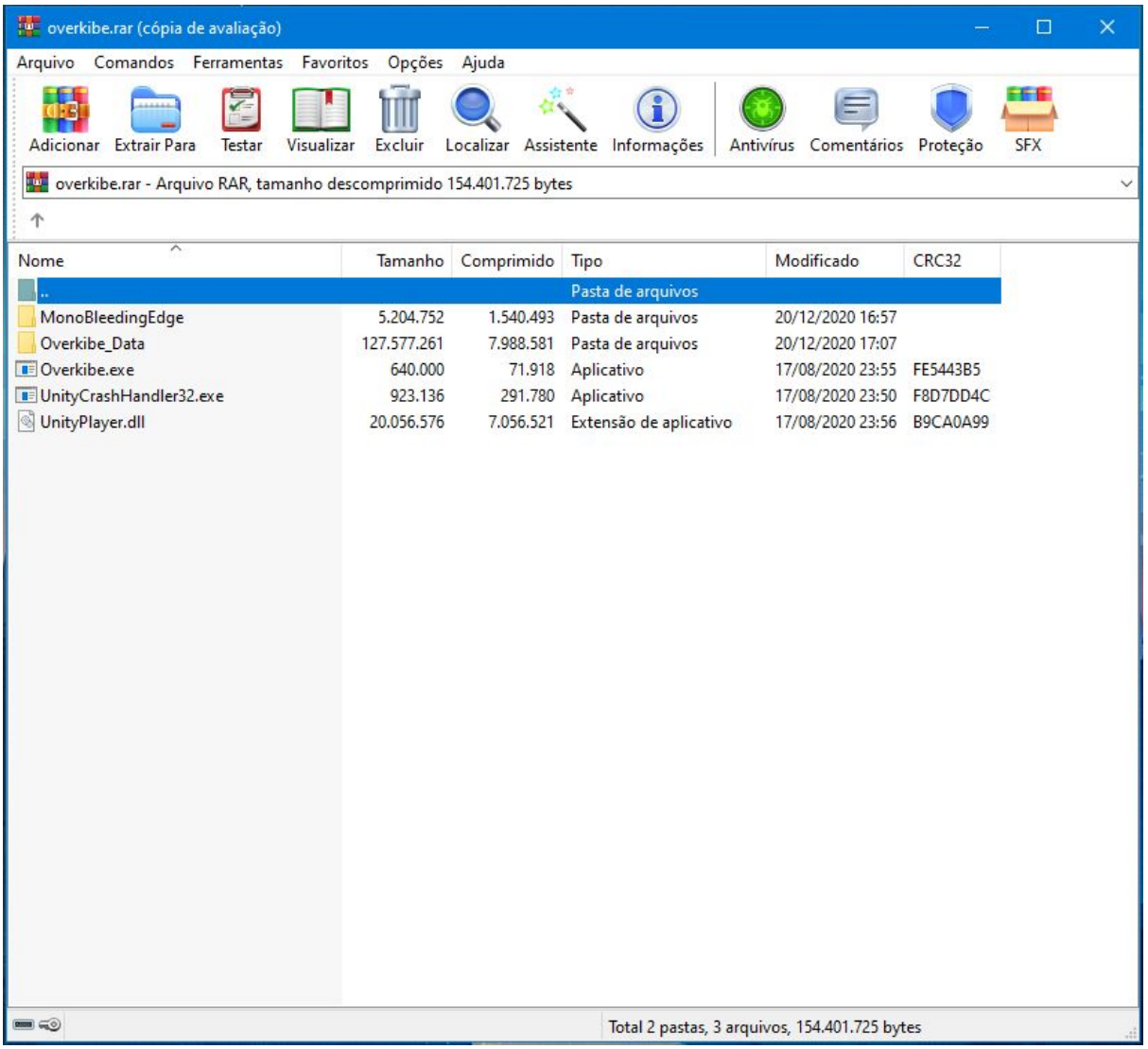
Leonardo Antonetti da Motta	11275338
Gabriel Monteiro Ferracioli	11219129
Olavo Morais Borges Pereira	11297792
Andre Santana Fernandes	11208537

Como instalar

Primeiramente, baixe o arquivo zipado que está disponibilizado e abra-o com seu descompactador preferido. No caso usamos o Winrar.



