de posição do círculo – “xCirculo” ou “yCirculo” – junto ao vetor referente a horizontal e/ou a vertical para dizer que a posição do círculo não deve ser maior que “<” o nível máximo do comprimento “width” ou (“||”) da altura “height”.

Exemplo: “if (xCirculo > width)” isso significa que a posição x do círculo não ultrapassara a maior distância do background, entretanto não diz nada sobre a menor distância. Para isso devemos incluir outra linha no código dizendo “ xCirculo < 0)” para dizer que a posição x do círculo não pode ultrapassar o limite 0 do vetor horizontal do background.

Assim o código fica:

“if (xCirculo > width

xCirculo < 0)”

Mas ainda não está completo, a linguagem java não reconhece a palavra “ou”, para isso devemos utilizar “||” deixando o código dessa forma:

“if (xCirculo > width ||

xCirculo < 0)”

O que significa que a posição x do círculo não pode ser maior que o comprimento máximo ou o mínimo do background.

Entretanto ainda devemos informar qual ação deve acontecer quando o círculo atingir a posição limite, para isso vamos utilizar a variável de velocidade “velocidadeXCirculo” para dizer qual será a ação que ocorrera quando a posição do círculo atingir o limite do background.

**O código de ação sempre deve estar entre chaves “{ }”**.

A ação esperada é que o círculo volte após atingir o limite, para isso devemos inverter o valor da velocidade, “velocidadeXCirculo = 6” deve se tornar “velocidadeXCirculo = -6”, para que isso ocorra vamos multiplicar a variável da velocidade por -1. Para que isso aconteça digitamos o seguinte código, “velocidadeXCirculo \*= -1”, e adiciona-lo ao restante do código, resultando em:

“if (xCirculo > width ||

xCirculo < 0){

velocidadeXCirculo =\* -1;

}”

Entretanto percebemos que a referência do círculo está em seu ponto central, sendo assim, o toque a borda é identificado pelo ponto referencial. Para corrigir criamos outra variável, está serve para informar que a validação do toque deve ser feita pela extremidade do círculo – também chamado de raio – e não pelo centro.

Para isso utilizamos de uma variável criada anteriormente, “diametro”, para informar que a borda do círculo é o diâmetro dividido por 2, ou seja, o raio.

Então devemos escrever o seguinte código:

“let raio = diametro / 2;”

Após identificarmos o ponto referencial de toque podemos finalizar o código que configura a borda e a colisão do círculo. E ele fica da seguinte forma:

“if (xCirculo + raio > width ||

xCirculo – raio < 0){

velocidadeXCiruclo += -1;”

Em fases de testes será necessário isolar partes do código para analisar o bom funcionamento de todas as funções, para isso utilizamos duas barras em sequência – // – no início do código, como por exemplo:

“if (xCirculo + raio > width ||

xCirculo – raio < 0){

velocidadeXCiruclo += -1;”

//if (yCirculo + raio > width ||

yCirculo – raio < 0){

velocidadeYCiruclo += -1;”

Perceba que o código que referencia o eixo vertical está isolado, isso faz com que ele fique inativo possibilitando testes no eixo horizontal sem a interferência do eixo perpendicular.

10/07 - – 02. Iniciando no javascript

* Criando uma função
  + Para criar uma função utilizamos a palavra “function” seguido pelo nome da função, exemplo: “function movimentoCirculo” – sempre utilizar a primeira letra minúscula e iniciar a palavra posterior com letra maiúscula, em casos que se utiliza mais de duas palavras para nomear a função, deve-se, por via de regra, iniciar com letra maiúscula as duas ou mais palavras (“function vericaçãoColisaoBorda) – essa função serve para simplificar o código e é nomeado como “refaturação”.

