Universidade Federal Rural do Rio de Janeiro

Curso Ciência da Computação

Nova Iguaçu

07/04/2025

Davi Anderson do Carmo de Souza Mat.: 20230024145

Hélio Henrique Lemos Magalhães Mat:. 20230000288

Mariana Ferreira Ferrarez Ribeiro Mat:. 20230001113

Pedro Henrique Ribeiro de Araujo Guedes Mat.: 20230000279

Pedro Vitor de Souza Dantas Mat:. 20230004204

Jogo CG

Disciplina: TM421 - COMPUTAÇÃO GRÁFICA- T01 (2025.1)

Professor: Bruno José Dembogurski

Game Design Document (GDD)

1. Informações Gerais

- **Nome do Jogo: Boa noite, Geraldo.**
- **Gênero:** Sobrevivência
- **Plataforma-Alvo:** PC
- **Ferramentas Utilizadas:** Unity, C++, C#, Blender, Vs Code, Open GL
- **Equipe:**
    - Programador: Davi Anderson, Hélio Lemos, Pedro Vitor
    - Artista: Pedro Guedes e Mariana Ribeiro
    - Designer: Pedro Guedes e Mariana Ribeiro.
    - Enredo: Mariana Ribeiro.

2. Conceito do Jogo Descrição Geral:

Geraldo é um homem comum que se muda para uma nova casa em busca de recomeço. No entanto, logo após sua chegada, eventos estranhos começam a acontecer. Objetos mudam de lugar, sussurros surgem à noite, e a atmosfera da casa se torna cada vez mais opressora a cada vez que ele dorme.

Com o passar dos dias, Geraldo descobre que a casa é mal-assombrada e parece estar viva. Preso em um ciclo de pesadelos e aparições, ele deve explorar cada cômodo, resolver puzzles sobrenaturais e coletar itens para tentar sobreviver aos horrores que espreitam na escuridão.

Conforme a casa revela sua verdadeira natureza, ele percebe que talvez escapar não seja uma opção. Na fase final, mesmo após completar todos os puzzles de fuga, Geraldo descobre que a porta de saída o leva de volta para dentro — e que ele nunca realmente deixou a casa.

O jogo mergulha o jogador em um ciclo psicológico de medo, onde a sanidade se desfaz junto com a realidade.

- **Objetivo Principal do Jogador:**

  Sobreviver aos eventos sobrenaturais da casa, desvendar sua maldição e encontrar uma forma de escapar — apenas para descobrir que a fuga é impossível.

- **Diferenciais:** A casa como personagem viva, puzzles sobrenaturais integrados à narrativa e Progressão de assombração ligada ao sono.

3. Aspectos Visuais (Computação Gráfica)

- **Estilo Visual:** Horror

  A casa é escura, antiga, cheia de detalhes clássicos: móveis de madeira pesada, cortinas de veludo, quadros antigos, espelhos rachados e paredes com papel de parede desbotado.

  Ambientes apertados, claustrofóbicos e com pouca luz natural.

  O visual segue um tom realista, mas com leves distorções que representam a sanidade do personagem (ex: sombras que se mexem, rachaduras que somem e voltam, objetos que piscam e desaparecem).

- **Modelagem 3D:**
  - Quantidade de modelos: 2 (Fase 1: Casa Velha e Fase 2: Casa Destruída e mal assombrada).
  - Ferramentas de criação: Blender.
  - Nível de detalhamento: Moderado.
- **Texturização:**
  - Técnicas utilizadas: Texturização Procedural e Texturing Mapping
- **Iluminação e Sombras:**

    - Texturização Procedural e Texturing Mapping
- **Câmera:**
    - Tipo: Primeira pessoa
    - Comportamento: Fixo ao personagem

4. Mecânicas de Jogo

- **Movimentação:** Caminhar, correr e se esgueirar.
- **Interação:** Mover e coletar objetos, resolver puzzles e combinar itens.
- **IA:**

    Entidade Observadora: aparece em pontos fixos, desaparece se observada por muito tempo.

    Aparição do Sono: surge quando o jogador está próximo de dormir ou com sanidade baixa, perseguindo-o até ele usar um item específico.

    A IA monitora ações do jogador (como ruído, iluminação e tempo sem dormir).

    A dificuldade aumenta progressivamente de forma invisível — mais sustos, aparições, eventos paranormais e alteração nos cômodos da casa.

5. Implementação Técnica

Engine ou Biblioteca:

Engine Principal:

- **Unity** – Utilizada como motor gráfico principal, responsável por renderização, física, input e gerenciamento de cenas.

Linguagens Utilizadas:

- **C# ou C++** – Lógica do jogo, interações, controle de IA, HUD, sistema de sanidade e inventário.

- **ShaderLab (HLSL)** – Escrita de shaders personalizados para distorções visuais, efeitos sobrenaturais e sanidade.

APIs / Bibliotecas / Sistemas Auxiliares:

- **Unity Input System** – Para controles personalizáveis e detecção de gamepads, teclado e mouse.

- **Cinemachine** – Para controlar a câmera em primeira pessoa com transições suaves e efeitos como tremores.