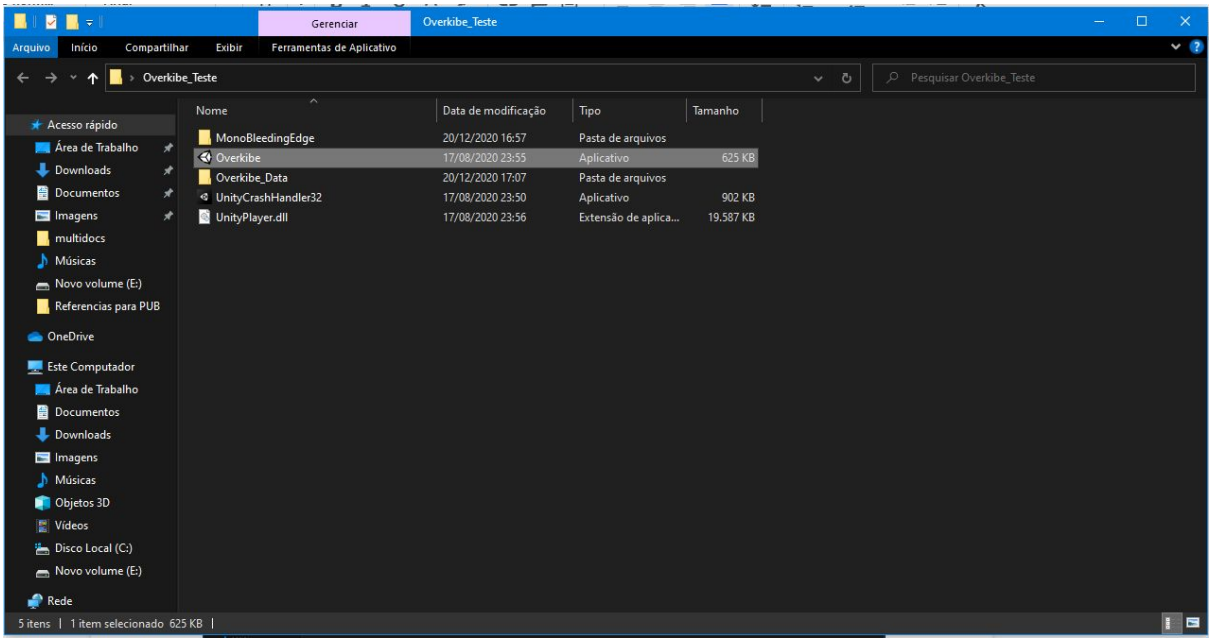
O arquivo zipado deve estar dessa forma, com os diretórios:

- MonoBleedingEdge
- Overkibe_Data

E os arquivos:

- Overkibe.exe
- UnityCrashHandler.exe
- UnityPlayer.dll

Agora descompacte os arquivos todos em um novo diretório à sua escolha. No nosso caso, descompactamos num diretório de nome Overkibe_Teste que criamos no Desktop.



Para executar o jogo, basta clicar duas vezes no executável de nome Overkibe.exe (o ícone selecionado na imagem). Isso inicializará o jogo, com uma tela splash da Unity.

Após a tela splash, o jogo é inicializado e está pronto para receber comandos do teclado e ser jogado.

Como jogar

Controles do Player 1

Identificado por iniciar no lado esquerdo da tela e ter cor mais escura.

W: Mover para cima

A: Mover para esquerda

S: Mover para baixo

D: Mover para direita

J: Interagir

Controles do Player 2

Identificado por iniciar no lado direito da tela e ter cor mais clara.

↑: Mover para cima

←: Mover para esquerda

↓: Mover para baixo

←: Mover para direita

Espaço: Interagir

Objetivo

O objetivo do jogo é simples, basta entregar o maior número de pratos corretos possíveis (o botão de interação é responsável por todas as entidades que devem ser usadas no jogo).

