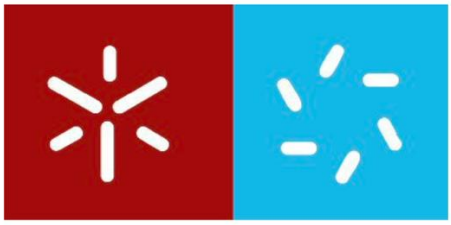


31 DE MAIO DE 2024

PROJETO
SIMULADOR DE COMPRA E VENDA
2023/2024

LICENCIATURA EM CIÊNCIAS DA COMPUTAÇÃO
UNIVERSIDADE DO MINHO

EDUARDO ANDRÉ SILVA CUNHA A98980
FÁBIO ALEXANDRE MAGALHÃES RIBEIRO A100058
GONÇALO EMANUEL FERREIRA MAGALHÃES A100084

Índice

1. Introdução

O presente relatório refere-se ao projeto desenvolvido ao longo do semestre e visa dar a conhecer e explicar a nossa metodologia e a implementação de certas ideias. O projeto consiste num simulador de compra e venda direcionado a crianças. Para a criação deste simulador, foi maioritariamente utilizado JavaScript, com o auxílio do “framework Phaser”.

O relatório está dividido em três partes principais. A primeira parte é relativa ao ficheiro "main.js", que por sua vez se divide em duas secções distintas: uma aborda a perspetiva do comprador, focando-se nas opções disponíveis ao realizar o pagamento de um determinado produto; a outra parte concentra-se na perspetiva do vendedor, analisando as opções disponíveis para fornecer o troco correto ao cliente. Seguidamente, explicamos a conexão do simulador a uma base de dados existente, fornecida pelo professor.

2. Main.js

Inicialmente, é importada a “Framework Phaser”, utilizada para o desenvolvimento deste código, assim como dois outros ficheiros relativos à ligação do simulador a uma base de dados, como veremos de seguida.

Posteriormente, definimos algumas características da página, como o tamanho do jogo e a cor de fundo. Criamos uma classe chamada “GameScene” que conterá todo o jogo. O “constructor()” e o método “preload()” desta classe carregam todas as imagens que serão utilizadas no jogo.

Iniciamos o jogo na função “create()”. Primeiramente, criamos uma variável chamada "menuInicial" que determina se estamos ou não no menu inicial. Em seguida, criamos o layout do menu, que possui o fundo de um supermercado, um menino com um carrinho que inicia a simulação de compra, um homem na caixa que inicia a simulação de venda e ainda alguns botões de login, créditos e informações. Neste layout inicial, também é calculada e atualizada a percentagem que é mostrada no topo do ecrã.

Figura 1: “Loyaut” do Menu Inicial

2.1. Simulação de Compra

Definimos a interação com o "Menino no Carrinho", configurando o botão clicável e, após o clique, passamos para outro layout que corresponde à simulação de compra. Neste layout, colocamos um produto com preço, uma carteira com algum dinheiro ao lado e uma caixa registadora vazia. O jogador tem então de arrastar as moedas para a caixa de forma a somar o valor mínimo possível e clicar em "OK" para obter o seu resultado. Após isto, tem a opção de jogar novamente ou sair da partida.

