Antes de iniciar o desenvolvimento do jogo, decidi documentar todo o processo de planejamento. Escolhi o pattern MVC (Model-View-Controller) para separar a lógica e os dados, utilizando behaviors e scriptable objects. Para a visualização, separei a interface do usuário (UI) da lógica do jogo o máximo possível, e criei managers que são intermediários entre o modelo e a visualização, gerenciando a entrada do usuário. Logo no início, percebi que seria essencial para a performance do jogo utilizar object pooling.

Como se trata de um jogo do tipo Tower Defense, se um inimigo chegar à linha de chegada, o jogador deve receber dano. Inicialmente, pensei em usar o padrão observer, mas depois entendi que ele não seria aplicável nesse caso e optei por utilizar eventos com delegate. Abri o projeto, criei um chão, reposicionei a câmera e comecei a codar. O primeiro passo foi criar um script de pooling genérico e scriptable objects para manter os dados usados nos inimigos e no spawner. Em seguida, criei o comportamento dos inimigos.

Quando consegui fazer um inimigo básico andar reto, criei um trigger na área que deve ser defendida, para que, quando o inimigo chegasse nessa área, ele retornasse para a pool e chamasse um Action de OnEnemyHit para atualizar a UI e outras futuras possíveis necessidades. Depois, criei uma UI simples para a vida e tornei-a responsiva. Para ter mais um tipo de inimigo eu criei um inimigo que fica invisível e invulnerável de tempo em tempo. Uma coisa que eu poderia ter melhorado se tivesse mais tempo era fazer o tempo de invisibilidade e invulnerabilidade customizável.

Em seguida, fiz spawners na cena para pegar uma lista de possíveis transforms para spawnar inimigos. Inicialmente, pensei em criar vários objetos para definir spawn points e pegá-los através de um [SerializeField], mas decidi criar um objeto com todos os spawn points e dentro do código pegar cada spawn point filho dele.

Para que o jogo tivesse uma tela de Game Over, decidi diminuir a vida do jogador quando o inimigo chegasse ao final, e quando a vida chegasse a zero, a tela de Game Over apareceria. Também precisava de um botão para reiniciar a cena, então adicionei um listener no botão para chamar uma função de uma instância de scene management que eu criei. Coloquei o listener no botão porque é mais seguro caso eu queira futuramente mudar o que o botão faz sem o risco de perder referências fazendo alterações.

Em seguida fiz a estrutura de como seriam as waves, mas para prosseguir precisava de fazer as defesas, então estruturei e comecei a desenvolver a partir desse fluxo: Comprar defesa > posicionar defesa > defesa matar inimigos > todos os inimigos morrerem > ir para a próxima wave. Comecei criando a interface de usuário para a compra de defesas. Para isso, criei uma lista de itens da loja usando o sistema de object pooling, baseado nos dados contidos no ShopData. Isso permite adicionar novos itens na loja com facilidade, bastando apenas modificar o ShopData. Cada botão da lista está associado a uma função onClick que ativa o spawner de defesas. Ao clicar em um botão, o sistema identifica o raycast do mouse e faz spawn uma defesa a partir da pool naquele local do mouse.

Em seguida, modifiquei a IA dos inimigos para evitar as defesas, utilizando o sistema NavMesh da Unity, eu já usei packages de pathfinding em outros projetos que tendem a ser melhores soluções, mas para não usar soluções prontas acabei optando por usar o Navmash. Isso permitiu que os inimigos desviassem das defesas, que agora atuavam como obstáculos. Para evitar que o jogador pudesse construir em qualquer lugar, inclusive em cima de outras defesas, criei um gatilho que identifica quando uma defesa está ocupando um determinado espaço. Essa solução também será útil para implementar o sistema de upgrades.

A próxima etapa foi criar a mecânica de rotação das torretas em direção aos inimigos. Inicialmente, tentei fazer uma rotação suave, mas para manter a sensação arcade e melhorar a precisão, optei por deixá-la sem suavização. Em seguida, busquei melhorar a IA dos inimigos, sem precisar modificar o sistema NavMesh.

