2023

Análise de Projeto

Zombie Game

tarikipekci

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

EnemyHealth / PlayerHealth

Resumo

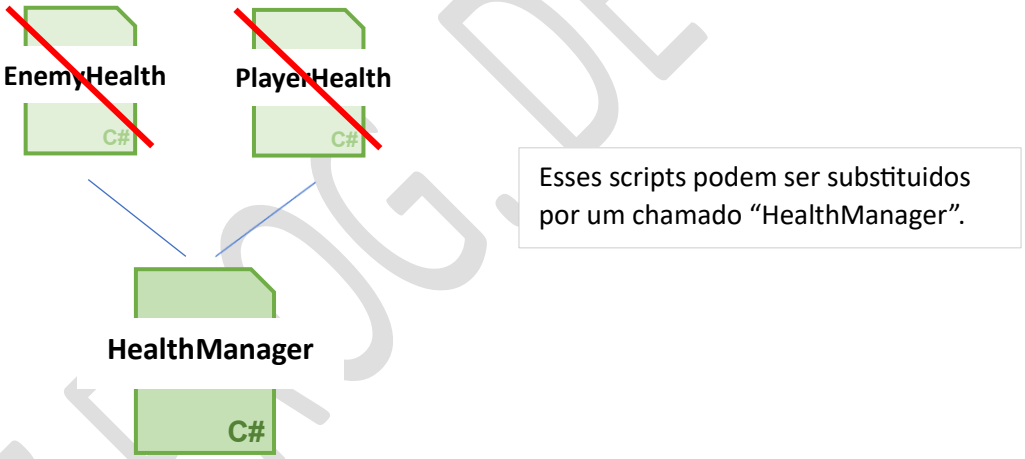
Nenhum problema encontrado, porém é possivel juntar os scripts em um só e criar uma classe abstrata com variáveis e métodos genéricos que permitem que um mesmo script seja usado tanto para o player quanto para os inimigos. O objetivo disso é melohorar a arquitetura do projeto.

Gravidade dos problemas: Baixa

Tempo médio para aplicar melhorias: 1h – 2h

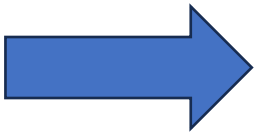
Dificuldade para aplicar melhorias: Baixa

Não há problemas de performance relacionado a esses scripts, porém a forma como eles funcionam não está ideal. Recomendo algumas alterações para ajudar na arquitura do jogo e deixar o script mais genérico e fácil de se lidar:



No HealthManager, importariamos de uma classe abstrata (**HealthBasics**) que teria implementado as variáveis básicas como “maxHealth”, “currentHealth” e “isAlive”, e também alguns métodos, como “OnHit” e “Die”.

```
HealthBasics

  int maxHealth;
  int currentHealth;
  bool isAlive;

  abstract void Hit();
  abstract void Die();

                                C#
```



```
HealthManager: HealthBasics

    override void Hit(){
            //Execução
    }
    override void Die(){
            //Execução
    }
                                C#
```

Com essas alterações podemos usar o mesmo script de controle de vida tanto no player quanto no inimigo. Isso acaba virando um ganho para os scripts que vão dar algum tipo de dano a entidade (como os das armas), já que no momento que for preciso pegar o componente de vida do objeto que se esta atingindo, o código pode procurar pela classe abstrata **HealthBasics** e chamar o método **Hit**. Para fazer a filtragem do que é player e do que é zumbi, usamos a tag do objeto.

Note que essas alterações não trazem melhorias na performace, mas faz com que seja mais fácil a manutenção do script de vida e facilita a interação de outros scripts com ele.

Weapon/ AmmoController/ WeaponSwitcher/ Weapon Zoom/ AmmoPickUpController

Resumo

Basicamente, quase todos esses scripts tem problemas na sua função Update(), onde existem operações desnecessárias ou que não são as mais performáticas sendo executadas todo frame.

O maior problema é quanto ao script Weapon que tenta fazer muita coisa ao mesmo tempo. Esse tipo de comportamento em um script pode se tornar uma dor de cabeça para a escalabilidade do projeto.