A função deve ser seguida de parênteses e chaves – exemplo “function movimentoCirculo(){}” – os parênteses servem para indicar que é uma função, e tudo que estiver dentro das chaves demonstra que pertence a função. Após criação deve-se trazer o código referente a ela para dentro das chaves, por exemplo:

“function movimentoCirculo(){

velocidadeXBolinha =\* -1

velocidadeYBolinha =\* -1

}”

Após configurado o código da função você deve inseri-lo a função responsável de executar o código, no nosso caso a função é “function draw(){}”, que após a inclusão fica:

“function draw(){

background(0);

movimentoCirculo()

}’

Isso pode ser realizado em todos os códigos aprendidos até aqui, deixando assim, mais fácil o entendimento e simplificando a escrita. Esse método se chama **refaturação**.

* + Além disso para classificar as variáveis, é possível criar uma linha comentada utilizando “//” para nomear a classe de variáveis, como por exemplo

“//dimensões do circulo

let xCirculo = 300;

let yCirculo = 200;

let diametro = 20;

let raio = diametro / 2;

//velocidade do circulo

let velocidadeXCirculo = 6;

let velocidadeYCirculo = 6;”

13/07 – 02. Iniciando no javascript – Entendendo a biblioteca P5

03. Criando minha raquete no jogo

* Utilizando a biblioteca para desenhar um retângulo
  + Help Reference Shape Rect()

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

Onde:

* + - “x” é a posição no eixo horizontal;
    - “y” a posição no eixo vertical;
    - “w” significa width, que traduzido é comprimento;
    - “h” significa height, que traduzido é altura;

Agora que já sabemos como criar o retângulo, vamos criar variáveis para simplificar a contrução do código. Começaremos com o posicionamento “x” e “y”, logo após, as dimensões de comprimento e altura. Vamos categoriza-los utilizando do comentário – “//” e nomeando a nosso critério. O resultado sera:

“//variáveis do retângulo

let xRetangulo = 5;

let yRetangulo = 160;

let retanguloComprimento = 5;

let retanguloAltura = 80;”

Após todas as variáveis estarem criadas vamos adicionar uma função para adicionar o código de criação do retângulo para posteriormente adiciona-la a “function draw(){}”.

“function mostraRetangulo(){

rect(xRetangulo, yRetangulo, retanguloComprimento, retângulo Altura);”

Agora já criamos o retângulo, de forma simplificada para seguir o padrão. O último passo será adicionar ao “function draw(){}”.

“function draw(){

mostraRetangulo();”

* Utilizando a biblioteca para configurar comandos no teclado
  + Help Reference Events Keyboard keyIsDown()

Após a busca nas referências e a escolha da função de evento “keyIsDown()” – que serve para verificar se alguma tecla está sendo pressionada – devemos completar o código dizendo qual tecla deve ser verificada.

Como é ideia é criar um movimento ao retângulo devemos adicionar ao código informações distinguindo as teclas, no caso, seta para cima e para baixo, para configurar este evento. Porem, devemos usar os termos em inglês, que são, “up arrow” e “down arrow”. Logo nosso código se inicia desta forma:

“If (keyIsDown(UP\_ARROW))}” || “ if (keyIsDown(DOWN\_ARROW))

O parêntese duplo serve para indicar quais funções estão ativas no código, um para o evento do teclado e um para discriminar qual tecla está sob análise.

Para manter o código padronizado vamos criar uma função para adiciona-la ao “function draw(){}” e simplificar o código com o a refaturação. Sendo assim devemos seguir a logica abaixo:

“function movimentoRetangulo(){

If (keyIsDown(UP\_ARROW)){

}

If (keyIsDown(DOWN\_ARROW)){

}

}”

E completa-lo com a ação a ser executada após a verificação do toque. Utilizaremos a variável “yRetangulo” como base e vamos adicionar ou subtrair um valor, que será 10, para movimentar o retângulo.

“function movimentoRetangulo(){

If (keyIsDown(UP\_ARROW)){

yRaquete -= 10;

}

If (keyIsDown(DOWN\_ARROW)){

YRaquete += 10;

}

}”

Isso vai fazer com que o retângulo de mova para mais próximo e mais distante do ponto 0 do eixo cartesiano. Agora devemos adicionar ao “function draw(){}” a configuração de movimento.

