



Escola Profissional
BENTO DE JESUS CARAÇA
DELEGAÇÃO DO BARREIRO

PROVA DE APTIDÃO PROFISSIONAL

CURSO PROFISSIONAL DE GESTÃO E PROGRAMAÇÃO DE SISTEMAS INFORMÁTICOS

CICLO DE FORMAÇÃO 2022/2025

Prova de Aptidão
Profissional



APOIO E SUPORTE INFORMÁTICO (SUBSTITUIR)

Julho de 2025

Ba2535 – Joel Matoso

Declaro que este trabalho se encontra em condições de ser apresentado a provas públicas.

O Professor Orientador

Barreiro, julho de 2025

Agradecimentos

Escola profissional bentos de Jesus caraça

Família

Júris

professor: Marcelo simão

professora: Marina Rocha

professora: Helena Nunes

professor: Diogo Oliveira

Índice

Índice	5
1 Introdução	6
1.1 Fundamentação da escolha do Projeto	6
1.2 Finalidades do Projeto	6
1.3 Enquadramento do Projeto	7
1.4 Cronograma	7
2 Projeto PAP – (The bleedin' globe of nigatsumi)	11
2.1 Ferramentas e linguagens utilizadas	11
2.2 Etapas e funcionalidades	11
3 Conclusão - análise crítica global da execução do projeto	37
3.1 Dificuldades	37
3.2 Problemas e obstáculos	38
3.3 Soluções encontradas	38
Bibliografia	39
Anexos	40

Índice de Ilustrações

Índice de Ilustrações	4
1 Introdução	5
1.1 Fundamentação da escolha do Projeto	5
1.2 Finalidades do Projeto	5
1.3 Enquadramento do Projeto	5
1.4 Cronograma	6
2 Projeto PAP – (The bleedin' globe of nigatsumi)	7
1.5 Ferramentas e linguagens utilizadas	7
1.6 Etapas e funcionalidades	8
2 Conclusão - análise crítica global da execução do projeto	35
1.7 Dificuldades	35
1.8 Problemas e obstáculos	35
1.9 Soluções encontradas	36
Bibliografia	37
Anexos	38

1 Introdução

1.1 Fundamentação da escolha do Projeto

Desde pequeno, sempre tive uma grande paixão por videojogos. Jogar sempre foi mais do que um hobby para o aluno; foi uma forma de explorar mundos, viver aventuras e sonhar em criar minhas próprias histórias interativas. Com o passar do tempo, esse sonho amadureceu e se tornou um objetivo: desenvolver o meu próprio jogo. Assim, quando surgiu a oportunidade de realizar a Prova de Aptidão Profissional (PAP), a escolha do tema foi natural. Criar um jogo não é apenas um desafio técnico, mas também uma realização pessoal e um marco na jornada para transformar um sonho de infância em realidade.

1.2 Finalidades do Projeto

As principais finalidades deste projeto são:

Aprendizado Técnico: Aplicar e aprimorar habilidades em programação, design gráfico e desenvolvimento de jogos adquiridas durante a formação.

Criatividade e Inovação: Desenvolver um jogo que combine uma narrativa envolvente com mecânicas de jogabilidade únicas, destacando-se pela criatividade e originalidade.

Experiência Prática: Simular o processo de desenvolvimento de um jogo real, incluindo planeamento, execução e solução de problemas.

Impacto Pessoal e Profissional: Criar um portfólio que reflita minha paixão e habilidades, servindo como um trampolim para oportunidades futuras na indústria de jogos.

1.3 Enquadramento do Projeto

O projeto "The Bleedin' Globe of Nigatsumi" está inserido no contexto de desenvolvimento de jogos independentes (indie), com foco em mecânicas inovadoras, narrativas envolventes e experiências visuais marcantes. Ele será desenvolvido utilizando ferramentas modernas, combinando aspectos técnicos como programação, arte e som, com uma visão criativa única.

Além disso, o projeto visa explorar temas como exploração, estratégia e combate, em um ambiente fictício que desafia os jogadores a pensar criticamente e se

envolver emocionalmente. O jogo reflete não apenas minhas habilidades técnicas, mas também minha visão criativa e minha paixão por contar histórias através de um meio interativo.

1.4 Cronograma

Semana 1:

Definição da ideia principal do jogo e das mecânicas.

Pesquisa de ferramentas e tecnologias a serem utilizadas (ex.: GameMaker, Unity, etc.).

Elaboração do documento de design do jogo (GDD - Game Design Document).

Semana 2:

Planejamento do cronograma e das tarefas.

Criação de esboços iniciais para personagens, cenários e interface do jogo (UI).

2. Desenvolvimento Inicial (Semana 3-6)

Semana 3-4:

Criação de um protótipo básico do jogo:

Implementação do sistema de movimentação do personagem principal.

Configuração de colisões e física do jogo.

Testes iniciais do protótipo para identificar bugs e ajustar mecânicas.

Semana 5-6:

Desenvolvimento de elementos básicos do ambiente (cenários, obstáculos, etc.).

Primeiros testes com as mecânicas de combate e interações.

3. Desenvolvimento Avançado (Semana 7-12)

Semana 7-8:

Design e animação dos personagens (principal, inimigos, NPCs).

Implementação das mecânicas avançadas, como ataques, habilidades especiais e interações com o cenário.

Semana 9-10:

Desenvolvimento de níveis/jogabilidade:

Design dos níveis iniciais e progressão de dificuldade.

Implementação de checkpoints e sistema de salvamento.

Ajustes na interface do usuário (menus, barras de vida, HUD).

Semana 11-12:

Adição de efeitos visuais e sonoros:

Música de fundo e efeitos sonoros para ações (ataques, saltos, etc.).

Ajuste da iluminação, partículas e outros detalhes gráficos.

4. Testes e Ajustes Finais (Semana 13-16)

Semana 13-14:

Testes intensivos para identificar bugs, falhas e problemas de jogabilidade.

Coleta de feedback de usuários externos e ajustes baseados nas observações.

Semana 15:

Revisão final do jogo:

Balanceamento de dificuldade e mecânicas.

Verificação da performance e otimização (FPS, tempo de carregamento, etc.).

Semana 16:

Preparação da apresentação e documentação do projeto.

Criação de materiais de apoio (ex.: slides, vídeo de gameplay, etc.).