- **Post-Processing Stack (Unity)** – Para efeitos visuais como bloom, grain, depth of field, vignette, color grading etc.

- **NavMesh (AI)** – Navegação de entidades dentro da casa, especialmente para a IA "Aparição do Sono".

- **Unity Timeline + Animator** – Para cutscenes leves, eventos de assombração e controle de transições visuais.

- **Unity Audio Mixer** – Para controlar ambientação sonora, eco, distorções auditivas conforme a sanidade.

- **TextMesh Pro** – Para HUD e interfaces com tipografia estilizada (ex: rabiscos, papel envelhecido).

- **Shader Graph (URP ou HDRP)** – Para desenvolvimento de superfícies e efeitos sobrenaturais com visual distorcido.

Sistemas Personalizados Desenvolvidos pela Equipe:

- **Sistema de Sanidade com Feedback Visual e Auditivo Dinâmico**

- **Gerador de Eventos Paranormais (modular e escalável)**

- **Sistema de Inventário com Combinação de Itens e Uso Contextual**

- **Gerenciador de IA com Comportamento Reativo (visão, som, sanidade)**

- **Sistema de Save baseado em tempo/sono (autosave em checkpoints invisíveis)**

Algoritmos Gráfico:

- **Iluminação Volumétrica:** Para criar neblina em cômodos, raios de luz entrando por janelas ou lanternas.

- **Pós-Processamento (Post-Processing Stack):** Para efeitos de distorção visual, aberração cromática, granulação, desfoque de movimento e mudança de cor com base na sanidade.

- **Sombras Dinâmicas e em Tempo Real:** Para criar tensão com movimentação de luzes e objetos.

- **Screen Space Reflections (SSR):** Para reflexos em espelhos quebrados e poças d’água.

- **Shader Graph (Unity):** Para criar superfícies que "respiram", rachaduras pulsantes, ou paredes que reagem ao jogador.

- **Normal Mapping & Parallax Occlusion Mapping:** Para aumentar o realismo de paredes antigas e objetos com relevo detalhado.

Performance e Otimização:

- **Occlusion Culling:** Para renderizar apenas o que está visível ao jogador, economizando processamento.

- **Level of Detail (LOD):** Modelos com menor complexidade para objetos distantes.

- **Baking de Iluminação Estática:** Para cômodos sem interação dinâmica de luz.

- **Object Pooling:** Para gerenciar sustos, aparições e partículas sem criar novos objetos em tempo real.

- **Compressão de Texturas e Áudio:** Para reduzir o tamanho do build e acelerar carregamentos.

- **Uso controlado de efeitos intensivos:** Como fumaça, reflexos e simulações físicas, limitados a eventos pontuais.

6. Interface e HUD

- **Elementos Visuais:**

  Indicador de sanidade: Representado por rachaduras e distorções na tela (como tremores, desfoque, sussurros audíveis e alteração de cores).

  Inventário rápido: Acessado por um menu lateral discreto, com slots limitados e visual semelhante a uma mochila ou diário antigo.

  Ícones contextuais: Aparecem apenas quando o jogador pode interagir com objetos (ex: mãos para mover, lupa para examinar).

  Notificações ambientais: Exibidas de forma integrada ao cenário — como rabiscos surgindo nas paredes, ou vozes sussurrando dicas.

- **Estilo de Interface:**

  Estética minimalista e imersiva: A interface evita elementos chamativos, priorizando o ambiente como principal forma de feedback.

  Visual envelhecido: Tipografia e texturas remetem a papel antigo e tinta borrada, reforçando o clima de mistério.

7. Progresso e Fases

Fases ou Dias: 2 dias

Progressão: Narrativa crescente com foco psicológico e simbólico

DIA 1 – A NEGAÇÃO

Objetivo Principal: Sobreviver à primeira noite e realizar tarefas básicas.

Eventos e Introdução de Mecânicas (Sutil):

SALA: Geraldo chega à casa com caixas de papelão empilhadas.

Diálogo interno: "Pequeno... mas vai servir. Só preciso de um lugar para descansar a cabeça."

COZINHA: Tarefa “Beber um copo d’água”. Uma figura feminina aparece na pia e desaparece.

Diálogo interno: "Preciso dormir. Já estou vendo coisas."

CORREDOR: Distorção visual súbita.

BANHEIRO: Reflexo distorcido no espelho durante tarefa de lavar o rosto.

QUARTO 2: barulho de algo se escondendo sob a cama.

Diálogo interno: "Chega. Cama. Agora."

QUARTO 1: Geraldo vai dormir. Fim do Dia 1.

DIA 2 – A ARMADILHA

Objetivo Principal: Encontrar a chave da porta da frente para tentar fugir.

Eventos e Mecânicas Ativas:

SALA: Porta da frente agora possui uma tranca estranha. Interagir causa queda de sanidade.

Diálogo interno: "O que é isso? Alguém me trancou aqui!"

COZINHA: A Mulher da Pia permanece de costas. Olhar direto causa sanidade cair.

QUARTO 2: Enigma no bilhete da cozinha leva ao quarto com A Criança Escondida.

