Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Desenvolvimento de Jogos Digitais

OTIMIZAÇÃO DE JOGOS PARA DISPOSITIVOS MÓVEIS COM UNITY3D

ADRIANO PEREIRA REZENDE

Americana, SP
2013

Faculdade de Tecnologia de Americana
Curso Superior de Tecnologia em Desenvolvimento de Jogos Digitais

OTIMIZAÇÃO DE JOGOS PARA DISPOSITIVOS MÓVEIS COM UNITY3D

ADRIANO PEREIRA REZENDE

adrianoprezende@gmail.com

Trabalho de Conclusão de Curso desenvolvido em cumprimento à exigência curricular do Curso Superior de Tecnologia em Desenvolvimento de Jogos Digitais, sob a orientação do Prof. Me. Kleber de Oliveira Andrade.

Área: Jogos Digitais

Americana, SP
2013

BANCA EXAMINADORA

Prof. Me. Kleber de Oliveira Andrade (Orientador)

Prof. Me. Cleberson Eugenio Forte

Prof. Gustavo Carvalho Gomes de Abreu

AGRADECIMENTOS

Agradeço primeiramente a Deus, meu Senhor e salvador Jesus Cristo, por ter permitido eu chegar até aqui. A Ele, pois, toda a glória!

Aos meus pais por todo apoio que me deram durante o período da faculdade, mas principalmente por terem acreditado nos meus sonhos e por terem me ensinado a lutar para alcançá-los.

Aos professores e grandes mestres Cleberson Forte e Kleber Andrade, por todo engajamento, fascínio e dedicação demonstrados pelo ensino na área de jogos.

A todos os meus amigos, colegas de classe e professores, os meus sinceros agradecimentos, pois sem dúvida todos foram essenciais nessa jornada.

DEDICATÓRIA

Dedico este trabalho á todos que têm lutado para fazer do Brasil uma potência no mercado de desenvolvimento de jogos.

RESUMO

O presente texto visa explorar as principais técnicas de otimização de jogos para dispositivos móveis desenvolvidos com o motor de jogo Unity3D. Com uma abordagem ampla, o estudo contido nesta pesquisa faz uma análise sobre o uso do motor de jogo Unity3D no desenvolvimento de jogos para estes dispositivos, elencando também os parâmetros e fatores, assim como os possíveis gargalos encontrados no desenvolvimento de jogos para este tipo de plataforma. Por fim, aplica-se as técnicas de otimização demonstrando de forma prática as métricas e os resultados obtidos.

Palavras Chave: Unity3D, Jogos, Otimização.

ABSTRACT

This paper aims to explore the major optimization techniques for mobile games developed with Unity3D game engine. With a broad approach, the study contained in this research performs an analysis about the use of Unity3D game engine in game development for these devices, enumerating also the parameters and factors, as well as possible bottlenecks found at game development for this platform type. Finally, applies the optimization techniques demonstrating in a practice way, the metrics and the results obtained.

Keywords: Unity3D, Games, Optimization.

SUMÁRIO

LISTA DE FIGURAS

1 INTRODUÇÃO

Os smartphones e tablets já fazem parte da vida da população mundial. O número de vendas desses dispositivos tem crescido cada vez mais e inclusive superado as de PCs, segundo pesquisa do instituto IDC (apud LINK, 2013). No Brasil, leis de redução de impostos diminuem os preços dos smartphones em até 30%, contribuindo para o acesso da população á esses aparelhos (TELES, 2013). Com toda essa alta demanda o mercado de aplicativos para os dispositivos móveis tem sido já há certo tempo, um setor muito interessante para os desenvolvedores, em especial os de jogos.

Desde o advento da App Store, lançada em 2008 e até os dias atuais, os jogos tem sido os aplicativos mais baixados para smartphones e tablets. Uma pesquisa feita pela consultoria americana Canalys mostrou que 50% de toda a receita obtida com a venda de aplicativos no mercado americano está concentrada em 25 empresas, sendo que destas, apenas uma não desenvolve games (DALMAZO; FERRARI, 2013).

E conforme o mercado de aplicativos para esses dispositivos foi crescendo, ampliaram-se também as possíveis formas de desenvolvimento para eles. As ferramentas de desenvolvimento que antes se restringiam apenas as nativas dos sistemas operacionais dos aparelhos, fornecidas pelos fabricantes, hoje passaram a ser uma opção do desenvolvedor. Ambientes de desenvolvimento multiplataforma como Corona, PhoneGap, Unreal e Unity3D passaram a incluir os principais sistemas operacionais para dispositivos móveis, Android e IOS, na lista de sistemas suportados, possibilitando que, um único jogo desenvolvido, seja exportado para mais de um sistema. Porém, smartphones e tablets são conhecidos por terem capacidades de hardware inferiores aos de um PC, e os jogos visados para essas plataformas, mesmo sendo desenvolvidos em ferramentas robustas e versáteis quanto às citadas acima, necessitam de otimizações para poderem funcionar. Sem as otimizações necessárias, um jogo exportado para dispositivos móveis não atingirá a mesma performance obtida ao ser executado em um computador pessoal, podendo em muitos casos, sequer funcionar.

Tendo em vista essa necessidade, a presente pesquisa foi desenvolvida com o objetivo de apresentar e demonstrar o uso de técnicas fundamentais para a otimização de jogos para dispositivos móveis desenvolvidos com a Unity3D. Para alcançar este objetivo, o trabalho foi dividido em quatro partes principais que se iniciam a partir do segundo capítulo, onde a Unity é apresentada, assim como o seu uso no desenvolvimento de jogos para dispositivos móveis e as necessidades de otimização. O terceiro capítulo conceitua a criação de um artefato conhecido como orçamento do jogo, que funciona como uma planta baixa para todas as otimizações que serão realizadas. O quarto capítulo trata de apresentar e demonstrar as técnicas de otimização, divididas em otimizações da parte gráfica e otimizações da parte lógica de um jogo. O quinto e último capítulo se reserva a expor as conclusões e destacar os pontos positivos da pesquisa.

2 A UNITY3D

A Unity3D (UNITY, 2013), também conhecida como Unity, é uma plataforma de desenvolvimento para a criação de jogos e conteúdo interativo em 3D, desenvolvido pela Unity Technologies. Popularmente também é definida como uma *game engine*[1]. Sua primeira versão foi lançada em 2005, porém começou a se popularizar entre os desenvolvedores a partir de 2007. Nos dias atuais possui cerca de 1,3 milhões de desenvolvedores registrados incluindo grandes empresas como Cartoon Network, Coca-Cola, Disney, Electronic Arts, LEGO, Microsoft, NASA, Ubisoft, Warner Bros e até mesmo o Exército dos Estados Unidos. Apesar de ser focada para o desenvolvimento de jogos, a Unity3D é também muito utilizada na criação de aplicações de conteúdo 3D interativas, como simuladores, visualizações arquitetônicas, anatomia, dentre outros. Vencedora de diversas premiações, a Unity atualmente é uma ferramenta consolidada e uma das mais escolhidas pelos desenvolvedores *indie*[2] para a criação de jogos.

