ATUANDO NA CONSTRUÇÃO DE APLICAÇÕES DE ALTO DESEMPENHO PARA PLATAFORMAS MÓVEIS UTILIZANDO A UNITY ENGINE.

Mauro Antônio Ubaldo Pereira Júnior[1]

Valeria Guerra[2]

Resumo

Esse artigo aborda uma série de técnicas de otimização para a construção de jogos para dispositivos móveis utilizando a Unity Engine. Durante o desenvolvimento do estudo foram aplicadas técnicas com o objetivo de construir um jogo de alta qualidade, e alto desempenho, otimizado o bastante para rodar em um dispositivo Android. Vale ressaltar que devido a isso, o dispositivo recebeu altas cargas de estresse, de forma que houvesse a possibilidade de avaliar seus limites, e extrair o melhor resultado possível com a construção da aplicação desenvolvida.

Palavras Chaves: Unity3D, Otimização, Plataformas Móveis, Programação de Jogos, Técnicas de Otimização.

Abstract

This article covers a number of optimization techniques for building mobile games using the Unity Engine. During the development of the study, techniques were applied in order to build a high-quality, high-performance game, optimized enough to run on an Android device. It is noteworthy that because of this, the device received high stress loads, so that there was the possibility of evaluating its limits, and extracting the best possible result with the construction of the developed application.

[1] Aluno do Curso de Pós-Graduação, em nível de Especialização, em Desenvolvimento de Software Com Metodologia Ágil, UNIVERSIDADE ANHEMBI MORUMBI.

[2] Professora do curso de Pós Graduação em Desenvolvimento de Software Com Metodologia Ágil da Universidade Anhembi Morumbi, Mestre em Comunicação - UAM.

Keywords: *Unity3D, Optimization, Mobile, Game Programming, Optimization Techniques*.

Introdução

Este estudo tem como objetivo mostrar e discutir uma série de técnicas de otimização para auxiliar na construção de aplicações de alto desempenho com Unity, tendo como alvo dispositivos móveis.

As técnicas apresentadas neste estudo podem ser aplicadas por qualquer pessoa com conhecimento prévio de Unity, isso porque são apresentadas com explicações teóricas, e práticas em através de um projeto de demonstração.

Durante a demonstração as técnicas são colocadas em prática, com o objetivo de gerar uma aplicação de alta qualidade, e alto desempenho, visando fazer com que ela possa ser executada em um Smartphone.

Ao final do estudo, é apresentada uma conclusão com os resultados de tudo que foi aplicado durante o desenvolvimento, junto com algumas boas práticas não mencionadas neste artigo e que também podem ser utilizadas para a otimização de aplicações para plataformas móveis utilizando a ferramenta Unity Engine.

1. Discussão Teórica

1.1 - Mesh Renderer

Mesh Renderer é o componente da Unity Engine responsável por renderizar a malha tridimensional selecionada no Componente *Mesh Filter*. No componente *Mesh Renderer* temos diversos campos que nos ajudam a trabalhar com objetos 3D na Unity Engine, isso faz dele um componente de alto valor valor, pois nele é possível fazer configurações que impactam especificamente na aparência de um GameObject portador de um modelo 3D e também em como esse GameObject vai interagir com o ambiente a sua volta, essas interações podem ser com luzes, sombras, e reflexos, também é possível fazer algumas configurações no *LightMap*, caso a opção *Contribute Global Illumination* esteja ativada, e o campo Receive Global Illumination esteja com a opção *LightMaps* selecionada.

1.2 - Mesh Renderer - Reflection Probe Mesh Renderer

O *Mesh Renderer* também possui uma propriedade denominada *Reflection Probe,* essa propriedade trabalha com a forma pela qual o objeto vai receber os reflexos oriundos do *Reflection Probe* encontrado em *GameObject > Lights > Reflection Probe*.[3]

De acordo com a documentação da Unity o campo *Reflection Probe* do *Mesh Renderer* contém as seguintes opções:

- *Off:* Desabilita a interação com a área *Reflection probe* da Unity, nesse caso o *Mesh Renderer* passa a utilizar o *Skybox* para a geração dos reflexos.

- *Blend Probes:* Habilita a interação do *Mesh Renderer* com as áreas de *Reflection Probe*, fazendo uma mistura de efeitos nos momentos em que existe a presença de mais de uma área de reflexão.

- *Blend Probes And Skybox:* Habilita interação do *Mesh Renderer* com as áreas do *Reflection Probe*, fazendo uma mistura de efeitos entre os próprios *Probes*, ou entre os Probes e os reflexos gerados pelo *Skybox*.[4]

- *Simple:* Habilita a interação do *Mesh Renderer* com as áreas do *Reflection Probe*, porém não faz nenhuma combinação caso haja mais de uma área de *Reflection Probe* sobreposta.

O uso de *Reflection Probes*, por mais que seja interessante do ponto de vista visual, traz algumas desvantagens quando o assunto é desempenho, isso porque segundo a própria documentação da Unity Engine ele leva uma grande quantidade de tempo para ser processado.

---

[3] É importante ressaltar que o campo *Reflection Probe* do *Mesh Renderer* é uma propriedade específica do componente, ela é diferente do Reflection Probe encontrado no menu de luzes. O *Reflection Probe* encontrado no menu de luzes trata-se de um objeto que cria uma área reflexiva em uma determinada região do cenário. Vale ressaltar que a forma como o *Mesh Renderer* está configurado influencia a geração de reflexos da área de reflexão.

[4] *Reflection Probe*: Componente da Unity Engine responsável por criar reflexos em determinadas áreas do nível de jogo. Esse componente pode ser encontrado no menu *Light*.

1.3 - Object Pooling

Object Pooling é um *Design Pattern*[5], citado em livros como: *Game Programming Patterns*, e *Hands-On game development patterns with Unity 2019*, e em vários outros lugares, trata-se de um *Design Pattern* que aplica o conceito de reutilização de objetos, seu uso normalmente ocorre em casos onde se faz necessário a criação e destruição de múltiplos objetos de maneira constante, é através da utilização desse *Pattern* que podemos evitar o uso massivo dos métodos *Instantiate* e *Destroy* que quando utilizados de maneira constante, podem causar problemas de desempenho devido ao alto uso da CPU.

No desenvolvimento de games o *Object Pooling* pode ser utilizado para diversas coisas como: Tiros, *Scroll* de objetos, criação de inimigos, power-ups, etc.

1.4 - Iluminação - Baked, Realtime e Mixed Lighting

A Unity possui um sistema de luz que nos permite obter bons resultados através de suas mais diversas configurações. A configuração de luz ideal, potencializada pelo *Post-Processing* pode mudar o resultado final de um jogo dando mais realismo, e aumentando a imersão, no entanto é preciso tomar cuidado, afinal uma iluminação com uma configuração inadequada pode gerar quedas no desempenho de um jogo.

Quando pensamos em luzes, é importante lembrar que, a Unity Engine nos oferece 3 tipos de luz que são:

- *Realtime:* Uma luz configurada como *Realtime* recebe atualizações em tempo real durante a execução do jogo.

- *Baked:* Segundo a própria documentação, quando uma luz é configurada como *Baked*, a Unity faz os cálculos de iluminação no editor e salva seus resultados em disco, esses dados apenas são carregados durante o jogo, e através disso o ambiente é iluminado. O simples fato dos cálculos complexos serem

---

[5] Segundo uma definição escrita por Iutti em um artigo publicado em opus-software.com.br, *Design Patterns* são padrões generalistas para resolver problemas recorrentes no desenvolvimento de um software.

executados com antecedência faz com que as *Baked Lights*, reduzam o custo de shading e também o custo da renderização das sombras.

- *Mixed:* Um artigo publicado no Unity Learn, aponta que uma luz configurada como *Mixed*, além de ser incluída nos *LightMaps*, ainda é capaz de contribuir com a iluminação em tempo real para objetos que não possuem a marcação *static*.

1.5 - Sombras

A construção da atmosfera de um jogo, leva em consideração muitos fatores, entre eles, a iluminação e o sombreamento.

Um trabalho de luz e sombras bem executado, aumenta o realismo e melhora a experiência do próprio jogador, levando em consideração tamanha importância, abordaremos elas neste tópico.

Um documento escrito por Cristiano Ferriera e Steve Hughes, publicado pela Intel durante o ano de 2015, menciona que elas podem consumir uma quantidade elevada de recursos da GPU. Levando isso em consideração, é muito importante dar atenção especial para sua otimização, especialmente em cenários com bastante complexidade. Um artigo escrito pela Google na documentação oficial do Android, aponta que o processo de sombreamento utilizado pela Unity faz o uso de uma técnica conhecida como *Shadow Map*[6].

De acordo com a própria documentação da Unity uma luz pode gerar dois tipos de sombra, são elas:

- *Hard Shadows:* A documentação oficial da Unity aponta que as *Hard Shadows* são responsáveis por criar sombras nítidas, mas com custo de processamento inferior ao das *Soft Shadows*.
- *Soft Shadows:* De acordo com a documentação oficial da Unity, as *Soft Shadows* são sombras mais realistas, e tendem a reduzir

---

[6] *Shadow Map*, segundo um artigo da AutoDesk, *Shadow Map* é uma técnica para geração de sombras.

o efeito de serrilhado criado pelo *Shadow Map*, ainda segundo a própria documentação, as *Soft Shadows*, geram uma sobrecarga nos processos de renderização superior a das *Hard Shadows*, nesse caso as sombras criadas, geram mais trabalho adicional para a GPU, do que para a CPU.

1.6 - Texture Importer

O *Texture Importer* é a ferramenta de importação e configuração de texturas dentro da Unity Engine, assim como várias outras ferramentas ele tem uma grande importância na hora de otimizar uma aplicação, uma vez que a otimização das texturas dentro da Unity é feita através dele.

A Unity possui uma série de configurações *Default* que são aplicadas ao *Texture Importer*, essas configurações nem sempre são as mais adequadas para cada situação, o que em muitos casos pode fazer com que uma textura perca a qualidade ou acabe utilizando mais recursos do que o necessário.

O *Texture Importer* possui uma série de opções, porém vamos nos atentar neste momento a região denominada *Platform - Specific Overrides*.