QUARTO 1: O Bau é aberto com um cubo.

QUARTO 1: Jogador clica no Cubo e lembra de um certo evento.

8.Técnicas implementadas pela equipe:

- Inventário e Interação com Itens
- Visualização de IA
- Texturização Procedural

9.Core Gameplay:

1. **Exploração em Primeira Pessoa**
    - O jogador explora livremente a casa mal-assombrada em primeira pessoa, investigando ambientes escuros e detalhados.
2. **Resolução de Puzzles Ambientais**
    - Para avançar pelas fases e desvendar o mistério, o jogador precisa resolver enigmas — como abrir portas trancadas, descobrir senhas escondidas, manipular objetos no

ambiente e usar lógica.

3. **Gerenciamento de Sanidade**

    - O jogador tem um medidor de sanidade que diminui em certos momentos (assustadores, estressantes, loops temporais).

    - Baixa sanidade altera a percepção (visões, sons, objetos mudam de lugar), dificultando o progresso e aumentando o terror psicológico.

4. **Loop Temporal e Narrativa Fragmentada**

    - O tempo passa de acordo com a conclusão das missões

5. **Sistema de Crafting**

    - O jogador coleta itens e os combina para criar ferramentas ou resolver puzzles (ex: montar uma chave, criar uma armadilha ou distração).

6. **Evitar Entidades Sobrenaturais**

    - Em certos momentos, o jogador é perseguido ou ameaçado por entidades. Não pode combatê-las diretamente — deve se esconder, correr ou manipulá-las

Descrição das atividades de cada integrante:

Helio Lemos:

1. Sistema de Inventário

O sistema de inventário foi projetado para ser flexível e robusto, permitindo ao jogador coletar, visualizar e interagir com itens essenciais para a progressão e sobrevivência.

1.1. Arquitetura e Componentes Chave

A arquitetura é modular, dividida em scripts com responsabilidades claras:

- **ItemData.cs**: Define o que é um item (seus dados e propriedades).
- **InventorySlot.cs**: Representa um "espaço" no inventário, contendo um item e sua quantidade.
- **InventoryManager.cs**: O "cérebro" que gerencia a lógica de adicionar e remover itens.
- **CollectibleItem.cs**: Transforma objetos na cena em itens que podem ser coletados.
- **InventoryUIManager.cs**: Controla toda a interface visual do inventário.
- **InventorySlotUI.cs**: Controla a aparência de um único slot visual.

1.2. Estrutura de Dados (ItemData.cs)

Para definir os itens, utilizamos ScriptableObjects. Isso nos permite criar "assets" de itens (como "Pilha Antiga", "Vinagre", "Chave Enferrujada") diretamente no editor do Unity.

- **Propósito:** O ItemData.cs armazena todos os atributos de um item, como itemName, description, o icon (Sprite 2D para a UI), isStackable (se pode ser empilhado), maxStackSize, e um enum ItemType para categorizá-lo (ex: KeyItem, Consumable).
- **Interação:** Este asset é a fonte de informação para todos os outros scripts do sistema.

1.3. Lógica Central (InventoryManager.cs)

Este script é o coração do sistema, gerenciando a lista de InventorySlot que o jogador possui.

- **Singleton:** Utiliza um padrão Singleton (Instance) para ser facilmente acessível por qualquer outro script que precise interagir com o inventário.
- **Funções Principais:**
    - **AddItem()**: Contém a lógica para adicionar um item. Primeiro, verifica se o item é empilhável e se já existe no inventário para adicionar à pilha (AddQuantity). Se não, verifica se há espaço disponível (items.Count < maxSlots) antes de criar um novo InventorySlot.
    - **RemoveItem()**: Localiza um item no inventário, diminui sua quantidade ou remove o slot por completo se a quantidade chegar a zero.
    - **HasItem()**: Um método auxiliar crucial para os puzzles, que verifica se o jogador possui uma determinada quantidade de um item.
- **Comunicação via Evento:** O InventoryManager possui um evento public event Action OnInventoryChanged. Este evento é disparado sempre que o inventário é modificado. O InventoryUIManager "escuta" este evento e atualiza a interface visual, mantendo a lógica e a UI desacopladas.

1.4. Interface do Usuário (InventoryUIManager.cs)

Este script gerencia toda a parte visual do inventário.

- **Layout:** Utiliza um Canvas com um InventoryPanel principal, que contém um SlotsContainer e um DetailsPanel. Para o layout dos slots, foi utilizado um Grid Layout Group para organizar 6 slots visuais de forma fixa e responsiva à esquerda do painel. A responsividade e o tamanho fixo em relação à imagem de fundo foram garantidos pelo uso combinado dos componentes Canvas Scaler e Aspect Ratio Fitter.
- **Funcionalidades:**
    - **ToggleInventory()**: Chamado ao pressionar a tecla "I", este método ativa/desativa o InventoryPanel e pausa/despausa o jogo (Time.timeScale).