Os jogos desenvolvidos na Unity3D podem ser exportados para múltiplas plataformas, tais como Android, IOS, PC, Mac, Linux, Web, Wii, Playstation, Xbox e Flash, de acordo com a licença adquirida pelo desenvolvedor. Com as principais licenças disponíveis, Unity Free e Unity Pro, é possível exportar os jogos e/ou aplicativos desenvolvidos para Windows, Mac, Linux e Web. A licença Unity Free é gratuita e possui diversas limitações, sendo o seu uso voltado á estudantes e desenvolvedores interessados em conhecer a plataforma, além disso, essa licença é restrita á companhias e empresas que obtiverem faturamento maior do que cem mil dólares em seu último ano fiscal. Já a licença Unity Pro é a versão profissional, completa e que possui diversos recursos inexistentes na licença gratuita. Apesar de ser uma ferramenta para o desenvolvimento multiplataforma, a Unity3D possui versões apenas para Mac e Windows.

---

[1] *Game engine* ou *engine*, é um programa de computador e/ou conjunto de bibliotecas, usado para simplificar e abstrair o desenvolvimento de jogos eletrônicos ou outras aplicações com gráficos em tempo real. Também é conhecido pelo termo em Português, Motor de Jogo.

[2] Do inglês, *indie* é a abreviação (no diminutivo) de *independent* (em Português, independente) e é aplicado aos desenvolvedores de jogos, assim como músicos, produtores e artistas, que não possuem contratos de publicação e distribuição e utilizam dos próprios recursos para produzirem e lançarem os seus projetos, de maneira independente.

Composta por um conjunto de ferramentas integradas, a Unity, como um motor de jogos, é constituída por vários “sub-motores” de funcionalidades distintas e que são responsáveis por tratar aspectos diferentes do jogo. O motor gráfico da Unity utiliza as APIs Direct3D, OpenGL, OpenGL ES e APIs proprietárias para a renderização, conforme a plataforma em que deseja-se exportar. Inclui suporte á técnicas de computação gráficas como *bump mapping*, *reflection mapping*, *parallax mapping*, *screen space ambient occlusion (SSAO)*, sombras dinâmicas usando mapas de sombras e efeitos de pós-processamento em tela cheia. Possui também a capacidade para importar objetos de jogo, como modelos 3D e imagens, diretamente de alguns dos principais softwares para arte, tais como: 3ds Max, Maya, Softimage, Blender, Modo, ZBrush, Cinema 4D, Cheetah3D, Adobe Photoshop e Adobe Fireworks. Uma biblioteca com diversos *shaders*, por padrão, também estão integrados á ferramenta. Para aqueles que desejarem, existe a possibilidade de programar seus próprios *shaders* utilizando as linguagens ShaderLab, CG ou HLSL. O conceito de *shader*, além de sua aplicação nos jogos digitais, é um assunto abordado em um dos capítulos posteriores deste trabalho.

Para a simulação da física, a Unity utiliza o motor de física PhysX da NVIDIA[3], possibilitando de modo simplificado o uso de gravidade nos objetos do jogo, assim como detecção de colisões, uso de cinemática, força, torque, fricção, elasticidade, simulação de roupa em tempo real e suporte a malhas de pele, dentre uma série de outras propriedades que contribuem para uma maior sensação de realismo dentro do jogo.

Seu sistema de audio foi construído com a biblioteca FMOD[4], sendo capaz de importar arquivos de audio nos seguintes formatos: aif, wav, mp3, ogg, xm, mod, it, and, s3m. Também possui um modo de som 3D, no qual o efeito de som aumenta com a proximidade e diminui com o afastamento do objeto, em relação á câmera. A licensa Pro ainda possui suporte á execução de videos.

Pode-se destacar ainda um motor próprio para a construção de terrenos e vegetações, o que elimina a necessidade do uso de ferramentas de modelagem externas para a criação de terrenos. O motor também possibilita a criação de

[3] http://www.geforce.com/hardware/technology/physx
[4] http://www.fmod.org/

árvores, incluindo detalhes de troncos e folhagens, assim como o uso de árvores prontas. Com o motor de terrenos e vegetações, é possível utilizar algumas técnicas bem interessantes como o *billboarding* de árvores, que é a transformação de um objeto 3D em uma imagem 2D (*billboard*) em tempo de execução, conforme a câmera se distância do objeto. Esta técnica é utilizada em jogos para economizar processamento (LUZ, 2004, p. 60 - 63).

Quanto às demais ferramentas integradas, vale ainda citar o sistema de partículas, que possibilita a criação de efeitos como: explosões, fumaça, cachoeira, água, fogo, laser, fogos de artifício, tempestades de areia, chuva, bolhas, neve e etc, a possibilidade do desenvolvimento de jogos com *multiplayer* em rede e a construção de mapas de luz e iluminação global, do qual a Unity faz uso do conjunto de ferramentas da Autodesk, Beast Lightmapping[5].

A programação na Unity é feita através de scripts que podem ser desenvolvidos utilizando até três linguagens diferentes: Javascript, C#, e um dialeto de Python chamado Boo. Por padrão, a partir da versão 3.0, a Unity traz a IDE MonoDevelop[6], integrada, para a criação e depuração dos scripts para o jogo.

O Suporte fornecido pela Unity Technologies inclui um fórum para a troca de informações e conhecimento entre os desenvolvedores participantes, que frequentemente contribuem para o enriquecimento de conteúdo para o uso da ferramenta, através de tutoriais, discussões, scripts, dentre outros recursos. É disponibilizada também, uma documentação extensa que aborda o uso de cada componente na Unity, com guias, separados em: Manual do Usuário, Referências de Componentes e Referências de Scripts. Além disso, existem diversos tutoriais oficiais, que ensinam passo a passo como desenvolver jogos de diversos gêneros como, corrida, plataforma, FPS, etc, com a Unity. É necessário frisar que, toda a documentação oficial está em inglês, porém a cada dia tem-se aumentado o número de tutoriais e videos na internet em português, sobre o desenvolvimento de jogos com a Unity, feitos por estudantes e desenvolvedores independentes, mas que possuem qualidade inferior aos oficiais. Há também um suporte *Premium* que é um serviço pago, onde os engenheiros da Unity estão sempre dispostos a ajudar.

[5] http://gameware.autodesk.com/beast
[6] http://monodevelop.com/

Existe ainda um recurso disponível dentro da Unity chamado *Unity Asset Store*, que nada mais é do que uma loja onde se concentram milhares de pacotes com *assets*[7], incluindo modelos 3D, texturas e materiais, sistemas de partículas, músicas e efeitos sonoros, tutoriais e projetos completos, pacotes de scripts, plugins, extensões e serviços online. Alguns são gratuitos, porém a maioria não.

2.1 A UNITY NO DESENVOLVIMENTO PARA DISPOSITIVOS MÓVEIS

A Unity passou a ter suporte ao desenvolvimento de jogos para IOS em 2008 e para Android apenas em 2010. Atualmente, ambas as plataformas contam com um grande número de títulos publicados, que foram desenvolvidos com a Unity. Segundo informações estatísticas contidas em Unity (2013a), a loja de aplicativos da Apple, App Store, possui mais de mil e quinhentos jogos desenvolvidos com a tecnologia Unity. Em 2012, dentre a lista dos melhores jogos do ano[8] publicados na App Store, divulgada pela Apple, dezessete deles foram desenvolvidos com a Unity (BRATCHER, 2013).

