APOCALIPSE-ZUMBI

apocalipse-zumbi

MahTF

Sumário

Informações Gerais

Versão do projeto	*2019.4.0f1*
Versão utilizada	2019.4.0f1
Acesso	Github https://github.com/MahTF/apocalipse-zumbi/
Hardware para builds	1. i7-13700H 2.40 GHz, RAM 16.00 GB, GeForce RTX 4060, SSD 1 TB 2. i5-4200U 1.60 GHz, RAM 6.00 GB, HD 500 GB 3. Intel(R) Celeron(R) N4020, RAM 8.00 GB, HD 500 GB

Objetivos da atividade

1. **Analisar** o projeto em suas diversas áreas: scripts, sprites, texturas, sons, Canvas, materiais, shaders entre outras;
2. **Identificar** problemas nas áreas citadas ou que não foram feitas da maneira mais adequada.

Objetivos do documento

1. Indicar os problemas encontrados;
2. Sugerir mudanças para melhorias em desempenho e arquitetura do projeto.

Autor

URZ Programação de jogos LTDA

Abnner Urzedo

A empresa ou pessoa não contratou o serviço.

Documento de exemplo.

Scripts

ControlaJogador

Resumo

Muitas tarefas feitas em um único script, sendo portanto um problema de estrutura e algumas oportunidades de otimização usando cachê de variáveis. O que deve ser feito é dividir as funções em outros scripts.

Gravidade dos problemas: Médio (performance e estrurura de projeto)

Tempo médio para aplicar melhorias: 6h -8h

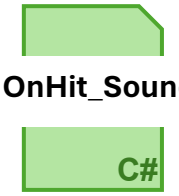
Dificuldade para aplicar melhorias: Médio

Este script faz muita coisa ao mesmo tempo, sendo ele responsável por lidar com a vida, movimentação e animação, além da parte de interface, como o a barra de vida. O que deve ser feito e dividir as tarefas em pelo menos 4 outros scripts diferentes, cada um dele responsável por uma das funções, sendo elas:

Responsável pelo controle da vida do personagem. Este script deve gerenciar as variáveis "**VidaInicial**" e "**VidaAtual**", atualmente presentes no script "**Status**". Implementaria as interfaces "**IMatavel**" e "**ICuravel**" com métodos semelhantes aos do script atual, mas sem as partes relacionadas ao som e à barra de vida. Em vez disso, sugiro criar eventos **Action** (OnHit, OnHeal, OnDeath, OnSpawn e OnHealthChange) para comunicação com os scripts **HealthBar** e **OnHitSound**.

Encarregado do movimento do jogador através de inputs. Este script combinaria funcionalidades dos scripts "**ControlaJogador**" e "**MovimentoJogador**". Além disso, adicionaria a variável de velocidade do script "**Status**", uma variável para a câmera principal, um cache para o **Transform** ("_transform"), utilizaria **raycastNonAlloc** em vez de raycast para melhor gerenciamento de memória e ajustaria a ordem do cálculo de rotação para melhorar a performance.

Responsável pelo controle visual da barra de vida do personagem. Ele recebe o evento **OnHealthChange** (HealthManager) que é acionado sempre que há uma alteração na vida atual do personagem (cura ou dano), transmitindo o valor atual de vida.



Controlaria os efeitos sonoros quando o personagem é atingido.

Basicamente o que estou sugerindo aqui é pegar as partes do script que exercem uma função e vamos colocá-las em um script diferente. Fazendo isso, consequentemente vamos eliminar a necessidade dos scripts “**Status**” e “**MovimentoJogador**”.

Mais alguns detalhes a se resolver. Quando o personagem morre, apenas seria acionado o evento de morte (**OnDeath**) que pode ser recebido pelo script de movimento (para desativá-lo) e pelo “**ControlaInterface**” (que será melhor desenvolvido adiante).

Agora falando sobre **PlayerMovement**, é preciso remover a **classe abstrata** “**MovimentoPersonagem**”. Embora esta classe seja usada também pelo “**ControlaInimigo**”, a existência dela não faz muito sentido visto que a forma de movimento do player e do inimigo é diferente, sendo um por input e outro por IA portando as necessidades também são diferentes. Logo, se for preciso mudar algo na classe abstrata, há um risco dessa alteração quebrar a lógica ou do player ou do inimigo.

