2023

Análise de Projeto

Catastrophe Island

micnekr

Sumário

Informações Gerais

Versão do projeto	*2020.3.35f1*
Versão utilizada	2021.3.24f1
Acesso	Github https://github.com/tarikipekci/ZombieGame
Hardware para builds	1. Intel Core i7-10750H , CPU 2.60GHz , RAM 8.00 GB, SSD 512, GeForce GTX 1650 2. Intel(Core i3-6100U CPU @ 2.30GHz 2.30 GHz, RAM 4,00 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

Scripts

LowPolyWater

Resumo

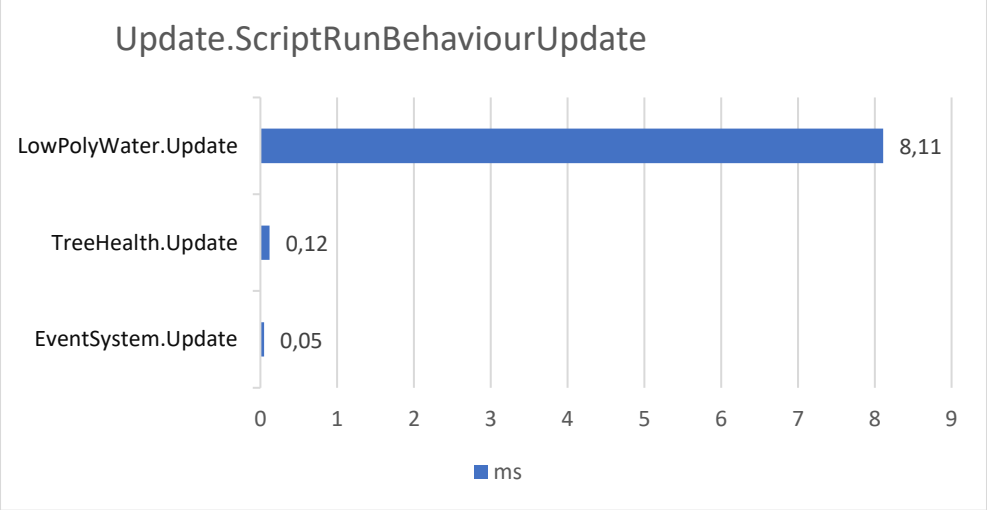
Esse script é o maior responsável pelo baixo FPS e os lags do projeto. Não há uma maneira efetiva de otimiza-lo, sendo preciso então usar algum **shader** para simular o efeito de água assim como o script faz. Para fazer o shader recomendo que o renderizador seja alterado para o **URP**, que tem o **Shader Graph** (editor de shader), onde é possivel criar o efeito de água low poly com facilidade. Porém mudar o renderizador traz alguns problemas de compatibilidade que teriam que ser resolvido caso a caso.

Gravidade dos problemas: Alto (performance)

Tempo médio para aplicar melhorias: 6h – 8h

Dificuldade para aplicar melhorias: Alto

Este script que cria o efeito de onda é o que causa o lag e drop de frame do jogo.



Dos 3 scripts que mais tem tempo de processamento, vemos que o **LowPolyWater** é disparado o mais demorado.

O melhor a se fazer é criar um **shader** que crie esse efeito de onda, sem pesar na performance. A maneira mais fácil de criar esse shader é trocando o **Render Pepiline** para o **URP,** e só então usar o Shader Graph para criar a onda facilmente O único porém, é que como o Render Pepeline vai mudar, todos os materiais do projeto teriam que ser trocados pois os shaders de materiais da Render Pepiline padrão são deferentes dos usados na URP.

Concluindo, é preciso mudar o projeto para usar a URP e criar um shader para substituir o script LowPolyWater. Sem fazer isto, qualquer outra mudança para tentar otimizar será em vão, pois a existencia deste script afeta demais o desempenho do jogo.

GameManager

Resumo

Como esse script tem uma manutenção e legibilidade muito dificil, por fazer muita coisa no mesmo código, a mudança sugerida seria de separar as funções em outros scripts, até para melhorar a estrutura do projeto, e usar o GameManager mais como um intermediador de ações que acontecem no jogo.

