

Introdução

Para o meu projeto, escolhi fazer um jogo baseado em **Space Invaders(1978)**, projetado por Tomohiro Nishikado.

Isto é, um jogo com as seguintes características:

- ☒ Diferentes inimigos posicionados na porção superior da tela.
- ☒ Um atirador(controlado pelo jogador) na porção inferior da tela.
- ☒ Os inimigos atiram aleatoriamente entre si.
- ☒ Os inimigos movem-se horizontalmente; ao alcançarem as paredes do canvas, descem de posição e aumentam a velocidade.
- ☒ Existem barreiras próximas ao jogador que são destruídas ao receberem tiros.
- ☒ O jogo é ganho quando todos os inimigos forem derrotados.
- ☒ O jogo é perdido quando o jogador recebe 1(um) tiro.

Dito isso, apenas não implementei duas coisas: a destruição das barreiras e o ato de determinado inimigo atirar somente quando o inimigo abaixo dele for abatido. Com isso, a dificuldade do jogo acabou ficando com o mesmo balanço do original, visto que todos os inimigos atiram e as barreiras permanecem protegendo o jogador. Ademais, fiz com que o jogador perdesse o jogo quando os inimigos alcançassem as barreiras. Acredito que tais especificações mantiveram uma dificuldade justa no jogo.

Estilização

Para os primeiros passos, eu decidi começar com os visuais do jogo. Eu queria que a temática fosse animais marinhos - o **Seashore Invaders**. E por estar me baseando em um jogo de arcade, decidi que tudo seria feito em pixels, por volta de 32x32. Para isso, utilizei a plataforma online [Pixilart](https://pixilart.com/). Para o personagem principal, escolhi uma tartaruga, um dos animais marinhos mais amados pelo público geral. Para os inimigos, escolhi os principais predadores das tartarugas marinhas: orcas e tubarões-tigres. Outro predador natural seriam os polvos, mas preferi desenhar lulas no lugar. Após os sprites dos animais, fiz tiros personalizados para cada um, além da criação das barreiras como um recife de corais.

Sprites para a tartaruga:



Sprites para a lula:



Sprites para a orca:



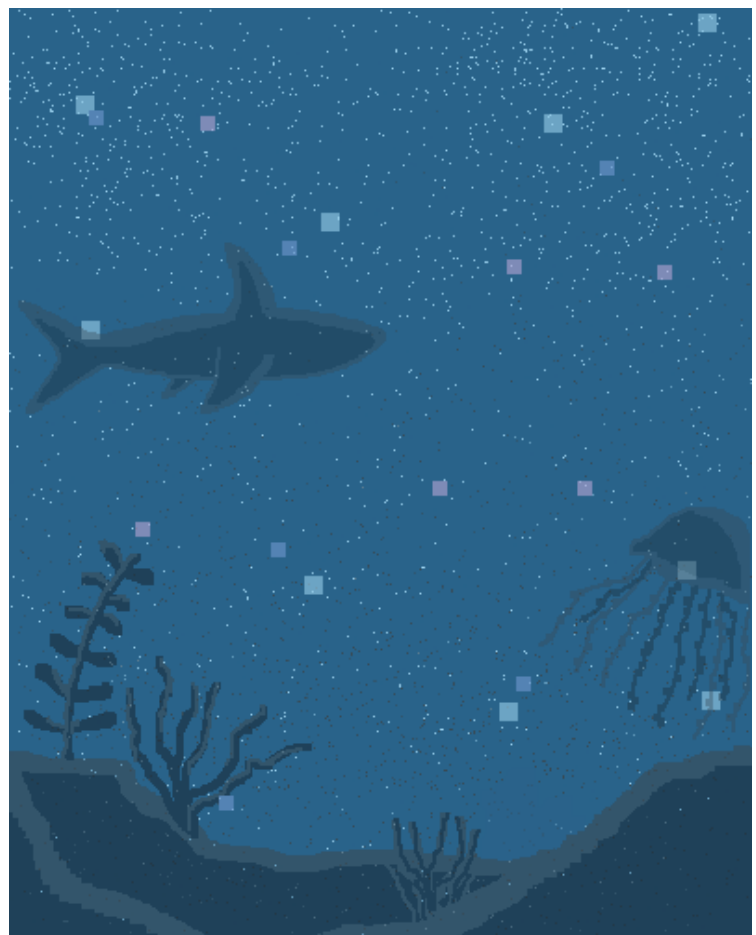
Sprites para o tubarão:



Coral:



Quanto ao fundo, desenhei um ambiente no fundo do mar 400x500 (tamanho do canvas)



Projeto Individual - Matemática Aplicada a Multimídia
Ana Carolina Furtado - 566030

Em seguida, criei as telas: menu, instruções, Game Over e Win. Para tal, utilizei o [Canva](https://www.canva.com/), com a fonte Press Start 2P e as cores #ff0000 e #efe01e. As imagens foram salvas com fundo transparente, visto que o fundo padrão já foi setado como background.



Tartaruga

Após finalizar toda a parte visual, comecei pela tartaruga. Fiz os movimentos dela serem controlados pelas setas do teclado e, de acordo com sua posição, o sprite mudaria. Quando estivesse andando, seriam os sprites para esquerda ou direita, enquanto, quando estivesse parada ou atirando, seria o sprite visto de cima. Ela se move com o método `keysDown` e tem velocidade 2.

Essa é a classe da **Tartaruga**:

```
class Tartaruga {
    constructor(x, y) {
        this.x = x;
        this.y = y;
        this.vel = 2;
    }
}
```

Esse é um método da classe **Player**:

```
mover() {
    if (keyIsDown(37) && tartaruga.x > -10) {
        image(tartarugaEsq, tartaruga.x, tartaruga.y);
        tartaruga.x -= tartaruga.vel;
    } else if (keyIsDown(39) && tartaruga.x < 340) {
        image(tartarugaDir, tartaruga.x, tartaruga.y);
        tartaruga.x += tartaruga.vel;
    } else if (keyIsDown(32) && !player.estaAtirando) {
        image(tartarugaCima, tartaruga.x, tartaruga.y, 70, 45);
        this.estaAtirando = true;
        this.tiro.y = tartaruga.y - 20;
        this.tiro.x = tartaruga.x + 32;
    } else {
        image(tartarugaCima, tartaruga.x, tartaruga.y, 70, 45);
    }
}
```



O tiro da tartaruga tem **velocidade -4** (negativa pois deve diminuir a posição y do elemento, visto que o tiro vai para cima). O tiro é criado a partir da classe **Tiro**, que ainda mostrarei.

```
class Tiro {
    constructor(pers, x, y, vel) {
        this.pers = pers;
        this.x = x;
        this.y = y;
        this.vel = vel;
        this.colidiu = false;
    }
}
```

Ainda na classe **Player**, é chamado o método **atirar()**, no qual a velocidade do tiro é somada à sua posição y e são checadas as colisões. Ao colidir, o tiro troca de sprite e tem sua posição reiniciada, bem como o **this.estaAtirando** é definido como **false** novamente.

```
constructor(tiro) {
    this.estaAtirando = false;
    this.tiro = tiro;
}

atirar() {
    this.tiro.y += this.tiro.vel;
    if (this.tiro.colidiu) {
        image(tartarugaTiroCol, this.tiro.x - 15, this.tiro.y - 15);
    } else {
        image(tartarugaTiro, this.tiro.x, this.tiro.y);
    }

    // Colisões barreiras e teto
    for (let i = 0; i < barreiras.length; i++) {
        if (
            this.tiro.y < 10 ||
            (this.tiro.y <= barreiras[i].y + barreiras[i].w &&
             this.tiro.x >= barreiras[i].x &&
             this.tiro.x <= barreiras[i].x + barreiras[i].w)
        ) {
            this.tiro.colidiu = true;
            this.estaAtirando = false;
            this.tiro.y = tartaruga.y - 20;
            this.tiro.x = tartaruga.x + 32;

            return this;
        } else {
            this.tiro.colidiu = false;
        }
    }
}
```