5. Entrega do Projeto (Semana 17)

Apresentação oficial do jogo e entrega da Prova de Aptidão Profissional (PAP).

Demonstração prática do jogo e explicação do processo de desenvolvimento.

2 Projeto PAP – (The bleedin' globe of nigatsumi)

1.5 Ferramentas e linguagens utilizadas

ferramentas utilizadas para pesquisa google youtube.

Linguagem de programação escolhida: (gms) Game Maker Studio e (gml) Game maker language

nome do Jogo The bleedin' globe of nigatsumi.

1.6 Etapas e funcionalidades

Introdução

Informações Gerais do Projeto

- Nome do Projeto:

The bleedin' globe of nigatsumi.

- Data de Início: - Data de Término:

01/10/2024

01/01/2025

- Tipo de Projeto: (Software/Hardware)

Vídeo Jogo rpg feito em software no Game Maker Studio

Jogo RPG

Jogo RPG 2D com foco na narrativa, exploração e combate estratégico, proporcionando uma experiência imersiva para os jogadores. o principal objetivo do projeto é com foco em proporcionar entretenimento aos jogadores.



Funcionalidades:

Descrição geral das principais funcionalidades desenvolvidas ou do hardware projetado.

ASDW para o boneco se andar, espaço para rolar shift para atacar esc para por no pausa

ctrl + espaço para lançar bombas Q, E para trocar de itens para o comando.

Joystick para o boneco se mover gp face2(bola) para rolar gp face3(quadrado) para atacar gp start para por no pausa gp padd(seta para baixo) depois bola para lançar bombas

e para outros itens como flecha e a garra gp padd(seta para baixo)L1(gp_shoulderl) e R1(gp_shoulder) para trocar de itens

—

Ferramentas Utilizadas para Pesquisa:

Google: Utilizado para buscar informações técnicas, tendências do mercado e soluções para desafios específicos no desenvolvimento do jogo.

YouTube: Fonte de tutoriais e exemplos práticos relacionados à criação de jogos no Game Maker Studio e à utilização da linguagem GML.

Linguagem de Plataforma e Programação Escolhida:

Game Maker Studio (GMS): Plataforma selecionada para o desenvolvimento do jogo devido à sua versatilidade e ao suporte para criação de jogos 2D.

Game Maker Language (GML): Linguagem de programação nativa do GMS, escolhida por ser específica para desenvolvimento de jogos na plataforma e oferecer uma curva de aprendizado apropriada para o projeto.

Controles do Jogador

```

15 keyLeft = keyboard_check(vk_left) or keyboard_check(ord("A"))
16 //or gamepad_axis_value(global.controls, gp_axis1b) > 0.25;
17 keyRight = keyboard_check(vk_right) or keyboard_check(ord("D"))
18 //or gamepad_axis_value(global.controls, gp_axis1b) > 0.25;
19 keyUp = keyboard_check(vk_up) or keyboard_check(ord("W"))
20 //or gamepad_axis_value(global.controls, gp_axis1b) > 0.25;
21 keyDown = keyboard_check(vk_down) or keyboard_check(ord("S"))
22 //or gamepad_axis_value(global.controls, gp_axis1b) > 0.25;
23 keyActivate = keyboard_check_pressed(vk_space)
24 //or gamepad_button_check_pressed(global.controls, gp_face2);
25 keynot_activate = keyboard_check_pressed(ord("X"))
26 //or gamepad_button_check_pressed(global.controls, gp_face1);
27 keyAttack = keyboard_check_pressed(vk_shift)
28 //or gamepad_button_check_pressed(global.controls, gp_face3);
29 keyItem = keyboard_check_pressed(vk_control)
30 //or gamepad_button_check_pressed(global.controls, gp_padd);
31 keyItemSelectUp = keyboard_check_pressed(ord("E"))
32 //or gamepad_button_check_pressed(global.controls, gp_shoulderl);
33 keyItemSelectDown = keyboard_check_pressed(ord("Q"))
34 //or gamepad_button_check_pressed(global.controls, gp_shoulderr);

```

A primeira parte do código define variáveis que detectam a entrada do jogador:

Fig. 1: Código dos controlos do personagem principal.

keyLeft e keyRight: Detectam o movimento horizontal (esquerda e direita). A verificação é feita para as teclas do teclado (vk_left, ord("A"), etc.), mas linhas comentadas indicam que há suporte planejado para um joystick ou gamepad.

keyUp e keyDown: Detectam o movimento vertical (cima e baixo), novamente para teclado e joystick.

keyActivate e keyAttack: Estas variáveis monitoram ações específicas:

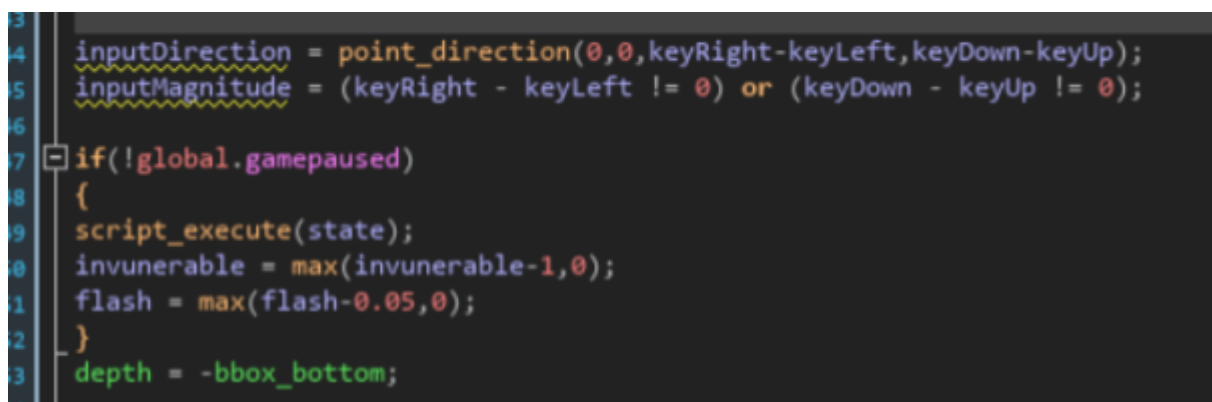
keyActivate: Ativação de interações (tecla Space ou botão do controle configurado).

keyAttack: Realiza um ataque (tecla Shift ou equivalente no controle).

keyItem e Seleção de Itens:

KeyItem verifica se o jogador pressionou Ctrl para usar um item.

keyItemSelectUp e keyItemSelectDown: Usadas para alternar itens no inventário (E para subir, Q para descer).