O loop do jogo segue conforme: receber o pedido na janela à esquerda, pegar os ingredientes conforme o pedido nos balcões da parte inferior da tela até ter todos os necessários na ordem certa, montar o prato na mesa de montagem na parte superior da tela, e entregar o prato na janela à direita.

Quem entregar o prato corretamente recebe 10 pontos, quem entregar o prato incorretamente recebe -10 pontos. Os pontos são marcados nos cantos superior esquerdo e superior direito (Player 1 e Player 2 respectivamente).

Ao terminar o tempo que está sendo contado no canto superior central da tela (atualmente dois minutos), o Player com mais pontos vence o jogo.

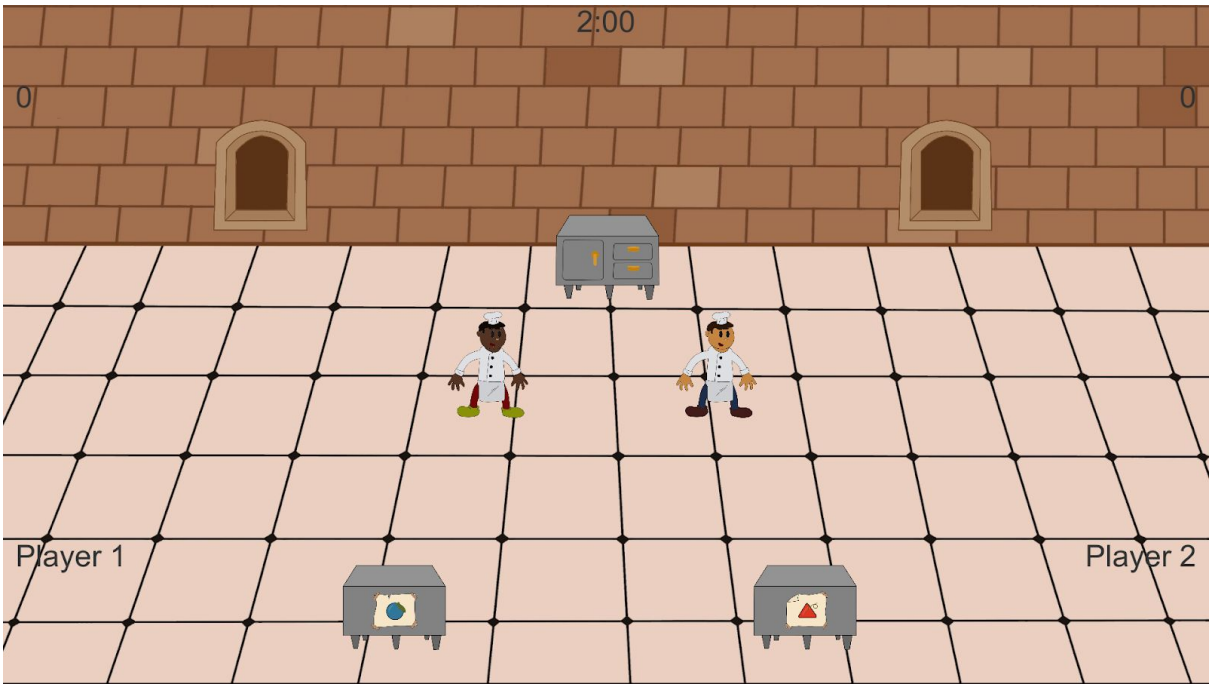
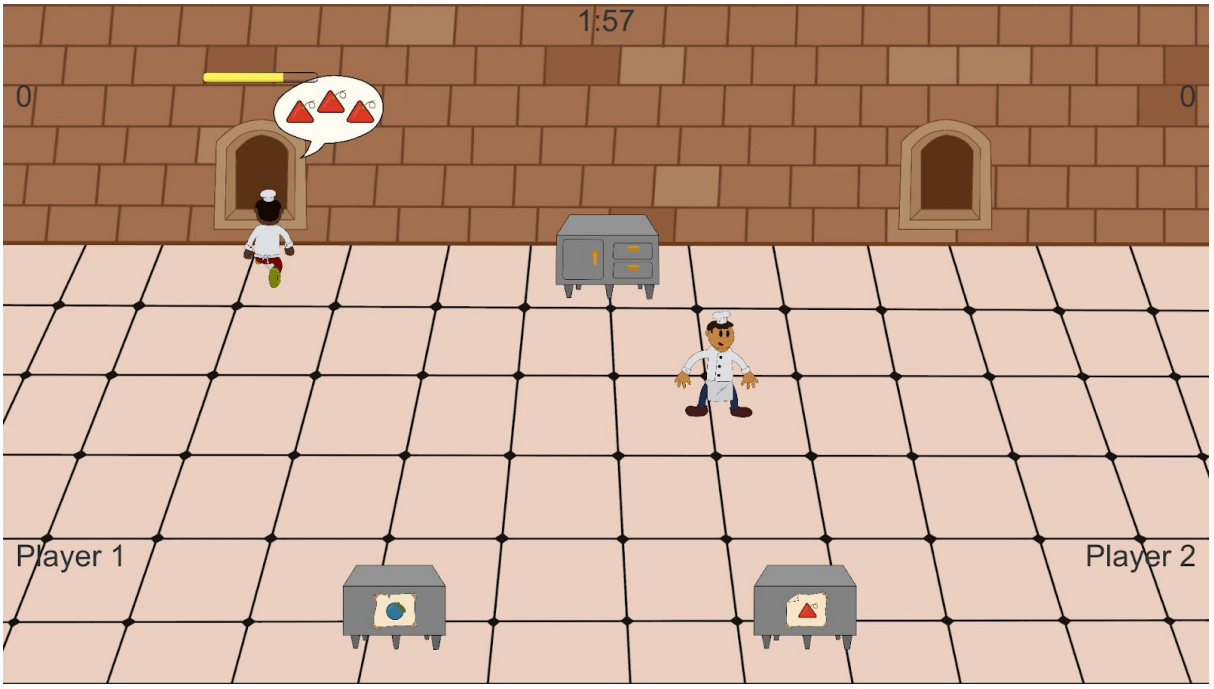
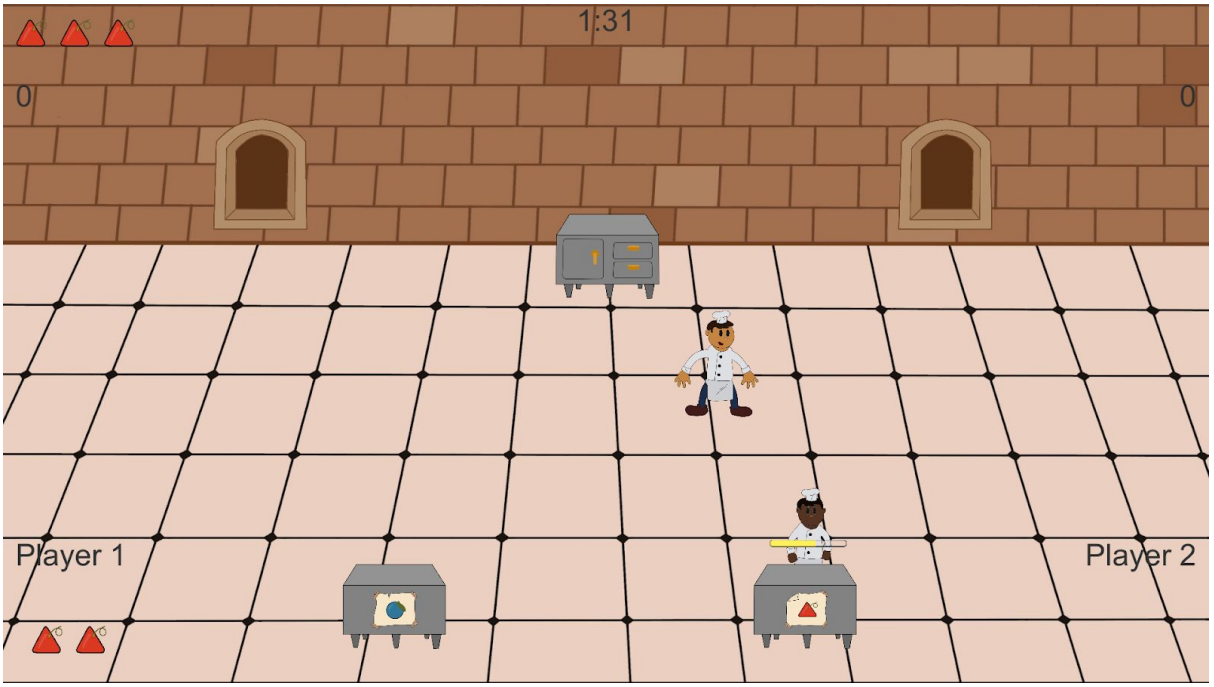