Aplicando estas mudanças, a estrutura do projeto fica mais robusta e escalável. Note que, criando esses scripts com funções mais especificas é possível usá-los em outras partes do projeto que não o jogador. Por exemplo, a vida do inimigo pode ser controlada também pelo **HealthManager**, assim como a barra de vida do chefe, pode ser administrada pelo **HealthBar**.

ControlaInimigo

Resumo

Muitas tarefas feita em um único script que compromete a flexibilidade da estrutura do projeto. Chamadas de “GetComponent<Transform>()” desnecessárias. Por fim, há uma oportunidade de se usar Behaviour Trees para programação da IA e o AI Navigation.

Gravidade dos problemas: Médio (estrutura de projeto)

Tempo médio para aplicar melhorias: 8h – 16h

Dificuldade para aplicar melhorias: Alta

Assim como o "**ControlaJogador**", o script "**ControlaInimigo**" atualmente está sobrecarregado com múltiplas responsabilidades, violando o princípio de segregação de tarefas e comprometendo a flexibilidade da estrutura do projeto.

Para o controle de vida do inimigo, sugiro utilizar o mesmo script **HealthManager**, proposto anteriormente, aproveitando seus eventos. A partir desses eventos, sugiro criar mais alguns scripts para exercer algumas funções que hoje são feitas pelo "**ControlaInimigo**", sendo eles:

OnHit_Particles

Receberia o evento **OnHit** e ativaria um sistema de partículas para efeitos visuais de impacto.

OnDeath_SpawnPrefab

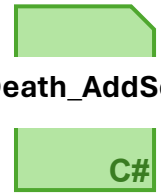
Responsável por uma chance de **spawnar** um prefab, no caso o kit médico, quando o inimigo morre.

OnSpawn_EnableRandomObject

Encarregado de habilitar um objeto aleatoriamente, como uma **skin**, selecionada a partir de uma **array** de opções. Pode manter a funcionalidade atual de selecionar uma skin entre os filhos de um GameObject.

OnDeath_Sound

Tocaria um áudio quando o inimigo morre.



OnDeath_AddScore

Responsável pelo aumento de pontuação do jogador. Integra um sistema de pontuação mais amplo, que será explicado mais adiante.

Criando esses scripts e tirando essas funções do "**ControlaInimigo**", agora ele seria focado apenas em controlar a IA. Mantendo a estrutura da IA como está hoje, as mudanças que sugiro fazer é pegar o **Transform** do jogador ao invés do GameObject, para evitar o "jogador.**transform**..." (o mesmo que "jogador.**GetComponent<Transform>()**..."), pois é uma função de busca de componente influenciando a performance. O mesmo deve ser feito para o **Transform** do próprio inimigo, criando uma variável de cachê ("**_transform**"), evitando chamadas de função desnecessárias no **FixedUpdate**.

ControlaInimigo

```
Transform Jogador;
Transform _transform; -> _transform = transform;
void FixedUpdate(){
(...)
direcao = Jogador.position - _transform.position;
(...)
}
```
C#

Exemplo de substituição para o uso de variáveis de cachê.

Apenas como sugestão, proponho que a lógica da IA seja implementada usando **Behaviour Trees**. Embora a IA atual do inimigo possa ser simples o suficiente para ser gerenciada por uma Máquina de Estado, considerando a possibilidade de adição de mais comportamentos no futuro, a Máquina de Estado pode se tornar difícil de manter. Nesse sentido, a utilização de Behaviour Trees se tornaria uma solução mais escalável e facilitaria a manutenção e adição de novos comportamentos.

Outra sugestão seria utilizar o **AI Navigation** da Unity para o movimento do inimigo. Configurá-lo não é tão complexo e proporcionaria à IA a capacidade de desviar de obstáculos e selecionar caminhos para se aproximar do jogador.

É importante notar que as mudanças propostas no script do jogador também afetariam o inimigo, uma vez que ambos poderiam utilizar o mesmo script de vida, por exemplo. Portanto, as alterações sugeridas aqui para o "**ControleInimigo**" têm como objetivo melhorar a estrutura do projeto e aumentar a versatilidade e reusabilidade dos componentes.

ControlaChefe

Resumo

Basicamente os mesmos problemas do “ControlaInimigo”.