Gravidade dos problemas: Médio (estrutura de projeto e manutenibilidade)

Tempo médio para aplicar melhorias: 7h – 9h

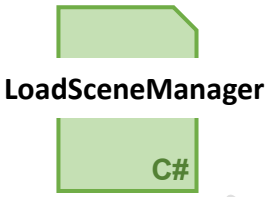
Dificuldade para aplicar melhorias: Médio

Esse GameManager faz muita coisa em um só script, o que dificulta sua leitura e principalmente sua manutenção.

Podemos dividi-lo em pelo menos 3 outros scripts:

Responsável por cuidar da vida do player. Nele podemos ter um evento **Action** que o GameManager usa para saber o status do player, o tipo de dano que ele tomou, se morreu etc.

Responsável por salvar os dados do score da partida. Provavelmente teria métodos estáticos para assim o GameManager chamar o save facilmente.



Responsável por ter métodos que permitem carregar cenas. Fazendo esse script de maneira mais genérica, até mesmo os botões podem usá-lo para carregar cena, não sendo mais necessário a existência de um menu manager.

Outro elemento que pode ser alterado é a remoção da corotina **StartNewMapCoroutine** que instancia o jogador e câmera, sendo que eles poderiam já está na cena. No caso da transição de câmera, é possivel usar o próprio **Cinemachine** para criar as transições.

No Update, poderiamos usar o **Animator** da Unity ao invés de fazer a animação de morte no script. Para chamar o método **KillPlayerImmediate()** seria usado o sistema de eventos nas animações.

Por fim, usariamos o GameManger para fazer algumas transições e filtros de ações. Por exemplo, no script **PlayerItems** existe uma condição no Update que se jogador estiver com status “Dead” ele não pode executar. Ao invés de ter esse filtro no PlayerItems, assim que jogador morrer o **HealthManager** avisaria o GameManger, e ele poderia chamar um método no PlayerItems para impedir qualquer ação, deixando o PlayerItem mais indenpendente e o projeto como um todo mais bem estruturado

TreeHealth

Resumo

São muitas árvores na cena, chamando o método **Update** o que é um problema. A solução é refazer a lógica do script para não ser mais preciso um Update e ainda tentar deixar o script mais genérico.

Gravidade dos problemas: Médio (performance)

Tempo médio para aplicar melhorias: 7h – 9h

Dificuldade para aplicar melhorias: Médio

Existem 245 árvores na cena, cada uma delas chamando o método Update todo frame. Mesmo que esse script não faça nada no Update a maior parte do tempo, existe um pequeno custo para a Unity só de chamar este método, o que torna um custo de desenpenho considerável quando temos tantos objetos com Update em seu script.

Visando tirar o Update do script e ainda melhorar a estrutura do projeto para o futuro, sugiro as seguintes alterações:

```
ResourceBasics


  Action OnHit;

  Action OnResourceExhausted;


  abstract int Collect();

  abstract void ResourceExhausted();
```

C#

Criar uma **classe abstrata** com o nome **ResourceBasics** e adicionar nela alguns eventos action como **OnHit, OnResourceExhausted** e outros, além dos métodos **Collect** e **ResourceExhausted.**

```
~~TreeHealth~~:ResourceBasics
  Resource

  Action OnHit;

  Action OnResourceExhausted;


  override int Collect(){

  //Executa a lógica que está atualmente no
        método Damage

  }


  abstract void ResourceExhausted(){

  (...)

  OnResourseExhausted?.Invoke();

  }
```

C#

Essa classe abstrata seria importada pelo TreeHealth (que inclusive poderiamos chama-lo de **Resource**), e então mudamos o que está no método “Damage” e colocamos em **Collect**. Quando o resurso de madeira estiver esgotado, chamamos o **ResourceExhausted** e fazemos com que não seja mais possivel coletar mais.