Gravidade dos problemas: Médio

Tempo médio para aplicar melhorias: 7h – 9h

Dificuldade para aplicar melhorias: Baixa

Os códigos **WeaponZoom** e **WeaponSwitcher** tem um problema em comum que é a atribuição de um parâmetro Bool no Animator das armas todo frame nem necessidade .

WeaponZoom

```
private void Update(){

(...)

animForMachineGun.SetBool("zoomIn", true);
animForShotgun.SetBool("zoomIn", true);
animForPistol.SetBool("zoomIn",true);

(...)

animForMachineGun.SetBool("zoomIn", false);
animForShotgun.SetBool("zoomIn", false);
animForPistol.SetBool("zoomIn",false);

}
```

C#

WaponSwitcher

```
private void Update(){

(...)

// ReSharper disable once
Unity.PreferAddressByIdToGraphicsParams
animForMachinegun.SetBool("selected", _machineGunSelected);
// ReSharper disable once
Unity.PreferAddressByIdToGraphicsParams
animForShotgun.SetBool("selected", _shotgunSelected);
// ReSharper disable once
Unity.PreferAddressByIdToGraphicsParams
animForPistol.SetBool("selected", _pistolSelected);

(...)

}
```

C#

Há um custo de performance chamar o **SetBool**, por isso não pode ser feito sem que seja preciso.

Quanto ao **Weapon,** o primeiro problema é que a variável *Player* é do tipo *GameObject* quando poderia ser do tipo *Transform*, evitando que o script tenha que todo frame buscar o componente *transform* do objeto player, sendo portanto uma variável de cachê. O mesmo deve ser feito com a variável *fpCamera.*

Weapon

```
Camera fpCamera -> Transform fbCamera

GameObject Player -> Transform Player
```

C#

Essas variáveis serão o cache do componente *transform* dos objetos.

Weapon usa o *raycast* padrão, o que pode ser perigoso já que ele gera lixo na memória e , considerando um cenário onde um jogador fica atirando por muito tempo sem parar, isso pode ser o culpado por drops de fps quando *Garbage Colletor* entrar em ação e limpar os lixos na memória.

Outro problema neste código é que tenta fazer tudo ao mesmo tempo, tendo nele o modo “Shotgun”, “Pistol” e “MachineGun”, sem contar que ele também cuida dos efeitos da arma.

Weapon

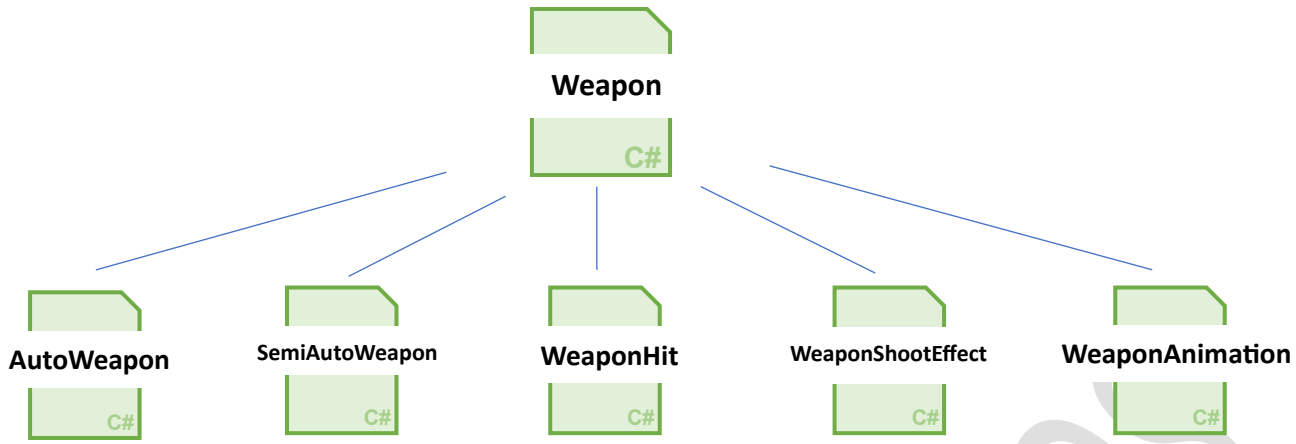
```
if (gameObject.CompareTag("MachineGun"))
{
        (...)
}
else if (gameObject.CompareTag("Pistol"))
{
        (...)
}
else if (gameObject.CompareTag("Shotgun"))
{
        (...)
}


(...)

private void PlayMuzzleFlash()
{
        muzzleFlash.Play();
}

(...)

private void CreateHitImpact(RaycastHit hitInfo)
{
        (...)
}
```