Segundo pesquisa publicada pela GDMag (apud GAMASUTRA, 2012, tradução nossa):

> A Unity é a *game engine* mais utilizada no desenvolvimento de jogos para dispositivos móveis, com 53,1% de uso entre desenvolvedores. A estatística segue com 39,8% de desenvolvedores que utilizam ferramentas customizadas, 17,7% usam a Cocos2D, 5,3% usam Marmalade e 5,3% utilizam Corona (Esta questão não foi exclusiva, por isso a contagem percentual chega a mais de 100%).
>
> Sobre quais os aspectos são mais importantes em uma engine para os desenvolvedores, a pesquisa relatou que “Rápido tempo de desenvolvimento” foi o mais votado, seguido por “Flexibilidade e fácil capacidade de extensão”, “Performance”, “Suporte e documentação” e “Uso anterior bem sucedido por outra equipe”.

Dos jogos que alcançaram grande sucesso e podem demonstrar as possibilidades e capacidades da Unity3D no desenvolvimento de jogos para

[7] Na Unity, *asset* é como são chamados os arquivos que serão usados para a criação do jogo. Por exemplo: imagens, modelos 3D, sons, scripts etc..
[8] A lista dos melhores jogos publicados na App Store de 2012, divulgada pela Apple, foi composta por um total de cento e sessenta e um jogos.

dispositivos móveis, estão alguns como: Bad Piggies (Figura 1a), desenvolvido pela empresa Rovio Mobile Ltd., Cowboy Vs Aliens Vs Ninja (Figura 1b), desenvolvido pela empresa brasileira Dead Mushroom, Ravensword: Shadowlands (Figura 1c), desenvolvido pela empresa Crescent Moon Games, ShadowGun (Figura 1d) e Dead Trigger 2, desenvolvidos pela empresa Madfinger Games, God of Blades desenvolvido pela empresa Whale Games e Ski Safari desenvolvido pela empresa Defiant Development.

Figura 1 – Exemplos de jogos para dispositivos móveis desenvolvidos com a Unity. A (Bad Piggies, Rovio Mobile Ltd.); B (Cowboy Vs Ninjas Vs Aliens, Dead Mushroom); C (Ravensword: Shadowlands, Crescent Moon Games); D (ShadowGun, Madfinger Games).

2.2 VANTAGENS

Em geral, a Unity demonstra ser uma ferramenta muito flexível, dinâmica e de grau de aprendizado relativamente baixo, o que contribui para um rápido tempo de desenvolvimento. São inúmeras as vantagens em seu uso na criação de jogos. Uma delas, que é um dos grandes destaques da ferramenta, é a possibilidade de compilação para diversas plataformas. Essa característica permite que, um único

jogo criado, seja exportado para cerca de dez plataformas diferentes, eliminando a necessidade de conhecimentos específicos para a criação de jogos, em cada uma dessas plataformas.

Uma das outras vantagens é a possibilidade do uso de até três linguagens de programação diferentes: C#, Javascript e Boo. Diferentemente de outras *engines*, qual se faz necessário o aprendizado de uma linguagem de programação própria da ferramenta, como é o caso do Game Maker e Unreal Engine, por exemplo, na Unity, o densenvolvedor pode optar em uma das três linguagens para escrever a lógica de seu jogo. É importante destacar também que C# e Javascript são linguagens muito difundidas e populares entre programadores no mundo todo, o que facilita ainda mais seu uso.

A Unity é extensível e permite a adição e/ou criação de *plugins*, ampliando a quantidade de ferramentas e possibilidades disponíveis. Fator este que também foi levado em consideração na escolha da Unity pelos desenvolvedores do jogo Bad Piggies da Rovio, segundo Haapasalo (2012).

A Unity Asset Store é também uma grande vantagem por permitir que o desenvolvedor possa comprar recursos prontos e vender os que criarem. Prática que inclusive, é utilizada por empresas responsáveis por grandes títulos como a Crescent Moon Games, segundo Presseisen (2013), ao ter utilizado por diversas vezes a Unity Asset Store para a compra de *shaders* para o jogo Ravensword: Shadowlands.

A Unity, apesar de ser tratada como uma ferramenta específica para o desenvolvimento de jogos em 3D, pode ser usada também para a criação de jogos em 2D, versatilidade que lhe confere grande vantagem se comparada com outras *game engines* que são, ou específicas para a criação de jogos 2D, ou específicas para a criação de jogos 3D.

Merece destaque também as frequentes atualizações realizadas pela Unity Technologies, que incluem diversas melhorias, correções, inúmeras funcionalidades e recursos que são acrescentadas à Unity á cada nova versão que é lançada, garantindo que a ferramenta esteja sempre otimizada e com o menor número possível de defeitos.

Outros fatores que conferem vantagem á Unity são: A compatibilidade com os principais formatos de arquivos referentes a recursos visuais e sonoros; Documentação extensa, com tutoriais, cursos online e vídeos para o aprendizado, assim como um grande fórum para discussões, dúvidas e assuntos relacionados à ferramenta.

2.3 DESVANTAGENS

A Unity também possui desvantagens. Para a execução dos jogos compilados para web, é necessária a instalação de um *plug-in* proprietário, para navegadores, conhecido como “Unity Web Player”. Assim como o Adobe Flash, a instalação do *plug-in* se faz de maneira simples, porém, muitos usuários por não conhecerem a ferramenta, podem ter receio de instalar um programa desconhecido em seus computadores e desistirem.

No desenvolvimento de jogos onde é utilizado ferramentas nativas á plataforma, como por exemplo, o Android e seu kit de desenvolvimento Android SDK, é possível a otimização de toda a estrutura de seu projeto de jogo, assim como a imediata correção de possíveis erros de arquitetura, correções de *bugs* e também a redução do tamanho do arquivo final. Com a Unity, assim como em qualquer outra *game engine*, essas são tarefas impossíveis de serem realizadas, pois os desenvolvedores não possuem acesso ao código-fonte da *engine*, fazendo com que as otimizações e correções de possíveis defeitos, fiquem á encargo da criadora da ferramenta, no caso, a Unity Technologies.

2.4 A REAL NECESSIDADE DE OTIMIZAÇÃO

Heineman (2008, tradução nossa) afirma que:

> “Um dos problemas mais comuns encontrados na criação de jogos de computador é o desempenho. Questões como o acesso ao disco, o desempenho da GPU, o desempenho da CPU, as condições de corrida, e largura de banda de memória (ou a falta dela) pode causar lentidão ou atrasos que podem transformar um jogo de trinta quadros por segundo em um jogo de nove quadros por segundo”.

A partir dessa afirmação, pode-se chegar a duas conclusões: O desempenho de um jogo em execução está relacionado ao hardware que o executa, e um jogo não tão bem projetado pode sofrer consequências como perda de desempenho.

A Unity possibilita o uso de recursos de alta qualidade, além de técnicas de computação gráfica avançadas na criação de jogos, porém, ao seu uso, deve-se atentar em quanto esses recursos e técnicas irão consumir a CPU, GPU e Memória, para que o jogo não venha a sofrer com os problemas mencionados acima.

Nos dispositivos móveis o problema de desempenho se agrava um pouco mais, devido ao fato de existirem diversos modelos de celulares e tablets com configurações de hardware completamente diferentes uns dos outros, como afirma Rabas (2011).