Gravidade dos problemas: Médio (estrutura de projeto)

Tempo médio para aplicar melhorias: 8h – 16h

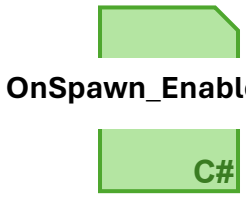
Dificuldade para aplicar melhorias: Alta

Os scripts "**ControlaChefe**" e "**ControlaInimigo**" compartilham essencialmente a mesma base, com pequenas variações de comportamento. No caso do "**ControlaChefe**", há a adição do controle da barra de vida, mas o problema de sobrecarga de tarefas em um único script persiste.

Para resolver essa questão, proponho que o sistema de vida seja gerenciado pelo **HealthManager**, e os eventos e scripts criados para os inimigos (OnHit_Particles, OnDeath_SpawnPrefab,

OnSpawn_EnableRandomObject, OnDeath_PlayAudio e OnDeath_AddScore) sejam aplicados também no "**ControlaChefe**", substituindo as funções dentro desse script.

No prefab do chefe, sugiro a inclusão de um **HealthBar** e a criação de um novo script:



Destinado a habilitar o texto de aviso. No contexto deste projeto, é interessante considerar a adição de um temporizador para controlar o tempo que o texto permanece na tela.

Quanto à IA do chefe, as sugestões previamente mencionadas de utilização de **Behaviour Tree** e **AI Navigation**, propostas para o "**ControlaInimigo**", são igualmente aplicáveis aqui.

É importante notar que, mais uma vez, uma sugestão de melhoria implementada em uma parte do projeto impacta diretamente outras áreas. Ao otimizar o comportamento do inimigo comum, é possível aprimorar a estrutura do chefe utilizando os mesmos scripts. Portanto, o objetivo principal dessas sugestões é sempre melhorar a organização e a eficiência do projeto como um todo.

GeradorZumbi/GeradorChefe

Resumo

Não é preciso usar um script "**GeradorZumbi**" para cada ponto de Spawn. Pode existir apenas um, com todas as opções de spawn em uma array. Além disso, o script é um dos responsáveis pela geração de lixo na memória do projeto ao Instanciar objetos sem reciclá-los, portanto, é necessária a utilização de algum sistema de **pooling**.

Gravidade dos problemas: Alta (estrutura de projeto e Garbage Collector)

Tempo médio para aplicar melhorias: 4h – 6h

Dificuldade para aplicar melhorias: Médio

Falando sobre os problemas menores dos scripts, a abordagem atual de ter um gerador de zumbis para cada ponto de spawn dificulta a contagem precisa dos inimigos presentes na cena, o que torna mais complexa a implementação de limites de spawn e, consequentemente, a progressão do jogo. Além disso, é necessário adicionar um cache para o **Transform** do jogador e para o ponto de spawn.

O principal problema do sistema de geração de inimigos está relacionado ao **instanciamento** de objetos na cena. Devido à grande quantidade de zumbis que são spawnados e posteriormente destruídos quando eliminados, ocorrem frequentes picos de coleta de lixo (**Garbage Collector**), resultando em possíveis quedas de frames durante a partida.

Para resolver os problemas menores, sugiro a criação de apenas um gerador de zumbis responsável pelo spawn de todos os inimigos. Isso simplificaria o controle da quantidade de zumbis na cena e facilitaria a implementação da **progressão do jogo**. O gerador teria uma **array** contendo as opções de pontos de spawn e o prefab dos zumbis a serem spawnados. A lógica de escolha do ponto de spawn

seria baseada na distância mínima do jogador. Como os chefes têm intervalos de spawn diferentes dos zumbis comuns, seriam necessários dois desses scripts no jogo: um para administrar os inimigos normais e outro para os chefes.

Gerador

```
Transform[] pontosDeSpawn;

void Update(){

for(int i=0; i < pontosDeSpawn.length; i++){

//Aqui o script iria passar por cada ponto e ver qual cumpre a distância mínima.

//Caso ache, é só fazer as operações da maneira que são feitas atualmente e então dar um break no loop.

}

(...)

}
```

C#

Um detalhe importante é o comportamento de spawn dos chefes, que não segue a lógica de escolher o ponto mais próximo do jogador, mas sim o mais distante possível. Nesse caso, seria possível criar uma variante do novo script para essa lógica específica de spawn dos chefes. No entanto, minha recomendação é utilizar uma **bool** (por exemplo, "**spawnarNoPontoMaisDistante**") e integrá-la ao mesmo script. Acredito que essa abordagem seja mais coerente, pois ambas as lógicas de spawn estão relacionadas à **distância**, então faz sentido mantê-las juntas no mesmo script.

