Relatório descritivo de desenvolvimento
Space Sprint

Christian Louzada[1], **Edson Luiz**[2]

[1]Universidade Federal De Ouro Preto (UFOP)

[2]Instituto de Ciências Exatas e Aplicadas
ICEA R. Trinta e Seis, 115 - Loanda, João Monlevade - MG, 35931-008

{christian.daniel@aluno.ufop.edu.br, edson.barbosa@aluno.ufop.edu.br}

Resumo. *O projeto Space Sprint começou com a ideia original de Christian, um jogo de surf espacial, mas evoluiu para algo mais acessível para a equipe, considerando a experiência limitada em design e programação. O resultado foi um jogo de plataforma 2D, onde os jogadores controlam um astronauta em um planeta alienígena, enfrentando obstáculos para retornar ao seu foguete. Cada fase oferece diferentes cenários e desafios, proporcionando uma jogabilidade simples e divertida, acessível a todos os públicos.*

1. Introdução

O relatório descreve o processo de desenvolvimento do jogo *Space Sprint*, um projeto colaborativo iniciado com a ideia de criar um jogo acessível e divertido. O conceito original era um jogo de ”surf espacial”, mas evoluiu para um jogo de plataforma 2D, no qual o jogador controla um astronauta em um planeta alienígena, superando obstáculos para retornar ao seu foguete. Esse projeto foi inspirado por jogos clássicos como Super Mario Bros, mas adaptado ao tema espacial, incorporando uma mecânica de jogabilidade simples, porém envolvente.

Durante o desenvolvimento, utilizamos um tutorial do YouTube como base para a implementação das principais mecânicas, como movimentação, colisões e animações. A experiência foi valiosa para aprimorarmos nossas habilidades em design e programação de jogos, resultando em um produto final que equilibra acessibilidade com uma experiência de jogo desafiadora e divertida para diferentes públicos. Este trabalho também se apresenta como uma oportunidade de explorar técnicas de desenvolvimento de jogos, utilizando ferramentas como o *GameMaker Studio*.

2. Desenvolvimento inicial

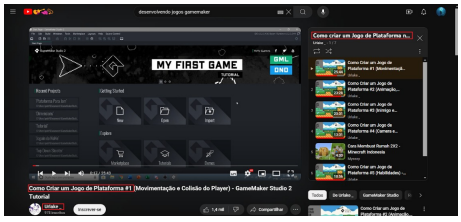

Figure 1. Tutorial seguido

As etapas de desenvolvimento do jogo, foi seguida de acordo com o tutorial do canal Urlake , criando inicialmente a sprite do *player* juntamente com seu objeto e em seguida, toda lógica de colisores, que seria seguida do início ao fim do projeto.

Conforme a própria trilha se conduzia, fomos seguindo junto à ela, de inclusão sprites e objetos tanto do *player*, quanto dos inimigos, juntamente com o esquemas de colisões do *player* e inimigos, tilesets e esquema de *checkpoint*.

Nem tudo que foi abordado durante o tutorial, foi implementado, somente o que precisamos, absorvemos o que necessitávamos há mais, fomos improvisando.

3. Desenvolvimento do *player*

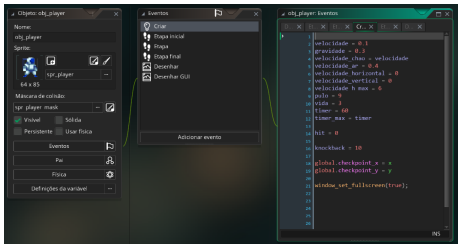

Figure 2. Objeto do *player*

O objeto *player* é criado e associado à ele, um sprite, que seria os frames em movimentos que dão a mágica da animação do *player*, foi coletado o Sprite Free do site para a nossa parte gráfica.

3.1. Evento: Criar

Neste evento, as variáveis essenciais para o comportamento do jogador são inicializadas:

- **Velocidades e gravidade**: Controlam o movimento horizontal, a gravidade que afeta o jogador e o impulso ao pular.
- **Pulo**: Define a força do salto.
- **Vida e timer**: O jogador começa com 3 vidas, e um *timer* controla a invulnerabilidade após uma colisão.
- ***checkpoint***: Armazena as coordenadas para *respawn* do jogador após a perda de todas as vidas.
- **Fullscreen**: Define o jogo em tela cheia.