A partir dessas argumentações, torna-se nítido a necessidade de conhecimento dos possíveis *hardwares* para quais se deseja desenvolver o jogo, a necessidade da criação e uso de conteúdo otimizado, assim como o uso adequado das técnicas e ferramentas disponibilizadas pela Unity, para que o jogo alcance uma performance satisfatória. O presente texto não abordará a análise de *hardware* para que as técnicas aqui exploradas possam ser utilizadas independentemente do *hardware* visado. Os demais assuntos serão tratados com maior profundidade nos capítulos posteriores.

3 DETERMINANDO O ORÇAMENTO DO JOGO

Antes de iniciar a criação do conteúdo para o desenvolvimento de um jogo, é necessário realizar uma análise do hardware visado e do tipo de jogo qual se deseja fazer, para definir os limites desse conteúdo e também quais os limites de recursos de hardware, que esse conteúdo poderá consumir, para que o jogo possa ter um bom desempenho. As conclusões provenientes dessa análise resultam em um artefato conhecido como orçamento do jogo.

Segundo Mcdermott (2012), orçamento de jogo é a planta baixa ou guia por meio do qual o conteúdo do jogo é criado. Ou seja, são os padrões que deverão ser

seguidos ao criar o conteúdo para o jogo, bem como codificá-lo. É no orçamento do jogo onde são definidos aspectos como a velocidade de projeção do jogo, o número máximo de vértices que poderão ser renderizados por quadro e a quantidade de recursos de memória que serão usados para as texturas. Por exemplo, para um conceito de jogo rápido, será necessário estipular alta velocidade de projeção no orçamento de jogo, como 30-60 quadros por segundo, e todo o conteúdo criado deverá ser otimizado para permitir que o jogo cumpra esse orçamento da velocidade de projeção.

Nesse ponto, é fundamental que o desenvolvedor possua sólida compreensão do hardware onde o jogo será executado, pois cada plataforma ou dispositivo possui suas próprias limitações, e o que pode rodar bem em uma plataforma não significa necessariamente que poderá rodar bem em outra. São as capacidades do hardware que ditarão, por exemplo, se imagens em alta definição poderão ser utilizadas nas texturas, ou a quantidade máxima de polígonos que poderão ser renderizados (MCDERMOTT, 2012).

Partindo desses princípios, esse capítulo irá abordar os passos fundamentais para a criação do orçamento de um jogo e quais os principais aspectos á serem levados em conta para o desenvolvimento deste artefato.

3.1 ORÇAMENTO DA VELOCIDADE DE PROJEÇÃO

A velocidade de projeção é determinada pelo número de quadros por segundo que o jogo deve rodar e isso depende do estilo do jogo que se pretende desenvolver. Quadros por segundo, também conhecido pelos termos em inglês *framerate* e *Frames Per Second* (*FPS*), é uma medida de como um vídeo de animação é exibido e é usado para indicar a quantidade de imagens que são reproduzidas por segundo em um vídeo. Cada quadro é uma imagem estática. Quanto menor a taxa de quadros, mais nítido torna-se ao olho humano a percepção da transição das imagens e quanto maior a taxa de quadros, maior é a sensação de fluidez da animação no vídeo.

Em jogos projetados para serem rápidos, como os de tiro em primeira pessoa, corrida e etc., o número de quadros por segundo precisa ser alto, para passar ao jogador a sensação de resposta rápida e fluidez. Um valor adequado seria de pelo menos 30 fps. Por outro lado, jogos que não foram projetados para serem rápidos como jogos de cartas, estratégia e etc., não necessitam de uma alta taxa de quadros, podendo ser menor do que 30 fps. Porém, após determinado, a taxa de quadros deve seguir esse orçamento, mantendo-se constante. É muito prejudicial para um jogo diminuir constantemente a taxa de quadros (MCDERMOTT, 2012).

3.2 ORÇAMENTO DO TEMPO DE QUADRO

O orçamento do tempo de quadro ou apenas orçamento do quadro, é determinado pelo tempo de quadro (*frametime*) do jogo, que nada mais é do que o tempo que um quadro leva para ser renderizado. Segundo Mcdermott (2012, p. 16) o orçamento do quadro é o orçamento mais importante para o jogo e quase todas as otimizações a serem realizadas serão para seguir esse orçamento.

O tempo de quadro é medido em milissegundos e para calculá-lo, deve-se dividir 1.000 pelo número de quadros por segundo. Por exemplo, supõe-se que a velocidade de projeção orçada seja de 30 fps, então teríamos um tempo do quadro de 33,3 milissegundos. Definido o orçamento do tempo de quadro, deve-se garantir que o valor estipulado não seja ultrapassado, pois isso implicaria em um problema de desempenho.

3.3 ORÇAMENTO DE VÉRTICES

O orçamento de vértices é o processo de decidir exatamente quantos vértices os objetos que compõem a cena irão conter, e é mostrado por quadro do loop de jogo, mantendo ainda a velocidade de projeção de seu orçamento do jogo. Esse é o segredo para o processo da modelagem, pois não se deve modelar nada sem conhecer as limitações do dispositivo visado, a velocidade de projeção visada e o orçamento geral dos vértices. É necessário ter uma contagem de vértices calculada antes que um único polígono seja criado no aplicativo 3D (MCDERMOTT, 2012, p. 20).

A Unity (2013b) recomenda usar um valor inferior a 10.000 vértices visíveis por quadro, porém Mcdermott (2012, p.20 - 21) afirma que o orçamento de vértices, tal como o orçamento da velocidade de projeção, é relativo e depende unicamente do jogo a ser criado. Se o jogo a ser desenvolvido possuir renderização pesada e for composto por muitos objetos nas cenas, as simulações da física deverão ser reduzidas. Em contrapartida, se o jogo for pesado nos cálculos da física, serão exigidos mais recursos da CPU e, assim, contagens de vértices menores e menos objetos em cena, com a finalidade de equilibrar o desempenho geral do jogo.

Uma prática extremamente útil que pode ser feita nos estágios iniciais do desenvolvimento do jogo, é criar uma cena de teste de desempenho, a fim de avaliar as capacidades do hardware visado. Essa cena de teste consiste basicamente em um cenário simples com um único modelo 3D próximo do nível de detalhes desejado para o personagem principal, dotado de malha de pele com animações, script para movimentação padrão do modelo 3D e um script responsável por criar novas instâncias desse modelo na cena, ao toque de um botão, criado na GUI. Com a cena compilada e rodando no dispositivo visado, basta pressionar o botão que cria uma nova instância do modelo 3D na cena, para gerar um clone desse modelo. Dessa forma é possível criar diversos clones, aumentando a contagem de vértices e malhas de pele em tempo de execução. Com o *Internal Profiler* deve-se monitorar as informações do tempo de quadro (*frametime*) e quando o valor exibido ultrapassar o valor do tempo de quadro orçado anteriormente, o valor encontrado do número de vértices será então o limite estimado que o hardware pode suportar. Essa então seria uma estimativa aproximada do desempenho da cena, já que não se leva em consideração os scripts ou a física usados em um jogo completo (MCDERMOTT, 2012, p. 21 - 23). A figura 2 exemplifica a construção de uma cena de teste de desempenho no editor da Unity, feita a partir dos recursos do tutorial Penélope, disponível no Unity Asset Store.