Ainda dentro da classe, para auxiliar na verificação da resposta, definimos duas funções: “calcularPagamentoMinimo” e “calcular”, para verificar efetivamente se a resposta dada pelo utilizador está correta.

```
// calcular o pagamento minimo desejado
function calcularPagamentoMinimo(precoProduto, dinheiro) {
    // Ordena as moedas pelo valor em ordem decrescente
    dinheiro.sort((a, b) => imagensValoresDinheiro[b].valor - imagensValoresDinheiro[a].valor);

    // Inicializa o pagamento mínimo como o maior valor possível
    var pagamentoMinimo = Number.MAX_SAFE_INTEGER;

    // Função recursiva para calcular o pagamento mínimo
    function calcular(pagamentoAtual, indiceMoeda) {

        // Se o pagamento atual for maior ou igual ao preço do produto, atualiza o pagamento mínimo
        if (pagamentoAtual >= precoProduto) {
            pagamentoMinimo = Math.min(pagamentoMinimo, pagamentoAtual).toFixed(2);
            return;
        }
        // Se não houver mais moedas disponíveis, retorna
        if (indiceMoeda >= dinheiro.length) {
            return;
        }
        // Tenta adicionar a moeda atual e chama a função recursivamente
        for (var i = indiceMoeda; i < dinheiro.length; i++) {
            calcular(pagamentoAtual + imagensValoresDinheiro[dinheiro[i]].valor, i + 1);
        }
    }
    // Inicia o cálculo do pagamento mínimo
    calcular(0, 0);
    // Retorna o pagamento mínimo encontrado
    return parseFloat(pagamentoMinimo);
}
```

Figura 2: Função que calcula o pagamento mínimo

Configuramos de forma que, sempre que se arrasta alguma moeda ou nota para uma zona dentro da caixa, estas são colocadas em posições específicas, possuindo também um contador para o caso de sobreposição.

Para possuir imagens aleatórias, definimos uma lista com todos os nomes das imagens e, através de um número aleatório entre 0 e o número de imagens possíveis, selecionamos a imagem. Para cada imagem possível, definimos uma posição e tamanho específicos, bem como uma gama de preços possíveis (aleatórios, entre determinados valores).
Definimos ainda um dicionário com o nome de cada imagem e com o respetivo valor em inteiro que representa, para poder utilizar como soma do valor pago.

```
var imagensValoresDinheiro = {
    "centimo1": { valor: 0.01 },
    "centimos2": { valor: 0.02 },
    "centimos5": { valor: 0.05 },
    "centimos10": { valor: 0.10 },
    "centimos20": { valor: 0.20 },
    "centimos50": { valor: 0.50 },
    "euro1": { valor: 1.00 },
    "euros2": { valor: 2.00 },
    "euros5_completos": { valor: 5.00 },
    "euros10_completos": { valor: 10.00 },
    //"euros20_completos": { valor: 20.00 }
};
```

Figura 3: dicionário que liga cada moeda ao seu próprio valor

Criamos também contadores para cada moeda para definir a sua posição inicial, sendo esta a posição definida para cada tipo de moeda mais um pouco relativamente ao número

de contadores, para que, quando há mais de uma moeda igual, pareça que estão empilhadas.

Criamos ainda uma estrutura que armazena a última moeda colocada na caixa, bem como a sua posição antiga. Assim, através da "seta para trás", podemos remover a última moeda colocada na caixa e esta regressa à sua posição de origem.

```
// Adicionar a moeda arrastada ao array de moedas arrastadas
moedasCaixa.push({
    moeda: novaMoeda,
    x: initX,
    y: initY,
    scale: originalScale
});
```

Figura 4: remoção da moeda da caixa para a sua posição na carteira

Posto isto, temos tudo para a boa execução do simulador. Sempre que uma moeda é adicionada à caixa, é somado um valor ao contador de valor pago e, cada vez que é retirada, o contador é decrementado pelo valor da moeda. Quando o jogador acha que tem a resposta correta, prime a tecla "OK" e recebe a informação de que ganhou ou se o preço está abaixo ou acima. Caso ganhe, pode ainda jogar novamente com outro valor na carteira e outro item para pagar com outro preço.

Figura 5: Acerto no "loyaut" comprador

2.2. Simulador Venda

Para a simulação de venda, inicialmente, definimos o homem da caixa como interativo. Ao clicar nele, novamente é apresentado um layout semelhante ao da simulação de compra. No entanto, nesta versão, as moedas e notas, em vez de estarem ao lado da carteira, encontram-se dentro da caixa registadora, onde também aparece o preço com que o cliente pagou. O objetivo desta simulação é arrastar para a carteira as moedas para fornecer o troco exato.