3.2. Evento: Etapa Inicial

Uma verificação simples é realizada aqui para reiniciar a sala caso o jogador pressione a tecla ”R”, permitindo ao jogador reiniciar o jogo manualmente.

3.3. Evento: Etapa

Esta é a parte principal do código, onde o comportamento do jogador durante o jogo é calculado. Inclui os seguintes aspectos:

- **Movimentação horizontal**: As teclas ”A” e ”D” são detectadas para mover o jogador para esquerda ou direita, e a função `lerp` é usada para suavizar a transição da velocidade.

- **Pulo**: A tecla ”W” é usada para fazer o jogador pular quando ele está no chão.
- **Controle de sprites**: Dependendo do estado de movimento e colisão do jogador, o sprite correto (parado, correndo, pulando ou caindo) é selecionado.
- **Colisões e dano**: O código verifica colisões com inimigos (`obj_enemy`, `obj_enemy2`, `obj_enemy3`). Quando ocorre uma colisão, o jogador recebe dano, perde vida e é empurrado (knockback) na direção oposta ao inimigo. O *timer* impede que o jogador seja atingido continuamente em curtos intervalos.
- ***checkpoint* de vida**: Caso o jogador perca todas as vidas, ele retorna ao último *checkpoint*, com a vida restaurada para 3.

3.4. Evento: Etapa Final

Este evento lida com colisões com o chão e paredes, além de aplicar as velocidades calculadas ao jogador:

- **Colisões horizontais**: Se o jogador encontrar um obstáculo horizontalmente, o código ajusta sua posição até que ele pare corretamente antes da colisão.
- **Colisões verticais**: Funciona de forma semelhante, ajustando a posição do jogador ao encontrar o chão ou o teto.
- **Aplicação de movimento**: No final da etapa, as velocidades horizontais e verticais são aplicadas às posições `x` e `y` do jogador.

3.5. Esquema de Colisões

O esquema de colisão usa a função `place_meeting` para verificar a interseção entre o jogador e outros objetos (como `obj_block` e `obj_enemy`). Quando há uma colisão, o código ajusta a posição do jogador para evitar que ele atravesse objetos sólidos, aplicando penalidades como perda de vida e *knockback*.

- **Colisão horizontal**: Verifica se o jogador está colidindo com objetos à esquerda ou direita. Se houver colisão, ele é movido para a posição anterior à colisão.
- **Colisão vertical**: Ajusta o movimento do jogador ao colidir com o chão ou o teto.

3.6. Dano e Reinício de Vidas

Quando o jogador colide com um inimigo, ele perde uma vida e é afetado pelo *knockback*. O sistema de *timer* impede que o jogador receba dano repetidamente em um curto intervalo, garantindo um tempo de invulnerabilidade.

- **Perda de dano**: Se o jogador colide com um inimigo, a vida diminui. Se `vida <= 0`, o jogador é teleportado de volta ao último *checkpoint* salvo (`global.checkpoint_x` e `global.checkpoint_y`), com 3 vidas restauradas.
- **Troca de fases**: O código não inclui um tratamento explícito para a troca de fases, mas o reinício de vidas ocorre sempre que o jogador perde todas as suas vidas e volta ao *checkpoint*.

4. Desenvolvimento dos inimigos

O jogo apresenta três tipos de inimigos, cada um com comportamentos específicos. Todos os inimigos utilizam a função *place_meeting* para detectar colisões e movimentação, assim como o jogador. Abaixo está o detalhamento de cada um deles:

4.1. Inimigo 1

Este inimigo é caracterizado por se mover horizontalmente e detectar colisões com o jogador. Caso o jogador pule sobre ele, o inimigo é destruído.

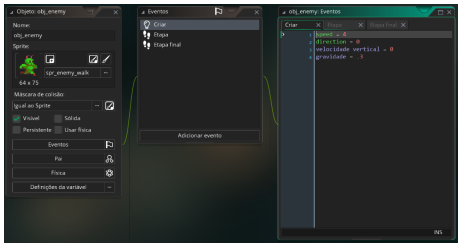

Figure 3. Inimigo 1

Criar: Inicialmente, as variáveis do Inimigo 1 são definidas para controlar sua velocidade horizontal (*speed*), direção e gravidade. A gravidade influencia o movimento vertical, enquanto a direção determina para onde ele está se movendo.