C#

Um mesmo script, têm três modos de armas, cuida do efeito de hit e do efeito de tiro.

Isso não é melhor a ser feito.

As soluções para esses problemas começa com o **Weapon**. Primeiramente, devemos **dividir as funções** exercidas por esse script, criando um script para **armas automáticas** e um para **armas semi-automáticas**, um para cuidar dos **efeitos quando atira**, um para **efeitos quando atingir algo** e outro para as **animações das armas**.



Para ter um certo padrão em scripts de armas, criamos também uma classe abstrata, que podemos chamar de **WeaponBasics**, que vai conter as informações básicas como o tipo de arma, tag alvo, range, dano, fire rate, transform da camera, força do recoil e máximo de munição no pente, também alguns eventos básicos como o **OnShoot**, "OnHit, **OnReload**, **OnEnable** e **OnDisable**, e por fim métodos como **AddAmmo**, **RemoveAmmo**, **Reload**, **Shoot**, **EnableWeapon** e **DisableWeapon**.

HealthBasics

```
WeaponType maxHealth;
string tagetTag;
float range;
int damage;
Transform camera;
Vector2 minMaxRecoil;
int magazineCapacity;

Action OnShoot;
Action OnHit;
Action OnReload;
Action OnEnable;
Action OnDisable;

abstract void AddAmmo();
abstract void RemoveAmmo();
abstract void Reload();
abstract void Shoot();
abstract void EnableWeapon();
abstract void DisableWeapon();
```

Essa classe abstarata seria importada pelos scripts **AutoWeapon**, **SemiAutoWeapon** e qualquer outro script de arma no futuro.

Trocamos o raycast padrão pelo *RayCastNonAlloc* que vai evitar que gerar lixo na memória enquanto atira.

```
private void Shoot()
{
        Physics.Raycast -> Physics.RaycastNonAlloc

}
```

C#

Seguindo a lógica que seria feita nos scripts de vida acima, deixamos de tentar pegar o component **EnemyHealth** e pegamos o **HealthBasics**, chamando logo em seguida o método **Hit** passando o dano da arma.

```
EnemyHealth target = hit.transform.GetComponent<EnemyHealth>();
HealthBasics target = hit[0].GetComponent<HealthBasics>();
```
C#

Ao atirar, o acionamos o evento **OnShoot** que seria recebido pelo script que cuida do efeito de tiro, e se o raycast atingir algo, chamamos o **OnHit** que seria recebido pelo script resposável pelo efeito de hit.

Neste script do hit, substituiriamos a lógica de instanciar e destruir objetos para uma que envolve uma **pool** para **reciclar os objetos** e evitar lixo na memória.

Voltando ao **WeaponSwitcher**, para acompanhar as mudanças acima, criamos neste script uma array de WeaponBasics. Com isso basicamente o que pode ser feito é chamar o método **EnableWeapon** da arma que foi selecionada e também chamar o método **DisableWeapon** na arma antiga. Nesses métodos, fazemos os processos de ativar ou desativar a arma além de chamar o evento **OnEnable** que será recebido pelo script que cuida das animações para rodar a animação de troca de arma. Esse evento também seria últil para o WeaponZoom que resetaria o nível de zoom quando este evento é chamado.

Apesar de toda essas mudanças, AmmoController e AmmoPickUpController permaneceriam basicamente a mesma coisa.