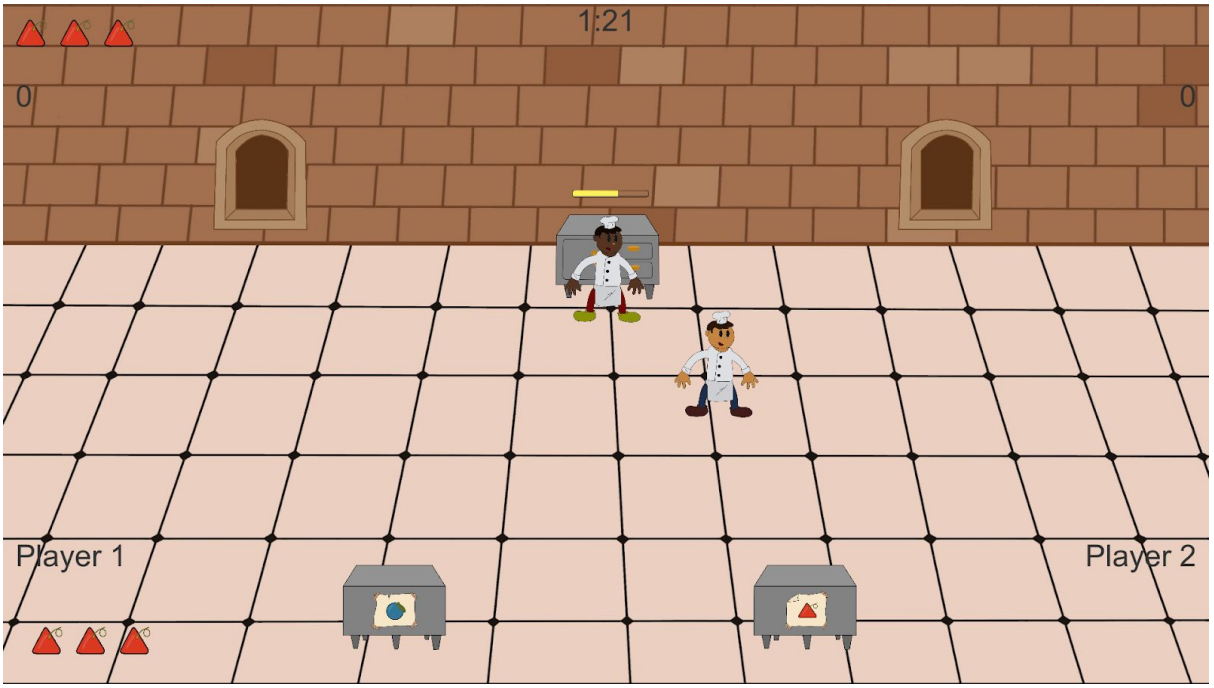
Imagem padrão do jogo, com Player 1 na esquerda e Player 2 na direita