1.7 - Texture Importer - Platform-Specific Overrides - Max Size

Um dos recursos mais importantes dessa região é o campo *Max Size*. Por padrão a Unity faz sua configuração em 2048 pixels, porém como muitas vezes utilizamos texturas com menor e maior tamanho, é importante ficar atento a esse detalhe para que seja possível efetuar a configuração adequada para cada textura, lembre-se que a medida em que esse valor aumenta, o uso de memória da textura também tende a aumentar. Vale ressaltar que em alguns casos, aumentar esse valor pode não produzir melhorias na qualidade da textura, isso ocorre quando o *Max Size* é redefinido para um valor maior que o suportado pela própria textura. Em contrapartida, em alguns casos reduzir esse valor pode ocasionar em perda de

qualidade, pelo simples fato de que, o valor atribuído é inferior ao mínimo necessário para uma boa qualidade da textura.

1.8 - Texture Importer - Format

Segundo a documentação da própria Unity Engine, as próprias GPUs fazem o uso de formatos de imagem diferentes dos comumente utilizados cotidianamente por nós, isso ocorre devido ao fato de que, os formatos suportados pela GPU, são mais otimizados. A documentação ainda prossegue dizendo que o formato de compactação de textura afeta diversos aspectos de um jogo, podemos citar alguns deles como: o uso de memória, a qualidade visual, o tempo de compilação, entre outros.

O Unity possui alguns formatos de compressão que podem ser selecionados de acordo com a plataforma alvo, basta selecionar a plataforma e fazer um *Override* no próprio *Texture Importer*.

1.9 - Texture Importer - Compression

Segundo a documentação da Unity Engine, texturas comprimidas resultam em tempos de carregamento mais rápidos, menor consumo de memória e desempenho de renderização aumentado.

Por padrão, quando utilizamos o preset *Default* da Unity, a compressão é definida como *Normal Quality*, porém segundo a própria documentação, a Unity possui algumas opções de compressão que são:

- *None:* Onde não há compressão da textura.
- *Low Quality:* Onde a compressão da textura é feita em formato de baixa qualidade.
- *Normal Quality:* Onde a compressão da textura é feita em um formato padrão.

- *High Quality:* Onde a compressão da textura é feita em formato de alta qualidade.

Quando utilizados o *preset override for Android*, as opções de compressão podem mudar de acordo com o formato de compressão escolhido.

2.0 - Texture Importer - MipMap Streaming

O Sistema de *MipMap Streaming*, é um sistema da Unity Engine que permite reduzir a quantidade de memória utilizada para efetuar o carregamento das texturas, uma vez que de acordo com a própria documentação, a Unity carrega apenas os *MipMaps* necessários para renderizar o campo de visão da câmera, ainda de acordo com sua documentação, esse recurso garante que seja utilizado uma pequena fração dos recursos da CPU, para que seja feita uma grande economia dos recursos da GPU.

2.1 - Occlusion Culling

Segundo a documentação da própria Unity Engine, o processo de *Occlusion Culling*, é um processo que impede a Engine de realizar a renderização de determinados objetos que estão ocultos por outros.

O sistema de *Occlusion Culling* trabalha com o conceito de objeto *Occluder* e *Occludee*.

Segundo um documento escrito por Cristiano Ferriera e Steve Hughes, e publicado pela Intel durante o ano de 2015, os conceitos de *Occluder Static* e *Occludee Static* tratam-se de:

> *Occluder Static*: Objeto o qual recebe a marcação de *Occluder Static*, ele atua como uma espécie de obstáculo impedindo o objeto marcado como *Occludee* de ser renderizado.

Occludee Static: Objeto o qual recebe a marcação de *Occludee Static*, essa marcação faz com que esse objeto possa ser ocluído por um objeto com a marcação de *Occluder Static*.

Em outras palavras, *Occluders* são objetos que ocultam, e *Occludees* são objetos que são ocultos.

Occlusion Area: É um componente que faz o uso de uma área para gerar o efeito de *Occlusion Culling* em uma determinada região do cenário, a própria Unity em sua documentação menciona sobre o uso da *Occlusion Area* para a oclusão de objetos em movimento uma vez que, através dela é possível definir uma região cuja a qual os objetos em movimento estarão presentes.

A Unity utiliza os conceitos de oclusão para garantir que apenas os objetos dentro do campo de visão sejam renderizados.

2.2 - Câmera

Segundo um artigo escrito por Bertrand Guay-Paquet em um web blog oficial da própria Unity, uma câmera trabalha com o campo de visão do usuário, ainda nesse mesmo artigo é mencionado o fato de que a câmera também pode determinar o conjunto de objetos visíveis, através do processo denominado *Oclusion Culling* (Processo já mencionado anteriormente neste artigo).

Como mencionado, a câmera define o campo de visão do usuário, quando criamos uma Câmera na Unity, ela vem com valores *Default*, esses valores podem criar um campo de visão grande demais ou pequeno demais, o que varia de acordo com a necessidade de cada projeto, um campo de visão maior do que o necessário, acaba por gerar uma área de renderização maior, o que pode resultar em uma área quantidade maior de *Draw Calls*, uma vez que a Unity vai renderizar todos os objetos dentro dessa área, caso não exista oclusão.

2.3 Discussão Prática

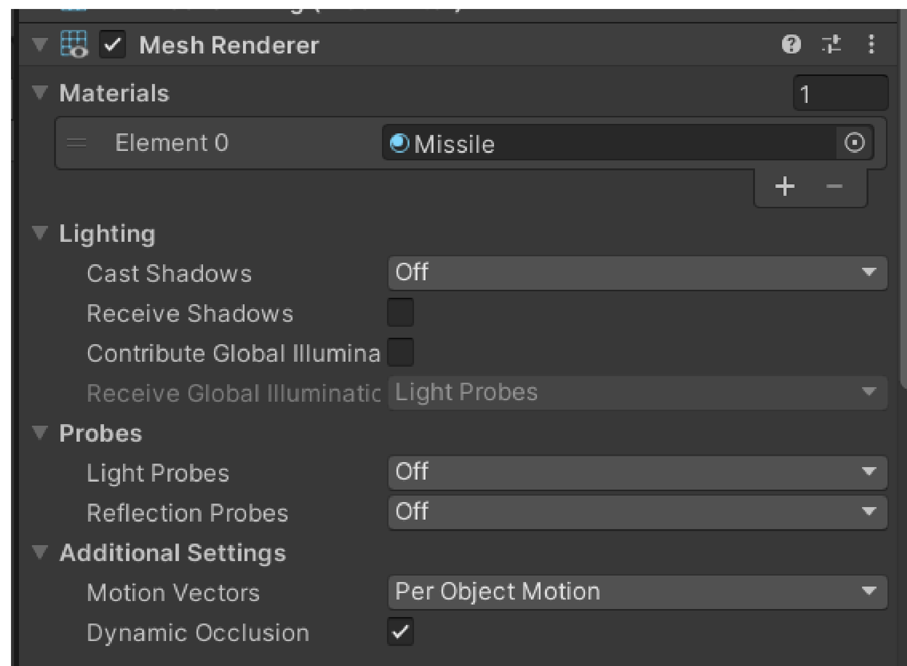
2.4 - Otimizando Mesh Renderer

Neste tópico será abordado de maneira breve a configuração do *Mesh Renderer* para esta demonstração, vale ressaltar que devido ao desligamento das sombras no menu *Quality*, este tópico abordará apenas a forma como foi configurado o campo *Reflection Probe*.

2.5 - Mesh Renderer - Reflection Probe Mesh Renderer

Na demonstração não foram utilizados *Reflection Probes*, então por padrão todos os objetos da cena tiveram seu campo *Reflection Probe* atribuído como *Off*.

Figura 1 - Demonstrando Configuração de um Mesh Renderer de um objeto.



FONTE: (autor, 2021)

2.6 - Object Pooling

Para implementar o pattern *Object Pooling* foram necessários alguns campos, que podem ser vistos na imagem abaixo.

Figura 2 - Campos Para a Implementação do *Pattern Object Pooling*.

```
public GameObject originalPrefab;

public GameObject[] projectiles;

public int objectPool;
public int objectHound;
```

FONTE: (autor, 2021)

Cada um desses campos possui uma função muito importante na implementação, essas funções podem ser conferidas a seguir:

- *OriginalPrefab:* Campo do tipo *GameObject* o qual vai conter o objeto original, que será replicado.
- *Projectiles:* *Array* do tipo *GameObject* que vai conter os objetos criados.
- *ObjectPool:* Campo do tipo inteiro responsável por ser o indexador do *Array Projectiles*. O objetivo do campo *ObjectPool* é garantir que novos objetos sejam colocados em uma posição vazia do *Array*, durante a etapa de criação de novos objetos.
- *ObjectHound:* Campo do tipo inteiro responsável por re-posicionar os objetos do *Array Projectiles* de acordo com seu índice (importante ressaltar que quando esse campo atingir à quantidade máxima de objetos definidos pelo *Length* do *Array Projectiles* seu valor é reiniciado, isso garante que os objetos criados uma vez, sejam reaproveitados sempre que necessário).

Na segunda fase da implementação o código responsável por gerenciar nossos objetos é implementado, podemos dividir a implementação em duas etapas que podemos chamar de:

Criação: Tem como objetivo criar novos objetos enquanto o valor presente em *ObjectPool* for menor que o valor presente no *Length* do *Array Projectiles*.

Reutilização: Tem como objetivo, reposicionar os objetos criados enquanto o valor presente em *ObjectHound,* for menor do que valor presente no *Length* do *Array Projectiles*, quando o valor presente no campo *ObjectHound* for igual ao do *Length* de *Projectiles,* seu valor volta a ser 0, o que garante que seja criado um ciclo de reutilização constante. [7]

Figura 3 - Implementação do Pattern Object Pooling.

```
void PoolingShoot(Vector3 angleForInstance)
{
    if (objectPool < projectiles.Length)
    {
        GameObject objectForCreate = Instantiate(originalPrefab, cannon.position,
                                                 Quaternion.Euler(angleForInstance.x, angleForInstance.y, angleForInstance.z));
        projectiles[objectPool] = objectForCreate;
        objectPool++;
        if (gameInfosInstance == null)
            gameInfosInstance = GameObject.Find("Utilities").GetComponent<GameInfos>();
        gameInfosInstance.RefreshObjectNumberCounter();
    }
    else
    {
        if (objectHound >= projectiles.Length)
            objectHound = 0;

        if (!projectiles[objectHound].activeSelf)
            projectiles[objectHound].SetActive(true);

        projectiles[objectHound].transform.position = cannon.position;
        objectHound++;
    }
}
```

FONTE: (autor, 2021)

Durante a demonstração o *Design Pattern Object Pooling* foi usado na criação dos tiros, tanto do *Player* quanto dos Inimigos, dessa forma foi possível garantir que cada inimigo disparasse 5 tiros, e o Player disparasse 20 tiros, impedindo o alto uso dos recursos da CPU, afinal uma vez instanciados, os tiros apenas são reaproveitados.