A screenshot of a code editor showing Lua-like code for character movement. The code is as follows:

```
3
4  inputDirection = point_direction(0,0,keyRight-keyLeft,keyDown-keyUp);
5  inputMagnitude = (keyRight - keyLeft != 0) or (keyDown - keyUp != 0);
6
7  if(!global.gamepaused)
8  {
9    script_execute(state);
10   invulnerable = max(invulnerable-1,0);
11   flash = max(flash-0.05,0);
12  }
13  depth = -bbox_bottom;
```

Fig. 2: Código dos controles do personagem principal.

inputDirection = point_direction(0,0,keyRight-keyLeft,keyDown-keyUp);

Objetivo: Determinar a direção do movimento do jogador.

Funcionamento: A função point direction calcula o ângulo (em graus) do vetor formado pelas entradas horizontais (key Right - key Left) e verticais (keyDown - keyUp).

Se key_Right e/ou keyDown forem pressionados, o ângulo será ajustado de acordo.

O resultado é usado para orientar o personagem ou determinar a direção de uma ação, como atirar ou mirar.

inputMagnitude = (keyRight - keyLeft != 0) or (keyDown - keyUp != 0);

Objetivo: Verificar se o jogador está em movimento.

Funcionamento: A expressão avalia se existe qualquer entrada ativa (horizontal ou vertical).

Retorna true se pelo menos uma das teclas de movimento estiver sendo pressionada.

Pode ser usado para alternar entre estados do personagem (parado/movendo).

if(!global.gamepaused)

Objetivo: Garantir que o bloco de código seguinte só será executado se o jogo não estiver pausado.

Funcionamento: O uso de !global.gamepaused verifica a variável global gamepaused, que indica o estado atual do jogo.

```
{  
  
    script_execute(state);  
  
    invulnerable = max(invulnerable-1,0);  
  
    flash = max(flash-0.05,0);  
  
}
```

script_execute(state):

Executa um script baseado no estado atual do jogador (state). Isso permite modificação dinâmica de comportamento, como ações de combate, exploração, ou transições de estado (parado/movendo).

Controle de Invulnerabilidade (invulnerable):

A variável invulnerable é decrementada em 1 a cada ciclo, até atingir o valor mínimo de 0.

Uso Comum: Representa um período de imunidade temporária após sofrer dano.

Controle de Flash (flash):

A variável flash é reduzida gradualmente em 0.05, até o mínimo de 0.

Uso Comum: Indica um efeito visual, como um piscar do sprite do personagem enquanto ele está invulnerável.

depth = -bbox_bottom;

Objetivo: Ajustar a profundidade (ordem de renderização) do objeto com base em sua posição vertical.

Funcionamento: Atribui a depth o valor negativo de bbox_bottom (a borda inferior do bounding box do sprite do objeto).

Efeito Visual: Objetos mais baixos na tela aparecem na frente de objetos mais altos, criando uma ilusão de perspectiva.

Câmera do Jogador

```
1  if(instance_exists(follow))
2  {
3      xto = follow.x;
4      yto = follow.y;
5
6  }
7  //atualizacao da posicao da camera
8  x += (xto - x)/ 15;
9  y += (yto - y)/ 15;
10 //manter a camera no centro
11 x = clamp(x, view_w, room_width-view_w);
12
13 //y = clamp(y, view_h, room_width-view_h);
14 y = clamp(y, view_h, room_height-view_h);
15 //ecra treme
16 x += random_range(-shake_remain,shake_remain);
17 y += random_range(-shake_remain,shake_remain);
18
19 shake_remain = max(0,shake_remain - ((1/shake_length) * shake_magnitude));
20
21 camera_set_view_pos(cam,x - view_w, y - view_h);
```

Fig. 3: Código da câmera que segue o jogador.

```
if(instance_exists(follow))
```

```
{
```

```
    xto = follow.x;
```

```
    yto = follow.y;
```

```
}
```

Descrição: Esta parte do código verifica se uma instância chamada follow existe no momento em que o código é executado.

Caso a instância exista, as coordenadas de destino da câmara (xto e yto) são definidas como a posição da instância follow (follow.x e follow.y).

Propósito: Permitir que a câmara siga um objeto específico, cuja existência não é garantida. Assim, evita erros de referência nula.

```
x += (xto - x) / 15;
```

```
y += (yto - y) / 15;
```

A posição atual da câmara (x e y) é ajustada gradualmente em direção às coordenadas de destino (xto e yto).

A fórmula $(xto - x) / 15$ cria um movimento suave, onde a câmara percorre 1/15 da distância restante a cada atualização.

Propósito: Este tipo de interpolação linear suaviza o movimento da câmara, evitando transições bruscas.

```
x = clamp(x, view_w, room_width - view_w);
```

```
y = clamp(y, view_h, room_height - view_h);
```

A função clamp restringe os valores de x e y para que a câmara não ultrapasse os limites definidos pelo cenário.

Os limites são calculados com base no tamanho da visão (view_w e view_h) e do cenário (room_width e room_height).

Propósito: Evitar que a câmera exiba áreas fora dos limites do cenário, mantendo a visualização focada dentro da área de jogo.

```
x += random_range(-shake_remain, shake_remain);
```

```
y += random_range(-shake_remain, shake_remain);
```

```
shake_remain = max(0, shake_remain - ((1 / shake_length) *  
shake_magnitude));
```

A posição da câmera (x e y) é alterada aleatoriamente dentro de um intervalo definido por shake_remain.

O intervalo de tremor é reduzido gradualmente, diminuindo o efeito ao longo do tempo.

A redução é controlada por $(1 / \text{shake_length}) * \text{shake_magnitude}$.

Propósito: Implementar um efeito visual de "tremor" para a câmera, que pode ser usado em momentos de impacto, explosões ou eventos dramáticos no jogo.

```
camera_set_view_pos(cam, x - view_w, y - view_h);
```

Define a posição final da câmera com base nos valores calculados de x e y.

A subtração de view_w e view_h ajusta o ponto de referência para manter o objeto de interesse no centro da tela.

Propósito: Atualizar a posição da câmera no ambiente do jogo, garantindo que o efeito de seguimento, os limites e o tremor sejam aplicados corretamente.