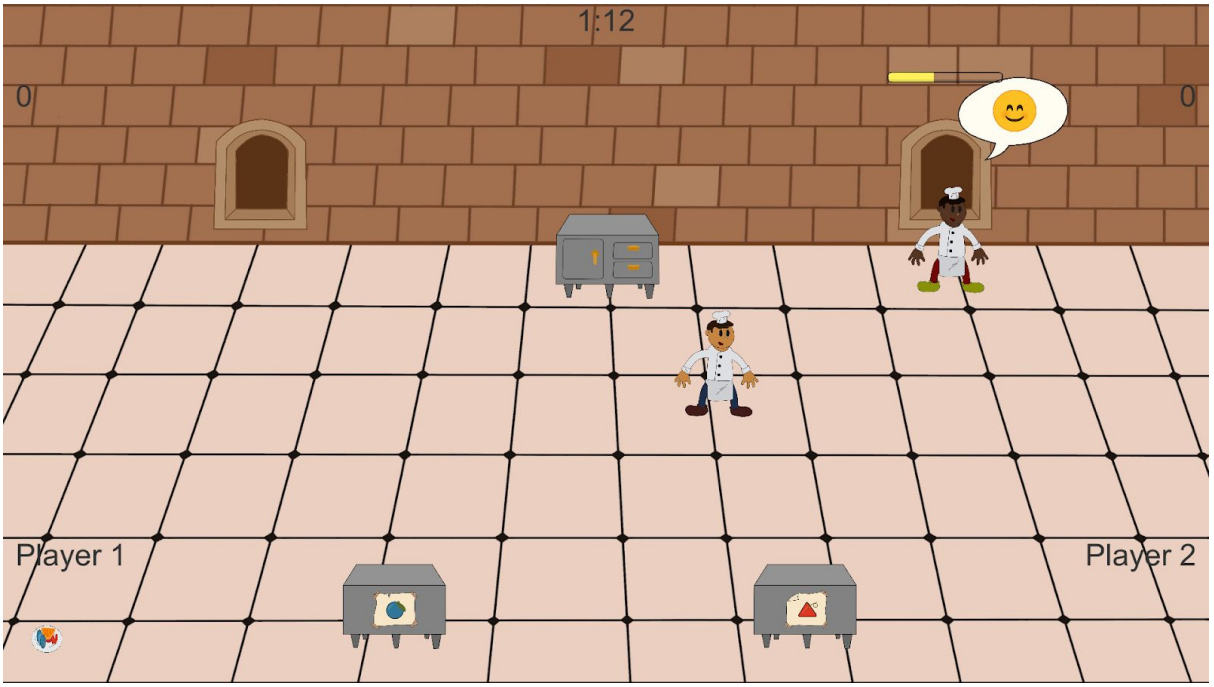
Player 1 recebendo pedido na janela de entrada