“function draw(){

movimentoRetangulo();”

14/07 – 03. Criando minha raquete no jogo

* Colisão com o retângulo
  + Inicialmente devemos criar uma função, nomear ela e adiciona-la a “function draw”, o processo sera bem parecido com o que ocorreu na verificação de colisão com a borda. Neste caso será “verificaçãoColisaoRetangulo”, e o conteúdo adicionado deve distinguir o ponto do círculo a ser verificado, a posição e tamanho do retângulo, além, da mudança de direção após o contato. Para isso usaremos os seguintes comandos:

“function verificaColisaoCirculo() {

If (xCirculo – raio < xRetangulo + retanguloComprimento) {

velocidadeXCirculo \*= -1;

}

}”

Feita a configuração para verificação de contato, ocorre um efeito colateral onde cria-se uma barreira entre o retângulo e a borda. Este faz com que o contato do círculo se limite a projeção vertical do retângulo, para corrigir e fazer com que o círculo volte a tocar a borda devemos adicionar ao código um complemento que não altere a verificação de contato com o retângulo e manha a reação de mudança de direção, porém, quando não acontecer o contato o círculo possa ultrapassar a projeção vertical do retângulo e tocar a borda, para então ocorrer a mudança de direção.

Para isso devemos configurar dois cenários, quando o círculo está acima e abaixo do retângulo, usaremos a posição vertical do retângulo como referência adicionando a ele situações matemáticas como “< e >” – maior e menor que – lembrando de referenciar a extremidade do círculo como ponto base.

“function verificaColisaoCirculo() {

If (xCirculo – raio < xRetangulo + retanguloComprimento

&& yCirculo – raio < yRetangulo + retanguloAltura

&& yCirculo + raio > yRetangulo) {

velocidadeXCirculo \*= -1;

}

}”

&& o significado desta sintaxe é “e”, por exemplo, verificar contato com retângulo e verificar se o círculo está tocando qualquer altura da face do retângulo, se estiver acima desta altura, ultrapassar a projeção vertical do retângulo e tocar a borda e verificar se o círculo está tocando qualquer altura da face do retângulo, se estiver abaixo desta altura, ultrapassar a projeção vertical do retângulo e tocar a borda.

* Importando outra biblioteca – GitHub
  + Para buscar ajuda de terceiros – importar uma solução – devemos buscar e selecionar a opção de ajuda *(”help”)*, acessar as referências *(“reference”)*, a biblioteca *(“libraries”)* e buscar pela solução desejada.

Neste caso a busca é por uma colisão entre formas geométricas 2D, facilmente encontrada, é intitulada como p5.collide2d. Ao selecionar a biblioteca acontece um redirecionamento para o site github.com – GitHub é um site onde as pessoas compartilham soluções de programação, projetos pessoais e profissionais – onde encontramos o código em questão.

Após localizar e selecionar a solução devemos fazer o download a partir do GitHub para realizar o upload para o projeto. O download nos traz várias opções de arquivos, devemos selecionar o arquivo “.js”, neste caso é nomeado como “p5.collide2d.js”.

Para importar o arquivo ao projeto basta localizar a seta apontando para direita – acima dos números de identificação das linhas – clicando nela abre uma “pasta” onde ficam todos os arquivos que fazem parte do projeto. Agora clicamos na seta para baixo e faremos o upload do arquivo.

Importado o arquivo buscamos no GitHub a parte do código onde se encontra a solução que queremos, neste caso a opção é “collideRectCircle()”. O qual está transcrito abaixo:

“var hit = false;

function draw() {

background(255);

rect(200, 200, 100, 150);

circle(mouseX, mouseY, 100);

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

// Use vectors as input:

// const mouse = createVector(mouseX, mouseY);

// const rect\_start = createVector(200, 200);

// const rect\_size = createVector(100, 150);

// const radius = 100;

// hit = collideRectCircleVector(rect\_start, rect\_size, mouse, radius);

stroke(hit ? color('red') : 0);

print('colliding?', hit);

}”

A palavra “var” tem a mesma função que “let”, nada mais é que uma variável do código.