Por fim, em relação à criação e destruição de inimigos, a solução mais adequada é a implementação de um sistema de **Pooling**, também conhecido como **PoolManager**. Em resumo, esse sistema permite a reciclagem de objetos instanciados, evitando a geração de lixo na memória e melhorando o desempenho geral do jogo. Existem vários tutoriais disponíveis para criar um **PoolManager personalizado**, mas também existem opções prontas disponíveis para uso (uma busca no Google com o termo site:github.com "PoolManager.cs" trará alguns exemplos).

Para aplicar esse sistema, é importante considerar alguns pontos. Como os objetos são reciclados, seus estados **não são resetados**. Isso significa que um zumbi que foi morto e teve sua vida reduzida a zero permanecerá com vida zero quando reciclado, já que em vez de ser destruído, ele é apenas desativado e posteriormente reativado.

Uma abordagem simples para lidar com isso, dependendo do script utilizado, é utilizar os métodos da Unity **OnEnable** e **OnDisable** para resetar os valores quando o objeto é reciclado. Por exemplo, seguindo as sugestões de alteração nos scripts anteriores, o reset da vida no **HealthManager** pode ser realizado da seguinte forma:

HealthManager

```
void OnEnable(){
VidaAtual = VidaInicial;
OnSpawn?.Invoke();
}
```

C#

Em resumo, ao implementar essas alterações, o objetivo é melhorar a performance do jogo, controlar de forma mais eficaz a quantidade de zumbis, simplificar a programação da progressão do jogo, reduzir a quantidade de chamadas do método **Update** e facilitar a atualização e manutenção da lógica de spawn de inimigos.

ControlarArma/Bala

Resumo

Usa variáveis publicas ao invés de **SerializedField.** Também apresenta problema de geração de lixo.

Gravidade dos problemas: Alta (Garbage Collector)

Tempo médio para aplicar melhorias: 5h – 8h

Dificuldade para aplicar melhorias: Baixa

Este script, semelhante aos outros, utiliza variáveis públicas em vez de **SerializedField**, que quebra o padrão convencional de programação na Unity. Além disso, enfrenta o mesmo problema de instanciação de objetos observado no gerador de zumbis. Por fim, apesar de ser um script de arma, ele também lida diretamente com o som, ou seja, desempenha mais de uma função no mesmo script.

Para resolver o problema da geração excessiva de lixo na memória através da criação de objetos na cena, é necessário empregar o sistema de **pooling**, já mencionado anteriormente. Em vez de deixar as variáveis públicas, o padrão **SerializedField** é usado para evitar acesso direto a elas.

Quanto ao som do tiro, pode-se criar um evento (**OnShoot**) que será recebido por um script específico:

OnShoot_PlaySound | Encarregado de reproduzir um som quando a arma é disparada

C#

Além da arma, o próprio projetil precisa de algumas adaptações para se adequar ao sistema de pool. Em vez de utilizar o método **Destroy**, os **PoolManagers** disponíveis no projeto devem ter um método para reciclar esses objetos.

Por fim, há algumas melhorias pontuais adicionais no script da bala. A primeira delas é a criação de um cache do componente **Transform** e a otimização da ordem de operações no **FixedUpdate**.

A segunda melhoria diz respeito à detecção de colisão da bala. Atualmente, a bala verifica se colide com objetos marcados com as **tags** "Inimigo" ou "ChefeFase" e, em seguida, acessa os componentes "**ControleInimigo**" ou "**ControleChefe**" para infligir dano. No entanto, ambos os scripts possuem a interface "**IMatavel**", então não há necessidade de distinguir entre inimigos comuns e chefes. Sugiro que todos os inimigos compartilhem uma única tag (por exemplo, "Inimigos") e que a bala acesse a interface "**IMatavel**" do objeto atingido para aplicar o dano por meio do método "**TomarDano**".

Essas mudanças contribuiriam para resolver a questão do lixo na memória e tornar os scripts mais genéricos, aprimorando ainda mais a estrutura do projeto como um todo.

ControlaCamera

Resumo

É preciso apenas uma variável de cachê para o **Transform** da câmera e sugiro que mude a lógica de **offset** do posicionamento da câmera.

Gravidade dos problemas: Baixa (performance e estrutura de perojeto)

Tempo médio para aplicar melhorias: 30 m

Dificuldade para aplicar melhorias: Baixa

Este script não apresenta problemas significativos, no entanto, é válido mencionar duas sugestões de melhoria. Primeiramente, sugiro adicionar uma variável de cache para o **Transform** da câmera.