Figura 6: "Loyaut" Vendedor

Para obter o produto aleatório, mais uma vez, definimos um "array" com as imagens possíveis e, através de um índice aleatório, determinamos o artigo que vai ser vendido. Para cada imagem possível, há posições específicas e preços que variam entre determinadas gamas de valores.
É então calculado um valor aleatório de pago pelo cliente, que é igual ao valor do produto mais um valor adicional aleatório. Com este valor, além de ser calculado o troco (resultado esperado), são também escolhidas as moedas/notas que vão ser colocadas na caixa de forma a satisfazer esse valor e a possibilitar o troco correto.

```
var trocoCerto = 0.00;
var moedasEscolhidas = [];

for (var chave in moedasDisponiveis) {
    var moeda = moedasDisponiveis[chave];
    var moedaValor = moedasDisponiveis[chave].valor;

    while (moedaValor <= troco && troco > 0) {
        moedasEscolhidas.push(moeda); // Adiciona a moeda ao troco
        trocoCerto += moedaValor; // Atualiza o troco certo
        troco -= moedaValor; // Subtrai o valor da moeda do troco restante
        troco = parseFloat(troco.toFixed(2)); // Arredonda o troco para duas casas decimais
        trocoCerto = parseFloat(trocoCerto.toFixed(2));
    }
}
```

Figura 7: definição do troco correto para ser dado

A posição das moedas dentro da caixa é feita em posições aleatórias dentro da sua secção, de forma a garantir que, caso haja mais de uma moeda, todas sejam visíveis.
As moedas são arrastáveis até uma área perto da carteira e, aí, posicionadas em locais definidos, com a posição incrementada conforme a contagem de cada moeda, para dar uma sensação de empilhamento.
Para o caso de voltar atrás (premir a "seta para trás"), é novamente criada uma estrutura que armazena as moedas movidas, bem como a sua última posição.

Posto isto, se o jogador arrastar as moedas de forma a gerar o troco certo e apertar o botão "OK", recebe uma mensagem de parabéns e aparece um botão de "Jogar Novamente" caso queira jogar com outros valores e produtos.

Figura 8: Acerto no "loyaut" vendedor

3. Base de dados

3.1. loginInfo.js

Os arquivos "loginInfo.js" e "backEndconnector.js" trabalham em conjunto para gerir e sincronizar informações do utilizador com a base de dados no "backend". O "loginInfo.js" armazena localmente as informações de "login" (tais como: "user", "firstName", turma, escola) no "sessionStorage" do navegador, garantindo que o estado do login seja preservado entre sessões. Este arquivo também possui métodos para recuperar e guardar essas informações, como o "getLocalData", que verifica se o navegador suporta "sessionStorage" e tenta recuperar dados armazenados sob a chave "loginInfo" para enviar os dados guardados entre sessões. O "setLocalData" converte propriedades da classe numa string "JSON" e armazena-as no "sessionStorage". O método "logout" tem a função de redefinir as informações do utilizador como vazias e atualizar o "sessionStorage". Finalmente, o "parseData" é responsável por verificar se as chaves (user, firstName, turma e escola) existem e atualizar as propriedades da classe com esses valores, processando também os dados guardados no "sessionStorage".

```
export class loginInfo {
    /**
     * Create inicial loginInfo
     */
    constructor() {
        this.user = '';
        this.firstName = '';
        this.turma = '';
        this.escola = '';
    }

    // sessionStorage format:
    // "user": string
    // "firstName": string
    // "turma": string
    // "escola": string

    /**
     * Retrieve data saved in the browser's sessionstorage if it exists
     */
    getLocalData(){
        if(typeof(Storage) === "undefined") {
            return;
        }

        let dataAux = sessionStorage.getItem('loginInfo');
        if(dataAux != null){
            let data = JSON.parse(dataAux);
            this.parseData(data);
        }
    }
    /**
     * Set browser's sessionstorage accordingly to the current class data
     */
    setLocalData(){

        if(typeof(Storage) === "undefined") {
            return;
        }
        let storeInfo = {
            'user': this.user, 'firstName': this.firstName,
            'turma': this.turma, 'escola': this.escola};
        let info = JSON.stringify(storeInfo);
```