Para fazer a lógica que antes estava no Update deste script (de criar um Rigidbody, ativa-lo e depois destroir o objeto depois de um tempo), podemos chamar o evento **OnResourceExhausted** que seria recebido por outros outros scripts que fariam ações especificas:

OnResourceExhasusted_EnableRigidbody

```
void Awake(){
(...)
resource.OnResourceExhausted += (Método);
}
override int Método(){
//Executa a lógica
}
```

C#

Habilita um rigidbory do objeto quando os recursos se esgotam.

OnResourceExhasusted_DestroyAfterTime

```
void Awake(){
(...)
resource.OnResourceExhausted += (Método);
}
override int Método(){
//Executa a lógica
}
Ienumerator CorotinaDeDelay(){
//Timer
}
```

C#

Destroi o objeto depois de um certo tempo que os recursos acabaram

Note que, agora podemos criar scripts para fazer ações especificas quando um recurso se acaba, o que nos permite criar um script para, por exemplo, criar um efeito quando os recursos se acabam ou até tocar um som especifico. Com essas mudanças, deixamos o script mais genérico, de modo que, caso o jogo no futuro tenha outros tipos de recursos (pedra, terra, etc) poderiamos usar esse mesmo script para todos (claro que precisariamos criar alguma Enum ou algo do tipo para indentificar o tipo de recurso).

PlayerItems

Resumo

Apesar de não ter nenhum problema relacionado a performance, o script deve ser alterado para melhorar a estrutura do projeto. A primeira mudança seria tirar essa animação feita no Update e usar o **Animator** da Unity.

A outra é deixar de salvar a quantidade atual de resursos do player neste script e criar outro, chamado **ResourceManager,** que será responsável por isso.

Gravidade dos problemas: Médio (estrutura de projeto)

Tempo médio para aplicar melhorias: 2h – 3h

Dificuldade para aplicar melhorias: Baixa

No Update há uma animação feita a mão no script, o que não preciso já que podemos usar o **Animator** junto aos eventos de animação para fazer o movimento do item e ainda executar a lógica de coleta de recurso. Falando em recurso, idealmente deveria ser criado um **ResourceManager** que guardaria as informações de recursos atuais do jogador, sendo o PlayerItems apenas responsável por avisar (por meio de um evento) que uma quantidade de recursos foi coletada.

Para segregar ainda mais as funções, sugiro que a parte visual da quantidade de recursos seja feita em outro script que seria veinculado mais ao **ResourceManager.**

O objetivo dessas mudanças é , primeiramente, tira o Update do script e ainda deixar o script mais genérico para melhorar a estrutura do jogo. Isso permite que, por exemplo, outras ferramentas de coleta de recursos possam ser criadas e usar esse mesmo script. Seguindo a lógica de sugestões de mudanças no script **TreeHealth** chamamos o método **Collect** para coletar um recurso, e o filtro do que a ferramenta que o jogador esta usando pode coletar ou não pode ser feito por meio de tags ou enum.

Builder

Resumo

Apesar de não ter nenhum problema relacionado a performance, o script deve ser alterado para melhorar a estrutura do projeto. A primeira mudança seria tirar essa animação feita no Update e usar o **Animator** da Unity.

A outra é deixar de salvar a quantidade atual de resursos do player neste script e criar outro, chamado **ResourceManager,** que será responsável por isso.

Gravidade dos problemas: Médio (estrutura de projeto)

Tempo médio para aplicar melhorias: 2h – 3h

Dificuldade para aplicar melhorias: Baixa

Este script tem algumas oportunidades para otimização. A primeira envolve o cache do component **Transform** do holograma do item:

Atualmente sempre que se quer alterar a posição do holograma ou a rotação é feita uma busca do component transform do objeto e só então as alterações são aplicadas.

Então além da variável que salva o gameObject do holograma (**_buildingHologramGameObject**), criamos uma para salvar o Transform, chamada **_buildingHologramTransform.** Então substituimos tudo que tem "**_buildingHologramGameObject.transform...**" por "**_buildingHologramTransform**".