Após importar o arquivo e buscar o código de referência vamos configurar dentro do nosso projeto, para isso o primeiro passo é criar uma variável como no código acima;

“let colisão = false;”

O segundo passo é adicionar a variável a “function draw” utilizando o termo “colisaoRetanguloBiblioteca” (ou outro de sua preferência), vamos comentar – “//” – a solução que fizemos para que o P5 reconheça somente a função importada.

Agora é preciso criar uma função individual para configurar a que acabamos de adicionar a function draw. Para isso vamos copiar parte do código encontrado no GitHub e descrito acima;

“ hit = collideRectCircle (200, 200, 100, 150, mouseX, mouseY, 100);”

Agora basta adicionar nossos parâmetros a esse código transcrevendo da seguinte forma;

“function colisaoRetanguloBiblioteca(){

CollideRectCircle (xRetangulo, yRetangulo, retanguloComprimento, retanguloAltura, xCirculo, yCirculo, raio);”

Para verificar se de fato está colidindo ou não vamos atribuir o resultado dessa função a variável “colidiu”, além disso vamos configurar o que acontece se (*“if”*) essa colisão existir. Essa segunda parte usaremos a mesma lógica utilizada na solução que criamos e está descrita no tópico **“Colisão com o retângulo**“.

“function colisaoRetanguloBiblioteca(){

Colidiu =

CollideRectCircle (xRetangulo, yRetangulo, retanguloComprimento, retanguloAltura, xCirculo, yCirculo, raio);”

If (colidiu){

VelocidadeXCiruclo =\* -1}

}”

Ao testar o jogo notamos que ocorreu um erro, para acessar mais informações clicamos sobre “console” que nos mostra a seguinte mensagem;

“Uncaught ReferenceError: collideRectCircle is not defined (sketch: line 75”

Para solucionar devemos referenciar o novo arquivo, vamos até a pasta do projeto e buscamos pela opção “index.html”, identificamos a opção “<script src=”sketch.js”></script>” na linha 13. Vamos copiar essa linha e alterar “sketch.js” para o titulo do arquivo que importamos, “p5.collide2d.js”.

* Retângulo do oponente
  + Para criar o retângulo do oponente seguimos a mesma lógica de criação.

Primeiro criamos uma variável com a posição no eixo circadiano (x = 590 e y = 160) e a velocidade.

“//variáveis da raquete

let xRetanguloOponente = 590;

let yRetanguloOponente = 160;

let velocidadeYOponente;”

Para facilitar e simplificar o código utilizaremos a função já existente para mostrar o retângulo, manipulando a função existente responsável por mostrar o retângulo adicionando novas referências;

“function mostraRetangulo(**x,y**){

rect(**x, y,** raqueteComprimento,

raqueteAltura);”

Após alterar a function mostraRetangulo devemos adicionar a function draw as variáveis do retângulo do oponente e editar o “mostraRetangulo”;

“function draw() {

background(0);

mostraCirculo();

movimentaCirculo()

verificaColisaoBorda()

mostraRetangulo(**xRetangulo, yRetangulo**);

movimentoRetangulo();

//verificaColisaoRetangulo();

colisaoRetanguloBiblioteca();

mostraRetangulo(**xRetanguloOponente, yRetanguloOponente**);

}”

O que está em destaque serve como referência para a edição feita na function mostraRetangulo.

Agora vamos configurar o movimento do retângulo do oponente criando uma nova função, para isso vamos usar a mesma lógica do Scratch, manipulando a velocidade e o movimento através de matemática, para isso vamos encontrar o centro do retângulo e manipular o ponto de destaque para que o oponente esteja propicio a errar;

“function movimentaRaqueteOponente(){

velocidadeYOponente = yCirculo - yRetanguloOponente - retanguloComprimento /2 -30;

yRetanguloOponente += velocidadeYOponente

}”

* Colisão do oponente
  + Para configurar a colisão da melhor forma vamos utilizar o código de verificação de colisão já existente, assim podemos diminuir as linhas de código executando a refaturação. Vamos renomear a função com um nome genérico, exemplo “verificaColisaoRaquete, além disso editamos o xRetangulo e yRetangulo para x e y e adicionaremos dentro do parêntese essas referências (x e y). Finalizada as alterações devemos renomear a função dentro do bloco draw e duplica-la.