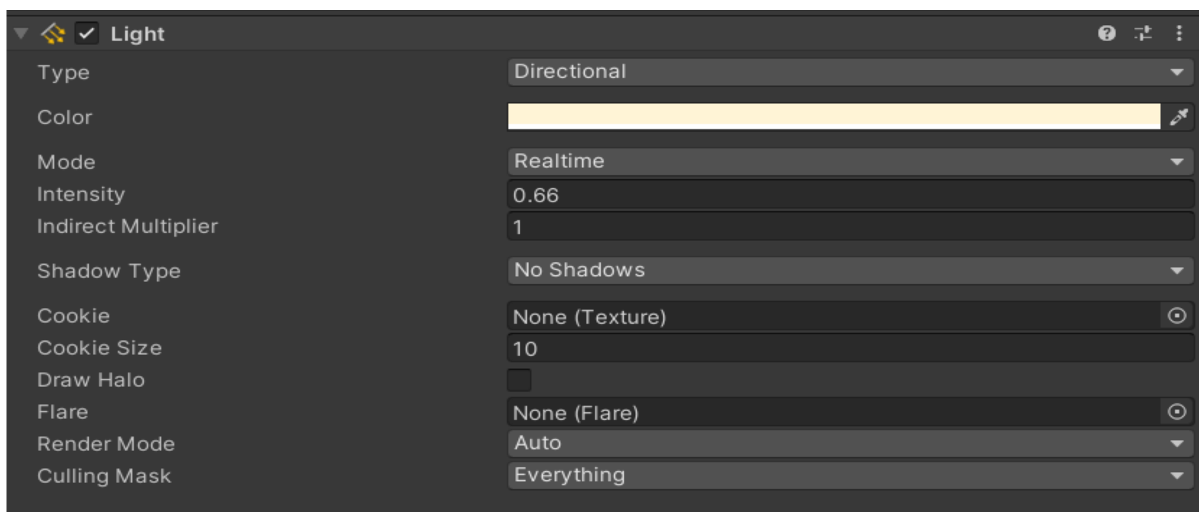
[7] Importante ressaltar que a operação de destruição não ocorreu em momento algum durante a construção da demonstração, nesse caso os tiros apenas são desativados através do método *SetActive* quando o sistema identifica a colisão dos projéteis com o *Player* ou dos projéteis com os Inimigos. É possível notar logo abaixo da instrução *else* a existência de uma chamada para *SetActive* sendo efetuada com o parâmetro *true.*

2.7 - Iluminação - Baked, Realtime e Mixed Lightning

Para o desenvolvimento desta demonstração, foi utilizado apenas uma *Directional Light* aplicada em *Realtime*, com uma intensidade de 0.66.

No caso da demonstração presente neste estudo, não foi necessário fazer a utilização de um sistema de iluminação complexo, sendo assim, foi utilizado apenas uma *Directional Light*.

Figura 4 - Configurações da *Directional Light*.



FONTE: (autor, 2021)

2.8 - Sombras

A Unity oferece diferentes escopos na hora de configurar as sombras que são:

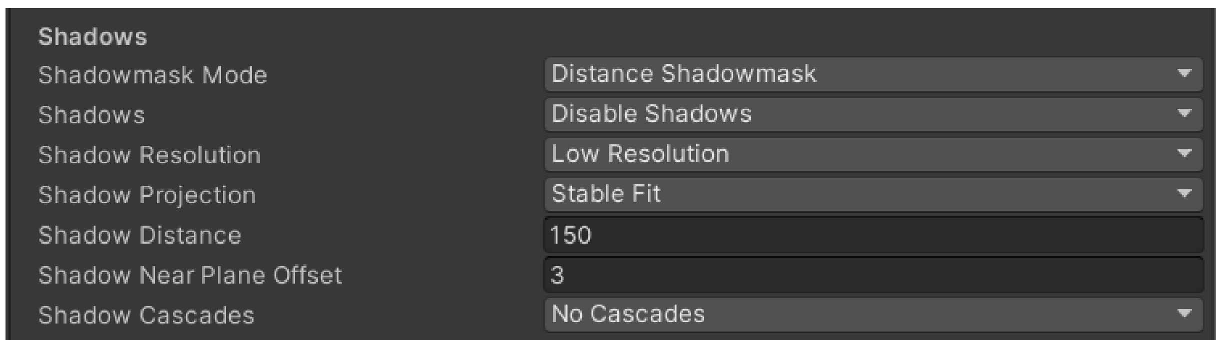
- Componente *Light*
- *Mesh Renderer*
- Seção Quality (*Project Settings* > *Quality*)

Cada um desses escopos possui uma maneira de atuação, e formas de configuração distintas diferentes. A configuração pelo componente *Light* trabalha as sombras dentro do alcance de uma fonte de luz. A configuração pelo *Mesh Renderer*

modifica a forma como a sombra interage com um determinado objeto, enquanto que, as configurações da seção Quality influenciam todo o contexto do projeto.

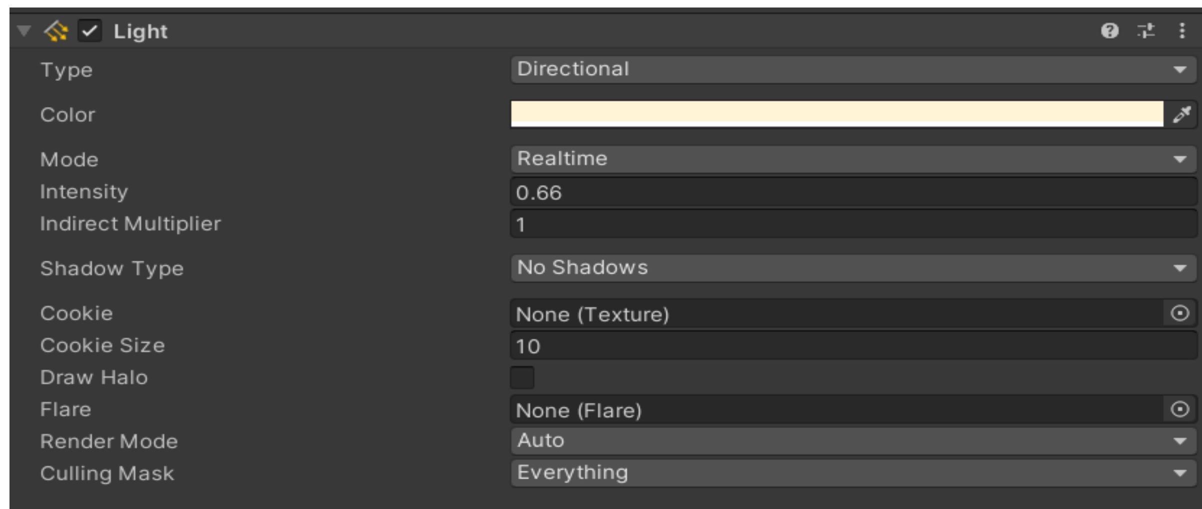
Na demonstração as sombras foram desabilitadas no escopo geral do projeto, como mostra a imagem abaixo.[8]

Figura 5 - Configurando as sombras através do menu Quality presente em Project Settings.



FONTE: (autor, 2021)

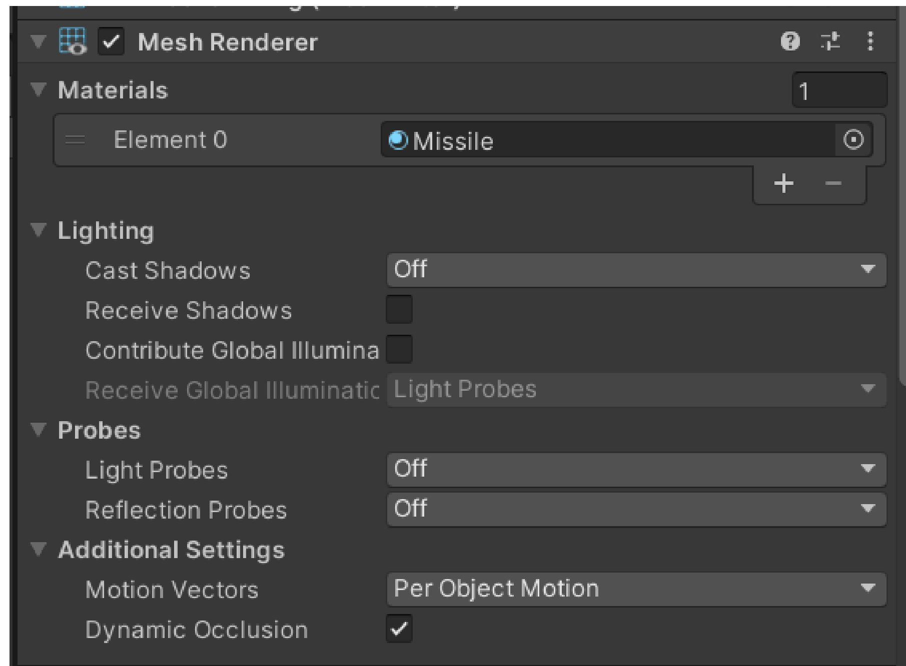
Figura 6 - Imagem Ilustrativa das sombras no Componente *Light*.



FONTE: (autor, 2021)

[8] Por uma questão meramente ilustrativa serão colocadas algumas imagens subsequentes para mostrar as configurações no *Mesh Renderer* e no Componente *Light*.