'transform' é o mesmo que '**GetComponent<Transform>()'**

```
Builder
   (...)
   _buildingHologramGameObject.transform.position = placement;
                              ⬇
   _buildingHologramTransform.position = placement;
   (...)
```
C#

Outra oportunidade de otimização é susbtituir o **Physics**.**Raycast**, que gera lixo na memória, pelo **Physics.RaycastNonAlloc.**

Como sugestão adicional, eu diria para usar o novo sistema de input da Unity para tentar parar de usar o Update. Com o novo sistema de input, é possivel detectar inputs e executar ações sem ser preciso o uma verificação todo frame no Update, porém aumentaria o nível de trabalho para fazer as mudanças (INICIALMENTE), mas pode facilitar a manutenção e estrutura do projeto.

Essas mudanças tem o objetivo de diminuir um pouco o tempo de processamento do script e também diminuir a quantidade de lixo na memória.

PlayerController

Resumo

Para esse script só existem alguns pontos para otimizar-lo um pouco mais. Mas no geral, não são muitas mudanças.

Gravidade dos problemas: Baixa (performance)

Tempo médio para aplicar melhorias: 30m – 1h

Dificuldade para aplicar melhorias: Baixa

Para esse script a primeira coisa a se fazer é criar uma variável de cache para o valor de **fixed update** e do componente **Transform** do player.

Para o cache do fixed update, criariamos uma variável, chamada **fixedUpdateValue**, do tipo **float.** Já para o cache do component Transform, criamos uma variável, com nome **_transform**, do tipo **Transform**.

```
PlayerController
   (...)
   _previousJumpSecs = Time.fixedTime;
                              ⬇
   _previousJumpSecs = fixedUpdateValue;
   (...)
```
C#

```
PlayerController
   (...)
   Vector3 position = transform.position;
                              ⬇
   Vector3 position = _transform.position;
   (...)
```

Agora é só trocar tudo que usava 'Time.fixedTime' para **fixedUpdateValue** e tudo que usava o 'transform' para **_transform.**

Também é preciso usar casts **NonAlloc** para evitar lixo na memória:

No caso de 'Physics.CheckCapsule' podemos usar o **Physics.OverlapCapsuleNonAlloc**. Já no caso do 'Physics.CheckSphere', substituimos por **Physics.OverlapSphereNonAlloc**.

Outro elemento que vale muito a pena substituir são as funções **math**, que embora agilizem o processo na hora de criar os códigos, não são otimizadas na versão .NET que o Unity usa. Alguns exemplos de como poderiam ser substituidos são:

`Mathf.Abs` ➡

```
float Abs(float value){
if(value < 0) value *= -1;
return value;
}
```
C#

`Mathf.Clamp` ➡

```
float Clamp(float value, float min, float max){
if(value < min) value = min;
else if (value > max) value= max;
return value;
}
```
C#

Agora falando especificamente de melhorar a estutura do projeto, sugiro que principalmente a ação de agachar seja colocada em outro script. Para fazer as limitações (por exemplo, enquanto estiver agachado não pode correr ou pular) use eventos e o GameManager para ser o intermediário de comunicação.

CameraPostEffectScript

Resumo

Não tem porque este script existir. Com um **Image** e uma animação fazemos o mesmo que esse script faz de maneira muito mais rápida e fácil de se entender.

Gravidade dos problemas: Baixa (manutenibilidade e estrutura de projeto)

Tempo médio para aplicar melhorias: 30m – 1h

Dificuldade para aplicar melhorias: Baixa

Esse script não precisa existir. No lugar dele, sugiro que seja criado um método no próprio GameManager para a animação de morte.

A animação pode ser simplificada também. Ao invés de usar o **OnRenderImage** e ficar alterando cor de material, sugiro que seja criado um **Image** no canvas que cubra e tela toda, criar uma animação alterando o alfa da imagem e dar um play quando o player morrer.

CameraRotate

Resumo

A maior mudança para esse script seria tirar a animação de morte por afogamento dele e usar o **Animator** para isso. No mais, é criar uma variável de cache para o **Transform** e substituir o **math**.