O objetivo de todas essas alterações é novamente melhorar a arquitetura do jogo, tornando a lógica das armas mais escalável e fácil de alterar ou fazer manutenção. Além de claro, melhorar um pouco a performance e a geração de lixo na memória.

FirstPersonController

Resumo

O script é bem feito e só precisa de alguns poucos ajustes. O mais importante é fazer um cache do componente *Transform* do player. Fora isso, também é preciso criar uma variável do tipo Transform e usa-la para evitar calculos com vetor, que não são performaticos. E a última alteração, é a substituição de uma função math por uma lógica com o processamento mais rápido.

Gravidade dos problemas: Baixa

Tempo médio para aplicar melhorias: 10m – 30m

Dificuldade para aplicar melhorias: Baixa

Esse script fica o tempo todo buscando o componente *Transform* do objeto por meio do **transform** (“transform” é o mesmo que “gameObject.GetComponent<Transform>()”), o que não é o ideal a se fazer. FirstPersonController também gera lixo na memoria pois usa *CheckSphere* para veficar se o jogador está no chão. Falando nesta verificação, é feito um calculo para determinar o ponto de inicio da verificação (*transform.position.y – GroundedOffset*), que poderia ser substituido por uma variável (**GroundCheckPoint**) do tipo *Transform* e evitar calculo com vetor, que não

performatico. O último problema a se observar é que o script usa a função matemática *math.clamp* que não recomendada de se fazer todo frame pode causar problemas na performance.

A primeira coisa a se fazer no FirstPersonController é criar uma variável de cache para o *Transform* e substituir tudo que usa **transform**:

FirstPersonController

```
Transform _transform = GetComponent<Transform>();

(...)

Vector3 spherePosition = new Vector3(transform.position.x, transform.position.y - GroundedOffset,
transform.position.z);

Vector3 spherePosition = new Vector3(_transform.position.x, _transform.position.y -
GroundedOffset, _transform.position.z);
```

Um exemplo do uso da variável de cache. Tudo que era "transform" foi alterado para _transform, a variável de cache.

Também criamos uma ***Transform* groundCheckPoint** para guardar a po
removemos a linha 128 do script e usamos o "OverlapSphereNonAlloc" passando como parametro da posição inicial a variável criada.

Weapon

```
Grounded = Physics.OverlapSphereNonAlloc(groundCheckPoint, GroundedRadius,results,
GroundLayers, QueryTriggerInteraction.Ignore);
```

C#

Por fim, substituiriamos o "math.clamp" por uma lógica simplificada que faça o mesmo que ele.

Weapon

```
private static float ClampAngle(float lfAngle, float lfMin, float lfMax)
{
        if (lfAngle < -360f) lfAngle += 360f;
        if (lfAngle > 360f) lfAngle -= 360f;
        return Mathf.Clamp(lfAngle, lfMin, lfMax);
}


private static float ClampAngle(float lfAngle, float lfMin, float lfMax)
{
        if (lfAngle < -360f) lfAngle += 360f;
        if (lfAngle > 360f) lfAngle -= 360f;

        if(lfAngle <= lfMin) lfAngle = lfMin;

        if(lfAngle >= lfMax) lfAngle = lfMax;

        return lfAngle;
}
```

C#

Essas mudanças visam melhorar a eficiência do script e aumentar levemente a sua performance.

Enemy AI

Resumo

Da maneira que o script foi feito, basicamente não há nenhum problema nele fora a necessidade de uma variável de cache para o *Transform.*

Porém recomendaria usar a lógica de *Behaviour Tree* ao invés de State Machine para montar a inteligência artificail do inimigo.

Gravidade dos problemas: Baixa

Tempo médio para aplicar melhorias: 10m – 15m

Tempo médio para aplicar melhorias com a Behaviour Tree: 5h – 6h

Dificuldade para aplicar melhorias: Baixa

Dificuldade para aplicar melhorias com a Behaviour Tree: Médio

Não há tantas mudanças neste script da forma que ele está. Na linha 31 ele usa o transform, então recomendo fazer um cache do componente *Transform*.Também não é preciso usar o método “FaceTarget”, já que o próprio NavMeshAgent tem a configuração de rotacionar para a direção que se esta movendo.