Além disso, é recomendável alterar a lógica do offset da câmera para ser uma variável ajustável, em vez de um valor definido no Start. Isso proporcionaria maior flexibilidade e facilidade de ajuste do posicionamento da câmera conforme necessário.

ControlaInterface

Resumo

O script faz muitas tarefas, tem uma leitura difícil e não é prático de se fazer manutenções. Seguindo as mudanças que seriam feitas acima, o script já perderia muitas destas funções a mais que ele tem. Recomendo criar um sistema a parte só para a parte de pontuação.

Gravidade dos problemas: Médio (estrutura e manutenibilidade)

Tempo médio para aplicar melhorias: 7h – 14h

Dificuldade para aplicar melhorias: Baixa

Esse script foi o último a ser mencionado pois as sugestões acima, o impactam diretamente. O principal problema deste script é a sobrecarga de funcionalidades, o que torna a leitura e manutenção do código mais complexas.

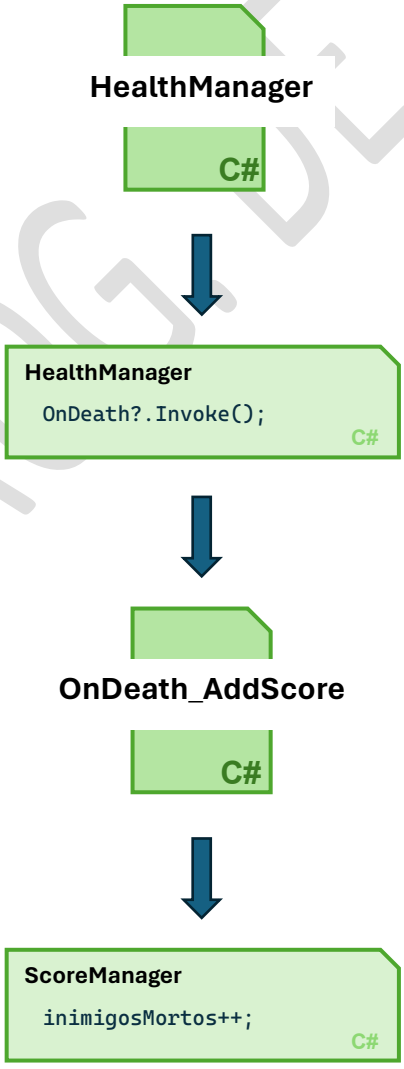
Seguindo as sugestões nos scripts acima, o "**ControlaInterface**" já não seria responsável por algumas funções, como:

A barra de vida do jogador, que agora seria controlada pelo **HealthManager** juntamente com o **HealthBar**.

O texto relacionado ao chefe, que seria gerenciado pelo **HealthManager** do chefe em conjunto com o **OnSpawn_EnableText**.

Sobraria então para o "**ControlaInterface**" adimistrar botões da interface.

Quanto a pontuação, o que eu sugiro é criar um outro script para cuidar da parte de pontuação, incluindo o high score. Para pegar a quantidade de inimigos mortos, por exemplo, cada inimigo poderia ter um script que recebe o evento quando ele é morto e adiciona um valor a uma **variável** que está neste administrador de score.



O tempo de jogo, pode ser medido da mesma forma que está hoje, porém o timer para quando recebe o evento de morte do jogador.

Quanto à representação visual das pontuações na tela, essa funcionalidade pode ser implementada no próprio script do administrador de pontuação. No entanto, para seguir o **princípio de segregação de tarefas**, esse script poderia emitir eventos que alertam outros scripts para atualizar um valor de pontuação.

Essas alterações no script "**ControlaInterface**" é mais uma consequência das mudanças sugeridas anteriormente em outros códigos. Ainda assim, o fato desse script executar tantas tarefas é um ponto de atenção que precisava ser corrigido.

Aplicando essas modificações, será mais fácil fazer a manutenção do projeto além de auxiliar a própria leitura e compreensão do código.