Função Morte do Jogador

```
1 function hurt_player(_direction, _force, _damage){
2   if(o_player.invulnerable <= 0)
3   {
4     global.playerhealth = max(0, global.playerhealth - _damage);
5     if(global.playerhealth > 0)
6     {
7       with(o_player)
8       {
9         state = playerstate_bonk;
10        direction = _direction-180;
11        movedi_res = _force;
12        screen_shake(2,10);
13        flash = 0.7;
14        invulnerable = 60;
15        flash_shader = shred_flash;
16      }
17    }
18    else
19    {
20      with(o_player) state = playerstate_dead;
21    }
22  }
23 }
```

Fig. 4: Código quando o jogador morre.

A função `hurt_player` implementa a lógica para causar dano ao jogador em um jogo. Esta função simula o impacto no personagem controlado pelo jogador (`o_player`) quando este sofre um ataque ou colisão. Ela gerencia a redução da vida (`global.playerhealth`), as

mudanças de estado, efeitos visuais e temporários (como invulnerabilidade), além de aplicar alterações visuais, como "screen shake" e "flash".

Funcionamento do Código:

Condição de Invulnerabilidade (`invulnerable`):

A função inicia verificando se o jogador não está em um estado de invulnerabilidade (`o_player.invulnerable <= 0`). Caso contrário, o jogador não sofre dano ou alterações.

Redução de Vida:

O dano causado é subtraído de `global.playerhealth`, garantindo que o valor mínimo de saúde seja 0 (usando a função `max`).

Verificação de Morte:

Se a vida do jogador for maior que 0:

O jogador entra em um estado de "bonk" (`playerstate_bonk`) que representa provavelmente uma reação ao dano recebido.

O jogador é "empurrado" na direção oposta ao impacto, com uma força determinada pelo parâmetro `_force`.

Aplicam-se efeitos visuais, como:

Agitação da tela (`screen_shake`).

Flash de dano (definido pela variável `flash` e o shader `flash_shader`).

O jogador entra em um estado temporário de invulnerabilidade (`invulnerable = 60`), impedindo danos imediatos subsequentes.

Se a vida do jogador for 0 ou menor:

O estado do jogador é alterado para `playerstate_dead`, indicando que o personagem morreu.

Parâmetros da Função:

`_direction`: Direção do impacto.

`_force`: Intensidade da força aplicada no jogador.

`_damage`: Quantidade de dano a ser infligido.

Pontos Fortes:

Modularidade:

A função encapsula toda a lógica de aplicação de dano, o que facilita sua reutilização em diferentes partes do jogo.

Tratamento Completo de Situações:

Inclui lógica para tanto o estado de sobrevivência quanto o estado de morte do jogador.

Gerencia invulnerabilidade, garantindo que o jogador tenha um intervalo para reagir antes de sofrer mais danos.

Imersão e Feedback Visual:

A presença de elementos como "screen shake" e "flash" melhora a experiência visual e o feedback para o jogador ao sofrer dano.

Manutenção da Vida do Jogador:

Uso da função max evita que a vida vá para valores negativos.

Função Ataque do Jogador

```
1
2 + function attack_slash(){
27
28 + function attack_spin(){
33
34 + function calc_attack(argument0){
62
63 + function hurt_enemy(_enemy,_damage,_source,_knockback){
92
93
94
95
```

Fig. 5: Código quando o jogador ataca.

Função attack_slash()

Essa função lida com a mecânica de um ataque do tipo "slash" (corte).

Principais funcionalidades:

Início do ataque:

Verifica se o sprite atual não é splayeattack_slash. Caso não seja, define esse sprite, reinicia o índice da animação (image_index) e cria/limpa a lista de objetos atingidos (hitby_attack).

Chamada à função calc_attack:

Calcula o efeito do ataque utilizando a função calc_attack(splayer_attackslashhb).

Animação e estado:

Chama playeran_sprite() para atualizar a animação.

Após o fim da animação (animation_end), o estado do jogador é alterado para playerstate_free, indicando que ele está livre para outra ação.

Função attack_spin()

Essa função está vazia e parece ser um esqueleto para um ataque de tipo "spin" (giro). Ainda não possui lógica implementada.

Função `calc_attack(argument0)`

Essa função realiza o cálculo e a execução de um ataque baseado na colisão com outros objetos.

Definição de máscara de colisão:

Atribui `argument0` à propriedade `mask_index`, determinando a área do ataque.

Detecção de colisões:

Utiliza `instance_place_list` para detectar todos os objetos na área de colisão. Os resultados são armazenados na lista temporária `hitby_attacknow`.

Iteração sobre os objetos atingidos:

Para cada objeto atingido:

Verifica se já foi adicionado à lista de objetos atingidos (`hitby_attack`).

Caso contrário, adiciona-o à lista e aplica os efeitos do ataque.

Aplicação de dano:

Se o objeto for um inimigo (`p_enemy`), aplica dano por meio da função `hurt_enemy` e executa um script adicional se definido (`entityhit_script`).

Limpeza de recursos:

Após o processamento, a lista temporária `hitby_attacknow` é destruída para evitar vazamentos de memória.

Função `hurt_enemy(_enemy, _damage, _source, _knockback)`

Essa função aplica dano e efeitos a inimigos atingidos.

Redução de vida do inimigo:

Se o estado do inimigo não for DIE, reduz sua vida (enemy_hp) com base no dano recebido.

Verificação do estado do inimigo:

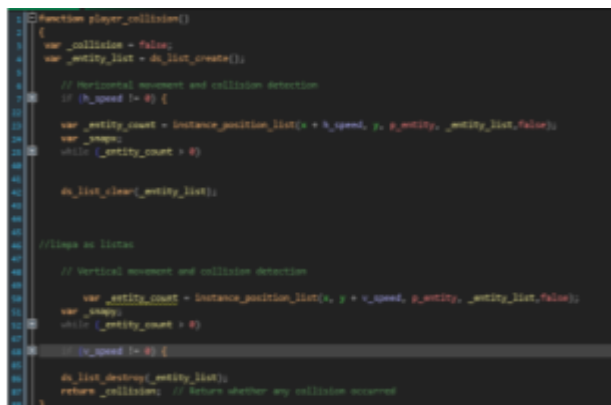
Caso a vida do inimigo (enemy_hp) seja menor ou igual a 0, altera o estado para DIE.

