→ **O jogo produzido**: O objetivo desse projeto foi desenvolver um jogo na Unreal Engine 4. Nele, o jogador controla uma bolinha vermelha que se move por um campo 2D enquanto coleta moedas e disputa contra o computador para ver quem consegue chegar ao número máximo de moedas determinado no começo do jogo.

→ **O processo de criação**: A primeira parte do projeto foi criar um código que gerasse o campo de maneira procedural. Inicialmente usei valores fixos, mas que poderiam ser facilmente alterados no futuro. Como o resultado final é um jogo isométrico, tive que criar um sistema de posicionamento para determinar onde cada elemento da tela estava em relação aos outros. Esse sistema ganhou a forma de um par de inteiros que repesentava as posições horizontal e vertical daquele objeto em relação ao campo. O jogador começa no ponto (0,0) (o canto inferior esquerdo), e cada quadrado possui uma coordenada diferente.

Após o campo ficar pronto, comecei a trabalhar no posicionamento do personagem. Como eu usei um Blueprint padrão para personagens, toda a parte de colisão com o mouse foi realmente simples de ser implementada, e pouco depois do personagem estar pronto, ela já podia se mover pelo campo quando algum quadrado era clicado.

Como já havia implementado um sistema para mudar a cor de todos os quadrados válidos para movimento, atualizei-o para também mudar a cor dos quadrados que estão sob o cursor do mouse, deixando o movimento mais fácil, pois é possível ver para qual quadrado o personagem vai se mover após o clique.

Agora que eu já tinha um personagem que podia ser controlado, parti para a implementação das moedas. Utilizando o mesmo sistema de coordenadas e um node escrito em C++, um quadrado livre do campo é escolhido aleatoriamente, e nele uma moeda é posicionada. Ao detectar uma colisão com um personagem, a moeda avisa ao jogo que a colisão ocorreu, e o jogo atribui os pontos ao time correto antes de deletá-la.

A UI foi muito simples de implementar, já que ela conta com apenas elementos textuais. Criei duas variáveis para guardar a pontuação de cada time, e sempre que alguma moeda é coletada, a pontuação é atualizada. Foi nesse momento também que implementei a tela de fim de jogo e as condições de encerramento.

Nesse momento eu já tinha um jogo simples, com um personagem controlável e objetivos definidos. Criei então um menu inicial para ajustar as configurações existentes antes da partida começar. Pela tela de configuração é possível ajustar o tamanho do campo, a quantidade de pontos que precisa ser feita para o jogo acabar, e o nível de dificuldade (que muda a quantidade de inimigos presentes durante a partida).

Estava pensando em como salvar o highscore do jogador em um arquivo de texto, até ler a documentação oficial e ver que a Unreal já possui métodos próprios para salvar e carregar o jogo. No fim da partida, caso o jogador tenha conseguido quebrar seu recorde pessoal, sua pontuação é atualizada e exibida sempre que uma partida tem fim.

Deixei a parte mais complicada para o final: a implementação dos inimigos. Como eu já estava planejando desde o começo do projeto o modo como ambos personagem e inimigos se comportariam, consegui reaproveitar boa parte do código já existente. Antes de se mover, o inimigo seleciona todos os quadrados possíveis para o movimento (usando a função *GetValidCells*, a mesma usada pelo jogador), e após selecionar o quadrado desejado, chama a função *MoveCharacter*, também usada pelo jogador. A maior diferença em sua implementação está na inteligência artificial que decide para qual quadrado se mover. Essa decisão é tomada da seguinte forma: dentre todos os

quadrados que possuem moedas, seleciona-se aquele mais próximo do inimigo em questão. Depois, de todos os quadrados válidos para se mover (geralmente todos os adjacentes), escolhe-se aquele que está mais próximo do quadrado com a moeda. Cada inimigo se move em momentos diferentes, pois após o movimento, é preciso esperar um tempo entre 2.5 e 3.5 segundos para que se possa se mover novamente (deixando o jogo mais natural, pois os inimigos parecem se mover de forma independente).

Após a implementação de todas as funcionalidade, peguei um sprite sheet que minha namorada desenhou, criei FlipBooks para as animações idle e de pulo, e criei uma máquina de estados simples para o jogador e para os inimigos. Durante o movimento, o personagem (que é uma bola), pula até o quadrado escolhido antes de parar na pose idle. Novamente, foi possível reaproveitar muitas coisas entre personagem e inimigos, até mesmo os FlipBooks.

→ **C++**: A principal linguagem usada no jogo foi Blueprint, mas em casos em que a implementação de uma função seria desnecessariamente complexa, optei por criar uma função em C++ e chamá-la de dentro do Blueprint desejado. Além dos Getters e Setters padrões, as função criadas por esse motivo foram: *GetValidCells*, *GetFreeCell*, *CompareArray*, e, *GetNextCellInPath*. Todas essas funções servem para varrer um array de objetos de retornar um array ou objeto desejado.

→ **Configurando o gameplay**: Para configurar o gameplay, clique no botão Options no menu inicial e mude as configurações como preferir. Não é possível alterar as configurações após o jogo ter início. A escolha de dificuldade altera a quantidade de inimigos presente (1 inimigo no Fácil, 2 no Médio, e 3 no Difícil)

→ **Coisas que podem melhorar**: Têm algumas classes inutilizadas na pasta de classes. Eu as criei e depois mudei de ideia quanto à implementação, tornando-as obsoletas;

Eu comecei nomeando todas as variáveis relacionadas aos quadrados do campo com a palavra Square, mas depois achei que Cell ficaria melhor. É possível diferenciar variáveis e funções que foram criadas no começo da produção (como a função GetSquareById) daquelas que foram criadas depois (como a variável *ValidCells*);

A máquina de estados que controla as animações é extremamente simples, mas quebra o galho. É possível fazer algo mais elaborado, que leve em conta a direção do movimento para alterar as animações, mas no momento ela só leva em conta se houve movimento mesmo