Canvas

Resumo

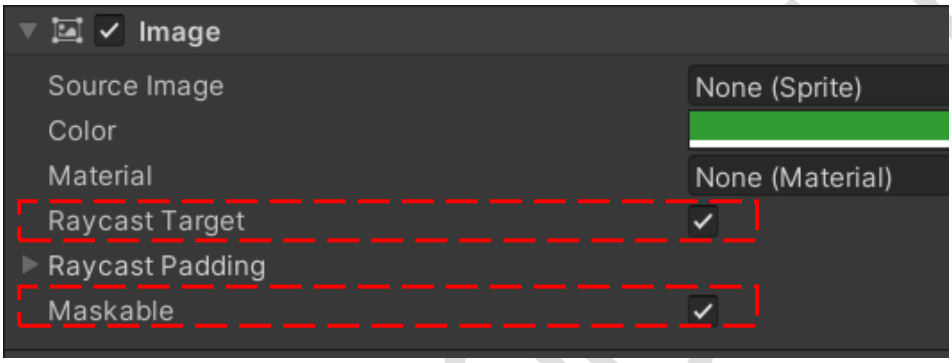
No canvas desse projeto só é preciso desabilitar as opções de *Raycast Target* e *Maskable* dos elementos que não usam essas funções.

Gravidade dos problemas: Baixa (performance)

Tempo médio para aplicar melhorias: 10m – 30m

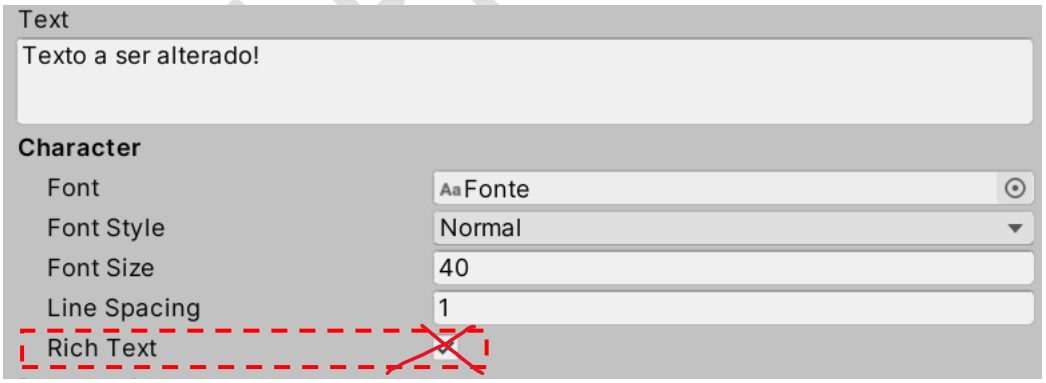
Dificuldade para aplicar melhorias: Baixa

No canvas, o ideal é desmarcar a opção de **raycastTarget** e **maskable** para aqueles elementos que não precisam de uma dessas opções ativa. No caso do projeto, todos os elementos podem ficar com *maskable* desligado pois não usam mascarás em nenhum momento.



As opções **Raycast Target** e **Maskable**.

Os textos especificamente, podem ficar com opção de **raycastTarget** deligada, já que não precisam receber nenhum evento do Graphic Raycaster. Também não é preciso que a opção **richText** fique ligada já que nenhum texto usa tag HTML.



Todas essas mudanças evitam processamento desnecessário na UI.

Modelos/Texturas

Materiais e texturas dos assets da cena

Gravidade dos problemas: Baixa (tamanho de arquivo)

Tempo médio para aplicar melhorias: 1h

Dificuldade para aplicar melhorias: Baixa

A seguinte sugestão se aplica a todos os assets presentes na cena, incluindo ruas, prédios e carros:

- Utilize a compressão **Crunch Compression**, especialmente projetada para texturas com formato **POT** (power of two).
- Reduza a configuração de **Max Size**.

Isso além de diminuir o tamanho de arquivo, melhora o processamento.