Dessa forma, para permitir o funcionamento da tartaruga, é chamada a seguinte função, **iniciarPlayer()**, no **setup()**:

```
function iniciarPlayer() {
    tartaruga = new Tartaruga(width / 2 - 30, height - 60);
    tiroPlayer = new Tiro(tartaruga, tartaruga.x + 32, tartaruga.y - 20, -4);
    player = new Player(tiroPlayer);
}
```


O jogo possui os estados: menu, instrucoes, instrucoes2, jogando, gameOver e win. O método **player.atirar** só será chamado quando o estado for jogando no **draw()**, ou seja, depois do jogo ser iniciado a partir do botão da tela inicial.

```
else if (estados.jogandoOn) {
    if (player.estaAtirando) player.atirar();
}
```

Barreiras


Cada barreira é individualmente gerada a partir da classe **Barreira**:

```
class Barreira {  
  constructor(x, y) {  
    this.x = x;  
    this.y = y;  
    this.w = 66;  
  }  
  
  exibir() {  
    image(coral, this.x, this.y);  
  }  
}
```



No `setup()`, é atribuída à variável global 'barreiras' a seguinte função:

```
function gerarLinhaBarreiras(x, y, qtd, espaco) {  
  const linha = [];  
  for (let i = 0; i < qtd; i++) {  
    linha.push(new Barreira(x + i * espaco, y));  
  }  
  return linha;  
}
```



```
barreiras = gerarLinhaBarreiras(10, height - 160, 4, 105);
```

O código acima cria uma linha de 4 corais, que começam no `x = 10`, e de altura `500 - 160`. O espaçamento entre eles é de 105 pixels, sendo que cada coral é 66x66. Em um loop com iteração até 4 (quantidade de corais), são colocados em um **array** objetos individuais derivados da classe `Barreira`, com posição `x` variando de acordo com o espaçamento definido.

Para que cada barreira apareça no canvas, a função a seguir é chamada no `draw()`, quando o estado for jogando:

```
function barreirasDisplay() {  
  for (let barreira of barreiras) {  
    barreira.exibir();  
  }  
}
```

Inimigos

Agora a parte que exigiu mais tempo, visto que são diferentes tipos de inimigos, diferentes tiros, com movimentação, velocidade, colisões, animações internas, etc.

Inicialmente, criei uma classe **Inimigo** que tratasse cada inimigo individualmente. A classe armazena **posição(x, y)**, tipo de inimigo, lista de sprites, **tamanho(w, h)**, método de animação interna(**MEF**), método **exibir()**, movimentação horizontal e vertical, colisão com tiros da tartaruga, e outro método **atirar()**. O código continua na página seguinte.

```
class Inimigo {
    constructor(tipo, x, y) {
        this.tipo = tipo;
        this.x = x;
        this.y = y;
        this.spritesLista = {
            tubarao: [tubarao1, tubarao2, tubarao3],
            lula: [lula1, lula2, lula3],
            orca: [orca1, orca2, orca3],
        };
        this.spriteAtual = 1;
        this.sprites = this.spritesLista[this.tipo];

        switch (tipo) {
            case "tubarao":
                this.w = 64;
                this.h = 58;
                break;
            case "orca":
                this.w = 64;
                this.h = 64;
                break;
            case "lula":
                this.w = 58;
                this.h = 60;
                break;
        }
    }

    MEF() {
        if (frameCount % 10 === 0 && this.spriteAtual === 1) {
            this.spriteAtual = 2;
        } else if (frameCount % 20 === 0 && this.spriteAtual === 2) {
            this.spriteAtual = 3;
        } else if (frameCount % 30 === 0 && this.spriteAtual === 3) {
            this.spriteAtual = 1;
        }
    }
}
```



```
exibir() {
    image(this.sprites[this.spriteAtual - 1], this.x, this.y);
    this.MEF();
}

mover() {
    this.x += velInimigos * direcaoInimigos;
}

descer() {
    if (this.y < height - 60) {
        this.y += 10;
    }
    if (this.y == tartaruga.y) {
        gameOver();
    }
}

recebeuTiro() {
    if (
        tiroPlayer.x > this.x - 5 &&
        tiroPlayer.x < this.x + this.w - 5 &&
        tiroPlayer.y < this.y + this.h
    ) {
        player.estaAtirando = false;
        tiroPlayer.y = tartaruga.y - 20;
        tiroPlayer.x = tartaruga.x + 32;
        return this;
    }
}

atirar() {
    let novoTiro = new Tiro(this.tipo, this.x + this.w / 2, this.y + this.h, 3);
    tirosInimigosArr.push(novoTiro);
}
}
```