Figura 7 - Imagem Ilustrativa das sombras no *Mesh Renderer*



FONTE: (autor, 2021)

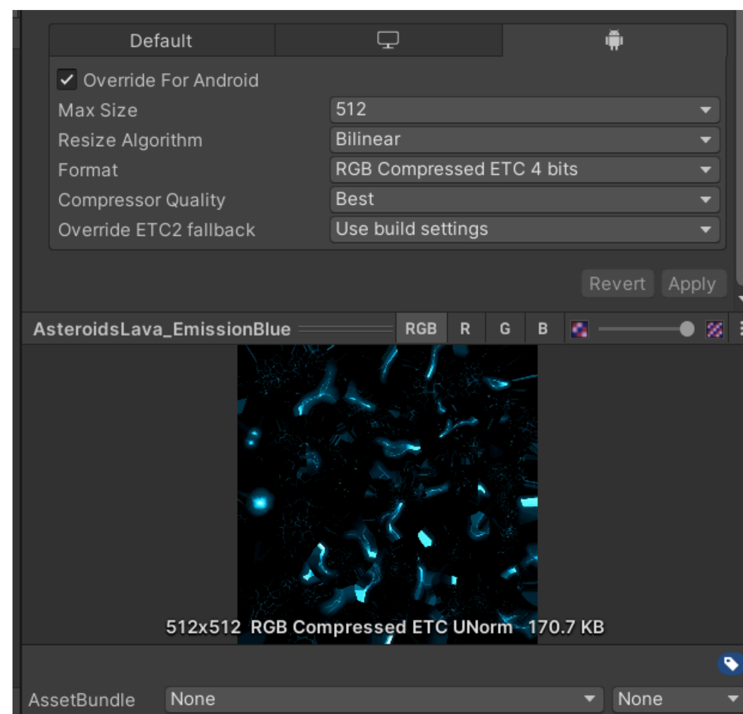
2.9 - Texture Importer

Para a aplicação na demonstração o *Texture Importer* foi trabalhado de maneira cuidadosa, visando fazer com que as texturas fossem otimizadas, sem sacrificar a qualidade gráfica do jogo, nas próximas seções será possível entender melhor sobre as configurações aplicadas na aba *Platform-Specific Overrides*, que foram de extrema importância para garantir que fosse possível alinhar qualidade gráfica e performance, sem que fosse possível sacrifícios em qualquer uma dessas duas frentes, nos exemplos abaixo é possível notar que a qualidade do compressor, e as opções de formato são diferentes do padrão, devido ao fato de que a opção *Override For Android* foi habilitada, além da aba Platform-Specific Overrides, também será mostrado como foi configurado o *MipMap Streaming* para a aplicação nesse projeto de demonstração.

3.0 - Texture Importer - Platform-Specific Overrides - Max Size

A aplicação do *Texture Importer* nos asteroides se deu através da seguinte configuração: foi utilizado um *Max Size* de 512 Pixels, que resultou em uma redução (Visível no próprio *Texture Importer*) de 0.7 MB, para 170.7KB, o formato de compressor escolhido foi o *RGB Compressed ETC 4 Bits* devido ao fato de ser um formato leve, e que mesmo sendo um formato de baixa qualidade não gerou perdas gráficas perceptíveis, por fim a qualidade do compressor foi utilizada em *Best* para que fosse obtido um resultado sem perda de qualidade, essa configuração não resultou em aumentos de memória segundo o próprio *Texture Importer*. Essas Configurações foram aplicadas de maneira padrão em todas as texturas do asteroide.

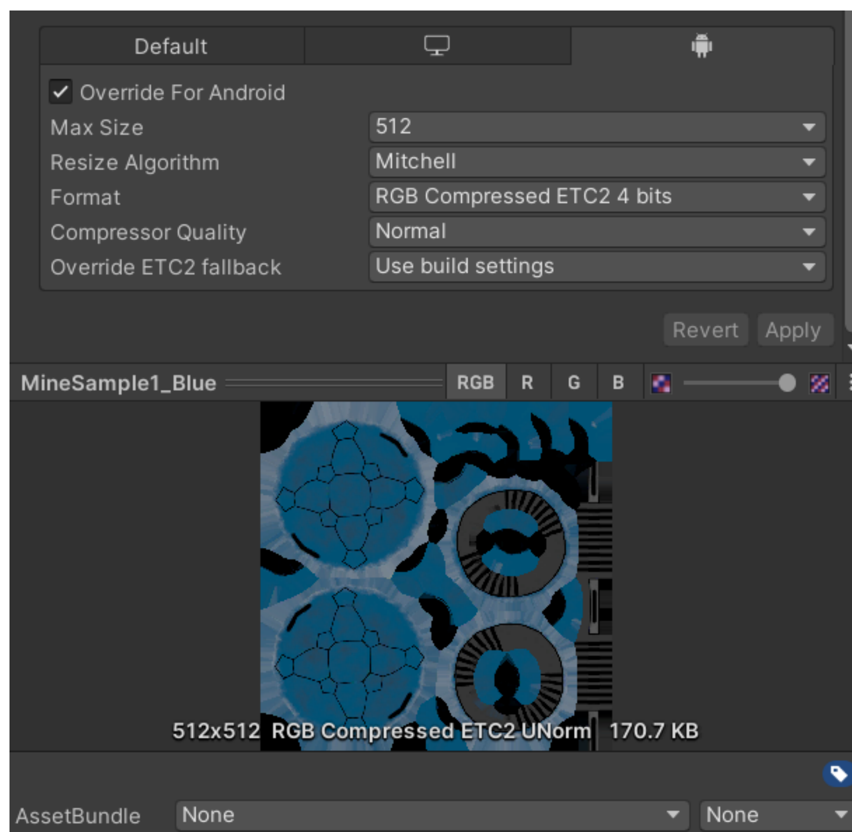
Figura 8 - Configuração da janela Platform Overrides nas Texturas dos Asteroides.



FONTE: (autor, 2021)

A aplicação do *Texture Importer* nas minas se deu através da seguinte configuração: foi utilizado um *Max Size* de 512 pixels, o que nesse caso não resultou em alterações de tamanho na memória, devido ao fato de que o tamanho definido da configuração *Default* era maior do que o suportado pela própria textura, nesse caso o *Max Size* apenas foi ajustado para o tamanho corresponde ao tamanho da textura, o formato de compressor escolhido foi o *RGB Compressed ETC 4 Bits* devido ao fato de ser um formato leve, e que mesmo sendo um formato de baixa qualidade não gerou perdas gráficas perceptíveis, por fim a qualidade do compressor foi utilizada em *Best* para que fosse obtido um resultado sem perda de qualidade, essa configuração não resultou em aumentos de memória segundo o próprio *Texture Importer*. Essas configurações foram aplicadas de maneira padrão em todas as texturas das minas.

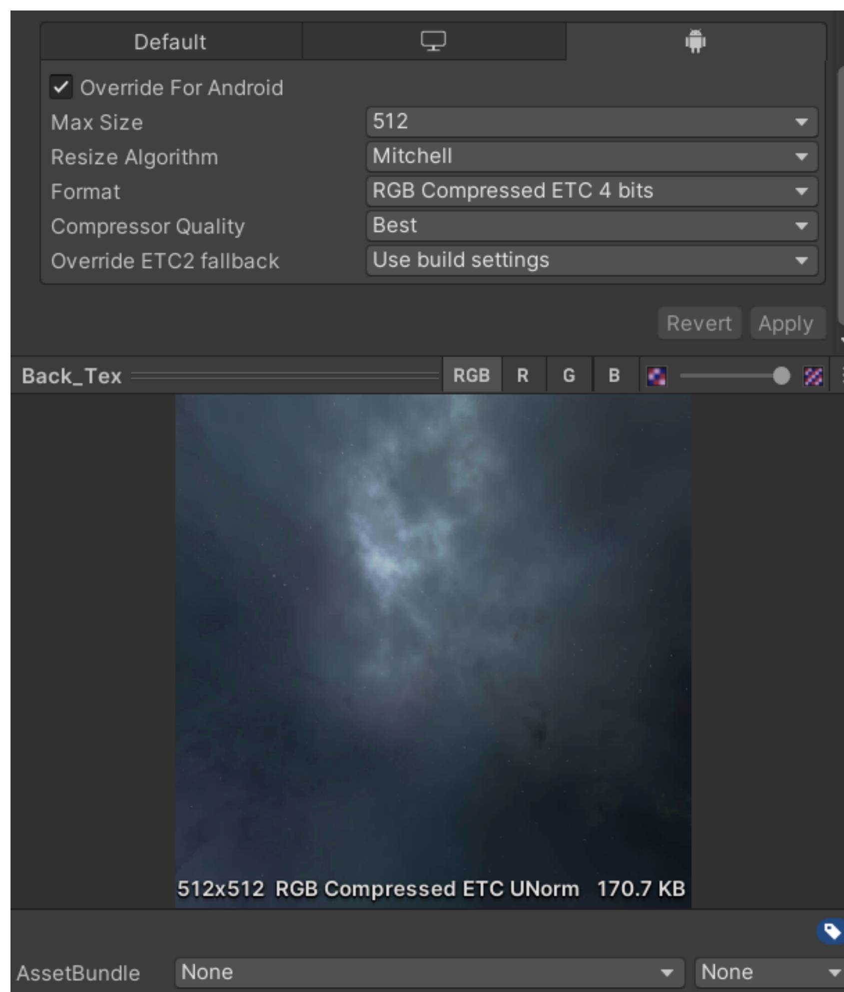
Figura 9 - Configuração da janela Platform Overrides nas Texturas das Minas.



FONTE: (autor, 2021)

A aplicação do *Texture Importer* no *Skybox* se deu através da seguinte configuração: foi utilizado um *Max Size* de 512 pixels, o que resultou em uma redução (Visível no próprio *Texture Importer*) de 2.7 MB, para 170.7KB, o formato de compressor escolhido foi o *RGB Compressed ETC 4 Bits* devido ao fato de ser um formato leve, e que mesmo sendo um formato de baixa qualidade não gerou perdas gráficas perceptíveis, por fim a qualidade do compressor foi utilizada em *Best* para que fosse obtido um resultado sem perda de qualidade, essa configuração não resultou em aumentos de memória segundo o próprio *Texture Importer*. Essas configurações foram aplicadas de maneira padrão em todas as texturas do *Skybox*, que possuía um total de 6 texturas.