Player 1 pegando ingredientes no balcão correspondente

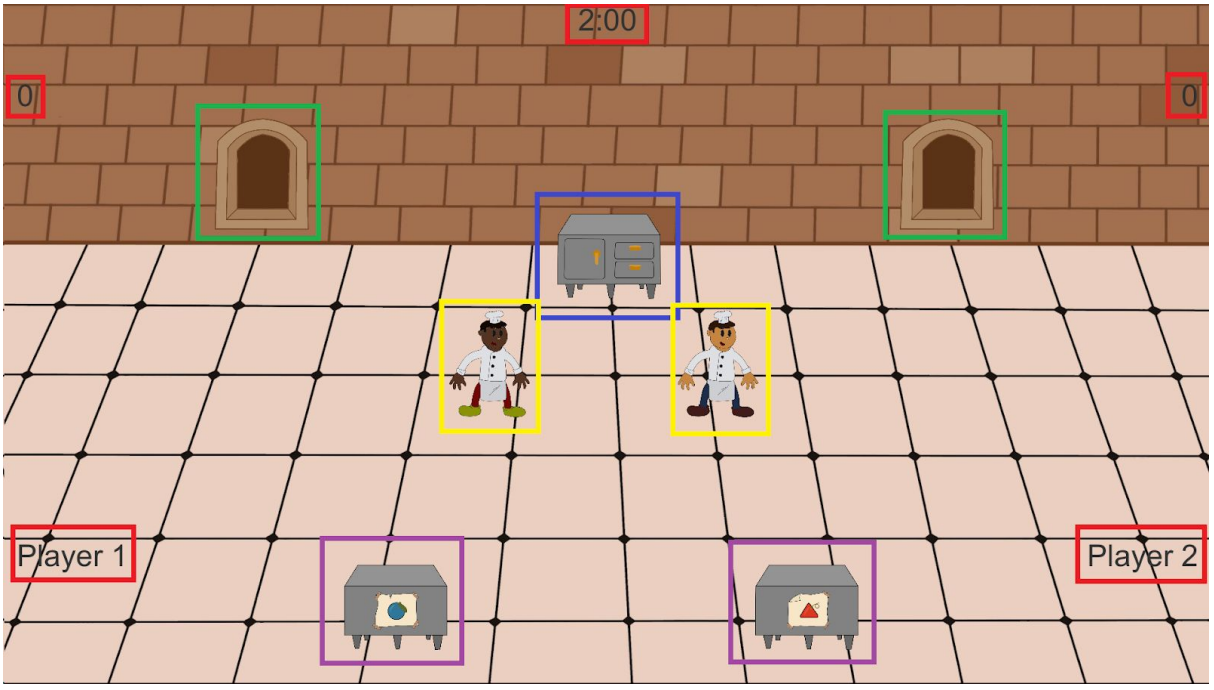


Player 1 montando pratos na mesa de montagem



Player 1 entregando pedido correto na janela de saída, os 10 pontos são adicionados no contador que atualmente marca 0 no topo superior esquerdo.

Restrições de jogo



Localização da HUD em vermelho, janelas de entrada e saída em verde, balcões de ingredientes em roxo, mesa de montagem em azul, Players em amarelo.

Só existe uma entidade de cada interação para dois Players. Dessa forma, um Player não pode usar a mesma entidade enquanto o outro Player não terminar a sua tarefa (Semáforo). Contudo, se o Player 1 interage com uma entidade enquanto o Player 2 já estava usando-a,

ele entrará numa fila de uso e começará a usar a entidade assim que o Player 2 terminar sua tarefa.

Cada Player pode carregar no máximo três ingredientes, que são usados na montagem do prato específico. Caso o Player tenha pego o ingrediente errado, ele montará um prato ruim. Os ingredientes corretos para o prato atualmente na agenda do Player são mostrados no canto superior respectivo ao Player. Os ingredientes que estão no bolso do Player são mostrados no canto inferior respectivo ao Player.

Módulos

Multithreading

Por que

Poder processar o movimento, interação e animação dos Players, visto que decidimos colocar dois Players que são *independentes* um do outro.

Como

Os personagens se comportam como threads únicas, sendo co-processadas por um core só. Não conseguimos utilizar as threads com todo seu potencial visto que a API da Unity só permite interações com ela por meio da thread principal, então tivemos que adaptar o código, utilizando as threads como "barramentos", que repassam as informações de forma assíncrona.

A lógica que usamos se assemelha ao conceito de threads lógicas num processador com Hyper-Threading (isso não afeta a utilidade do nosso jogo, ele pode ser jogado num processador normal sem Hyper-Threading sem problema).

O Unity, obrigatoriamente, vai permitir utilizar uma única thread, e isso não pode ser alterado sem alterar os padrões da framework - o que acarretaria outras mudanças necessárias, e não está dentro dos termos de uso da plataforma - portanto na ótica prática, realmente só é dado ao nosso programa uma thread (o padrão da maioria dos jogos hoje em dia, o que dá a processadores da Intel vantagem sobre os da AMD, por terem single-threading mais forte).