Gravidade dos problemas: Baixa (performance)

Tempo médio para aplicar melhorias: 1h – 2h

Dificuldade para aplicar melhorias: Baixa

Para esse script é importante que haja uma variável de cache do component **Transform**:

```
CameraRotate
  (...)
  Vector3 upEuler = transform.rotation.eulerAngles;

  Vector3 upEuler = _transform.rotation.eulerAngles;
  (...)
```
C#

O math deve ser substituido por uma lógica mais otimizada:

`Mathf.Clamp`

```
float Clamp(float value, float min, float max){
if(value < min) value = min;
else if (value > max) value= max;
return value;
}
```
C#

Por fim, não é preciso que a animação de morte (morte por afogamento) seja executada neste script. Ao invés disso, podemos colocar a animação no GameManager e usar o **Animator** para fazer a animação em si.

Essas mudanças visam melhorar um pouco o tempo de processamento do script e deixar aprimorar estrutura do projeto.

DisasterController/Floods/Tornado/Floods/Tornadoes

Resumo

Os scripts que cuidam da parte de criar os eventos de desastres não estão escaláveis, ao ponto de que, para criar um novo desastre por exemplo, como terremoto ou outro, seria uma dor de cabeça para alterar principalmente o DisasterController para encaixar um novo evento. Sem contar que, qualquer alteração pode causar bugs aos eventos já existentes.

Para melhorar, seria preciso refazer a lógica dos eventos do desatres, usando classe abstrata para deixar os eventos indenpendentes entre si e ajustando a função do DisasterController. Com isso, melhoramos a escalabilidade do projeto e sua manutenção futura.

Gravidade dos problemas: Alta (estrutura de projeto e manutenibilidade)

Tempo médio para aplicar melhorias: 5h – 8h

Dificuldade para aplicar melhorias: Médio

Embora não nada para apontar em questão de problemas de performance, os scripts relacionados aos desastres não estão escaláveis, no sentido de, para adicionar um novo tipo de desastre o esforço será muito grande e ainda há chance de causar bugs nos desastres já existentes.

Para melhorar isso, sugiro uma reformulação dos scripts:

Para começar, primeiro seria criada uma classe abstrata, que podemos chamar de **DisasterBase** que vai conter nele os métodos **ExecuteDisaster()** e **EndDisaster(),** onde ExecuteDisaster() seria uma corotina. Além disso, podemos incluir algumas variáveis do para guardar o nome do desastre, se se ele já esta ativo ou não e tempo minimo e máximo de duração.

Todos os desastres importariam essa classe abstrata e fariam sua lógica dentro da corotina **ExecuteDisaster.**

Então no **DisasterController** teriamos uma array do tipo **DisasterBase** , uma variável do delay entre os eventos e uma corotina para escolher o próximo evento. No inspector colocariamos os disastres programados (Tornado, Floods e outros futuros) na array criada.

Então agora seria só escolher aleatóriamente um dos valores dessa array e chamar **'StartCoroutine (array[random].ExecuteDisaster())'** que a lógica do desastre escolhido seria executada.

Para saber quando o evento do desastre escolhido acabou, poderiamos adicionar um **Action** (OnEndEvent) que seria enviado pelo script DisasterBase, recebido pelo DisasterController e executaria um novo loop para chamar o próximo evento de disastre.

Resumindo a ideia:

.Criamos uma classe abstrata que será a base para os scripts de eventos;

.Cada desastre teria seu próprio script, que importaria essa classe abstrata e teria sua lógica dentro da corotina **ExecuteDisaster()** (por exemplo, no script Floods, a lógica que decide a força do alagamento e faz subir a água seria colocada nesta corotina);

.Quando acabar o disastre, o script avisaria o **DisasterController** por meio de um Action;

.O papel do **DisasterController** agora seria apenas saber quando um evento terminou para criar um delay do um próximo evento, escolhido aleatóriamente, e então dar um StartCoroutine(ExecuteDisaster()) e mandar o script do evento escolhido executar a lógica do desastre em si.

Com essas mudanças, seria muito mais fácil criar novos tipos de desastres no futuro.

PlayerHealthController

Resumo

Esse script faz muita coisa em um só código. Para facilitar a leitura e manutenção, podemos dividi-lo em 3:

.Um para controlar a barra de vida;

.Um para dano de queda;

.Por fim, um para o dano de afogamento.