Mesmo não tendo tantas mudanças, eu recomendaria já começar a usar **Behaviour Tree** para montar a inteligência artificial dos zumbis ao invés da lógica atual que se aproxima mais da **State Machine**. A Behaviour Tree é muito mais escalável, com o lado negativo de ser mais demorado criar comportamentos NO INICIO, mas que depois, devido a sua natureza de comportamentos divididos em etapas, fica muito mais simples criar novos comportamentos, aumentar sua complexidade ou fazer manutenções.

Canvas

Resumo

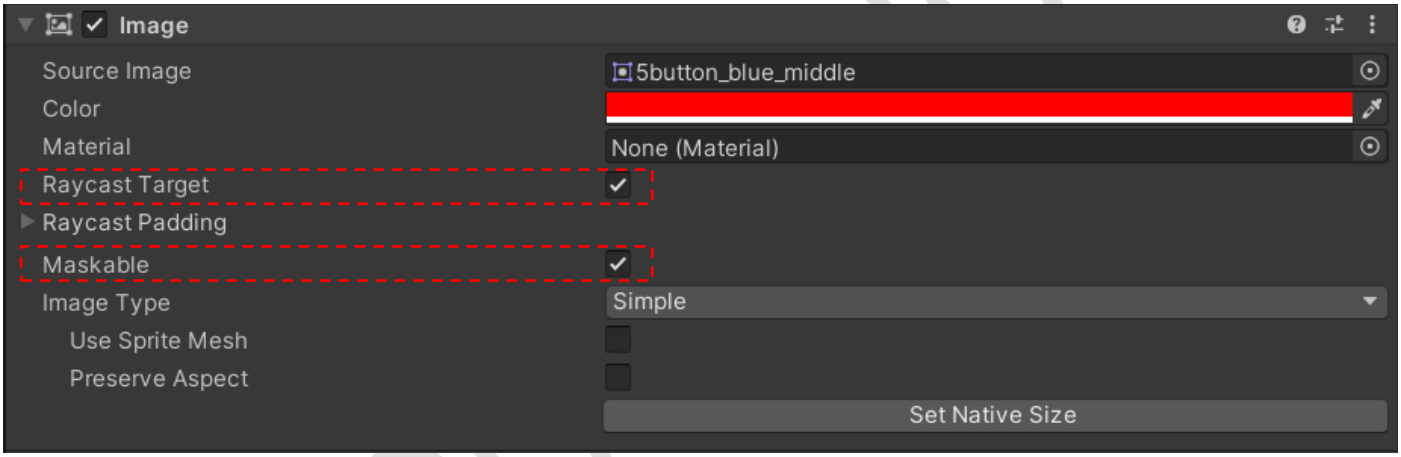
No canvas desse projeto só é preciso desabilitar as opções de *Raycast Target* e *Maskable* dos elementos que não usam essas funções.