Foi necessária a criação de variáveis globais para armazenar os tipos de inimigos separadamente, seus respectivos tiros, e velocidade e direção para alterá-las conforme necessário.

```
let tubaroes = [];
let orcas = [];
let lulas = [];
let inimigos = [];
let direcaoInimigos = 1;
let velInimigos = 0.5;
let tirosInimigosArr = [];
```


Os inimigos são iniciados com a função `iniciarInimigos()`, que chama a função `gerarLinhaInimigos()`, no `setup()`.

```
function iniciarInimigos() {
  tubaroes = gerarLinhaInimigos("tubarao", 50, 5, 4, 73);
  orcas = gerarLinhaInimigos("orca", 50, 60, 3, 110);
  lulas = gerarLinhaInimigos("lula", 50, 125, 5, 60);
  inimigos = [tubaroes, orcas, lulas];
}

function gerarLinhaInimigos(tipo, x, y, qtd, espaco) {
  const linha = [];
  for (let i = 0; i < qtd; i++) {
    linha.push(new Inimigo(tipo, x + i * espaco, y));
  }
  return linha;
}
```

O código acima segue a mesma linha de raciocínio utilizada na criação das barreiras, com o diferencial de que os arrays gerados são armazenados em outro array principal.

Em seguida, na função `inimigoDisplay()`, são manipulados os métodos da classe previamente definida. Dessa forma, são atualizados os inimigos atingidos, o `gameOver`, as colisões e a movimentação horizontal e a vertical. O código continua na próxima página.

```
function inimigoDisplay() {
  let atingiuBorda = false;
  for (let tipos of inimigos) {
    for (let inimigo of tipos) {
      // Inimigo move-se
      inimigo.mover();
      if (inimigo.x <= 0 || inimigo.x + inimigo.w >= width) {
        atingiuBorda = true;
      }

      // Inimigo aparece
      inimigo.exibir();

      // Inimigo morre
      if (inimigo.recebeuTiro()) {
        tipos.splice(tipos.indexOf(inimigo), 1);
        if (inimigos.every((tipo) => tipo.length === 0)) {
          estados.jogandoOn = false;
          estados.winOn = true;
        }
      }
    }
  }

  for (let barreira of barreiras) {
    if (
      inimigo.y + inimigo.h >= barreira.y &&
      inimigo.x + inimigo.w >= barreira.x &&

```

```
        inimigo.x <= barreira.x + barreira.w
    ) {
        estados.jogandoOn = false;
        estados.gameOverOn = true;
        return;
    }
}
}
}

if (atingiuBorda) {
    direcaoInimigos *= -1;
    velInimigos += 0.1;
    for (let tipos of inimigos) {
        for (let inimigo of tipos) {
            inimigo.descer();
        }
    }
}
}
```

Utilizei um **for of** aninhado para que, para cada inimigo de cada array dentro de **inimigos** fosse acessado devidamente, de acordo com seus métodos de classe.

Fiz outro **for of** para checar colisões barreira por barreira, visto que essas também estão armazenadas em um array principal.

Caso o inimigo receba um tiro da tartaruga, será removido do array.

Caso todos os inimigos sejam eliminados, o estado **win** é ativado.

Quanto à movimentação, defini uma variável booleana local **atingiuBorda** para confirmar se a fila de inimigos encostou nas paredes da tela, para que esses tenham sua direção invertida e desçam certa quantidade de pixels. Essa função é, por fim, chamada no **draw()**.