    - **UpdateUI()**: É chamado pelo evento OnInventoryChanged. Ele percorre a lista de slots visuais e, para cada um, verifica se há um item correspondente no InventoryManager para exibir, chamando os métodos do InventorySlotUI.
    - **Painel de Detalhes:** Ao clicar em um InventorySlotUI (que possui um componente Button), a função OnSlotClicked() é chamada. Esta, por sua vez, chama DisplayItemDetails() no InventoryUIManager, que atualiza a imagem e a descrição do item no painel de detalhes à direita. Clicar em um slot vazio chama ClearItemDetails() para limpar o painel.

2. Sistema de Sanidade e Efeitos

Para refletir o estado mental de Geraldo, um sistema de sanidade foi implementado como a "barra de vida" do jogador.

2.1. Lógica Central (PlayerSanity.cs)

- **Gerenciamento:** Este script, anexado ao jogador, utiliza um Singleton e gerencia a variável CurrentSanity, que é mantida entre 0 e maxSanity usando Mathf.Clamp.
- **Dreno Contínuo:** No método Update(), a sanidade diminui gradualmente com o tempo (-sanityDrainPerSecond * Time.deltaTime), aumentando a tensão do jogo. Um booleano isDraining permite desativar essa perda em áreas seguras.
- **Comunicação via Evento:** Assim como o inventário, ele possui um evento public static event Action<float, float> OnSanityChanged que é disparado sempre que a sanidade muda.

2.2. Feedback Visual e Auditivo

- **Barra de Sanidade (SanityBarUI.cs):** Um Slider da UI do Unity "escuta" o evento OnSanityChanged. Quando o evento é disparado, o método UpdateSanityBar() atualiza o valor do slider (currentSanity / maxSanity), refletindo visualmente o estado da sanidade em tempo real. A ordem de execução dos scripts foi ajustada no Project Settings para garantir que o PlayerSanity inicialize antes da UI.

- **Efeitos de Pós-Processamento (SanityEffects.cs):** Para criar a sensação de insanidade, um sistema de pós-processamento foi configurado. Outro script "escuta" o OnSanityChanged e ajusta dinamicamente a intensidade de efeitos como Vignette (escurecimento das bordas) и Chromatic Aberration (distorção de cores) com base na porcentagem de sanidade do jogador.

3. Tela de Fim de Jogo (Game Over)

Para dar um desfecho à falha do jogador, uma condição de fim de jogo foi implementada de forma simples e direta, sem a necessidade de carregar uma nova cena.

- Implementação: Um GameOverPanel (um painel de UI preto que cobre toda a tela com um texto) foi criado dentro do Canvas principal e mantido desativado.
- Gatilho: No script PlayerSanity.cs, o método ChangeSanity() verifica a cada alteração se CurrentSanity <= Mathf.Epsilon (uma forma segura de verificar se o valor é zero ou muito próximo de zero).
- Ação: Se a condição for verdadeira, um método Morrer() é chamado. Ele pausa o jogo (Time.timeScale = 0), libera o cursor do mouse e ativa o GameOverPanel (gameOverPanel.SetActive(true)), mostrando a tela de fim de jogo sobre a cena atual.

4. Modularidade e Escalabilidade para o Futuro

Um dos principais objetivos durante a implementação foi criar sistemas que pudessem ser facilmente expandidos sem a necessidade de reescrever a lógica central. A seguir, detalhamos como cada sistema foi preparado para o crescimento futuro do jogo.

4.1. Adição de Novos Itens (Sistema de ScriptableObject)

A decisão de usar ScriptableObjects para o script ItemData.cs foi fundamental para a escalabilidade do inventário.

- Fundamento: Cada tipo de item no jogo é um *asset* independente no projeto. Isso desacopla completamente a definição de um item da lógica do jogo.
- Processo de Expansão: Para adicionar um novo item ao jogo (seja uma chave, um consumível, um documento ou uma ferramenta de puzzle), não é necessário escrever uma única linha de código novo. O processo é inteiramente visual e feito no editor do Unity:
    1. No menu do projeto, navega-se para Create > Inventory > Item Data.
    2. Um novo asset de item é criado. No Inspector, preenchem-se os campos: nome, descrição, ícone, se é empilhável, seu tipo, etc.
    3. Para colocar o item no mundo, cria-se um CollectibleItem na cena e arrasta-se este novo asset de ItemData para o seu campo correspondente.
- Vantagem: Este fluxo de trabalho permite que designers de níveis ou outros membros da equipe adicionem dezenas de itens ao jogo de forma rápida e segura, sem a intervenção de um programador. O sistema de inventário, UI e puzzles já está pronto para reconhecer e trabalhar com qualquer novo ItemData criado.

4.2. Interação e Uso de Itens (Puzzles e Consumíveis)

O sistema foi construído com a interação em mente, criando padrões reutilizáveis.