Etapa: - O inimigo verifica constantemente se há chão sob ele. Se não houver, ele aplica a gravidade para simular a queda. - Além disso, a colisão com outros inimigos (*obj_collision_enemy*) faz com que ele altere sua direção ao longo do eixo horizontal. - A colisão com o jogador (*obj_player*) é verificada, e se o jogador estiver pulando sobre o inimigo, o Inimigo 1 é destruído.

Etapa final: - O inimigo ajusta sua posição vertical ao colidir com o chão, garantindo que ele não atravesse o chão e se mova corretamente ao longo da superfície.

4.2. Inimigo 2

O Inimigo 2 apresenta uma lógica semelhante ao Inimigo 1, porém com a diferença de ter uma velocidade horizontal negativa, movendo-se inicialmente para a esquerda.

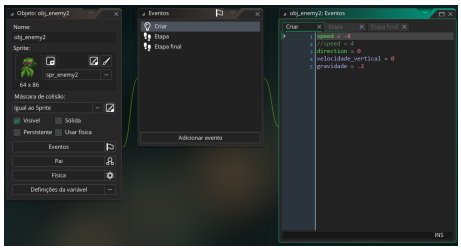

Figure 4. Inimigo 2

Criar: A velocidade e gravidade do Inimigo 2 são definidas, com *speed* negativo para fazê-lo se mover para a esquerda.

Etapa: - Assim como o Inimigo 1, ele altera sua direção ao colidir com outro inimigo, porém mantém a mesma lógica de movimentação horizontal e vertical. - A colisão com o jogador também é tratada da mesma forma: se o jogador cair sobre o inimigo, ele é destruído.

Etapa final: - A lógica de colisão com o chão e ajuste da posição vertical funciona de forma idêntica ao Inimigo 1.

4.3. Inimigo 3

O Inimigo 3 é único, pois ele permanece fixo no chão e elimina o jogador na primeira colisão.

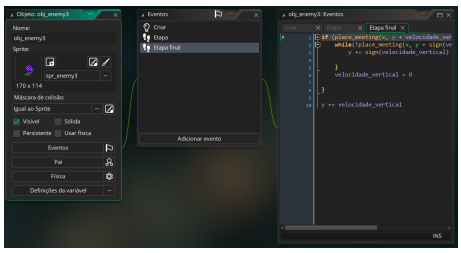

Figure 5. Inimigo 3

Criar: - Inicialmente, o Inimigo 3 tem sua gravidade definida para evitar que ele ”flutue”, mantendo-se no chão.

Etapa: - A colisão com o jogador não depende da altura ou da velocidade do jogador. Assim, se o jogador tocar no inimigo, independentemente da situação, ele será eliminado imediatamente. - Como o Inimigo 3 não se move horizontalmente, o controle de direção não afeta sua posição.

Etapa final: - Ele ajusta sua posição vertical para se manter fixo ao chão, assim como os outros inimigos. No entanto, ao contrário dos outros, ele não se move horizontalmente e atua como um obstáculo estático.

5. Colisores, cenas e tilesets

Nesta seção, exploramos como os colisores, as cenas e os tilesets se integram ao desenvolvimento dos inimigos e do jogador. Esses elementos são essenciais para a construção de um mundo interativo e dinâmico dentro do jogo.

5.1. Colisores

Os colisores são elementos fundamentais na definição das áreas de colisão dentro do jogo, desempenhando um papel crucial tanto para o jogador quanto para os inimigos. Eles garantem que as interações físicas no jogo, como movimentação, saltos e colisões com blocos e inimigos, ocorram de forma consistente e previsível. O uso da função placeMeeting no código é essencial para detectar essas colisões, permitindo que o jogador interaja corretamente com objetos sólidos e evite atravessá-los. Quando o jogador colide com um bloco, por exemplo, o colisor ajusta automaticamente sua posição, evitando que fique preso ou atravesse o objeto.