A respeito dos tiros dos inimigos, foram necessárias 3 funções: **sortearAtirador()** para randomizar o inimigo escolhido, **tirosInimigosDisplay()** para manipular a exibição do tiro de cada tipo de inimigo, as colisões e a interação com o array global, e **tirosInimigosVel()** para exibir os tiros de acordo com o frameCount do programa. Todas essas funções se relacionam à classe Tiro. O código continua na próxima página.

```
class Tiro {
    constructor(pers, x, y, vel) {
        this.pers = pers;
        this.x = x;
        this.y = y;
        this.vel = vel;
        this.colidiu = false;
    }
}

function sortearAtirador() {
    for (let tipos of inimigos) {
        let atirador = tipos[Math.floor(Math.random() * tipos.length)];
    }
}
```

```
    if (tipos.length !== 0) {  
        atirador.atirar();  
    }  
}  
}  
  
function tirosInimigosDisplay() {  
    for (let i = tirosInimigosArr.length - 1; i >= 0; i--) {  
        let tiroInimigo = tirosInimigosArr[i];  
        tiroInimigo.y += tiroInimigo.vel;  
  
        if (tiroInimigo.pers === "tubarao") {  
            image(tubaraoTiroImg, tiroInimigo.x, tiroInimigo.y);  
        } else if (tiroInimigo.pers === "orca") {  
            image(orcaTiroImg, tiroInimigo.x, tiroInimigo.y);  
        } else if (tiroInimigo.pers === "lula") {  
            image(lulaTiroImg, tiroInimigo.x, tiroInimigo.y);  
        }  
  
        // Colisões com barreiras  
        for (let j = 0; j < barreiras.length; j++) {  
            if (  
                tiroInimigo.y > height ||  
                (tiroInimigo.y >= barreiras[j].y &&  
                 tiroInimigo.x >= barreiras[j].x &&  
                 tiroInimigo.x <= barreiras[j].x + barreiras[j].w)  
            ) {  
                tirosInimigosArr.splice(i, 1);  
                return this;  
            }  
        }  
  
        // Colisões com tartaruga  
        if (  
            tiroInimigo.y >= tartaruga.y &&  
            tiroInimigo.x >= tartaruga.x &&  
            tiroInimigo.x <= tartaruga.x + 58  
        ) {  
            tirosInimigosArr.splice(i, 1);  
            estados.jogandoOn = false;  
            estados.gameOverOn = true;  
        }  
    }  
}  
  
function tirosInimigosVel() {  
    if (frameCount % 100 == 0) {  
        sortearAtirador();  
    }  
}
```

Dessa forma, no **for of** de **inimigos**, será sorteado aleatoriamente um inimigo de cada tipo enquanto esses existirem($!= 0$), ou seja, três devem atirar a cada 100 fps(1,4 segundo). Quando sorteados, será chamada a função **atirar()** de **Inimigo**, que criará um novo tiro e o adicionará no array principal.

No display dos tiros, acessamos cada tiro criado por meio de uma variável local **tiroInimigo**, para poder obter as propriedades(**pers, x, y**) de cada objeto, permitindo a troca de imagens para cada tipo e o teste de colisões. As funções **tirosInimigosVel()** e **tirosInimigosDisplay()** são chamadas no **draw()**.

Configurações gerais

As configurações dos estados foram feitas dessa maneira:

```
let estados = {  
  menuOn: true,  
  jogandoOn: false,  
  instrucoesOn: false,  
  instrucoes2On: false,  
  gameOverOn: false,  
  winOn: false,  
};
```

Os estados são manipulados, majoritariamente, no `draw()`

```
function draw() {  
  background(fundo);  
  if (estados.menuOn) {  
    menu();  
  } else if (estados.instrucoesOn) {  
    instrucoes();  
  } else if (estados.instrucoes2On) {  
    instrucoes2();  
  } else if (estados.jogandoOn) {  
    player.mover();  
    if (player.estaAtirando) {  
      player.atirar();  
    }  
  
    barreirasDisplay();  
    inimigoDisplay();  
    tirosInimigosVel();  
    tirosInimigosDisplay();  
  } else if (estados.gameOverOn) {  
    gameOver();  
  } else if (estados.winOn) {  
    win();  
  }  
}
```

No `setup()`, são chamados apenas os botões do menu, para que esses não sejam criados infinitamente no `draw()`. Os botões foram criados por meio de `createButton()`, sendo armazenados em um container(HTML). Estilizei-os no arquivo `style.css`.