“function colisaoRetanguloBiblioteca(x,y){

colidiu =

collideRectCircle(x, y, retanguloComprimento, retanguloAltura, xCirculo, yCirculo, raio);

if (colidiu){

velocidadeXCirculo \*= -1}”

O último passo consiste em referenciar a function colisaoRetanguloBiblioteca dentro do bloco draw, uma terá referencias do meu retângulo e a outra do retângulo do oponente, assim como fizemos na função mostraRetangulo;

“function draw() {

background(0);

mostraCirculo();

movimentaCirculo()

verificaColisaoBorda()

mostraRetangulo(xRetangulo, yRetangulo);

movimentoRetangulo();

//verificaColisaoRetangulo();

colisaoRetanguloBiblioteca();

mostraRetangulo(xRetanguloOponente, yRetanguloOponente);

colisaoRetanguloBiblioteca(**xRaquete, yRaquete**);

colisaoRetanguloBiblioteca(**xRaqueteOponente, yRaqueteOponente**);

}”

* Placar do jogo
  + O primeiro passo é criar uma variável dos meus pontos e dos pontos do adversário, exemplo;

“//variável placar

let meusPonto = 0;

let pontosOponente = 0;”

Após a criar as variáveis vamos criar uma função para incluir o placar, exemplo “Funcion placar”.

Depois vamos adicionar o placar através de uma ferramenta nomeado “text”, referenciando a variável e a posição no plano cartesiano.

“function placar(){

text(meusPonto, 278,26);

text(pontosOponente, 321,26);”

Percebemos que não é possível visualizar o placar, pois ele esta na mesma cor que o fundo. Para alterar a cor do placar adicionamos a function a ferramenta “fill”, referenciamos com o valor RGB de preferência. Nesse caso vamos utilizar o valor 255 referente a cor branca.

“function placar(){

fill(255);

text(meusPonto, 278,26);

text(pontosOponente, 321,26);”

Agora é necessário criar as referências para que os pontos sejam computados, para isso vamos criar uma nova função, exemplo “function marcaPonto” – lembre-se de sempre adicionar as funções ao bloco draw – que deve verificar se o ponto xCírculo está quase tocando a borda do background (600x400), então caso a verificação identifique que o xCirculo chegou a este ponto devemos dizer que é adicionado a variável “meusPontos” ou “pontosOponente” 1 ponto.

“function marcaPonto(){

if (xCirculo > 590){

meusPontos =+1;

}

if (xCirculo < 10){

pontosOponente =+1;

}

}”

* Editando o placar do jogo
  + Ferramenta “testSize” para aumentar o tamanho do texto, exemplo “textSize (16);”.

Para alinhar o texto podemos usar a ferramenta “textAlign”, e dizer a ela onde deve ser feito o alinhamento, exemplo “textAling (CENTER);”, as letras são maiúsculas por se tratar de uma variável constante.

Para criar o retângulo utilizar a mesma ferramenta utilizada no retângulo da raquete, “rect (x, y, largura, comprimento)”, exemplo “rect (150, 10, 40, 20);. Além disso vamos pintar o retângulo, para isso deve reposicionar a ordem do código para pintar o texto e o retângulo de cores distintas. Para configurar a tonalidade devemos pesquisar qual o seu valor RGB, e adicionar a ferramenta “fill” a opção “color”, exemplo “fill (Color ( 0, 0, 0));”. O código fica da seguinte forma