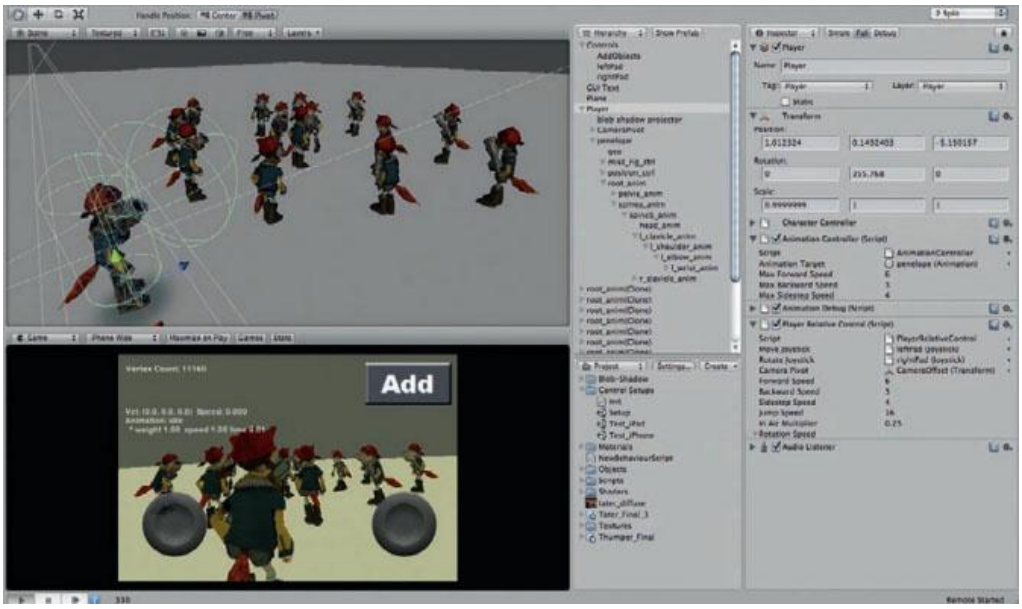


Figura 2 - Cena de teste de desempenho no editor da Unity (MCDERMOTT, 2012, p. 22).

3.4 ORÇAMENTO DE TEXTURAS

Ao determinar o orçamento de texturas, volta-se ao paradigma de que tudo depende do tipo de jogo que se deseja criar e do dispositivo qual se deseja exportar o jogo. Para Mcdermott (2012, p.56 - 57), a melhor solução é criar uma nova cena de teste para avaliar o desempenho e com o *Internal Profiler* ou com a janela *Statistics* do editor, monitorar a quantidade de memória que os objetos consumirem, tendo assim uma representação adequada de como os objetos do jogo e as texturas irão utilizar de recursos do sistema.

4 APLICANDO AS TÉCNICAS DE OTIMIZAÇÃO

As técnicas aqui apresentadas podem ser aplicadas na criação de jogos com Unity independente da plataforma visada, porém de maneiras específicas e com escopo distinto. O escopo das técnicas aqui demonstradas são os dispositivos móveis, considerando suas limitações de hardware e particularidades. Os dispositivos móveis são conhecidos por terem capacidades de hardware inferiores aos de um computador pessoal, portanto existe a possibilidade de que, para outras plataformas além dos dispositivos móveis, a aplicação de algumas dessas técnicas nem seja necessário, sendo fundamental outro estudo de caso específico.

A Unity possibilita que um único jogo desenvolvido seja exportado para diversas plataformas, porém sem as otimizações necessárias, um jogo exportado para dispositivos móveis não atingirá a mesma performance obtida ao ser executado em um computador pessoal. Devido a isso se faz necessário um estudo aprofundado sobre quais técnicas e métodos de otimização são adequados para cada plataforma visada.

As figuras 3 e 4 mostram as diferenças de performance obtidas ao se executar um jogo desenvolvido com a Unity em um computador pessoal e em um tablet, apenas exportando-os para ambas plataformas. No computador, a media de quadros por segundo obtida foi de 200 FPS (Figura 3), já o mesmo jogo rodando no tablet obteve uma media de aproximados 15 FPS (Figura 4), deixando claras as limitações de hardware dos dispositivos móveis. O computador pessoal utilizado foi um PC com processador AMD Athlon II X2 2.9GHz, 2GB de RAM, placa gráfica Nvidia GeForce 6150SE e sistema operacional Windows 7. Já o tablet é um Motorola Xoom 2 MZ 607 com processador da Texas Instruments OMAP 4430 1.2GHz, 1GB de RAM, placa gráfica PowerVR SGX540 e sistema operacional Android 4.0. O jogo utilizado nos testes foi o demonstrativo da Unity, AngryBots.

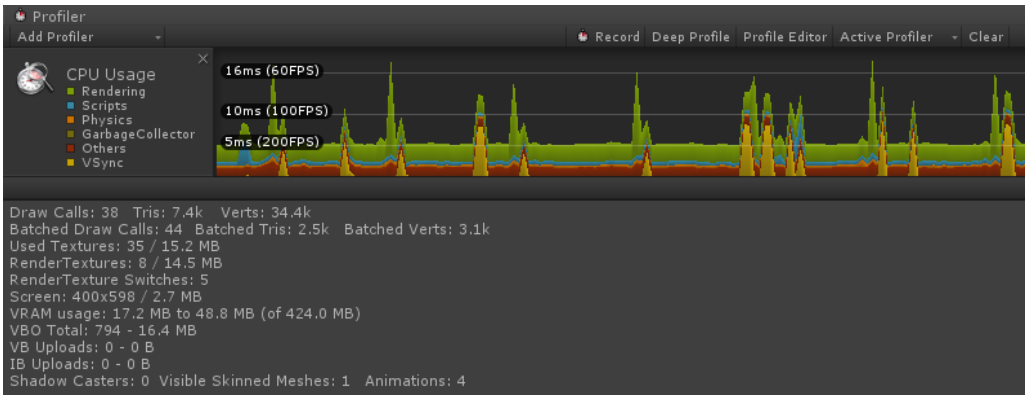


Figura 3 - Profiler do jogo demonstrativo da Unity, AngryBots rodando em um PC.

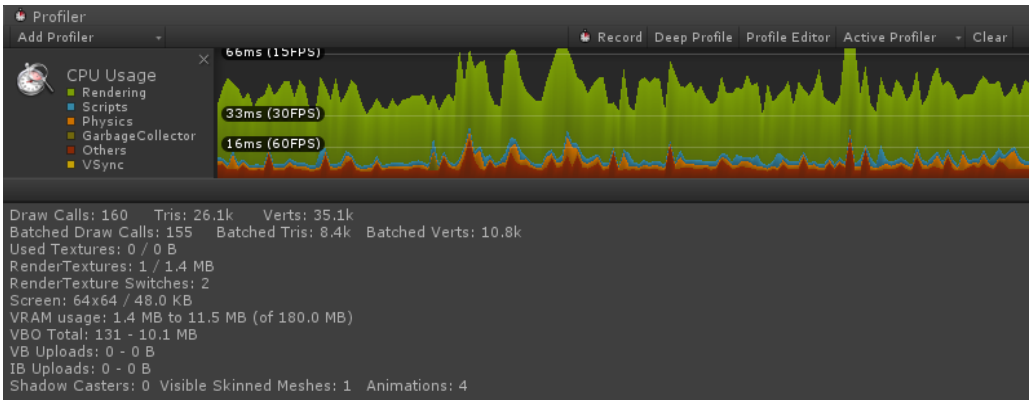


Figura 4 - Profiler do jogo demonstrativo da Unity, AngryBots rodando em um tablet.