Caso contrário, muda o estado para HURT e salva o estado anterior.

Efeito de "knockback":

Calcula a direção do impacto e ajusta a posição do inimigo para simular o recuo.

colisões do Jogador



```

function player_collision()
{
    var _collision = false;
    var _entity_list = ds_list_create();

    // Horizontal movement and collision detection
    if (h_speed != 0) {
        var _entity_count = distance_position_list(x + h_speed, p, _entity_list, false);
        var _vmap;
        while (_entity_count > 0)
        {
            ds_list_destroy(_entity_list);

            //links as listas
            // Vertical movement and collision detection
            var _entity_count = distance_position_list(x, y + v_speed, p, _entity_list, false);
            var _vmap;
            while (_entity_count > 0)
            {
                ds_list_destroy(_entity_list);
                return _collision; // Return whether any collision occurred
            }
        }
    }
}

```

Fig. 6: Código das colisões do jogo.

Este código implementa uma lógica de detecção e resposta a colisões para um jogador ou entidade dentro de um ambiente bidimensional.

A função player_collision trata tanto de movimentações horizontais quanto verticais, verificando colisões com

objetos no mapa (collision_map) e com outras entidades (p_entity).

Aqui está um detalhamento do funcionamento, aspectos positivos e possíveis melhorias do código.

Criação e Inicialização de Variáveis

A variável _collision é usada como um indicador para determinar se houve colisão durante a execução da função.

_entity_list é uma lista dinâmica (ds_list) criada para armazenar entidades detectadas nas proximidades do jogador.

Detecção e Resposta à Colisão Horizontal

O código verifica inicialmente se existe movimento horizontal ($h_speed \neq 0$).

Utiliza a função `place_meeting` para prever se a próxima posição horizontal do jogador causará colisão com o mapa.

Caso detectada, ajusta a posição horizontal do jogador para a borda da tile (através de um `while` que aproxima o jogador do ponto de colisão) e zera a velocidade horizontal (h_speed).

Caso contrário, aplica o movimento horizontal ao jogador.

Adicionalmente, verifica se há colisões com outras entidades (`p_entity`) utilizando `instance_position_list`. Caso haja, a posição é ajustada para não sobrepor a entidade.

Detecção e Resposta à Colisão Vertical

Similar ao processo horizontal, verifica colisões no eixo vertical caso exista velocidade nesse sentido ($v_speed \neq 0$).

Realiza ajustes à posição vertical do jogador ao encontrar colisões com o mapa ou entidades próximas, ajustando a borda para evitar sobreposição.

Destruição e Limpeza de Recursos

Ao final da função, a lista dinâmica `_entity_list` é destruída para liberar memória.

Retorno

A função retorna `_collision`, indicando se qualquer tipo de colisão ocorreu durante a execução.

Clareza e Organização

O código é organizado de maneira lógica, separando claramente as verificações horizontais e verticais.

O uso de funções específicas como `place_meeting` e `instance_position_list` facilita a compreensão do objetivo de cada bloco de código.

Trabalho com Entidades

A implementação inclui detecção de colisões não apenas com o mapa, mas também com entidades dinâmicas, ampliando a funcionalidade.

Prevenção de Sobreposição

A lógica ajusta a posição do jogador de forma precisa para evitar sobreposição com tiles ou entidades.

Gerenciamento de Recursos

O uso de `ds_list_destroy` e `ds_list_clear` garante que recursos temporários não fiquem ocupando memória desnecessariamente após o uso.

transição de níveis do Jogador

```
1
2 function room_transition(){
3     //transicao de sala
4     if(!instance_exists(o_transition))
5     {
6         with(instance_create_depth(0,0,-9999,o_transition))
7         {
8             type = argument[0];
9             target = argument[1];
10        }
11    }else show_debug_message("Trying to transition while transition is happening");
12
13 }
```

Fig. 7: Código das transições de sala .

O código apresenta uma função chamada `room_transition`, cujo objetivo principal é realizar uma transição entre salas (ou níveis) dentro de um jogo.

Essa função parece ser implementada na linguagem GML (GameMaker Language), frequentemente utilizada em projetos desenvolvidos no GameMaker Studio.

Verificação de Instância

A função começa verificando se uma instância do objeto `o_transition` não existe no jogo, usando o método `instance_exists(o_transition)`.

Caso não exista, a função cria uma nova instância desse objeto usando `instance_create_depth(0,0,-9999,o_transition)`.

Definição de Propriedades

Após criar a instância de `o_transition`, as propriedades `type` e `target` do objeto recém-criado são configuradas com os valores fornecidos pelos argumentos `argument[0]` e `argument[1]` da função.

Prevenção de Conflitos

Caso uma instância de `o_transition` já esteja presente no jogo, a função exibe uma mensagem de depuração (`show_debug_message`), indicando que já há uma transição em andamento e impedindo a criação de uma nova.

Função Principal:

`room_transition()` é a função principal, responsável por lidar com a transição de sala.

Objeto `o_transition`:

É o objeto que, aparentemente, gerencia a transição de salas. Não há informações no código sobre a implementação de `o_transition`, mas espera-se que ele seja responsável por animar ou processar a transição.

Uso de Argumentos:

`argument[0]` e `argument[1]` são usados para definir o tipo de transição e o destino da transição (provavelmente a sala para onde o jogo deve ir).

Métodos Utilizados:

`instance_exists(o_transition)`: Verifica se a instância de um objeto já está presente no jogo.

`instance_create_depth(0,0,-9999,o_transition)`: Cria uma nova instância do objeto `o_transition` em uma profundidade (`depth`) específica.

show_debug_message(): Exibe mensagens para depuração.

Diálogo de texto do Jogador

```
3 //respostas
4 function new_textbox(){
5
6     var _obj;
7     if (instance_exists(o_text)) _obj = otext_queued; else _obj = o_text;
8     with (instance_create_layer(0,0,"Instances",_obj))
53 with (o_player)
61
62 }
```

Fig. 8: Código do diálogo de texto .

A função, new_textbox(), cria uma nova caixa de texto no jogo com algumas opções adicionais, como definir mensagens e respostas associadas. Vou quebrar o código em partes e explicar cada seção.

Ele cria uma instância de um objeto (o_text ou otext_queued) e define seu comportamento com base nos argumentos fornecidos à função. Também há suporte para respostas interativas e scripts associados a essas respostas.