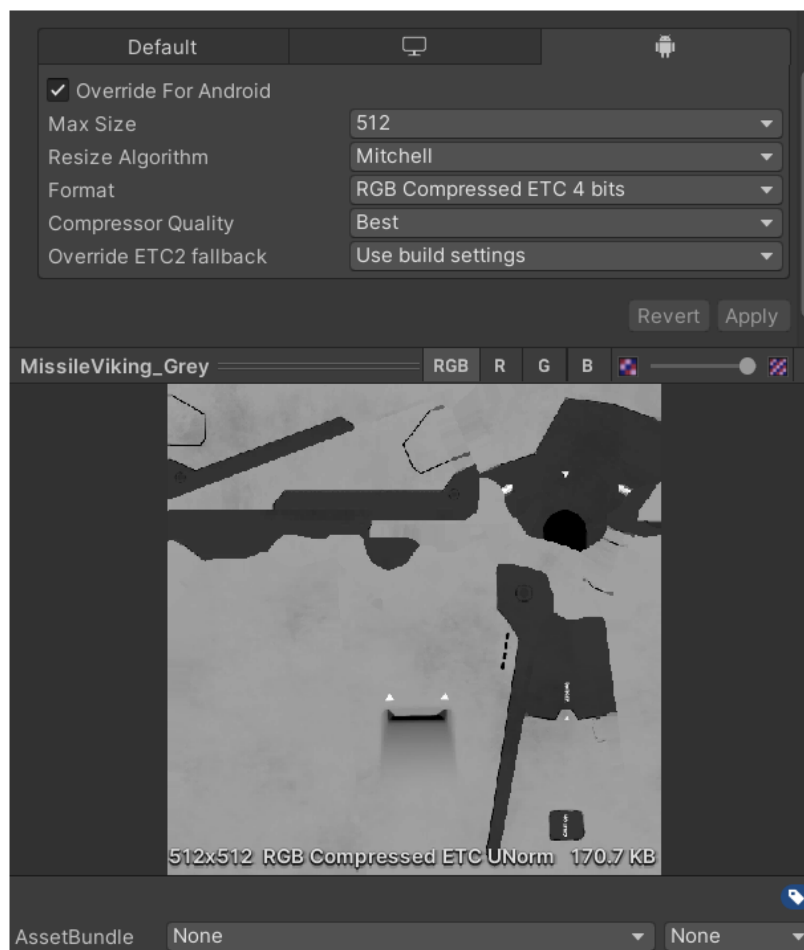
Figura 10 - Configuração da janela Platform Overrides nas Texturas do *Skybox*.



FONTE: (autor, 2021)

A aplicação do *Texture Importer* nos Mísseis se deu através da seguinte configuração: foi utilizado um *Max Size* de 512 pixels, o que nesse caso não gerou alterações de tamanho na memória, devido ao fato de que o tamanho da configuração *Default* era maior do que o suportado pela própria textura, o formato de compressor escolhido foi o *RGB Compressed ETC 4 Bits* devido ao fato de ser um formato leve, e que mesmo sendo um formato de baixa qualidade não gerou perdas gráficas perceptíveis, por fim a qualidade do compressor foi utilizada em *Best* para que fosse obtido um resultado sem perda de qualidade, essa configuração não resultou em aumentos de memória segundo o próprio *Texture Importer*. Essas configurações foram aplicadas de maneira padrão no *Albedo* e no *Normal Map* dos Mísseis.

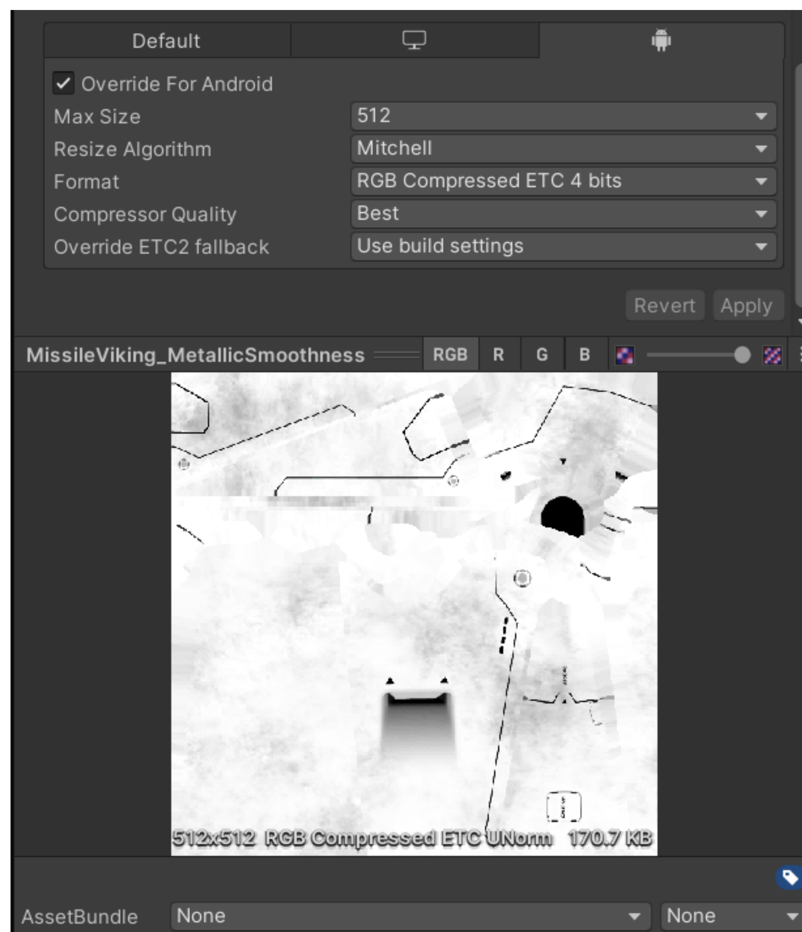
Figura 11 - Configuração da janela Platform Overrides nas Texturas de *Normal* e *Albedo* dos Mísseis.



FONTE: (autor, 2021)

A aplicação do *Texture Importer* no *Metallic* dos Mísseis se deu através da seguinte configuração: foi utilizado um *Max Size*, de 512 pixels, com formato de compressão *RGB Compressed ETC 4 Bits* devido ao fato de ser um formato leve, e que mesmo sendo um formato de baixa qualidade não gerou perdas gráficas perceptíveis, por fim a qualidade do compressor foi utilizada em *Best* para que fosse obtido um resultado sem perda de qualidade, essa configuração não resultou em aumentos de memória segundo o próprio *Texture Importer*. [9]

Figura 12 - Configuração da janela Platform Overrides nas Texturas de *Metallic* dos Mísseis.

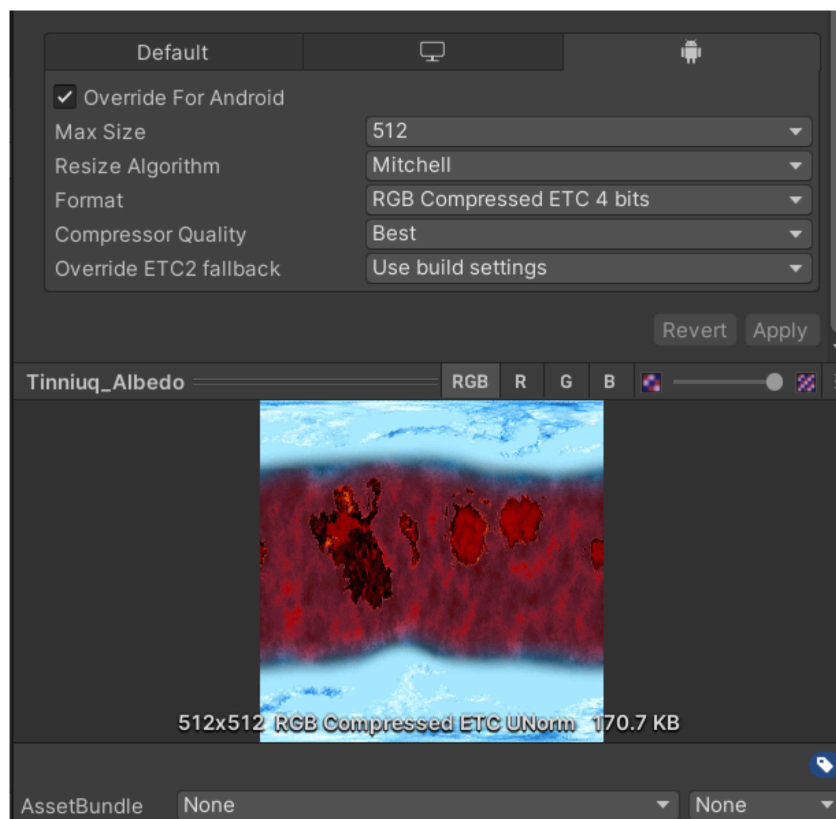


FONTE: (autor, 2021)

[9] No caso específico do *Metallic* dos mísseis podemos ver o compressor fazendo uma alteração significativa no tamanho da textura, quando o *Max Size* foi reduzido (Durante o desenvolvimento da demonstração), a textura se manteve em seu tamanho original de 341.4KB, porém após alterar o formato do compressor a textura passou a consumir apenas 170.7KB, o que abre a possibilidade para diferentes experimentos, que podem gerar outros resultados tanto em termos de otimização quanto em termos gráficos.

A aplicação do *Texture Importer* nos Planetas se deu através da seguinte configuração: foi utilizado um *Max Size* de 2048 para 512 pixels, o que resultou em uma redução (Visível no próprio *Texture Importer*) de 2.7 MB, para 170.7KB, nas texturas *Albedo* e *Normal* e de 1.3MB para 170.7KB na textura *Metallic*, com formato de compressão *RGB Compressed ETC 4 Bits* devido ao fato de ser um formato leve, e que mesmo sendo um formato de baixa qualidade não gerou perdas gráficas perceptíveis por fim a qualidade do compressor foi utilizada em *Best* para que fosse obtido um resultado sem perda de qualidade, essa configuração não resultou em aumentos de memória segundo o próprio *Texture Importer*. Essas Configurações foram aplicadas de maneira padrão em todas as texturas dos Planetas.