- Puzzles Escaláveis: O script Door.cs, com sua lista requiredItems, serve como um modelo para qualquer obstáculo que exija itens. Para criar um novo puzzle de "item-chave" (seja um baú trancado, um cofre ou uma máquina quebrada), basta criar um novo script que siga o mesmo padrão: ter uma lista de ItemDatas requeridos e usar InventoryManager.Instance.HasItem() para verificar se o jogador os possui.
- Sistema de "Uso" de Itens: O método public virtual void Use() no ItemData.cs é a "porta de entrada" para a funcionalidade de consumíveis e itens ativos. Para implementar o "Chá Calmante"

(do GDD) e fazê-lo restaurar sanidade, o caminho já está preparado:

1. No ItemData do Chá, marcar isUsable = true, affectsSanity = true e definir um sanityChange positivo.
2. Na UI do inventário, quando o jogador clica em um item e escolhe "Usar", o InventoryUIManager chamaria item.Use().
3. A lógica dentro do método Use() (ou em um sistema que o consome) então chamaria PlayerSanity.Instance.ChangeSanity(...), aplicando o efeito de cura de sanidade. O InventoryManager então removeria o item.

- Vantagem: Essa arquitetura cria um "contrato" claro: qualquer item pode se tornar "usável" ou parte de um puzzle. A lógica central não precisa ser alterada; apenas a lógica específica do item ou do objeto interativo é adicionada.

4.3. Sistema de Dano à Sanidade (Inimigos e Eventos)

O script PlayerSanity.cs foi intencionalmente projetado com um ponto de entrada único e público para alterar a sanidade, tornando-o um "serviço" robusto para o resto do jogo.

Interface Simples: Para que um inimigo, um evento paranormal (um *jumpscare*), ou uma área de escuridão causem dano à sanidade do jogador, o script responsável por essa ameaça precisa de apenas uma linha de código:
C#
// Exemplo no script de um Inimigo ou Gatilho de Susto

// Se a condição de dano for atendida (ex: jogador entrou no campo de visão)...

if (PlayerSanity.Instance != null)

{

PlayerSanity.Instance.ChangeSanity(-15.0f); // Causa 15 de dano à sanidade

}

-
- Vantagem: Graças ao padrão Singleton (PlayerSanity.Instance), qualquer sistema do jogo pode acessar e modificar a sanidade do jogador de forma fácil e segura. Isso torna a implementação de novas fontes de terror e perigo extremamente simples. A lógica de atualizar a UI (a barra de sanidade) e os efeitos visuais de pós-processamento já está pronta para reagir instantaneamente a qualquer uma dessas chamadas, graças ao sistema de eventos OnSanityChanged. Não é preciso conectar manualmente cada inimigo à UI.

Pedro Vitor:

Luz/lanterna

A construção da luz foi feita usando pontos de luz da própria unity porém o funcionamento foi feito por scripts feitos na linguagem C# por questão de escolha do grupo. Os códigos foram construídos a partir de documentação da própria linguagem, LLM e pesquisas na internet.
As luzes dos cômodos possuem um funcionamento de se manterem piscando para que a imersão seja de uma melhor forma.
A lanterna segue a câmera em todas as direções para ter um melhor gameplay. Outro ponto também é que a lanterna possui uma bateria que está ligada ao objeto BarraBateria da unity que serve para representar a bateria na UI para o jogador.

Tela início

A tela inicial do jogo possui apenas um botão, o que inicia o jogo, foi uma escolha do grupo para ser mais simples, o vídeo usado foi construído pela integrante Mariana, já a tela de início foi em dupla comigo.
Ao clicar no botão “play” é disparado um evento que reproduz um efeito musical e logo em seguida é carregado a cena principal e então é iniciado o jogo e com ele sua gameplay.
A tela inicial foi construída a partir de uma nova cena do projeto.

Tela final

A tela final foi construída a parte de LLM a parte das pessoas, utilizamos o mesmo esquema que foi utilizado na tela do menu. Uma nova cena e a partir dela as interações. Nesse caso se faz necessário toda uma lógica, assim como as outras presentes no jogo para que o final ocorra, e interagindo com um item, o final é apresentado.

Personagem seguindo

Em um dos trechos do jogo, ao entrar no quarto o jogador é perseguido por um dos mobs.
O mesmo mob é paralizado ao ter a luz da lanterna mirado no mesmo. Para isso utilizei o componente mesh de IA da própria unity. Com isso, setei a região em que o mob pode se mover, e quando o personagem entra nessa área, a perseguição começa, para que o mob pare, utilizei de um ponto com colisão que quando esse ponto é atingido a movimentação do mob para.

Personagem cozinha

O mob da cozinha tem uma mecânica que, ao chegar em uma distância próxima e ser observado, o personagem começa receber um tipo de dano, utilizei a mesma técnica do ponto e utilizei um raio de distância para que as configurações fossem aplicadas.