Criação de Objetos

var _obj;

if (instance_exists(o_text)) _obj = otext_queued; else _obj = o_text;

with (instance_create_layer(0,0,"Instances",_obj))

A função começa verificando se uma instância do objeto o_text já existe no jogo. Se sim, a variável _obj recebe o objeto otext_queued, caso contrário, _obj recebe o objeto o_text.

Em seguida, a função cria uma nova instância desse objeto na camada "Instances" nas coordenadas (0,0) (canto superior esquerdo do jogo).

Atribuição de Variáveis

```
message = argument[0];
```

```
if(instance_exists(other)) origin_in = other.id else origin_in = noone;
```

```
if(argument_count > 1) background = argument[1]; else background = 1;
```

A variável `message` é configurada com o primeiro argumento passado para a função.

Se existir uma instância chamada `other`, a variável `origin_in` recebe seu ID, senão, recebe `noone` (nenhum).

O segundo argumento é usado para definir o fundo da caixa de texto. Caso não seja fornecido, o fundo recebe o valor 1 por padrão.

Processamento de Respostas

```
if(argument_count > 2)
```

```
{
```

```
    var _array = argument[2];
```

```
    for(var _i = 0; _i < array_length(_array); _i++)
```

```
    {
```

```
        responses[_i] = _array[_i];
```

```
    } for(var i = 0; i < array_length_1d(responses); i++)
```

```
    {
```

```
        var _markerpos = string_pos(":",_array[i]);
```

```
        responses_scripts[i] = string_copy(_array[i],1,_markerpos-1);
```

```
        responses_scripts[i] = real(responses_scripts[i]);
```

```
        responses[i] = string_delete(_array[i],1,_markerpos);
```

```
        breakpoint = 10;
```



```
}  
  
}else  
  
{  
  
    responses = [-1];  
  
    responses_scripts = [-1];  
  
}
```

Caso um terceiro argumento seja fornecido (um array de respostas), o código o processa.

O array de respostas é copiado para a variável `responses`.

Para cada resposta, o código tenta encontrar a posição do caractere ":", que é usado como um marcador. Tudo que vem antes do ":" é extraído e armazenado em `responses_scripts`, enquanto a parte após o ":" é a resposta final e é armazenada em `responses`.

O valor de `responses_scripts` é convertido em um número real usando a função `real()`.

O `breakpoint = 10`; sugere um ponto de interrupção para depuração, mas não é utilizado no restante do código.

Caso Não Haja Argumentos para Respostas

```
else{  
  
    responses = [-1];  
  
    responses_scripts = [-1];  
  
}
```

Explicação: Se não houver um terceiro argumento (ou seja, um array de respostas), as variáveis `responses` e `responses_scripts` são definidas como arrays contendo o valor -1.

Bloqueio do Jogador

with (o_player)

```
{  
  
    if(state != playerst_locked) {  
  
        laststate = state;  
  
        state = playerst_locked;  
  
    }  
  
}
```

Explicação:

O código acessa o objeto `o_player` e verifica se o estado do jogador não é `playerst_locked`.

Se o jogador não estiver bloqueado, o estado do jogador é salvo em `laststate` e o estado do jogador é alterado para `playerst_locked`.

Isso provavelmente impede que o jogador interaja com o jogo enquanto a caixa de texto estiver visível.

Resumo

A função `new_textbox()` cria uma caixa de texto no jogo, atribui uma mensagem, define o fundo e processa um conjunto de respostas, se fornecidas.

Também bloqueia a interação do jogador enquanto a caixa de texto estiver ativa, impedindo que o jogador realize ações até que a caixa de texto seja descartada ou fechada.

Animação do Jogador

A screenshot of a code editor showing a JavaScript function named `playeract_outan()`. The code is as follows:

```
2 // https://help.yoyogames.com/hc/en-us/articles/360005277377 for mor
3 function playeract_outan(){
4
5     state = playerstate_act;
6     sprite_index
7     if(argument_ function playerstate_act() -> Undefined ;
8     local_frame
9     image_index = 0;
10    playeran_sprite();
11 }
```

Fig. 9: Código da animação do jogador.

A função `playeract_outan` parece estar associada ao controle da animação de um jogador (ou personagem) no jogo. Vamos partir o código linha por linha para

entender melhor o que cada parte faz:

function playeract_outan(){

A função `playeract_outan` é declarada. Ela é uma função que, possivelmente, é chamada em momentos específicos do jogo, como ao iniciar uma nova animação para o jogador.

state = playerstate_act;

Aqui, o valor da variável `playerstate_act` é atribuído à variável `state`. Isso sugere que a função está controlando o estado do jogador, possivelmente para indicar que o jogador está realizando uma ação ou uma animação específica.

sprite_index = argument[0];

O valor do primeiro argumento da função (representado por `argument[0]`) é atribuído à variável `sprite_index`. O `sprite_index` é uma variável importante que define qual sprite (ou conjunto de imagens) será mostrado para o personagem no jogo. Isso significa que a função `playeract_outan` muda a animação do jogador para um sprite especificado quando chamada.

if(argument_count > 1) animationendscript = argument[1];

Esta linha verifica se a função foi chamada com mais de um argumento. Se sim, ela atribui o valor do segundo argumento (`argument[1]`) à variável `animationendscript`. Isso sugere que a função pode ter um script que deve ser executado quando a animação terminar, caso o segundo argumento seja fornecido.

arremesso, possivelmente de um objeto ou item que ele levantou.

A função é definida por `function player_throw()`, o que indica que ela é um bloco de código que pode ser chamado quando necessário, presumivelmente quando o jogador faz o arremesso de um objeto.

A função manipula variáveis globais e locais, com um foco no controle do arremesso de um objeto levantado.

Bloco with(global.lifted):

O código dentro deste bloco afeta o objeto ou entidade armazenada em `global.lifted`, o que provavelmente representa o objeto que o jogador está segurando ou levantando.

Variáveis que controlam o estado do objeto levantado são manipuladas:

`lifted = false`: O objeto deixa de estar sendo levantado.

`persistent = false`: A persistência do objeto (provavelmente sobre seu status) é desativada.

`thrown = true`: Indica que o objeto foi arremessado.

`z = 13`: Um valor fixo é atribuído à variável `z`, talvez representando a altura ou uma posição no eixo `Z` no momento do arremesso.