Gravidade dos problemas: Baixa

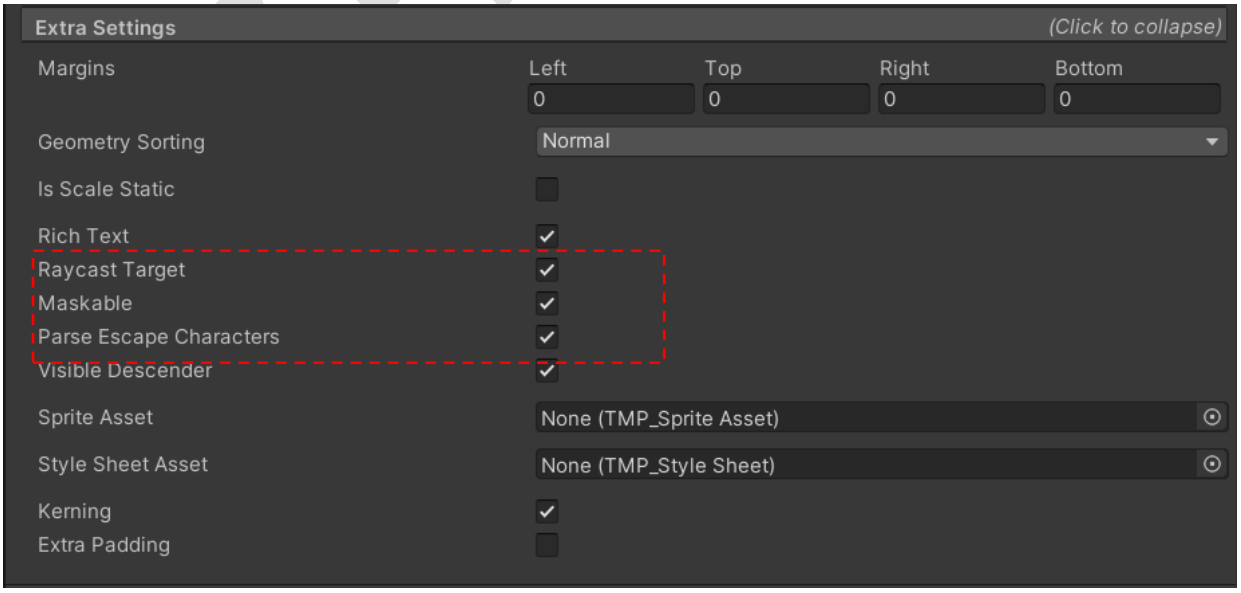
Tempo médio para aplicar melhorias: 10m – 30m

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.



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 e texturas

Tree9_4

Gravidade dos problemas: Médio

Tempo médio para aplicar melhorias: 1h – 1h 30m

Dificuldade para aplicar melhorias: Baixa

As texturas deste modelo precisam ser comprimidas com o **Crunch Compression** e ter seu **Max Size** alterado, tantando balancear tamanho de arquivo com qualidade de imagem.

Além disso, como essa árvore é ultilizada no terreno, é preciso que ela tenha **LOD**s afim de diminuir a contagem de vértices na cena. Existes assets que criam LODs automáticos sem que se seja preciso cria-los no software 3D

Texturas de terreno/Texturas das armas/ Texturas das caixas de munição/ Texturas dos zumbis

Gravidade dos problemas: Médio

Tempo médio para aplicar melhorias: 5m -10m

Dificuldade para aplicar melhorias: Baixa

Todos esse modelos precisam ser comprimidas com o **Crunch Compression** e diminuir o **Max Size**, tentando balancear tamanho de arquivo com qualidade de imagem.