- **Iluminação e Sombras:**
    - Texturização Procedural e Texturing Mapping

Pedro Guedes:

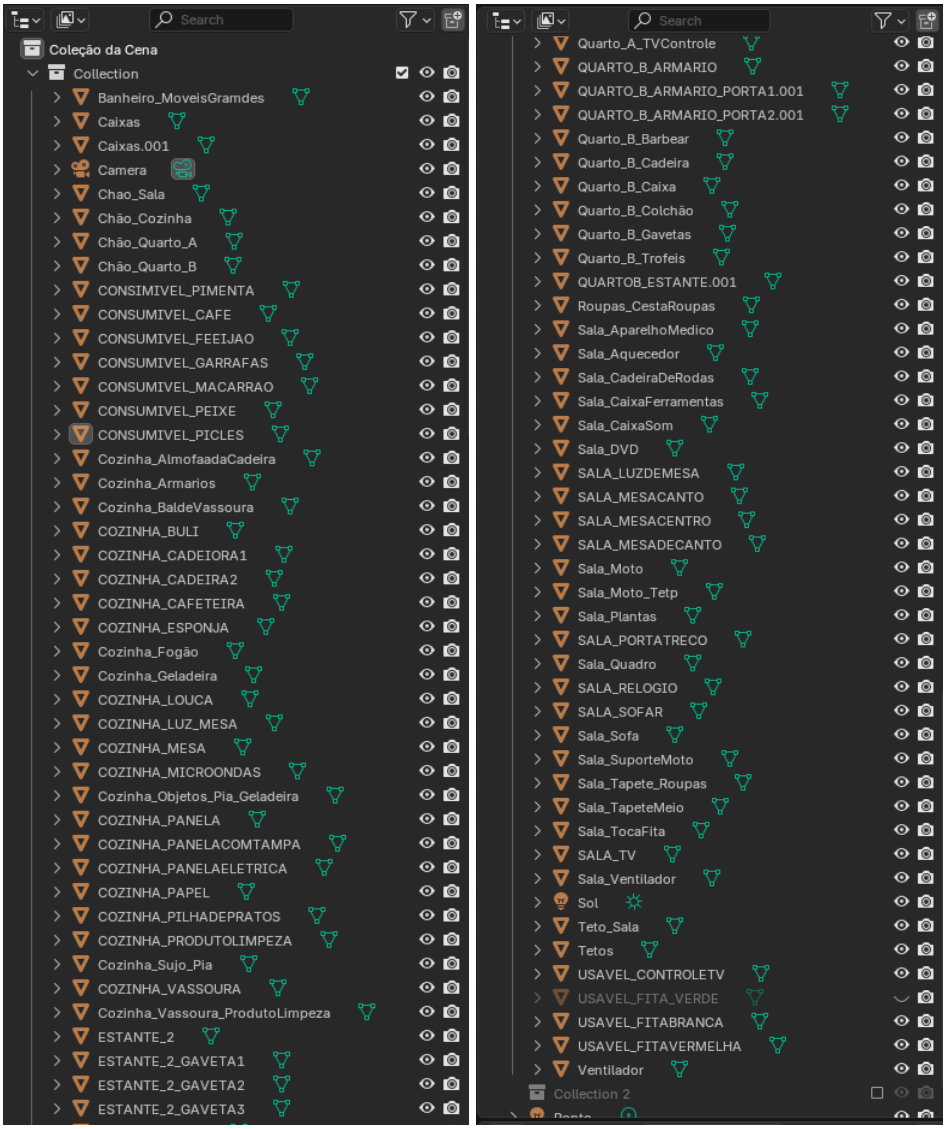
Escolha do Mapa

Após finalizarmos a parte inicial da casa, encontramos este modelo em um site especializado. Infelizmente, o valor estava muito acima e em dólares então foi necessário recorrer a métodos legalmente

questionáveis para adquiri-lo. No final, conseguimos obter o modelo.

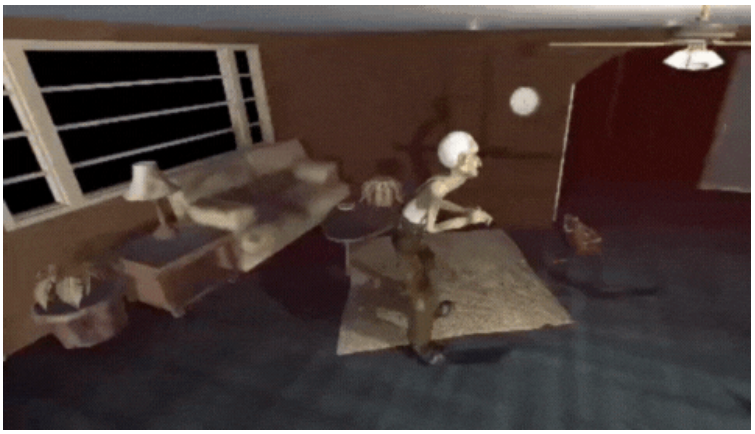
Segregação do Mapa

Depois de adquirir o mapa base, percebi que ele estava composto por um único objeto 3D de grande escala, incluindo toda a textura em um único material. Por isso, precisei segregar cada móvel e objeto manualmente, selecionando face por face, para que fosse possível manipular cada parte separadamente no Unity. Apesar do trabalho intenso, consegui separar os elementos de forma que se tornassem utilizáveis dentro do projeto.