`throw_peak_h = z + 10`: A altura máxima do arremesso é calculada com base no valor de `z`.

`throw_distance = entity_throwdistance`: A distância máxima que o objeto pode ser arremessado é atribuída pela variável `entity_throwdistance` (possivelmente definida em outro lugar do código).

`throw_dis_travelled = 0`: Inicializa a distância percorrida com valor 0.

`throw_start_percent = (13 / throw_peak_h) * 0.5`: Calcula uma porcentagem inicial do arremesso baseado na altura máxima.

`throw_percent = throw_start_percent`: A porcentagem do arremesso é inicializada com o valor calculado.

`direction = other.direction`: Define a direção do arremesso com base na direção do jogador (representado por `other.direction`).

`xstart = x` e `ystart = y`: Armazena as coordenadas iniciais do arremesso (posição atual do jogador).

Chamada da Função `playeract_outan(splayer_lift)`:

Após configurar o estado do objeto levantado, a função `playeract_outan()` é chamada, provavelmente para executar a ação de arremesso, passando o parâmetro `splayer_lift`, que pode representar o sprite ou a animação associada ao ato de levantar o objeto.

Alterações no Estado Global:

`global.lifted = noone`: O objeto levantado deixa de ser referenciado, o que significa que o jogador não está mais segurando ou levantando nada.

As variáveis de sprite do jogador são alteradas:

`sprite_idle = s_player`: O sprite de inatividade (`idle`) do jogador é configurado para o padrão `s_player`.

`sprite_run = splayer_run`: O sprite de corrida do jogador é configurado para o padrão `splayer_run`.

Resumo das Funções:

A função `player_throw()` gerencia o processo de arremessar um objeto que o jogador tenha levantado.

Ela configura várias variáveis para controlar a altura, distância e direção do arremesso.

Depois de configurar o arremesso, a função altera o estado do objeto (de levantado para arremessado) e reinicia o status do jogador para os sprites padrão (idle e running).

```

1 if (!global.gamepaused)
2 {
3     depth = -bbox_bottom
4     if(lifted) && (instance_exists(o_player))
5     {
6         if(o_player.sprite_index != splayer_lift)
13     }
14     if(!lifted)
50 }
51 flash = max(flash-0.04,0);

```

Fig. 11: Código para o jogador atirar objectos

O código implementa a lógica de comportamento de um objeto ou entidade em um jogo desenvolvido com GameMaker Language (GML). Ele lida com as condições e estados associados a: Suspensão no ar (lifted), Arremesso (thrown), Gravidade (grav),

if (!global.gamepaused)

O bloco só será executado se o jogo não estiver pausado, o que garante que o estado da entidade não será atualizado enquanto o jogo estiver em pausa.

depth = -bbox_bottom;

A profundidade da entidade (depth) é definida como o valor negativo do limite inferior da caixa delimitadora (bbox_bottom).

Isso pode ser usado para organizar a ordem de desenho no jogo, colocando objetos "mais abaixo" na tela com maior prioridade de desenho.

if(lifted) && (instance_exists(o_player))

Verifica se a entidade está sendo levantada (lifted == true) e se a instância do jogador (o_player) existe.

Se o jogador não estiver usando um sprite específico (splayer_lift), a posição da entidade é sincronizada com a posição do jogador (x, y) e a altura (z) é definida como 13.

A profundidade da entidade é ajustada para estar logo acima do jogador (o_player.depth -1).

if(!lifted)

Neste bloco, existem dois cenários principais: entidade arremessada e entidade caindo de volta ao chão.

throw_dis_travelled = min(throw_dis_travelled+3,throw_distance);

A distância percorrida pelo arremesso (throw_dis_travelled) é incrementada em 3, mas limitada pelo valor máximo (throw_distance).

x = xstart + lengthdir_x(throw_dis_travelled, direction);

y = ystart + lengthdir_y(throw_dis_travelled, direction);

Calcula a posição atual baseada na distância percorrida e na direção do arremesso, utilizando funções trigonométricas.

if(tilemap_get_at_pixel(collmap, x, y) > 0)

Verifica se o objeto colide com um tile do mapa (collmap). Em caso de colisão, o arremesso é interrompido (thrown = false) e a gravidade é ajustada (grav = 0.1).

z = throw_peak_h * sign(throw_percent * pi);

Calcula a altura com base na porcentagem de arremesso completada (throw_percent) e no pico da altura do arremesso (throw_peak_h).

if(throw_distance == throw_dis_travelled)

Quando a distância percorrida alcança o máximo permitido (throw_distance), o arremesso é encerrado (thrown = false). Se a entidade estiver marcada para destruir após o arremesso (entity_throwbreak), ela é destruída.

z = max(z - grav, 0);

grav += 0.1;

A altura da entidade é reduzida gradualmente pela gravidade (grav), que aumenta ao longo do tempo.

if(z == 0) && (entity_throwbreak) instance_destroy();

Quando a entidade atinge o chão (altura z igual a 0) e está marcada como destrutível após o arremesso, ela é destruída.

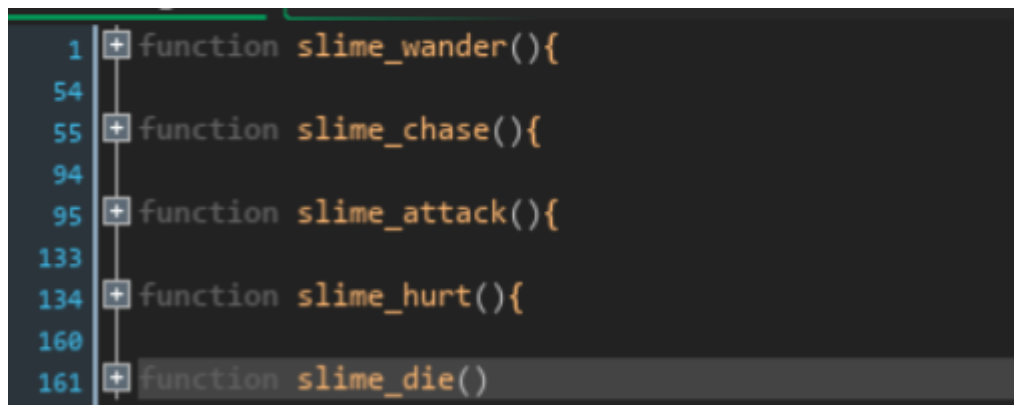
grav = 0.1;

Se a entidade já está no chão, a gravidade é redefinida.

flash = max(flash-0.04,0);

O valor da variável flash é reduzido progressivamente até atingir zero.