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

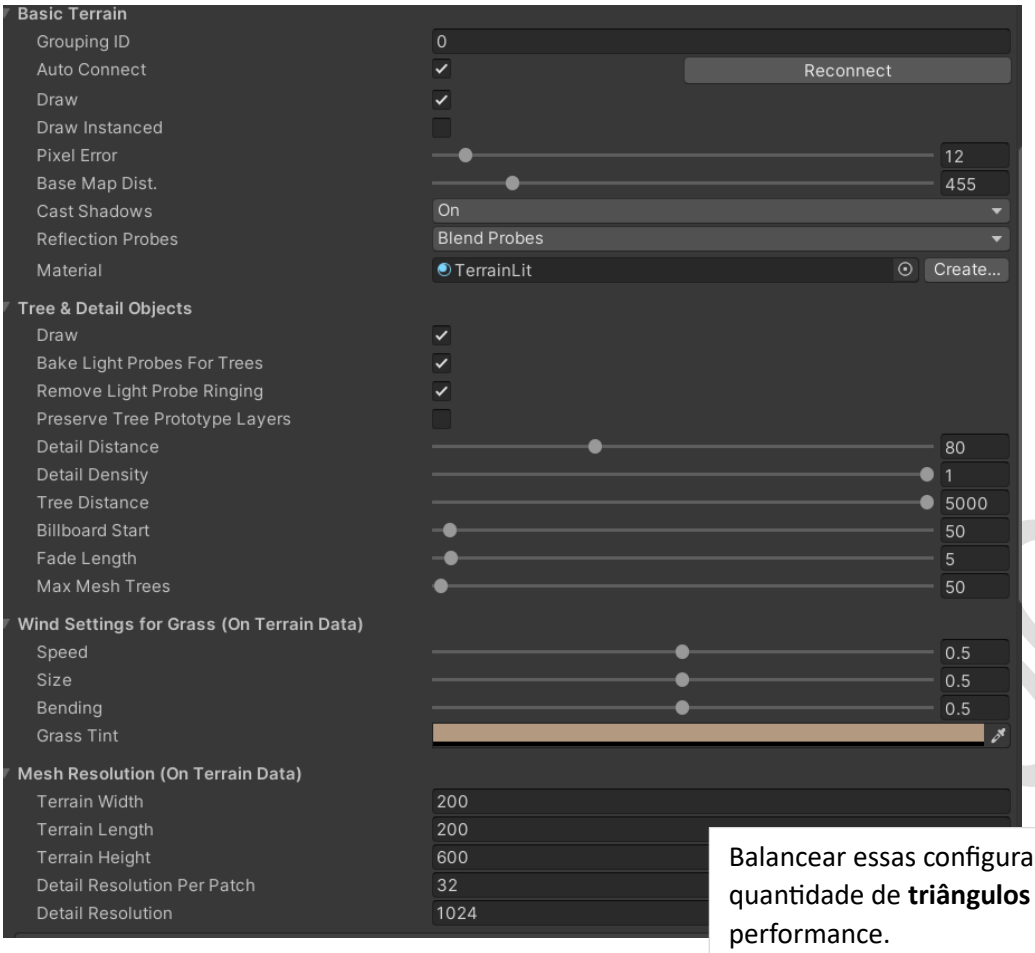
Terreno

Gravidade dos problemas: Médio

Tempo médio para aplicar melhorias: 4h – 5h

Dificuldade para aplicar melhorias: Baixa

As configurações de terreno não foram modificads ideais afim de melhorar o processamento. Essas configurações de terreno são mais complexas de mudar, pois exigem diversos testes tendo em vista que os valores ideais mudam de projeto para projeto. Porém, algumas das configurações que poderiam está melhor são **Pixel Erro**, **Base Map Dist**, **Tree Distance**, **Billiboard Start**, **Fade Length** e **Deatail Distance**, além das configurações de resolução de detalhes.



Balancear essas configurações diminuiem o ***draw call*** e a quantidade de **triângulos e vértices**, logo melhoram a performance.

HUD/Sprites 2D

Resumo

É preciso juntar os arquivos de srpite ultilizados na Hud em um *srpite sheet*. Além disso, este *sprite sheet* deve ser comprimido com o *Crunch Compression*.

Gravidade dos problemas: Médio

Tempo médio para aplicar melhorias: 30m – 1h

Dificuldade para aplicar melhorias: Baixa

No projeto, cada spirte ultilizado é um arquivo diferente o que aumenta a quantidade de **draw call**. Além disso, esses arquivos não estão comprimidos e no formato **POT** (arquivo com tamanho de 2 elevando a uma potência, como 512x512, 1024x1024, 2048x2048...), que é o mais recomendado para jogos.

Portanto o melhor a se fazer é um **sprite sheet** no formato **POT** com todos os sprites que serão ultilizados no jogo. Depois disso é só comprimir com o **Crunch Compression**, tentando manter a qualidade da imagem.

Fazendo isso, a quantidade de draw call diminuir quando o HUD está ativo, e ainda diminui o tamanho do arquivo para a build.

Desempenho Builds

Configurações de pc	***Média FPS***	***Média MS***
Intel Core i7-10750H *CPU 2.60GHz* *RAM 8.00 GB* *SSD 512* *GeForce GTX*	162	6.1
Intel Core i3-6100U *CPU @ 2.30GHz 2.30 GHz* *RAM 4.00 GB*	18	55

X _____

NAME