Gravidade dos problemas: Baixa (estrutura de projeto)

Tempo médio para aplicar melhorias: 2h – 4h

Dificuldade para aplicar melhorias: Baixa

Não existe nenhum grande problema em questão de performance, porém o script essa um pouco sobrecarregado de funções. O que eu sugiro é que ele seja divido em outros scripts: um para a barra de vida, um para dano de queda e outro para dano de afogamento.

A ideia é o script do PlayerHealthController seja responsável apenas por guardar o valor atual da vida do jogador e tenha nele um método que permite outros scripts alterar o valor da vida (DoDamange, OnHit ou qualquer outro nome) e uma forma de avisar que o valor atual da quantidade de vida foi alterada, por meio de um evento **Action.**

Então basicamente, a lógica de fall damage e de afogamento premaneceriam iguais, porém em scripts fora do PlayerHealthController. Mas agora, eles teriam que chamar o método para dar dano player que estaria no script de vida.

No meio disso, sempre que o valor de vida é alterado, o PlayerHealthController invocaria um evento **Action** (OnChangeHealthValue, OnNewHealthValue ou outro nome) que seria recebido pelo script de barra de vida.

Essas mudanças visão melhorar a estrutura do projeto e facilitar a manutenção.

Canvas

Resumo

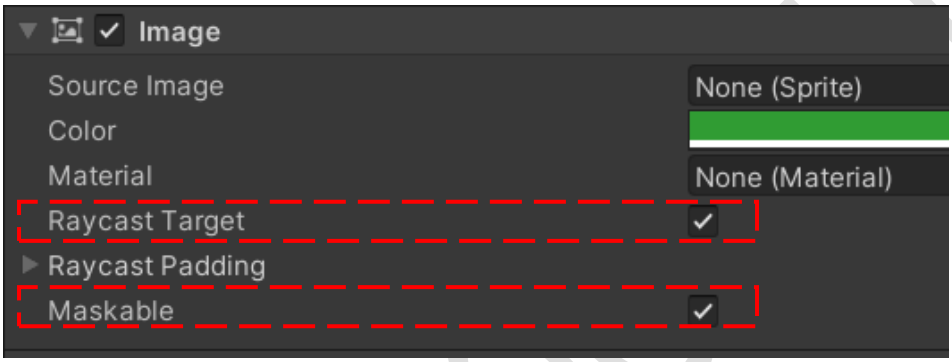
No canvas desse projeto só é preciso desabilitar as opções de *Raycast Target* e *Maskable* dos elementos que não usam essas funções.

Gravidade dos problemas: Baixa (performance)

Tempo médio para aplicar melhorias: 10m – 30m

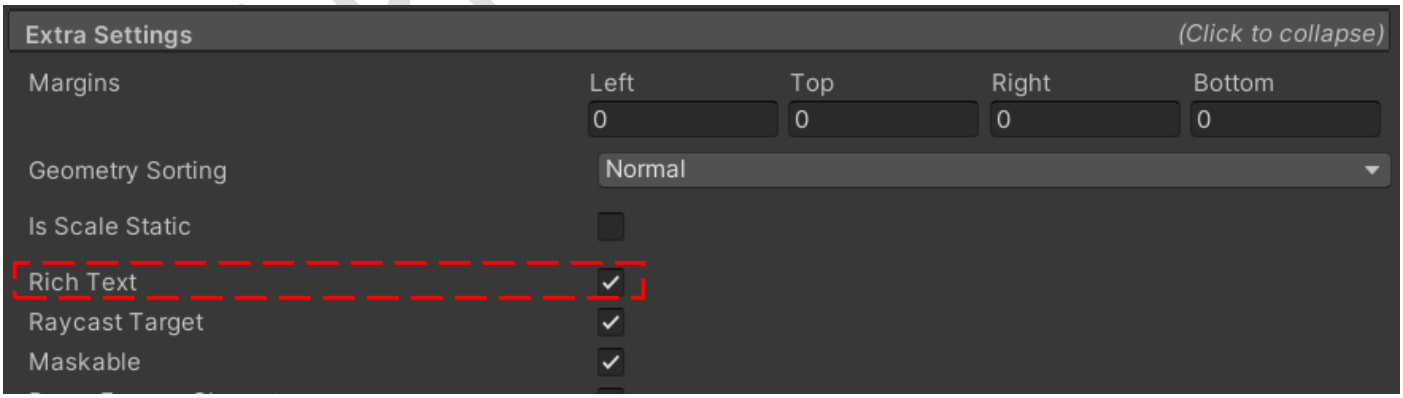
Dificuldade para aplicar melhorias: Baixa