Essa lógica pode estar relacionada a um efeito visual de piscar ou brilho, indicando alguma interação ou estado especial da entidade.

A screenshot of a code editor showing GML functions for a slime enemy. The code is as follows:

```
1  + function slime_wander(){
54
55 + function slime_chase(){
94
95 + function slime_attack(){
133
134 + function slime_hurt(){
160
161 + function slime_die()
```

Fig. 12: Código do inimigo slime

O código fornecido define o comportamento de uma entidade de "slime" (um inimigo) em um jogo, implementado em GML (GameMaker Language). Ele utiliza estados para modelar as ações do slime e controla suas animações, movimentos e interações com o jogador ou o ambiente. Abaixo está uma análise detalhada e estruturada de cada parte:

O código implementa uma máquina de estados para controlar o comportamento do slime, incluindo os seguintes estados principais:

slime_wander: Movimento aleatório enquanto patrulha.

slime_chase: Persegue o jogador quando este está dentro do raio de detecção

slime_attack: Ataca o jogador quando está próximo o suficiente.

slime_hurt: Comportamento ao ser atingido.

slime_die: Comportamento ao ser derrotado.

Cada função manipula variáveis relacionadas à posição, direção, velocidade e animação para implementar os comportamentos desejados.

slime_wander()

Objetivo: Controla o movimento aleatório do slime enquanto está em estado de patrulha.

Lógica:

Se o slime chegou ao destino atual ou excedeu o tempo limite, ele para, interrompe a animação de movimento e escolhe um novo destino aleatório dentro de um ângulo entre -45° e 45°.

Caso contrário, incrementa o contador de tempo, calcula a direção e ajusta a velocidade horizontal (h_speed) e vertical (v_speed) para se mover em direção ao destino.

Verifica colisões com paredes e interrompe o movimento se necessário.

Realiza uma verificação periódica para determinar se o jogador está dentro do raio de aggro, mudando para o estado de perseguição (ENEMYSTATE.CHASE) se for o caso.

slime_chase()

Objetivo: Persegue o jogador ao identificá-lo no raio de aggro.

Lógica:

Define o destino (xto, yto) como a posição do jogador e calcula a direção.

Move-se em direção ao jogador com velocidade limitada a enemy_speed.

Ajusta a escala horizontal da imagem (image_xscale) para refletir a direção do movimento.

Se o slime estiver perto o suficiente do jogador, muda para o estado de ataque (ENEMYSTATE.ATTACK), iniciando a animação de ataque.

slime_attack()

Objetivo: Realiza o ataque ao jogador, movendo-se para uma posição próxima e realizando um salto/impacto.

Lógica:

Controla a velocidade do movimento dependendo da fase da animação (ex.: parado enquanto prepara o salto).

Checa a distância até o destino. Se for pequena, acelera a animação de aterrissagem.

Move-se em direção ao destino com base na direção e velocidade calculadas.

Ao completar o ataque, retorna ao estado de perseguição ou espera.

slime_hurt()

Objetivo: Reage a um dano recebido, movendo-se para longe.

Lógica:

Calcula a direção oposta e move o slime com base na velocidade configurada.

Caso colida com uma parede, interrompe o movimento.

Retorna ao estado anterior ou ao estado de perseguição caso estivesse atacando.

slime_die()

Objetivo: Executa a lógica de morte do slime, incluindo animação e remoção da instância.

Lógica:

Move-se para o destino final enquanto executa a animação de morte.

Após o término da animação, destrói a instância do slime.

2 Conclusão - análise crítica global da execução do projeto

O desenvolvimento do projeto "The Bleedin' Globe of Nigatsumi" foi uma experiência enriquecedora, tanto em termos técnicos quanto pessoais. Durante a execução, tive a oportunidade de explorar várias áreas do desenvolvimento de jogos, como programação, design de personagens e mecânicas de jogabilidade, enquanto enfrentava desafios que me ajudaram a crescer como criador.

O projeto permitiu que eu consolidasse habilidades adquiridas ao longo da formação, aplicando-as em um contexto prático e criativo. Embora tenha sido um processo exigente, o resultado final reflete o meu empenho e paixão por criar algo único.

1.7 Dificuldades

Gerenciamento de Tempo:

Durante o desenvolvimento, foi difícil equilibrar o tempo entre diferentes etapas do projeto, como programação, design gráfico e teste de jogabilidade, além das demais responsabilidades acadêmicas.

Curva de Aprendizado:

Algumas ferramentas e técnicas utilizadas no projeto, como animação avançada ou otimização de desempenho, exigiram aprendizado adicional e pesquisa para serem implementadas de forma eficaz.

Complexidade do Design:

Integrar diferentes mecânicas e garantir que todas funcionassem de maneira coesa foi um grande desafio, especialmente para evitar bugs e problemas de balanceamento no jogo.

1.8 Problemas e obstáculos

Bugs e Erros:

Durante o processo de programação, vários bugs surgiram, desde erros simples de lógica até comportamentos inesperados em mecânicas de movimento e colisão.

Limitações Técnicas:

Algumas ideias planeadas inicialmente precisaram ser simplificadas ou descartadas devido às limitações de tempo, ferramentas ou recursos disponíveis.

Feedback dos Testes:

Durante os testes iniciais, jogadores apontaram problemas relacionados à dificuldade do jogo e à clareza das instruções, exigindo ajustes para melhorar a experiência do usuário.

1.9 Soluções encontradas

Planejamento e Priorização:

Para lidar com a gestão do tempo, estabeleci um cronograma detalhado e priorizei tarefas essenciais, garantindo que as funcionalidades principais do jogo fossem concluídas primeiro.

Pesquisa e Aprendizado:

Dediquei tempo para estudar e consultar tutoriais e documentações, o que foi crucial para superar dificuldades técnicas e aprender novas ferramentas e métodos de desenvolvimento.

Iteração e Feedback:

Após cada rodada de testes, implementei melhorias com base no feedback recebido, ajustando elementos como controles, dificuldade e design de níveis para criar uma experiência mais fluida e agradável para os jogadores.

Bibliografia

Anexos