Movimentação Player:

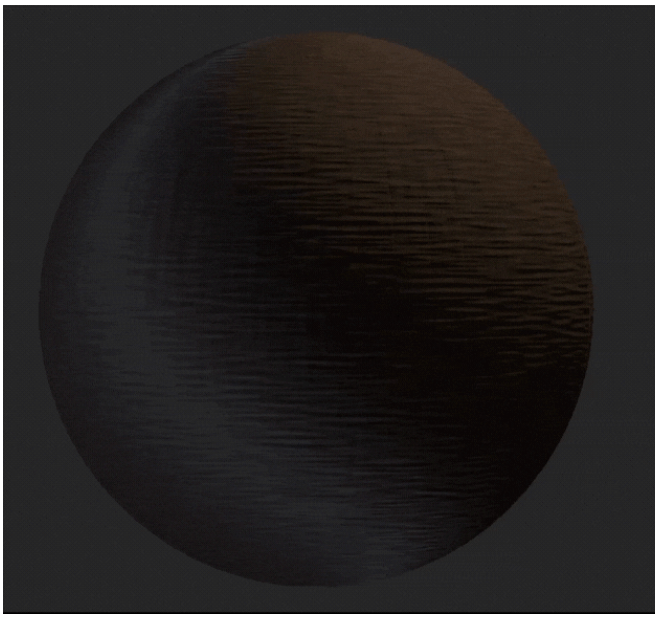
Implementei toda a movimentação do personagem, incluindo o controle de câmera, utilizando o **Cinemachine** do Unity. Esse mesmo recurso foi utilizado para realizar a gravação do efeito 360° em torno do personagem principal na tela inicial do jogo.



A qualidade está baixa pois precisei converter para .GIF

Texturização Procedural:

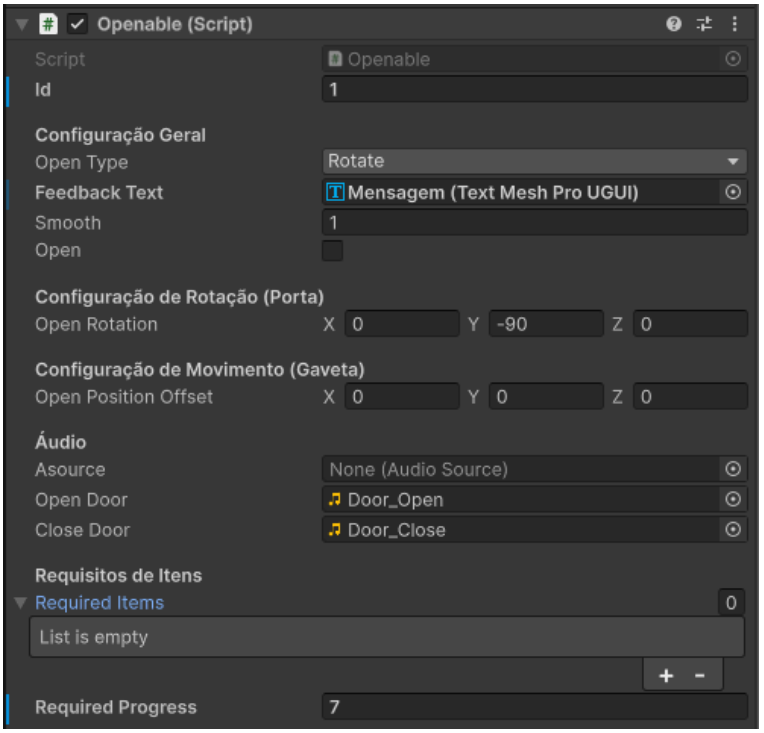
Como requisito do projeto, precisei aplicar texturização procedural no mapa. Inicialmente tentei gerar as texturas completamente via código, porém, após diversas tentativas sem sucesso, percebi que seria mais eficiente criar imagens procedurais com ruídos (noises) e aplicá-las como materiais no Blender. Após entender melhor o processo, também criei um **normal map** para dar mais realismo à textura. Abaixo está um exemplo de uma textura procedural de madeira que produzi:



Portas Armários e Gavetas:

Durante o processo de segregação dos objetos no Blender, percebi que seria necessário realizar mais ajustes para permitir animações de abrir e fechar portas, armários e gavetas. Além disso, como esses móveis eram modelados apenas com faces externas, no Unity as texturas não apareciam internamente. Por isso, precisei modelar o interior de todas as gavetas e armários e aplicar as texturas correspondentes para garantir consistência visual.

Implementei também os comandos de interação com essas portas e gavetas, de modo que pudessem ser reutilizados em qualquer local, além de adicionar efeitos sonoros para as ações de abrir e fechar.



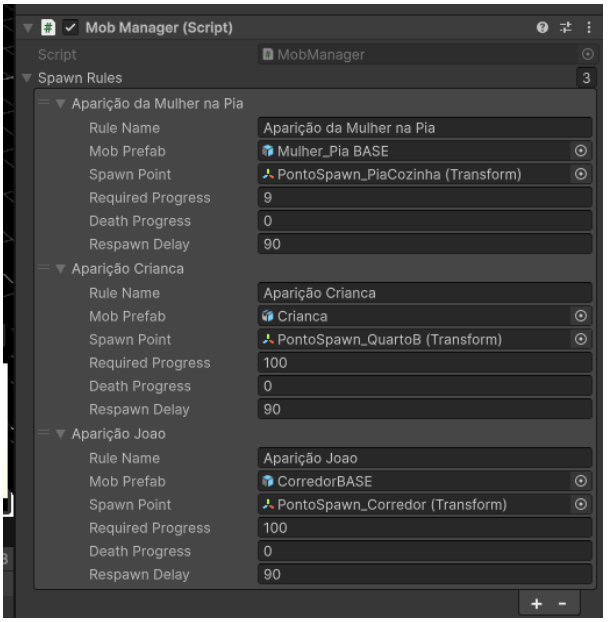
Mobs: Começando pelo Geraldo, finalmente escolhemos um modelo

para ele, pois até então era apenas representado por um cilindro que se movia pelo mapa.