Figura 10: Parte 1 do arquivo loginInfo.js

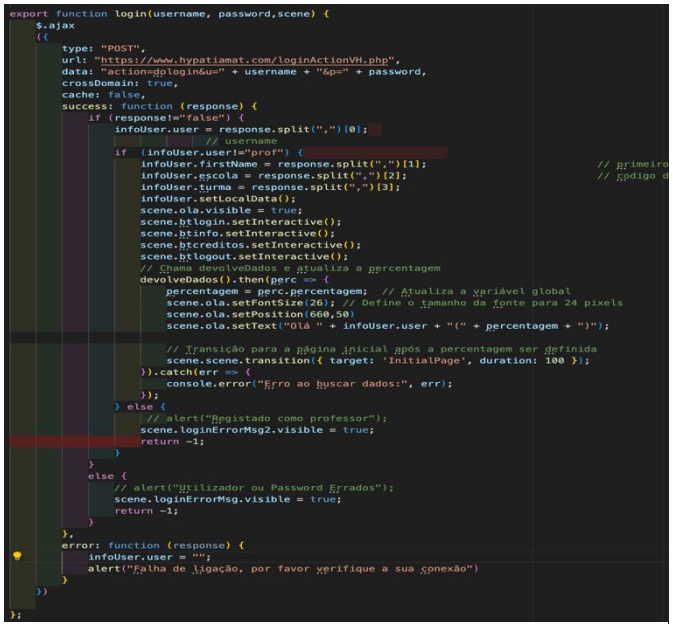
```
    /**
     * Delete user login info
     */
    logout() {
        this.user = '';
        this.firstName = '';
        this.turma = '';
        this.escola = '';
        this.setLocalData();
    }
    /**
     * Parse retriave data from browser's sessionstorage
     * @param {JSON} data - retrieved data in Json format
     */
    parseData(data) {
        if(data['user']){ // returns false if undefined/null
            this.user = data['user'];
        }
        if (data['firstName']) { // returns false if undefined/null
            this.firstName = data['firstName'];
        }
        if (data['turma']) { // returns false if undefined/null
            this.turma = data['turma'];
        }
        if (data['escola']) { // returns false if undefined/null
            this.escola = data['escola'];
        }
    }
}
```

Figura 9: Parte 2 do arquivo loginInfo.js

3.2. backEndconnecter.js

Por outro lado, o "backEndconnector.js" lida com a comunicação entre o “frontend” e o “backend”, enviando dados do utilizador e as suas atividades para o servidor. Funções como "saveScore" recebem dois parâmetros ("sc", "f"), onde "sc" pode ser "+" ou "-". No caso de ser "+", o "globalScore" é incrementado, e são chamadas as funções "saveDados" e "saveDadosTask", responsáveis por enviar diferentes tipos de dados ao "backend", garantindo assim que as atividades do utilizador sejam registadas. No caso de ser "-", são chamadas apenas as funções de salvar os dados. O parâmetro "f" indica se a simulação foi na parte vendedora ou compradora ("V" ou "D" respetivamente), para ser guardado na base de dados.