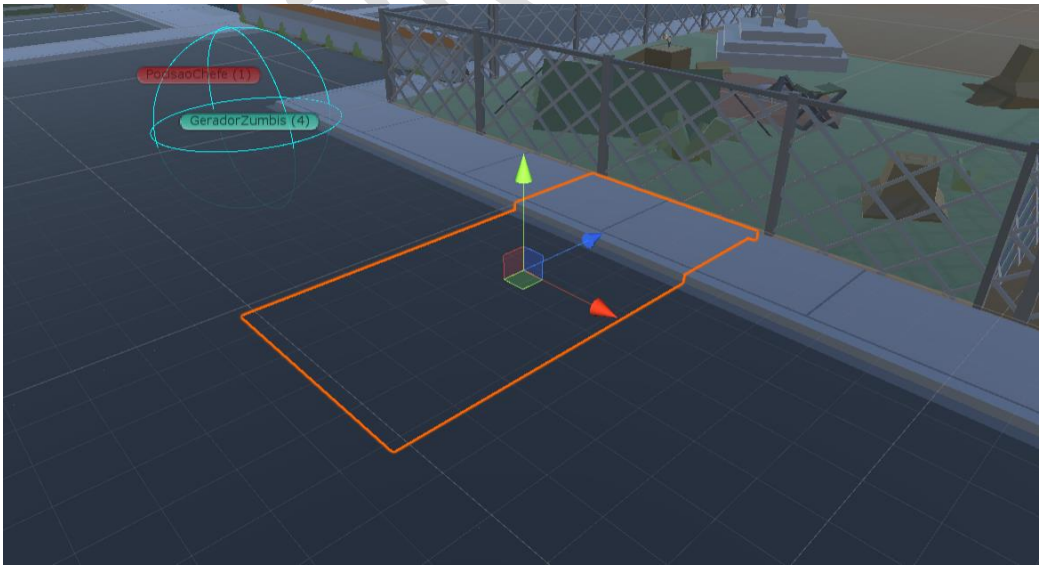
Ruas

Gravidade dos problemas: Alta (performance)

Tempo médio para aplicar melhorias: 5h -14h

Dificuldade para aplicar melhorias: Médio

O problema das ruas esta no número de objetos para compor esta estrutura, o que gera muitas chamadas de renderização.



Há duas opções para este problema:

Redesenhar a Estrutura:

A rua em si (área cinza), pode ser simplificada para um único plano com a textura cinza. Enquanto isso, a calçada pode ser construída utilizando cubos posicionados ligeiramente acima deste plano.

Ao invés de usar pequenas partes para montar a rua, use um grande **plano**.

Utilizar Modelos Especializados:

Em vez de simplesmente utilizar o cubo padrão da Unity e aplicar texturas, é preferível criar um modelo separado para a calçada ou utilizar ferramentas como o ProBuilder para criar uma calçada personalizada. Isso evita a criação de faces desnecessárias que nunca serão vistas durante a renderização.

Mesh Combine:

Uma alternativa adicional é empregar técnicas de "**mesh combine**" nos modelos que compõem a rua, fundindo-os em um único modelo. Essa abordagem, reduzirá o número de chamadas de renderização, melhorando assim o desempenho do jogo. Existem sistemas de "**mesh combine**" disponíveis na internet, inclusive na própria **Asset Store** da Unity.

Prédios e outras estruturas do ambiente

Gravidade dos problemas: Médio (performance)

Tempo médio para aplicar melhorias: 5h – 7h

Dificuldade para aplicar melhorias: Baixa

Sugiro que seja feito um **mesh combine** nestas estruturas também, pois isso vai diminuir as chamadas de renderização.

Zumbis e Personagem

Gravidade dos problemas: Baixa (tamanho de arquivo)

Tempo médio para aplicar melhorias: 1h – 6h

Dificuldade para aplicar melhorias: Baixa - Médio

Quanto às texturas dos zumbis e do personagem, recomendo aplicar a compressão **Crunch Compression**. Outra estratégia, é adotar o mesmo princípio utilizado para as texturas dos prédios e demais componentes do cenário, **juntar todas as texturas dos zumbis** em um único arquivo de textura. Essa abordagem resultará em uma significativa redução no tamanho total do projeto.

Material das partículas

Gravidade dos problemas: Baixa (performance)

Tempo médio para aplicar melhorias: 5m

Dificuldade para aplicar melhorias: Baixa

Sugiro mudar o material para um mobile, pois são mais otimizados. O material pode ser o mobile Alpha Blended, que não vai afetar o visual e ainda será um tipo de material mais otimizado.

HUD/Sprites 2D

Gravidade dos problemas: Baixa (performance)

Tempo médio para aplicar melhorias: 1h – 4h

Dificuldade para aplicar melhorias: Baixa

Para os sprites, recomento que seja criado um **sprite sheet,** pois isso diminui não só a quantidade de chamadas de renderização, mas também o tamanho do projeto. Além disso, aplique o **Crunch Compression** e diminua o **Max Size**.

Qualidade/Renderização

Static/Bake/Oclussion Culling

Gravidade dos problemas: Alta (performance)

Tempo médio para aplicar melhorias: 4h – 6h

Dificuldade para aplicar melhorias: Baixa

Recomendo marcar os objetos do cenário como **static**. Fazendo apenas isso, a e olhando na parte de Stats na Unity, já vai ser possível notar que a quantidade de **draw call** será reduzida, pois a Unity irá tentar renderizar os objetos estáticos juntos.