As técnicas abordadas nesse estudo estão divididas entre dois grandes grupos: otimizações que envolvem a parte gráfica e otimizações que englobam a parte lógica de um jogo.

4.1 OTIMIZANDO OS GRÁFICOS E A RENDERIZAÇÃO

Para Mcdermott (2012, p.18), as principais áreas a se focar na otimização de um jogo para dispositivos móveis, em termos de arte são: as chamadas de desenho, a contagem de vértices por quadros, a quantidade de texturas em memória e os *shaders*. Unity (2013c), considera as áreas citadas acima, em uma lista de diretrizes para a otimização da performance gráfica de um jogo. A partir dessas considerações, as áreas supracitadas foram escolhidas para serem abordadas com maior profundidade nas seções subsequentes.

4.1.1 CHAMADAS DE DESENHO

Chamada de desenho é o comando que diz a GPU para processar um determinado conjunto de vértices de triângulos com um determinado estado (*shaders*, dados posicionais, normais e etc.). Para desenhar um objeto na tela, a *engine* emite uma chamada de desenho para a API gráfica (por exemplo, OpenGL ou Direct3D), que então prepara os comandos para serem enviados para a GPU, para que por fim o objeto seja renderizado. Todo o processamento para que os dados dos vértices cheguem até a GPU para serem renderizados, é realizado pela CPU. Portanto, esse custo da CPU "por objeto", pode-se tornar um gargalo caso tenha-se muitos objetos a serem renderizados. Assim, por exemplo, com milhares de triângulos, será menos trabalhoso para a CPU se eles estiverem todos em uma única malha tridimensional[9], ao invés de se ter milhares de malhas individuais com um triângulo cada uma. O custo de ambos os cenários sobre a GPU será muito semelhante, mas o trabalho feito pela CPU para renderizar milhares de objetos (ao invés de um) será significativo. (Unity, 2013c).

A fim de fazer com que a CPU realize menos trabalho, deve-se manter o número de objetos vísiveis por quadro, reduzido. Para isso, a técnica de chamadas de desenho em *batching* pode ser aplicada.

4.1.1.1 BATCHING

A Unity pode combinar um número de objetos em tempo de execução e desenhá-los com uma única chamada de desenho. Esta operação é denominada *batching*. Quanto maior o número de objetos em que a operação de *batching* puder ser aplicada, melhor será o desempenho de renderização no lado da CPU e consequentemente o número de chamadas de desenho será reduzido.

Para que o *batching* seja aplicado, os objetos devem compartilhar o mesmo material, ou seja, utilizar a mesma instância de material. Portanto para obter um bom nível de *batching*, é necessário compartilhar o maior número possível de materiais

[9] Malha tridimensional ou *mesh*, são modelos computadorizados tri-dimensionais, constituidos de uma malha poligonal, aos quais são atribuidos materiais de acabamento (textura) e iluminação.

entre os objetos do jogo. Na Unity, existem dois tipos de *batch*: *batch* dinâmico e *batch* estático.

4.1.1.1.1 BATCH DINÂMICO

A Unity aplica o *batch* dinâmico nos objetos não estáticos automaticamente em tempo de execução, desde que a malha a ser renderizada contenha menos do que 900 vértices no total, dependendo do *shader* usado. Se o *shader* do material utilizar atributos de posição de vértice, normais, e atributos de mapeamento UV[10], então o batch só é aplicado se a malha conter até 300 vértices.

As figuras 5 e 6 exemplificam o uso da aplicação do *batch* dinâmico. Na figura 5, dois objetos compartilham o mesmo material, consumindo apenas uma chamada de desenho. Já na figura 6 os objetos não compartilham o mesmo material, fazendo com que o *batch* dinâmico não seja aplicado, gerando então uma chamada de desenho a mais, com relação aos resultados da figura 5.

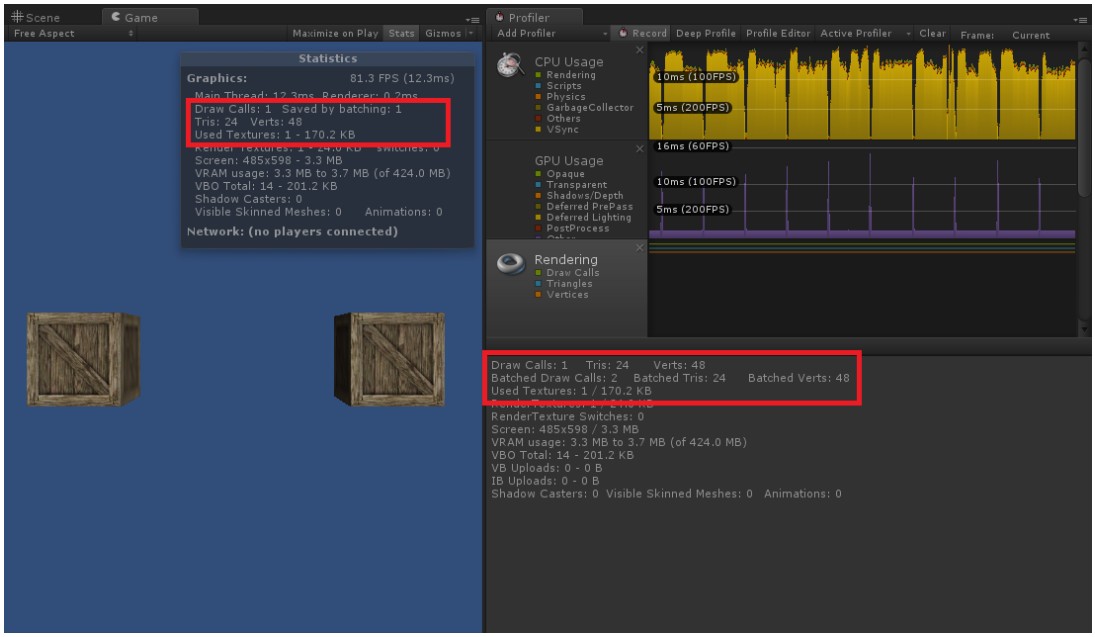


Figura 5 – Resultados da renderização de dois objetos onde o *batch* dinâmico é aplicado.

[10] Mapeamento UV é o processo de modelagem 3D que cria um mapeamento de um modelo 3D em uma imagem 2D. As letras U e V denotam os eixos cartesianos X e Y na imagem 2D, já que as letras X, Y e Z já são usadas para a representação espacial do modelo 3D.