O que fizemos foi adaptar a forma que o Unity processa seu código customizado interno para que ele trabalhe de forma assíncrona com uso de barramentos, fazendo assim com que, na visão do usuário final, seja dada uma thread para cada Player e eles trabalhem independentemente entre si. Uma simulação interna *muito* básica do Hyper-Threading da Intel.

```
Assets > Game > Scripts > C# GameController.cs
{

    [SerializeField] private int timer = 60;
    [SerializeField] private Text timerText = null;
    [SerializeField] private List<Sprite> ingredientes = null;
    [SerializeField] private List<Sprite> pratos = null;
    [SerializeField] private Player player1 = null;
    [SerializeField] private Player player2 = null;

    public Thread _t1;
    public Thread _t2;

    void Start() {

        this.timerText.text = (this.timer / 60) + ":" + (this.timer % 60).ToString("00");
        InvokeRepeating("setTimer", 1f, 1f);

        _t1 = new Thread(_ControlPlayer1);
        _t1.Start();

        _t2 = new Thread(_ControlPlayer2);
        _t2.Start();

    }

    private bool W = false;
    private bool A = false;
    private bool S = false;
    private bool D = false;
```

Na função Start, temos o setup do nosso multithreading, com cada Player tendo sua própria thread.

```
Assets > Game > Scripts > C# GameController.cs
    }

    void _ControlPlayer1() {

        while( _t1.IsAlive ) {

            int horizontalInput = 0;
            int verticalInput = 0;
            bool interacting = false;

            if( W ) verticalInput = 1;
            if( A ) horizontalInput = -1;
            if( S ) verticalInput = -1;
            if( D ) horizontalInput = 1;
            if( J ) interacting = true;

            player1.Move(horizontalInput, verticalInput);
            player1.Interact(interacting);

            Thread.Sleep(100);

        }

    }

    void _ControlPlayer2() {

        while( _t2.IsAlive ) {
```

As funções _ControlPlayer1() e _ControlPlayer2() são similares em sua lógica, só alterando os devidos valores para garantir independência de entidade. Nelas, checamos a flag .IsAlive da thread respectiva à função (_t1 para Player1 e _t2 para Player2) e se houve algum input do usuário controlador do Player, processamos esse input com Move ou Interact, dependendo da ação, e damos o Sleep para a execução ocorrer sem overlap.

Para ser realmente um simulacro, a thread seria pausada ao entrar na fila, e retomada na vez dela. Mas como isso não faz sentido para as regras do jogo, ela é pausada ao entrar na fila e só é retomada ao acabar o processo.

Semáforo

Por que

Poder criar uma fila de espera nas entidades de interação, visto que limitamos elas para apenas uma entidade para dois Players e o uso delas *não* pode ser simultâneo.

Como

Foi utilizado o conceito de filas para fazermos o semáforo das entidades de interação. Assim, quando um Player pede para usar uma entidade, ele entra na fila com sua posição adequada e, conforme a flag de aviso mudar, terá acesso à entidade ou não.

```
1 reference
public void EntrarNaFila(GameObject personagem) {

    fila.Enqueue( personagem );

    if( personagem.transform.name == "Player1" ) gameController._t1.Suspend();
    else gameController._t2.Suspend();

}
```

```
Assets > Game > Scripts > C# mesaSemaforo.cs > mesaSemaforo
    void Update()
    {

        // Se nao tiver ninguem anotando pedido e houver pessoas na fila
        if( fila.Count > 0 && !interagindo ) {

            interagindo = true;
            contadorTempo = 0f;
            personagemAtual = fila.Dequeue();

            progressBar.SetActive(true);

            OnStartInteraction.Invoke();

        } else if ( interagindo ) {

            if( contadorTempo < duracao ) {

                contadorTempo += Time.deltaTime;

            } else if( contadorTempo >= duracao ) {

                interagindo = false;
                contadorTempo = 0f;

                if( personagemAtual.transform.name == "Player1" ) gameController._t1.Resume();
                else gameController._t2.Resume();

                personagemAtual = null;

                progressBar.SetActive(false);

                OnExitInteraction.Invoke();

            }

        }

    }
```

Toda a parte de programação segue o exemplo dado em slide, com a utilização da lógica de fila. O Unity permite aplicar fila facilmente, e as partes customizadas existem para podermos aplicar as regras do nosso jogo no programa.

Vale notar, o C# tem suporte nativo para semáforo com multithreading, mas encontraríamos o mesmo problema da framework disponibilizar uma ferramenta que não se adequa às regras do jogo detalhados no documento enviado no Moodle.