Já para as telas das instruções, usei uma “colisão” para usar os botões das setas, que já havia criado dentro do design no Canva.

```
function instrucoes() {  
  image(instrucoesImg, 0, 0);  
  // Voltar pro menu  
  if (  
    mouseIsPressed &&  
    mouseX > 17 &&  
    mouseX < 63 &&  
    mouseY > 15 &&  
    mouseY < 50  
  ) {  
    estados.menuOn = true;  
    estados.instrucoesOn = false;  
    botoesMostrar();  
  }  
  if (  
    mouseIsPressed &&  
    mouseX > 332 &&  
    mouseX < 460 &&  
    mouseY > 445 &&  
    mouseY < 472  
  ) {  
    estados.instrucoesOn = false;  
    estados.instrucoes2On = true;  
  }  
}  
  
function instrucoes2() {  
  image(instrucoes2Img, 0, 0);  
  if (  
    mouseIsPressed &&  
    mouseX > 17 &&  
    mouseX < 63 &&  
    mouseY > 15 &&  
    mouseY < 50  
  ) {  
    estados.instrucoes2On = false;  
    estados.instrucoesOn = true;  
  }  
}
```



Para a função de reiniciar o jogo, fiz dessa maneira:

```
function reiniciarJogo() {  
  estados.gameOverOn = false;  
  estados.jogandoOn = false;  
  estados.menuOn = true;  
  botoesMostrar();  
  iniciarPlayer();  
  iniciarInimigos();  
  tirosInimigosArr = [];  
  sortearAtirador();  
  barreiras = gerarLinhaBarreiras(10, height - 160, 4, 105);  
  velInimigos = 0.5;  
}  
  
function keyPressed() {  
  if (keyCode === 82) {  
    reiniciarJogo();  
  }  
}
```

O player e os inimigos são reiniciados e os arrays de tiros e de barreiras são resetados, além da velocidade original dos inimigos. O menu é ativado e os botões reaparecem.

Conceitos matemáticos aplicados

Por fim, os conceitos de matemática aplicados nessa multimídia são:

1. **Movimento Retilíneo Uniforme(MRU)**: os tiros possuem uma velocidade fixa e retilínea, a qual é somada/subtraída ao vetor y para que o projétil percorra a tela. Os inimigos e a tartaruga também possuem velocidade e movem-se em linha reta em um dado espaço do canvas.
2. **Colisões**: o jogo inteiro baseia-se em colisões, com checagens, por exemplo, atribuídas ao x inicial, x final e y da tartaruga para saber se essa foi atingida. É testado se o tiro dos inimigos é maior que o x inicial E menor que o x final E igual ou maior que o y para que o acerto seja contabilizado. Essa lógica se repete para as barreiras, inimigos, teto, paredes, etc.
3. **Máquina de Estados Finitos(MEF)** dos inimigos: cada inimigo possui 3 sprites que se repetem em loop, gerando uma animação interna contínua - se o tempo for 10, o sprite 1 trocará para o 2; se o tempo for 20, o sprite 2 trocará para o 3; se o tempo for 30, o sprite 3 trocará para o 1 e a animação se reinicia.

Atenciosamente,
Ana Carolina Furtado

Referências

▶ Coding Challenge #5: Space Invaders in JavaScript with p5.js

▶ [Space Invaders 1978 - Arcade Gameplay](#)

▶ Programming Space Invaders in P5.js - Part 12 - boss alien

[Space Invaders – Wikipédia, a enciclopédia livre](#)

[Space Invaders Alternatives and Similar Games | AlternativeTo](#)

Bonates, Mara Franklin. (2025). *Aula 05 - Pong*. [Disciplina de Matemática Aplicada à Multimídia I, lecionada na UFC, Fortaleza]

Bonates, Mara Franklin. (2025). *Aula 06 - MEF - Apresentação*. [Disciplina de Matemática Aplicada à Multimídia I, lecionada na UFC, Fortaleza]