Depois disso, criei a mecânica de tiro das defesas. Criei um método para lidar com as colisões e os próximos passos: causar dano, despawnar a bala e o inimigo, caso tenha causado dano suficiente, calcular a pontuação, o ouro e a quantidade de inimigos restantes. Se o inimigo for morto, a wave continua até que todos os inimigos da wave sejam mortos.

Para gerenciar a interface do usuário, inicialmente criei vários scripts diferentes, mas decidi unificá-los em um só para evitar confusão e tornar a modificação mais fácil. Em seguida, trabalhei nas lógicas relacionadas às waves, incluindo a mudança de wave, alteração dos dados da próxima wave e o reset da interface do usuário. Criei também uma coroutine para aguardar alguns segundos entre as waves, que pode ser facilmente alterada por meio do ScriptableObject do MatchData.

Em seguida, criei a lógica para upgrades, adicionando um novo botão e modificando a interface do usuário para evitar que o jogador pudesse clicar no botão de fazer upgrade enquanto estivesse na loja e vice-versa. Quando o jogador clica no botão de fazer upgrade, o sistema retorna qual defesa o mouse está selecionando. Se nada estiver selecionado, o jogador é avisado. Caso contrário, o jogador pode fazer o upgrade se tiver dinheiro suficiente. O nível da defesa é aumentado, assim como o multiplicador de preço e de dano dos projéteis.

Por último, criei um leaderboard que salva até 10 highscores do usuário.

Com toda a estrutura do jogo pronta, eu precisava criar pelo menos mais um inimigo e um tipo de defesa nova. O inimigo que eu criei é foi o goldEnemy, ele é maior que os outros e tem bem mais vida. Em compensação ele é bem lento e por ser maior, ele tem dificuldade em passar entre estruturas. Mas por isso também ele pode acabar bloqueando as balas para outros inimigos passarem enquanto ele não morrer, o que cria uma mecânica interessante. Depois eu criei uma defesa que atira 3 balas de uma vez.
É interessante citar que para replicar novos tipos de estrutura acaba sendo bem prático, eu só tenho que colocar os dados necessários dos scriptableObjects e criar um comportamento específico do que fazer na ação de shoot, que é uma ação que se repete de acordo com o reload rate. Para repliciar inimigos é tão fácil quanto. Toda defesa herda de Defense e todo inimigo herda de Enemy. Também coloquei os

objetos estáticos da cena como estáticos para melhorar performance. De última hora também tive que corrigir um bug que ocorria quando transportava os inimigos para longe do navMash, mas isso era a solução para um problema visual no comportamento da torreta. Mas a correção para isso deve ser apenas resetar a posição da torreta sempre que não houver inimigo.

Infelizmente não tive tempo de implementar tudo que tinha em mente por falta de tempo pois só tinha o período da noite para trabalhar na test task, mas meus next steps eram:

- Fazer uma Coroutine para iniciar o jogo.
- Fazer uma mecânica para aumentar a vida dos inimigos conforme passa os rounds.
- Fazer torretas somente atirarem quando inimigos estivessem perto e parar todas as defesas de atirar entre waves.
- Salvar os scores com o nome do usuário.
- Melhorar algumas estruturas no código feitas com pressa, como por exemplo:
  # Em alguns lugares eu utilizo:
  if(other.gameObject.tag == "Enemy" || other.gameObject.tag == "Ghost")
  O problema disso é que toda vez tenho que verificar ambos, eu poderia fazer um padrão que compare os dois ao mesmo tempo. Mas só uso esse if no Detect Close Enemy e no PlayerAreaController.
- Colocar indicador visual de upgrades nas defesas.
- Fazer mais inimigos e defesas.
- Melhorar o comportamento da torreta.
- Alterar entrada do input para ser jogável tanto no celular quando no computador.