Marcando como estático, também sugiro que seja feito um **bake** na iluminação da cena, para reduzir cálculos em real time. Para que estas informações do bake afetem os objetos não estáticos, use o Light Probe.

Por fim, é muito importante, especialmente na Pepiline padrão da Unity, usar o **Occlusion Culling** para garantir que tudo o que não está dentro da tela do jogador não seja renderizado, salvando muito tempo de processamento.

Fisica

Resumo

Muitos colliders na cena, sendo grande parte deles do tipo **mesh collider**, o mais complexo quando se trada de cálculo de colisão. A solução é reduzir o número de colliders e sempre que possível usar colisores mais simples, como **Box** ou **Sphere**.

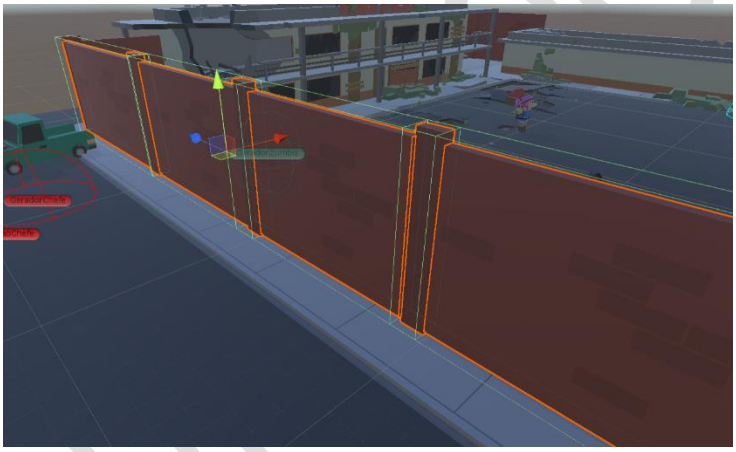
Gravidade dos problemas: Alta (performance)

Tempo médio para aplicar melhorias: 5h – 7h

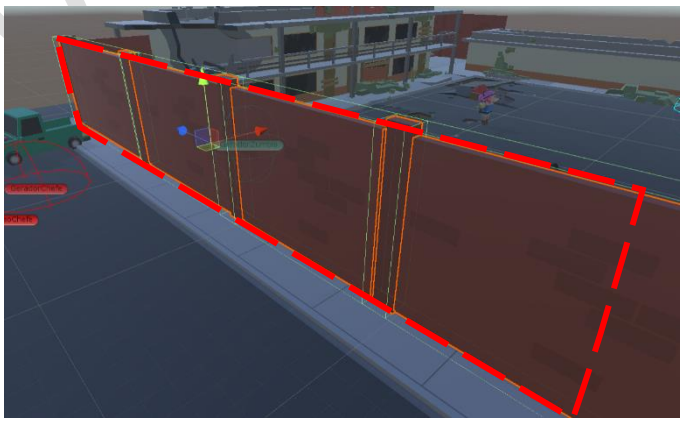
Dificuldade para aplicar melhorias: Médio

Neste projeto, há uma presença significativa de **mesh colliders**, que são tipos de colliders com cálculos de colisão mais complexos. Além disso, devido ao fato de cada peça do cenário possuir seu próprio collider, a quantidade de colliders é maior do que o necessário, o que, naturalmente, impacta a performance do jogo.

Minha sugestão inicial é substituir esses **mesh colliders** por opções mais simples, como colliders **Box** ou **Sphere**, sempre que possível. Além disso, é recomendável utilizar colliders que representem grupos de pequenas peças com colisão, em vez de atribuir colliders individuais a cada uma dessas peças. Essas medidas contribuirão para otimizar a performance do jogo e reduzir o tempo de processamento relacionada aos cálculos de colisão.



> Aqui por exemplo, cada parte do muro tem um collider próprio.



> O ideal é ter apenar um collider que cubra toda a extensão deste muro.

Desempenho Builds

Configurações de pc	*Média FPS*	*Média MS*
i7-13700H 2.40 GHz *RAM 16.00 GB* *GeForce RTX 4060* *SSD 1 TB*	580	1,7
i5-4200U 1.60 GHz *RAM 6.00 GB* *HD 500 GB*	35	10
Intel(R) Celeron(R) N4020 *RAM 8.00 GB* *HD 500 GB*	30	9,1

X

NAME