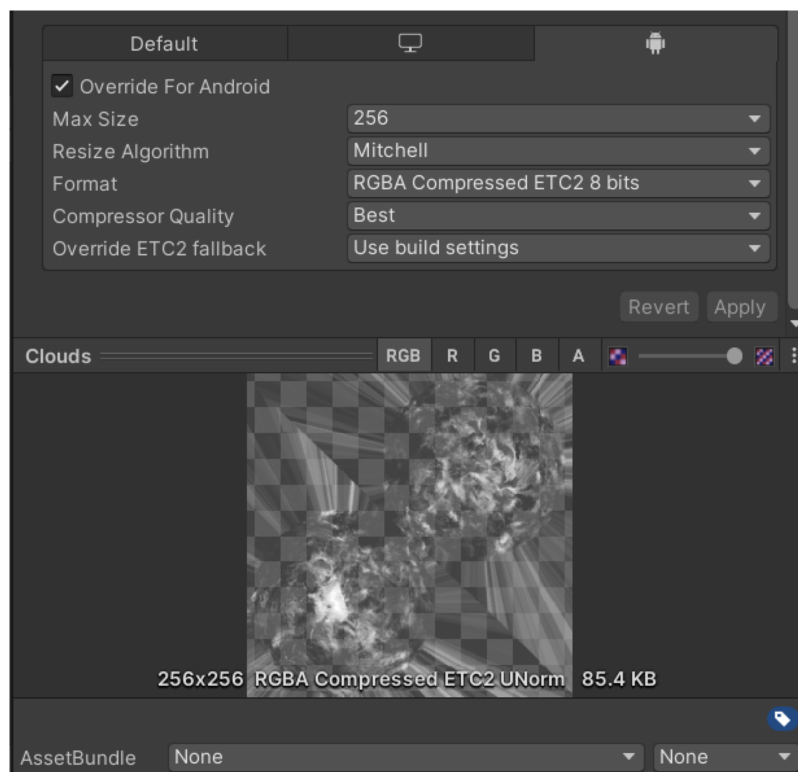
Figura 13 - Configuração da janela Platform Overrides nas Texturas dos Planetas.



FONTE: (autor, 2021)

A aplicação do *Texture Importer* nas nuvens dos planetas se deu através da seguinte configuração: aqui foi utilizado um *Max Size* de 2048 para 256 pixels, o que resultou em uma redução (Visível no próprio *Texture Importer*) de 5.3 MB, para 85.4KB, com formato de compressão *RGBA Compressed ETC2 8 Bits* devido ao fato de possuir um canal Alfa, e mesmo sendo um formato de média qualidade não gerou perdas gráficas perceptíveis no decorrer do desenvolvimento da demonstração, por fim a qualidade do compressor foi utilizada em *Best* para que fosse obtido um resultado sem perda de qualidade, essa configuração não resultou em aumentos de memória segundo o próprio *Texture Importer*..

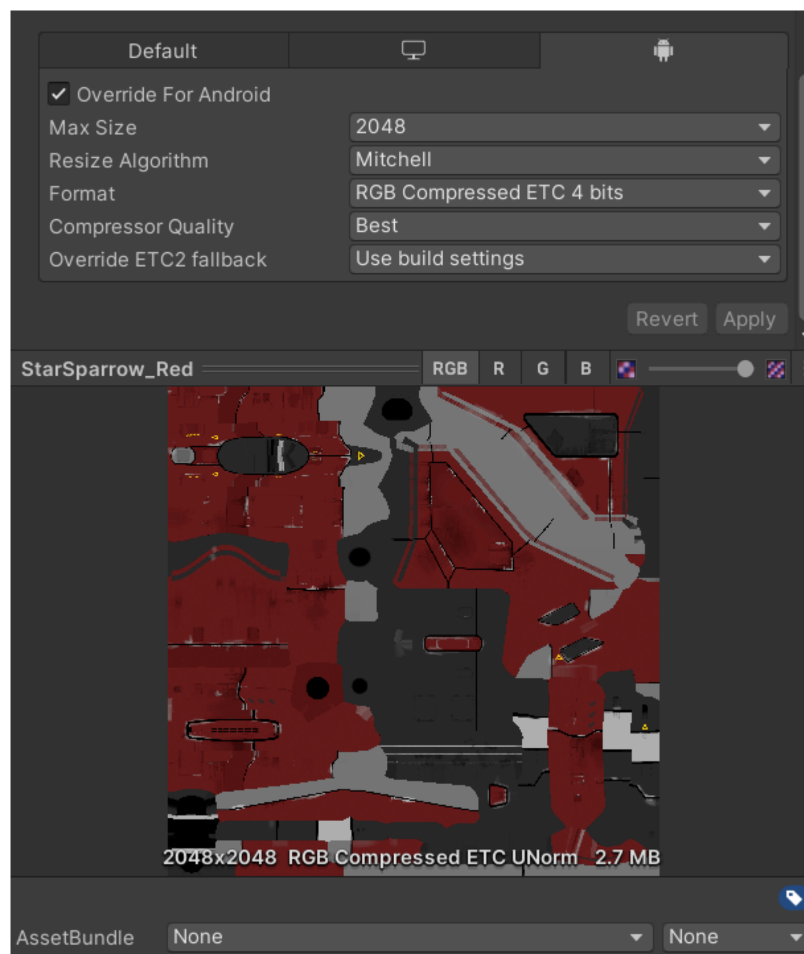
Figura 14 - Configuração da janela Platform Overrides nas Texturas de Nuvem dos planetas.



FONTE: (autor, 2021)

A aplicação do *Texture Importer* nas texturas *Albedo* e *Emissive* das Naves se deu através da seguinte configuração: foi utilizado um *Max Size* de 2048 pixels que era o tamanho padrão, com formato de compressão *RGB Compressed ETC 4 Bits* devido ao fato de ser um formato leve, e que mesmo sendo um formato de baixa qualidade não gerou perdas gráficas perceptíveis, por fim a qualidade do compressor foi utilizada em *Best* para que fosse obtido um resultado sem perda de qualidade, essa configuração não resultou em aumentos de memória segundo o próprio *Texture Importer*. Nesse caso não houve aumento nem diminuição do uso de memória da textura, por outro lado também não houve perda de qualidade.

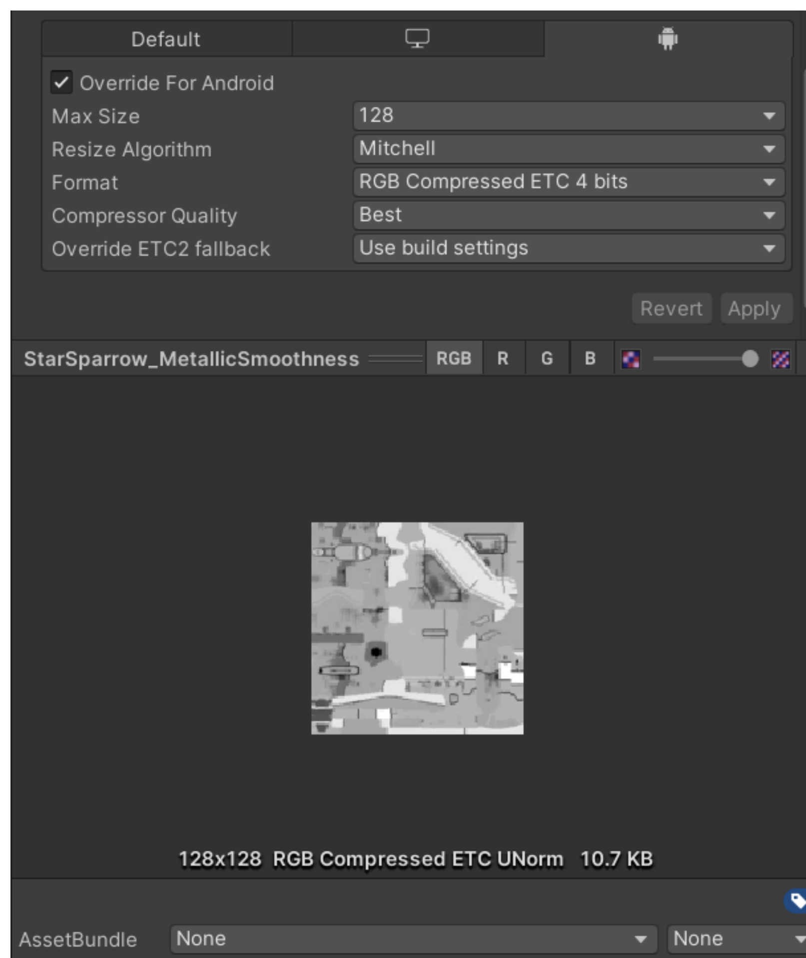
Figura 15 - Configuração da janela Platform Overrides nas Texturas de Albedo e Emissive das Naves.



FONTE: (autor, 2021)

A aplicação do *Texture Importer* no *Metallic* das naves se deu através da seguinte configuração: foi utilizado um *Max Size* de 2048 para 128 pixels, o que resultou uma redução (Visível no próprio *Texture Importer*) de 5.3 MB, para 21.4KB, com formato de compressão *RGB Compressed ETC 4 Bits* devido ao fato de ser um formato leve, e que mesmo sendo um formato de baixa qualidade não gerou perdas gráficas perceptíveis, após a troca do compressor o tamanho da textura caiu para 10.7KB, por fim a qualidade do compressor foi utilizada em *Best* para que fosse obtido um resultado sem perda de qualidade, essa configuração não resultou em aumentos de memória segundo o próprio *Texture Importer*.