Além de gerenciar as colisões, os colisores permitem que os inimigos reajam de maneira apropriada às interações com o jogador. Eles são utilizados para impedir que os inimigos ultrapassem barreiras, como plataformas e blocos, e para detectar a presença do jogador, possibilitando reações como mudar de direção ou atacar. Uma implementação cuidadosa dos colisores contribui para a fluidez da jogabilidade, criando um ambiente mais dinâmico e desafiador. Isso aumenta a imersão do jogador, pois cada salto, corrida e colisão se sente realista e integrada à experiência do jogo.

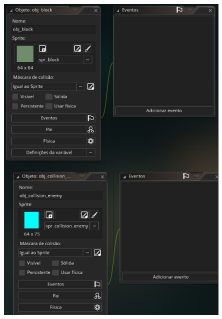

Figure 6. Colisores utilizados no jogo para controlar a interação entre o jogador, inimigos e o ambiente.

5.2. Cenas e Tilesets

As cenas e tilesets definem o ambiente visual e o layout das fases do jogo. Cada fase utiliza um tileset para compor os blocos, plataformas e outros elementos de cenário, que também possuem colisores para interagir com o jogador e os inimigos. A construção cuidadosa das cenas com esses elementos garante que o jogador possa se movimentar adequadamente pelo cenário, saltar sobre plataformas e evitar perigos. Além disso, o uso dos tilesets facilita a criação de múltiplas fases com variações visuais e estruturais, permitindo uma progressão natural do jogo.

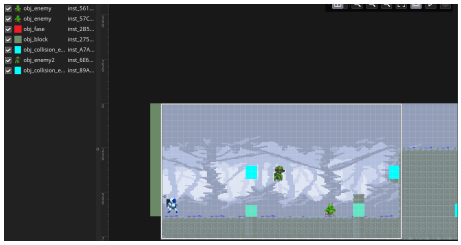

Figure 7. Exemplo de cena utilizando tilesets para compor a estrutura visual e as plataformas de uma fase.

Os tilesets fornecem uma base modular que facilita a criação de múltiplos níveis, e cada cena é montada a partir da combinação de blocos que podem ser tanto visuais quanto funcionais, com colisores definidos. Isso é importante para assegurar que as mecânicas de jogo, como a colisão entre o jogador, os inimigos e o ambiente, funcionem adequadamente em todas as fases.

6. Conclusão

Neste trabalho, foi apresentado o desenvolvimento de um sistema de jogo que envolve a construção de um personagem jogável e inimigos, utilizando colisores, cenas e tilesets para criar um ambiente interativo e desafiador. O comportamento do *player* foi cuidadosamente estruturado, levando em conta a física de movimentação, como a gravidade e a resposta a colisões, além de um sistema de *checkpoint* para gerenciamento de vidas. Os inimigos foram projetados para reagirem às interações com o jogador e o ambiente de maneira consistente, criando desafios variados a cada fase.

Os colisores desempenham um papel central no controle das interações físicas entre o jogador, inimigos e objetos do cenário, enquanto os tilesets e cenas fornecem a base

visual e estrutural de cada nível. A integração desses elementos resultou em um sistema coeso e flexível, que pode ser expandido e adaptado conforme novas fases e mecânicas de jogo forem adicionadas.

O desenvolvimento do jogo seguiu um tutorial disponível no YouTube [urlake286 2024], e os sprites utilizados foram retirados de sites de recursos gratuitos, como o Itch.io [Itch.io 2024] e o Adobe Stock [Adobe 2024].

Com isso, o trabalho desenvolvido até aqui oferece uma base sólida para a continuidade do projeto, abrindo espaço para a introdução de novas funcionalidades, aprimoramento das fases e a adição de inimigos e desafios mais complexos. O uso de uma arquitetura modular, com separação clara de comportamentos e interações, permite que o jogo se mantenha escalável e fácil de manter.

References

Adobe (2024). Adobe stock - stock photos, royalty-free images, graphics, vectors, videos, and music. Acesso em: 09 out. 2024.

Itch.io (2024). 2d game assets on itch.io. Acesso em: 09 out. 2024.

urlake286 (2024). Game Development Tutorial. [Acesso em: 09 out. 2024].