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.



Na cena Game Scene, o canvas pode ser divido em dois. Em um ficaria a barra de vida e o "Overlay" que são elementos de tela dinâmicos, ou seja, que sofrem alterações. O motivo de separar em dois canvas é que quando um elemento (button, text ou image) sofre alguma alteração, a canvas inteiro deve ser redesenhado. Por isso é uma boa prática separar em canvas de elementos dinâmicos e estáticos.

Todas essas mudanças evitam processamento desnecessário na UI.

Modelos e texturas

Ocean50x50W750H750

Gravidade dos problemas: Médio (performance)

Tempo médio para aplicar melhorias: 4h – 6h

Dificuldade para aplicar melhorias: Médio

Para diminuir a contagem de vértices do oceano, sugiro que a parte mais próxima da ilha continue a mesma coisa porém o que está mais afastado pode ser trocado por um **Plane** que tem menos vértices. Embora isso possa causar alguma discrepância visual, o FOG do jogo pode disafarçar um pouco.

O objetivo é tentar otimizar a quantidade de vértices na cena e acelerar um pouco o precessamento.

Tree_01/Tree_02/Tree_03/Rock_01...04/ Stump_01

Gravidade dos problemas: Médio (tamanho de arquivo e performance)

Tempo médio para aplicar melhorias: 4h – 6h

Dificuldade para aplicar melhorias: Médio

Para começar, o material desses modelos podem usar um shader mais otimizado, como o Mobile Vertex Lit, pois a diferença no visual não fica tão grande e ainda ajudar no processamento, especialmente em hardwares mais fracos.

Quanto a textura, o **Crounch Compression** deve ser aplicado e o **Max Size** alterado para um valor mais baixo e que não afete o visual.

Especialmente o modelos das árvores, precisam ter um **LOD**. Com isso, elementos mais distantes ficam na sua versão mais lowpoly o que ajudar a diminuir a quantidade de vértices renderizadas na cena.

Casas e contruções

Gravidade dos problemas: Médio (performance)

Tempo médio para aplicar melhorias: 6h – 8h

Dificuldade para aplicar melhorias: Médio

As contruções do jogo são peças separadas que formam o conjunto. Isso aumenta a quantidade de draw call e diminui o FPS.

Uma solução é colocar essas peças em um programa 3D, montar a casa dentro do programa e exportar para o Unity como um modelo único. Outra opção é usar algum asset de **Mesh Combine** para juntar as peças em um único modelo, assim diminuindo os draw calls.

Por fim, também é preciso aplicar o **Crounch Compression** e alterar o **Max Size.**

Terreno

Gravidade dos problemas: Médio (performance)

Tempo médio para aplicar melhorias: 6h – 8h

Dificuldade para aplicar melhorias: Baixa

O Terreno é um pouco complicado saber exatamente o que alterar, mas deixar os valores padrões nunca é o ideal. Algumas opções para se ajustar são:

.Pixel error;

.Detail Distance;

.Base Map Dist;

.Detail Resolution;

.Detail Resolution per patch.

No Terreno que está abaixo da água deve ter essas opções ainda mais alteradas com valores mais baixos.

Encontro um ajsute equilibrado, a quantidade de draw calls e vértices vão diminuir consideravelmente.