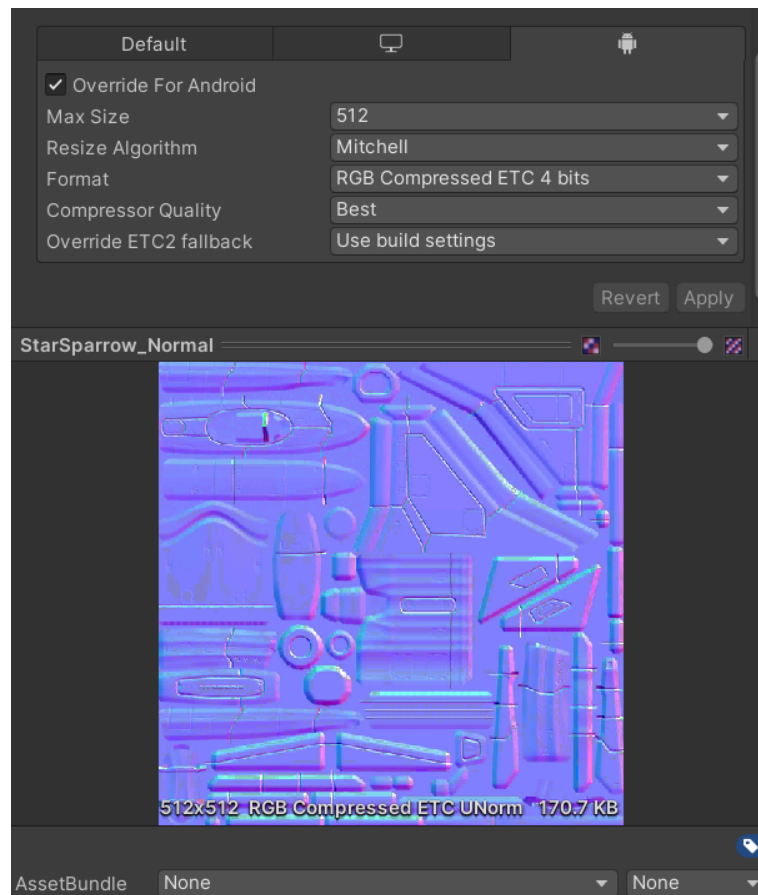
Figura 16 - Configuração da janela Platform Overrides nas Texturas de Metallic das Naves.



FONTE: (autor, 2021)

A aplicação do *Texture Importer* no *Normal* das naves se deu através da seguinte configuração: foi utilizado *Max Size* de 2048 para 512 pixels, o que resultou uma redução (Visível no próprio *Texture Importer*) de 2.7 MB, para 170.7KB, com formato de compressão *RGB Compressed ETC 4 Bits* devido ao fato de ser um formato leve, e que mesmo sendo um formato de baixa qualidade não gerou perdas gráficas perceptíveis por fim a qualidade do compressor foi utilizada em *Best* para que fosse obtido um resultado sem perda de qualidade, essa configuração não resultou em aumentos de memória segundo o próprio *Texture Importer*.

Figura 17 - Configuração da janela Platform Overrides nas Texturas de Normal das Naves.



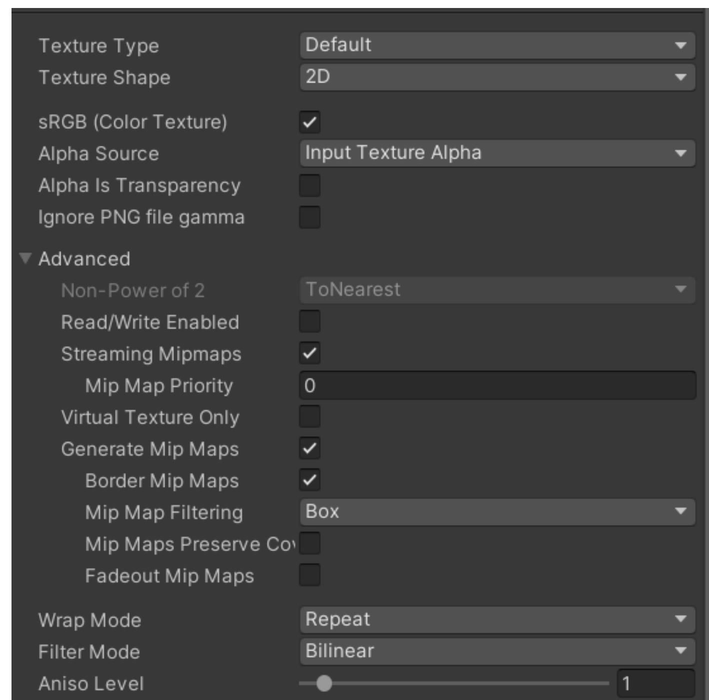
FONTE: (autor, 2021)

3.1 - Texture Importer - MipMap Streaming

Na demonstração, os *MipMaps* foram aplicados a todas as texturas, devido ao fato do *hardware* ser limitado.

O processo de aplicação do MipMap Streaming se deu através da seguinte configuração:

Figura 18 - Configuração do *Texture Importer* para aplicação dos *MipMaps*.

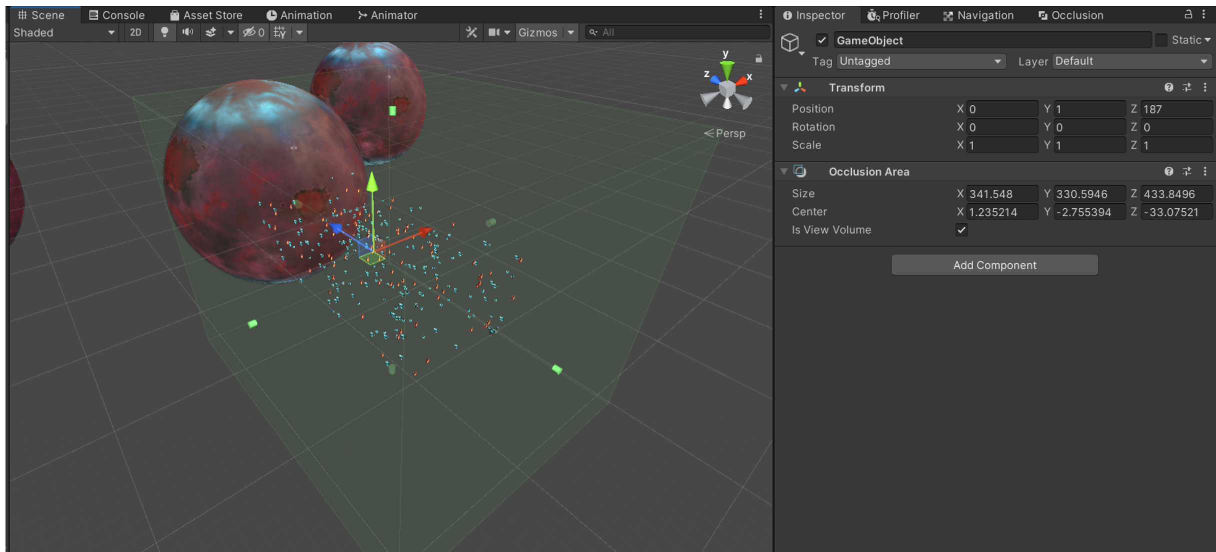


FONTE: (autor, 2021)

3.2 - Aplicando Occlusion Culling

A aplicação do sistema de oclusão se deu através da criação de uma *Occlusion Area* na área de jogo, cuja qual teria os objetos em movimento (asteróides, minas, etc.), após a criação da Occlusion Área, foi gerado um *Bake* através da janela Olucclusion, para que a Occlusion Culling fosse computada.

Figura 19 - Configurando a *Occlusion Area* do tamanho total do cenário de jogo.



FONTE: (autor, 2021)

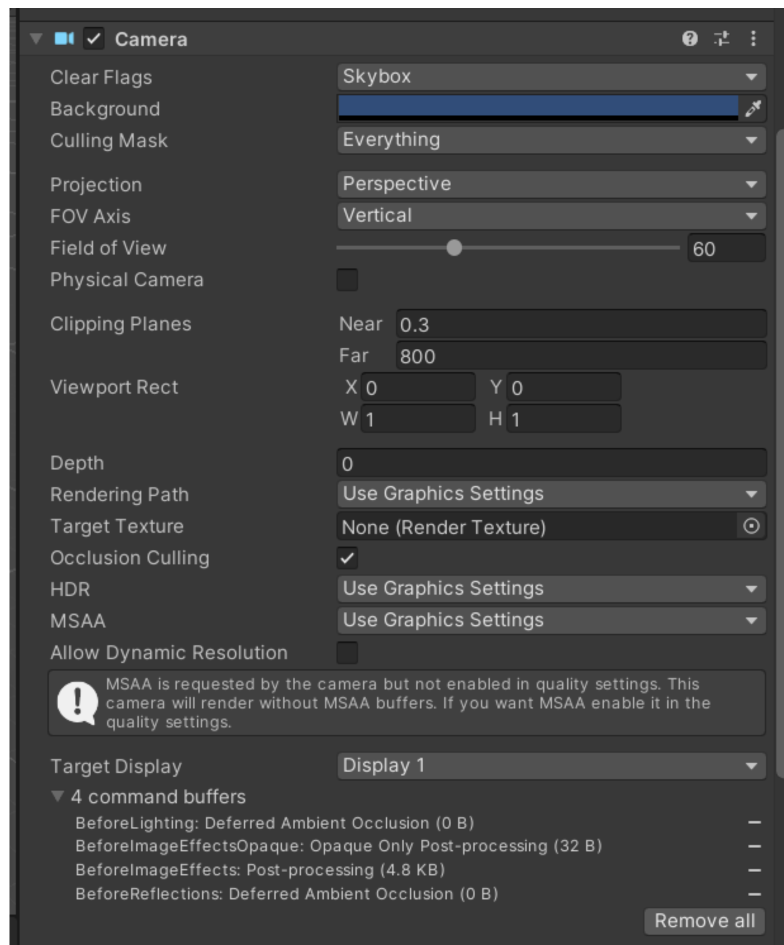
Esse processo aplicado no jogo garante que apenas os objetos na área da câmera sejam renderizados, com isso tudo que não pertence ao campo de visão da câmera é descartado, vale ressaltar que ocluir é diferente de reduzir o *Far* da câmera, uma vez que o *Far* reduz o campo de visão já o Occlusion, não reduz o campo de visão.[10]

[10] O *Occlusion Culling* "Recorta" a área do *Far*, ou seja, se o *Far* for grande demais, o *Occlusion Culling* vai garantir que apenas os objetos dentro do cone de visão sejam desenhados, descartando tudo que for encoberto por outros objetos, enquanto que o *Far* vai fazer com que todos os objetos dentro de uma determinada área sejam desenhados independente de estarem sendo oclusos por outros ou não.

3.3 - Câmeras

A otimização da câmera ocorreu através da redução do campo de renderização (*Far*) uma vez que por padrão a Unity atribui um valor maior do que o necessário no caso da demonstração, o que faz com que uma área maior do que o necessário seja renderizada, ainda que o usuário nunca tenha acesso a essa área.

Figura 20 - Configurando a *Main Camera*.