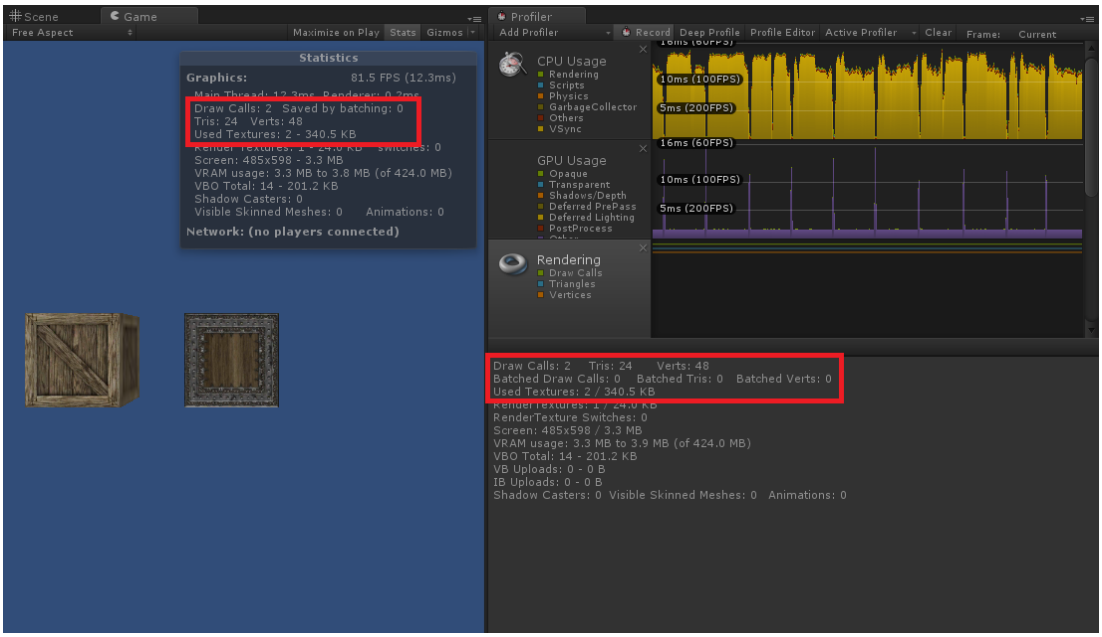


Figura 6 - Resultados da renderização de dois objetos onde o *batch* dinâmico não é aplicado.

4.1.1.1.2 BATCH ESTÁTICO

O *batch* estático permite a redução das chamadas de desenho para geometrias de qualquer tamanho, ou seja, não existe uma restrição para o número de vértices do objeto, porém só pode ser aplicado em objetos que não irão se mover, rotacionar ou sofrer alterações de escala durante o jogo. Para que o *batch* estático seja aplicado, os objetos devem ser marcados como estáticos usando a caixa de seleção *Static* da guia *Inspector* no editor da Unity, como mostra a figura 7.

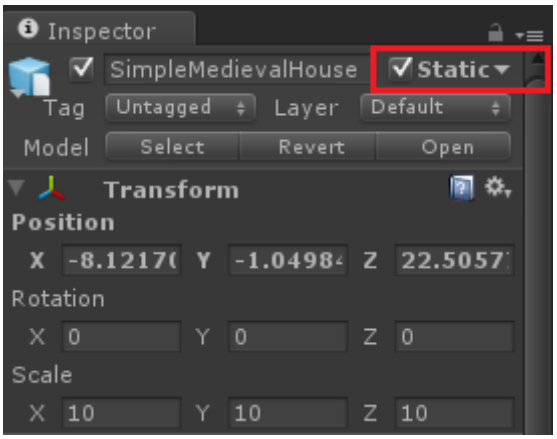


Figura 7 - Caixa de seleção *Static* da guia Inspector.

As figuras 8 e 9 exemplificam o uso da aplicação do *batch* estático. Na figura 8, dois objetos de geometrias idênticas que compartilham o mesmo material, não marcados como estáticos, consomem quatro chamadas de desenho, e não há aplicação de *batch*. Já na figura 9, os mesmo objetos são renderizados, mas agora marcados como estáticos, consomem apenas três chamadas de desenho.

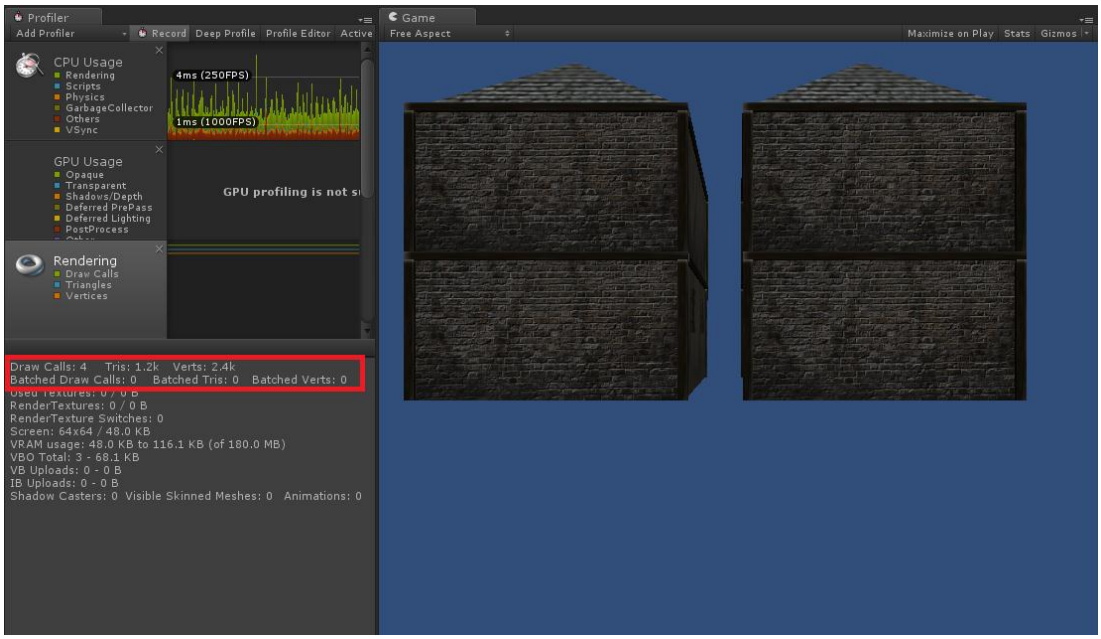


Figura 8 – Resultados da renderização de dois objetos onde o *batch* estático não é aplicado.