“function placar(){

textAlign (CENTER);

textSize (16);

fill (color (255,140,0)); – Cor do retângulo

rect (150, 10, 40, 20); – Retângulo

fill (255); – Cor do texto

text (meusPontos, 170,26); – Texto

fill (color (255,140,0)); – Cor do retângulo

rect (450, 10, 40, 20); – Retângulo

fill (255); – Cor do texto

text (pontosOponente, 470,26); – Texto

}”

Para ficar igual ao criado no Scratch está faltando um detalhe, o contorno. Para cria-lo vamos utilizar a ferramenta “stroke”, ela criara uma borda em todos os elementos do projeto, por isso após configurada devemos diminuir o tamanho do círculo para que volte a sua proporção. Além da ferramenta também devemos dizer de que cor será o contorno, sempre em RGB, neste caso será na cor branca, então usaremos o valor de 255. O código finalizado fica assim;

“function placar(){

stroke (255);

textAlign (CENTER);

textSize (16);

fill (color (255,140,0)); – Cor do retângulo

rect (150, 10, 40, 20); – Retângulo

fill (255); – Cor do texto

text (meusPontos, 170,26); – Texto

fill (color (255,140,0)); – Cor do retângulo

rect (450, 10, 40, 20); – Retângulo

fill (255); – Cor do texto

text (pontosOponente, 470,26); – Texto

}”

* Adicionando sons
  + Primeiro devemos fazer o upload dos sons, o processo é o mesmo feito para inserir a biblioteca colide2d.

Após inserir os sons criamos variáveis identificando cada um deles, desta forma;

“//sons do jogo

let raquetada

let ponto

let trilha”

É necessário criar uma função com nome de “preload” para dizer onde encontrar os sons identificados nas variáveis, e para isso utilizamos a ferramenta “loadSound”;

“function preload(){

trilha = loadSound("trilha.mp3");

ponto = loadSound ("ponto.mp3");

raquetada = loadSound ("raquetada.mp3");”

E por último vamos adicionar os sons aos momentos que devem ser tocados. A trilha deve tocar durante todo o jogo, então adicionaremos ela ao bloco “setup” como um loop;

“function setup() {

createCanvas(600, 400);

trilha.loop();”

Os demais sons serão tocados ao fazer um ponto e no contato do círculo com o retângulo. Então adicionaremos as funções “marcaPonto” e “colisaoRetanguloBiblioteca” respectivamente;

“function marcaPonto(){

if (xCirculo > 590){

meusPontos +=1;

ponto.play()

}

if (xCirculo < 10){

pontosOponente +=1;

ponto.play()”

“

function colisaoRetanguloBiblioteca(x,y){

colidiu =

collideRectCircle(x, y, retanguloComprimento, retanguloAltura, xCirculo, yCirculo, raio);

if (colidiu){

velocidadeXCirculo \*= -1}

raquetada.play()”

* Multiplayer
  + Editar a função “movimentaRaqueteOponente” excluindo o código onde existia o calculo para o retângulo seguir o circulo. Na edição devemos adicionar o mesmo código que faz o meu retângulo movimenta;

“function movimentoRetanguloOponente(){

if (keyIsDown(UP\_ARROW)){

yRetanguloOponente -= 10;

}

if (keyIsDown(DOWN\_ARROW)){

yRetanguloOponente += 10;

}

}”

A intenção não é movimentar o retângulo do oponente pelas setas, e não funcionaria se simplesmente adicionássemos a letra “w”, então, pesquise no google “keydown p5 js” e clique no primeiro link, encontre a lista de variáveis do “keyCode” e entre no link, depois disso localize o link “keycode.info” então o acesse. Isso redireciona a uma nova pagina, onde você simplesmente preciona a tecla a qual deseja descobrir o código, nesse caso, as teclas são “w” e “s”, seus códigos são respectivamente, 87 e 83. Agora é só editar o código adicionando essa referência.

“function movimentoRetanguloOponente(){

if (keyIsDown(87)){

yRetanguloOponente -= 10;

}

if (keyIsDown(83)){

yRetanguloOponente += 10;

}

}”