FONTE: (autor, 2021)

3.4 - Observações ao longo do experimento

Ao longo do estudo foi notado a princípio uma série de limitações relacionadas ao dispositivo utilizado, dentre essas limitações a ausência de espaço para a dissipação de calor, isso fazia com que as temperaturas do aparelho inicialmente fossem um problema, pois as altas temperaturas apareciam logo no início dos testes, com o decorrer do tempo, a medida em que as técnicas eram aplicadas, em conjunto com análises de performance constantes, a temperatura inicial do aparelho começou a diminuir, até que em determinado momento chegou em um patamar aceitável. Vale ressaltar que os objetivos deste estudo estavam voltados para a aplicação das técnicas de otimização, e como aplicá-las de forma que fosse possível construir uma aplicação que fizesse o uso de operações de física e criação, e deleção de objetos, em conjunto com modelos, texturas e efeitos de alta qualidade, em um dispositivo móvel.

3.5 - Especificações do dispositivo

Para a execução dos testes foi utilizado um smartphone Samsung Modelo M21s com as seguintes especificações:

- Memória Interna 64GB.
- Memória RAM 4GB.
- Processador Octa-Core (1.7GHz, 2.3GHz).
- Sistema Operacional Android 11.

3.6 - Execução dos testes

A execução dos testes se deu através da execução da aplicação no dispositivo alvo (Galaxy M21), e também dentro do Unity Editor, os testes foram conduzidos com diferentes objetivos sendo eles:

Dentro do Unity Editor

- Balanceamento e Calibragem.
- Correção de Bugs.
- Coleta dos valores dos seguintes campos:
    - Tris.
    - Vertex.
    - FPS.
    - Quantidade de objetos em cena.
- Análise e Ajustes de performance.

Figura 21 - Coletando dados da janela Stats do Unity Editor.

FONTE: (autor, 2021)

Figura 22 - Coletando dados do jogo no Unity Editor.

FONTE: (autor, 2021)

Dispositivo Alvo

- Temperatura do Aparelho (Antes e depois da execução do jogo).
- FPS.
- Quantidade de objetos em cena.

Figura 23 - Fazendo a coleta de dados do jogo na aplicação android.

FONTE: (autor, 2021)

3.7 - Considerações finais

Após a execução dos testes, os dados coletados apontam que nas condições apresentadas previamente, o dispositivo foi capaz de executar a aplicação sem quedas de FPS, e sem aumentos anormais de temperatura, porém após alguns minutos com o jogo em seu ápice, a temperatura do dispositivo apresentou um leve aumento, com esse resultado foi possível tirar duas conclusões:

A primeira conclusão é de que, as técnicas aqui descritas, são eficazes em seu papel, devido ao fato de que à medida em que elas eram aplicadas, o tempo de execução do jogo aumentava sem que fosse percebido aumentos de temperatura.

A segunda conclusão, é que mesmo com uma quantidade pequena de técnicas aplicadas, foi possível obter um bom resultado, tendo em consideração os objetivos pelos quais a demonstração foi construída.

O resultado apresentado abre a possibilidade para que sejam feitos novos estudos acerca do tema, através da apresentação de novas técnicas ou da combinação de outras técnicas.

Por fim é importante ressaltar que os conceitos utilizados neste artigo podem ser aplicados em várias plataformas. Vale ressaltar que em alguns casos a Unity Engine pode exigir uma série de configurações específicas para cada plataforma.

3.8 - Técnicas de Otimização não mencionadas durante a construção do artigo.

Com relação às Referências em Cache, é comum muitas vezes termos a necessidade de pesquisar por referências em tempo de execução através dos métodos *Find*, *Camera.Main*, e afins, porém é importante ressaltar que segundo um artigo publicado no site da Microsoft, essas funções são caras, e quando executadas em excesso podem acabar gerando gargalos de performance, para evitar esse problema, é uma boa prática salvar os valores necessários em cache, através de métodos *Start* ou *Awake*, em funções como *Updates*, *FixedUpdate*, ou *LateUpdate* esses valores devem ser apenas trabalhados.

A Criação e Destruição de Instâncias são demandas bastante comuns no desenvolvimento de jogos, para atender à essa demanda a Unity disponibiliza os métodos *Instantiate* e *Destroy*, que fazem instanciação e destruição, porém embora eles sejam capazes de atender perfeitamente à essas demandas, sua utilização de maneira incorreta acaba gerando uma alta carga de uso na CPU, o que pode ocasionar em travamentos nas aplicações, pensando em resolver esse tipo de problema e em atender às demandas, uma das recomendações é o uso do *Pattern Object Pooling*, que é recomendado em artigos escritos nos websites de empresas como Microsoft, Unity e Intel.

As chamadas *Debug.Log*, *Debug.LogError*, *Debug,LogWarning*, são muito úteis quando precisamos debugar nossos jogos, porém é importante ressaltar a necessidade de removê-los do código quando eles não forem mais úteis, uma vez que segundo um artigo publicado no Unity Learn, eles também são executados no aplicativo compilado (fora do editor).

É importante ressaltar que devemos tomar cuidado com o uso dos componentes *Mesh Colliders*, uma vez que, eles geram alta sobrecarga no processamento, a recomendação dada pela Unity na própria documentação é sempre tentar se aproximar do objeto usando colisores primitivos.

Um artigo publicado no site da Microsoft diz que um *LINQ* embora seja mais limpo em sua escrita, acaba exigindo mais computação e memória do que a escrita manual do código, um outro artigo publicado no site da Unity Learn aponta que,

LINQ e Regular Expressions acaba gerando lixo que deve ser coletado pelo garbage collector, devido ao fato de ocorrência de boxing nos bastidores da execução.

Segundo um artigo publicado pela Unity em sua documentação oficial, algumas funções matemáticas como *Mathf.Sin*, *Mathf.Pow*, *Normalize*, etc, levam em média 100 vezes mais o tempo de uma multiplicação.

É importante tomar cuidado durante a configuração do *Post-Processing Layer*, o uso das layers *Default* e *Everything* deve ser evitado durante a configuração dos efeitos, pois, segundo a própria Unity Engine (Software), o processo de blending será mais lento nesses casos, a própria engine aponta uma solução para o problema que é através da utilização de uma layer a parte para a configuração dos efeitos.

REFERÊNCIAS BIBLIOGRÁFICAS

Autodesk 29 de Novembro de 2021. https://help.autodesk.com/cloudhelp/2016/ENU/mental-ray-docs/mr_docs/manual/node18.html

Bertrand Guay-Paque 23 de Setembro de 2021. https://blog.unity.com/games/optimize-game-performance-with-camera-usage

Cristiano Ferriera e Steve Hughes 2015.

https://www.intel.com/content/dam/develop/external/us/en/documents/unity-optimizations-for-x86-android-v2-92.pdf

Iutti 26 de Setembro de 2018. https://www.opus-software.com.br/design-patterns/

Kent Sharkey; Vinayak, David Coulter, Harrison Ferrone, Vinnie Tieto 18 de Outubro de 2021.

https://docs.microsoft.com/en-us/windows/mixed-reality/develop/unity/performance-recommendations-for-unity#cpu-performance-recommendations

Mohamed Hijazi 11 de Agosto. https://medium.com/nerd-for-tech/tips-for-code-optimization-in-unity-947b1fd9b6a9

Praveen K Kundurthy 05 de Dezembro de 2016. https://www.intel.com/content/www/us/en/developer/articles/technical/unity-software-performance-optimizations-for-games-best-practices.html

Unity Technologies 2016. https://docs.unity3d.com/540/Documentation/Manual/class-TextureImporter.html

Unity Technologies 20 de Novembro de 2021. https://docs.unity3d.com/Manual/class-TextureImporter.html

Unity Technologies 20 de Novembro de 2021. https://docs.unity3d.com/Manual/ShadowPerformance.html

Google Developers 20 de Abril de 2021. https://developer.android.com/games/optimize/lighting-for-mobile-games-with-unity

Unity Technologies 20 de Novembro de 2021. https://docs.unity3d.com/Manual/shadow-mapping.html

Unity Technologies 20 de Novembro de 2021. https://docs.unity3d.com/Manual/texture-compression-formats.html

Unity Technologies 2017. https://docs.unity3d.com/560/Documentation/Manual/class-TextureImporterOverride.html

Unity Technologies 20 de Novembro de 2021. https://docs.unity3d.com/Manual/class-TextureImporterOverride.html

Unity Technologies 11 de Novembro de 2021. https://docs.unity3d.com/Manual/iphone-Optimizing-Physics.html

Unity Technologies 08 de Janeiro de 2021. https://learn.unity.com/tutorial/introduction-to-optimization-in-unity#5ff8ce16edbc2a0023134676

Unity Technologies 20 de Novembro de 2021.

https://docs.unity3d.com/ScriptReference/TextureCompressionQuality.html

Unity Technologies 26 de Novembro de 2021.

https://docs.unity3d.com/Manual/class-TextureImporter.html

Unity Technologies 2015.

https://docs.unity3d.com/510/Documentation/Manual/ShadowOverview.html

Unity Technologies 12 de Julho de 2017. https://docs.unity3d.com/560/Documentation/Manual/class-OcclusionArea.html

Unity Technologies 23 de Dezembro de 2020.

https://learn.unity.com/tutorial/working-with-occlusion-culling#5fe2b352edbc2a10f945f217

Unity Technologies 2016.

https://docs.unity3d.com/540/Documentation/Manual/class-OcclusionArea.html

Unity Technologies 02 de Abril de 2020.

https://learn.unity.com/tutorial/introduction-to-lighting-and-rendering#5c7f8528edbc2a002053b52a

Unity Technologies 26 de Novembro de 2021.

https://docs.unity3d.com/Manual/OptimizingGraphicsPerformance.html

Unity Technologies 25 de Maio de 2020.

https://learn.unity.com/tutorial/fixing-performance-problems#5c7f8528edbc2a002053b595

Nick Mower 27 de Novembro de 2021. https://www.techarthub.com/an-introduction-to-texture-compression-in-unity/

Agradecimentos

A Universidade Anhembi Morumbi, fundamental no processo da minha construção profissional, pelo suporte oferecido, e por tudo que aprendi durante o curso.

A professora e orientadora deste artigo Valéria Guerra, pelos conselhos e ensinamentos fundamentais para o desenvolvimento deste trabalho.

As empresas Ebal Studios, e Pulsar Bytes, pelos Modelos 3D e Texturas fornecidos, que foram essenciais para a construção da demonstração presente neste estudo.