Após a escolha do modelo, criei o **armature** (esqueleto) de todos os personagens para possibilitar a utilização de animações de movimento e de estados base. Também editei a textura do Geraldo original para criar sua versão Dark Geraldo.

Sistemema de Spawn:

Com os inimigos finalizados, implementei o sistema de spawn, responsável por controlar quando, onde e em quais condições cada mob poderia surgir nos ambientes. Para isso, criei presets base para cada inimigo, já contendo suas animações e mecânicas programadas pelos meus colegas. Em seguida, posicionei os pontos de spawn e defini as regras de ativação, como por exemplo: o inimigo só pode nascer em determinado momento do jogo; após spawnar, permanece vivo por no máximo três minutos; e, caso seja morto antes do jogador atingir o objetivo, ele irá respawnar em outro momento pré-determinado.



Davi:

Controlando o Progresso

O controle de progresso do jogo é gerenciado pelo script GameProgression. Este script foi implementado seguindo o padrão Singleton, para garantir que exista apenas uma instância de controle de progresso em todo o jogo, centralizando a lógica. O avanço é rastreado por meio de uma variável do tipo inteiro, cujo valor determina os eventos que ocorrem no mapa e as ações disponíveis para o jogador em cada etapa da jornada.

Textos/Falas

O sistema de textos e falas está diretamente vinculado ao progresso do jogador. O script PlayerLines é responsável por gerenciar as caixas de diálogo, preenchendo o conteúdo textual (string), definindo a duração de exibição da fala e utilizando a variável de progresso para determinar qual diálogo deve ser exibido em momentos específicos da história. Além disso, existem textos contextuais que são independentes do progresso principal, como avisos de "porta trancada" ou confirmações de "item coletado", que são ativados por interações diretas do jogador com o cenário.

Manipulando Cenários

A manipulação dos cenários é um elemento-chave para guiar a experiência do jogador. Utilizando a mesma variável de progresso do GameProgression, o jogo controla eventos dinâmicos no ambiente. Isso inclui a abertura e o fechamento de portas, a ativação de eventos programados e outras alterações que direcionam o jogador para a tarefa principal. Dessa forma, o design de nível mantém uma forte linearidade na narrativa, garantindo que o jogador siga o fluxo pretendido da história.

Sensores

Foram implementados múltiplos scripts que atuam como sensores para detectar a presença e as ações do jogador. Esses sensores têm a capacidade de manipular tanto o cenário quanto o próprio jogador. Por exemplo:

Um sensor pode ser ativado para prender o jogador em um corredor ao avistar um inimigo, criando uma situação de confronto.

Outros sensores são usados para acionar o surgimento (spawn) de inimigos específicos apenas quando o jogador entra em uma determinada área de detecção.

Uma característica importante desses scripts é que, após cumprirem sua função, eles são destruídos e removidos da cena. Essa abordagem otimiza o desempenho, pois evita a manutenção de objetos que não são mais úteis no cenário.

Mariana:

Documentações:

Durante a execução do trabalho em grupo, tive uma participação ativa em diversas etapas do projeto. Fui responsável pela construção do enredo da história, elaborando a lógica e a imaginação dos personagens, elementos essenciais para a imersão e coesão do jogo. Também atuei na definição da lógica geral do jogo, garantindo que os elementos narrativos estivessem integrados com a mecânica proposta.

Além da parte criativa, contribuí diretamente na elaboração e organização dos materiais de apoio, como o GDD, slides e roteiro, que serviram de base para os demais integrantes do grupo. Para isso, utilizei ferramentas como Microsoft Word e Canva.

CutScenes:

Na parte de recursos audiovisuais, fui encarregada da criação das cutscenes e da edição dos áudios do jogo. Para isso, utilizei ferramentas como inteligência artificial (IA), Hedra, GPT, Canva e sites diversos para extração e edição dos áudios, garantindo uma experiência mais envolvente para o jogador.

Código:

No desenvolvimento técnico, juntamente com o Pedro Vitor, fui responsável por implementar a lógica de transição entre o dia e a noite, além de desenvolver a funcionalidade que aciona as cutscenes no Unity. Também adicionei os vídeos diretamente na engine e contribuí com a criação de funcionalidades adicionais.

Alguns dos materiais e vídeos feitos:

https://drive.google.com/drive/folders/1Hen9G8Vgf6h0-N5e1_mV4IX3vaeW9BRR?usp=drive_link