Qualidade/Renderização

Camera Main/ HeldItemCamera

Gravidade dos problemas: Baixa (performance)

Tempo médio para aplicar melhorias: 1h – 2h

Dificuldade para aplicar melhorias: Baixa

Ambas as câmeras podem ter seu **Far Plane** diminuido para diminuir a distancia de renderização. Esta redução junto ao **FOG**, faz com que o jogador quase não note que há um limite de renderização e ainda pode salvar alguns **ms** de processamneto e aumentar o FPS.

A câmera HeldItemCamera especificamente, pode ficar com a distância de renderização bem baixa, já que só é usada para renderizar os itens.

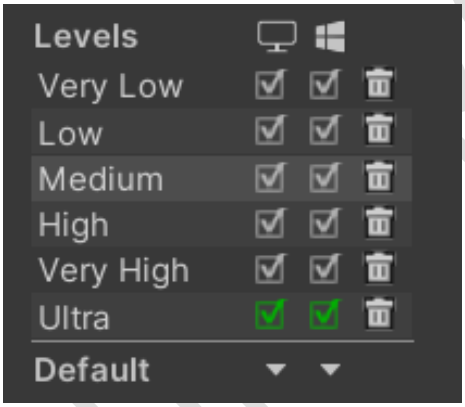
Configurações de qualidade

Gravidade dos problemas: Baixa (performance)

Tempo médio para aplicar melhorias: 1h – 2h

Dificuldade para aplicar melhorias: Baixa

As qualidade do projeto está no **Ultra** (ProjectSettings > Quality**)** o que não é nem um pouco necessário. Sugiro que haja uma mudança nas configurações de qualidade do projeto, algo entre **Medium** e **Low** já funciona bem.



Física

Resumo

Basicamente é preciso simplificar os colliders dos objetos do jogo, pois atualmente todos usam o **mesh collider**, que dentre todos é o que mais demanda processamento.

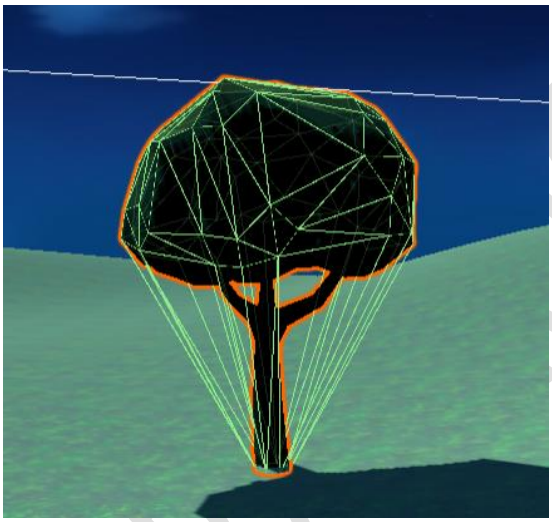
Gravidade dos problemas: Médio (performance)

Tempo médio para aplicar melhorias: 2h – 4h

Dificuldade para aplicar melhorias: Baixa

Todos os colliders usam o **mesh** collider que em grande quantidade podem afetar o processamento do projeto, ou gerar alguns picos de **ms** (algo que acontece de vez em quando no jogo). O que pode ajudar muito é usar **colliders mais simples**, como cubo, esfera, cilindro e capsulas, no lugar dos **mesh colliders** ultilizados por todos os modelos que estão na Game Scene.

A árvore, por exemplo, pode usar um collider de capsula e uma de esfera para compor seu grupo de colisão.



Como todos os modelos usam o mesh collider, que é o collider que demanda mais processamento, quando o Unity vai fazer calculos de colisão acaba gerando um drop de FPS significativo. Então sugiro fortemente que os colisores sejam mudados, sendo preciso ir de prefab em prefab e análisar qual a melhor forma de substituir o mesh collider.

Desempenho Builds

Configurações de pc	***Média FPS***	***Média MS***
Intel Core i7-10750H *CPU 2.60GHz* *RAM 8.00 GB* *SSD 512* *GeForce GTX*	86 / 90	11.5 / 11
Intel Core i3-6100U *CPU @ 2.30GHz 2.30 GHz* *RAM 4.00 GB*	X	X

X _____

NAME