Juntos, garantem que as informações do utilizador estejam sempre disponíveis para sincronização com o "backend": o "loginInfo.js" mantém os dados de "login" atualizados localmente, enquanto o "backEndconnector.js" envia essas informações e as atividades do utilizador para a base de dados já definida, mantendo tudo sincronizado e atualizado.
É importante salientar que estes dois arquivos foram fornecidos pelo professor, e as suas funções não foram praticamente alteradas.



```
export function login(username, password,scene) {
    $.ajax
    ({
        type: "POST",
        url: "https://www.hypatiamat.com/loginActionVM.php",
        data: "action=dologin&u=" + username + "&p=" + password,
        crosDomain: true,
        cache: false,
        success: function (response) {
            if (response!="false") {
                infoUser.user = response.split(",")[0];
                                // username
                if  (infoUser.user!="prof") {
                    infoUser.firstName = response.split(",")[1];                    // primeiro
                    infoUser.escola = response.split(",")[2];                       // código d
                    infoUser.turma = response.split(",")[3];
                    infoUser.setLocalData();
                    scene.ola.visible = true;
                    scene.btlogin.setInteractive();
                    scene.btinfo.setInteractive();
                    scene.btcreditos.setInteractive();
                    scene.btlogout.setInteractive();
                    // Chama devolveDados e atualiza a percentagem
                    devolveDados().then(perc => {
                        percentagem = perc.percentagem;  // Atualiza a variável global
                        scene.ola.setFontSize(26); // Define o tamanho da fonte para 24 pixels
                        scene.ola.setPosition(660,50)
                        scene.ola.setText("Olá " + infoUser.user + "(" + percentagem + ")");

                        // Transição para a página inicial após a percentagem ser definida
                        scene.scene.transition({ target: 'InitialPage', duration: 100 });
                    }).catch(err => {
                        console.error("Erro ao buscar dados:", err);
                    });
                } else {
                    // alert("Registado como professor");
                    scene.loginErrorMsg2.visible = true;
                    return -1;
                }
            }
            else {
                // alert("Utilizador ou Password Errados");
                scene.loginErrorMsg.visible = true;
                return -1;
            }
        },
        error: function (response) {
            infoUser.user = "";
            alert("Falha de ligação, por favor verifique a sua conexão")
        }
    })
};
```

Figura 11: excerto do código backEndconnecter.js

4. Testes

Utilizador está no modo comprador:

- Teste quando o valor pago não chega para a compra do produto

Figura 12: Pagamento mínimo não atingido

- Teste quando o valor pago supera o valor necessário para pagar o produto

Figura 13: Pagamento mínimo superado

- Teste quando o valor é igual ao valor mínimo esperado para pagar o produto.

Figura 14: Pagamento mínimo atingido

Utilizador está no modo Vendedor:

- Teste quando o troco é superior ou inferior ao esperado (no caso superior)

Figura 15: Erro ao dar o troco

- Teste quando o valor do troco é o correto perante o valor de pagamento

Figura 16: Troco dado de forma correta

5. Conclusão

Em resumo, desenvolvemos um simulador de compra e venda no qual o utilizador pode escolher se deseja simular uma compra ou venda. No caso de selecionar compra, pode adquirir um determinado artigo com o objetivo de determinar o valor mínimo possível para efetuar o pagamento desse artigo, dadas as moedas que possui na carteira. Se optar pela venda, recebe o preço de um artigo e o valor pago pelo cliente, tendo de determinar o troco correto para concluir a venda.

Acreditamos ter cumprido todos os objetivos, desde o desenvolvimento do simulador até à conexão com uma base de dados existente. No entanto, consideramos que o código poderia ser mais claro com um maior conhecimento da linguagem JavaScript e um planeamento mais detalhado antes do início do desenvolvimento do jogo, tendo a noção de que o código podia ter ficado mais claro e compreensível.

Este trabalho permitiu-nos desenvolver as nossas capacidades em JavaScript, uma vez que nunca tínhamos trabalhado nem estudado esta linguagem anteriormente. Além disso, proporcionou-nos a oportunidade de explorar diversas metodologias e abordagens no âmbito do desenvolvimento de jogos educativos.