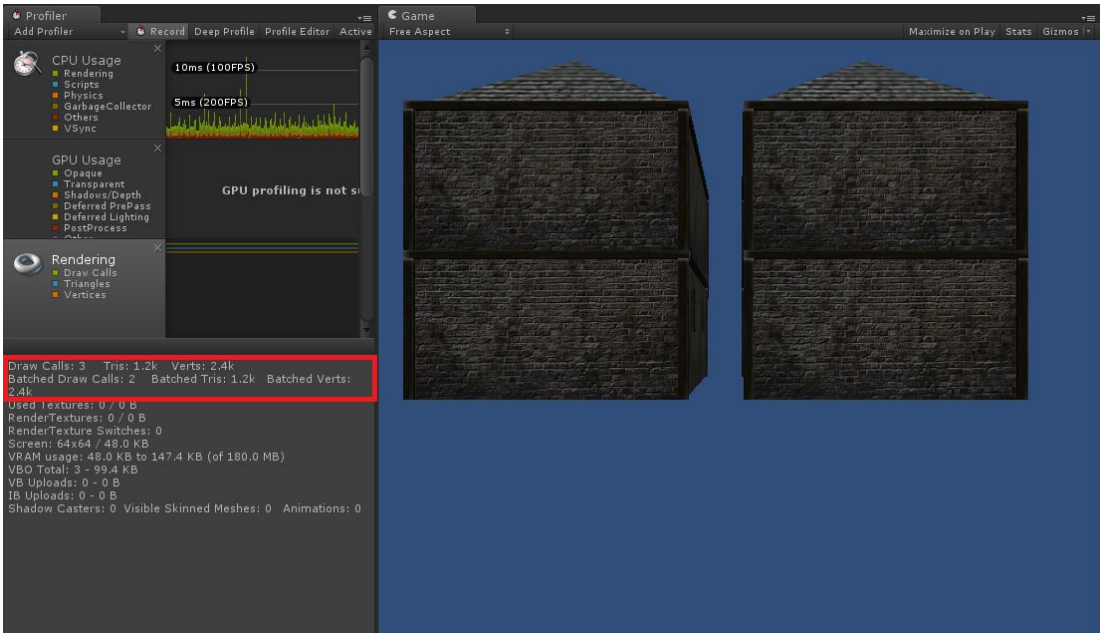


Figura 9 - Resultados da renderização de dois objetos onde o *batch* estático é aplicado.

4.1.2 CONTAGEM DE VÉRTICES

A contagem de vértices se restringe praticamente a modelagem de baixa resolução (*low poly*), devido aos recursos limitados dos dispositivos móveis. Os modelos 3D devem ser criados respeitando o orçamento de vértices pré-definido, atentando-se sempre aos valores do número de vértices que a Unity exibe na janela *Statistics* ou através do *Internal Profiler* e nunca pela contagem de vértices exibida pelo software 3D de modelagem. Segundo Mcdermott (2012, p.29), o número exibido na Unity é o número resultante de vértices enviados para a GPU, o que inclui os adicionados por luzes e *shaders* complexos. Além disso, outro fator que pode aumentar o número de vértices a serem renderizados pela GPU são as bordas duras.

Ao importar um objeto 3D para a Unity ou na própria utilização do software de modelagem, o ângulo de suavização dos polígonos deve ser ajustado. A figura 10 exibe os ajustes do ângulo de suavização na janela de importação da Unity. Como afirma Mcdermott (2012, p.30), sem suavização, os polígonos que compõem uma malha seriam visíveis como lados. Por isso a opção *Smoothing Angle* é utilizada, para determinar a tolerância de ângulo máxima entre os polígonos conectados. Porém, se o ângulo entre os polígonos conectados for maior que a definição do *Smoothing Angle*, não ocorrerá nenhuma suavização. A GPU então irá dividir os vértices nesse ângulo para criar a borda dura, resultando em uma costura de malha e consequentemente, aumentando o número de vértices do objeto. A figura 11 compara duas malhas com ângulos de suavização diferentes. Os polígonos da figura 11a possuem ângulo de suavização de 180 graus, já a figura 11b mostra a divisão dos vértices na criação de uma borda dura pela GPU, qual possui maior número de vértices.

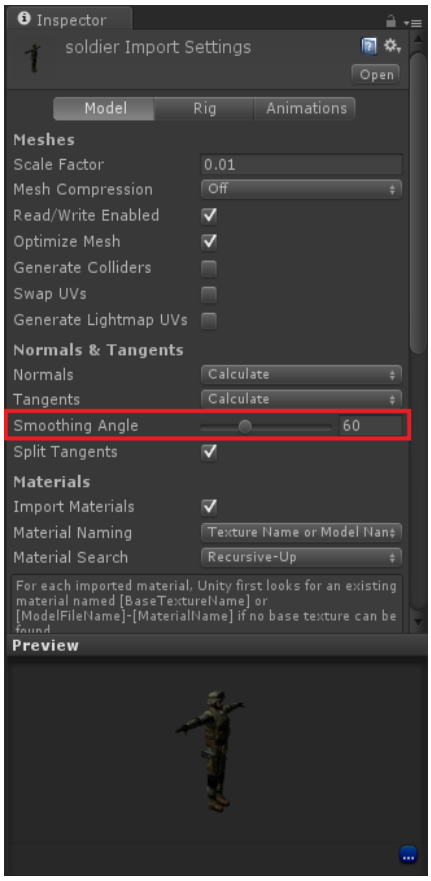


Figura 10 – Controle do ângulo de suavização nos ajustes de importação da Unity.

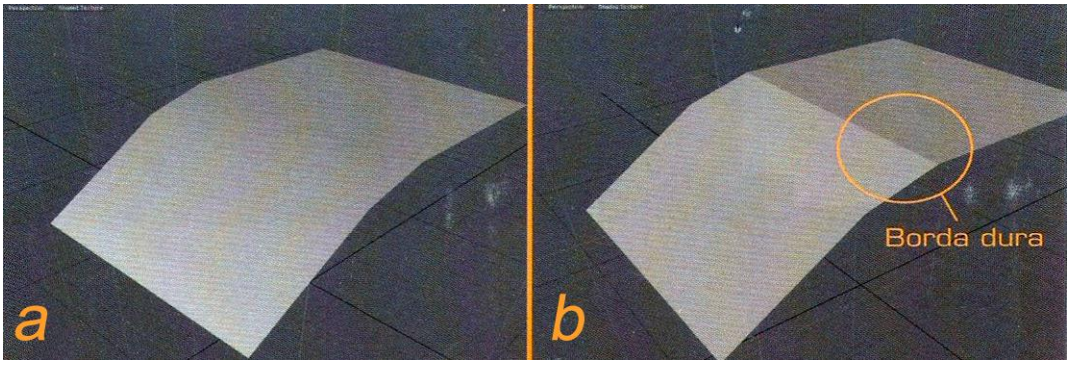


Figura 11 – Comparação de malhas com ângulos de suavização diferentes (MCDERMOTT, 2012, p. 31).

4.1.3 TEXTURAS

Segundo Mcdermott (2012, p.11), as texturas são os principais consumidores de memória RAM em um jogo, por isso é muito importante o uso de texturas

comprimidas[11] e otimizadas, para minimizar o uso de RAM do dispositivo durante a execução do jogo. A Unity possui o recurso de comprimir as texturas importadas no projeto, o que contribui para a redução do tamanho final do arquivo executável a ser gerado e principalmente para a redução do consumo de memória RAM.

Uma textura pode ter dimensões em potência de 2 (POT) ou não potência de 2 (NPOT). Uma textura de dimensão POT, significa que possui dimensão igual a 2 elevado a um número x. Alguns exemplos de dimensões POT são 1.024 pixels, 512 pixels, 256 pixels, 128 pixels. Performaticamente as texturas devem ter dimensões POT e não necessariamente precisam ser quadradas, por exemplo, a largura pode ser diferente da altura (salvo exceções onde a GPU do dispositivo exige o uso de texturas quadradas), conforme Unity (2013d), pois as texturas de dimensões NPOT consomem mais memória e são mais lentas para serem lidas pela GPU.

Para todas as texturas importadas em um projeto, a Unity automaticamente aplica uma configuração de compressão padrão, como mostra a figura 12. Dependendo da finalidade dessa textura, o seu tipo deve ser alterado na caixa de seleção *Texture Type* da guia *Inspector* do editor (Figura 13), para a obtenção de melhores resultados.

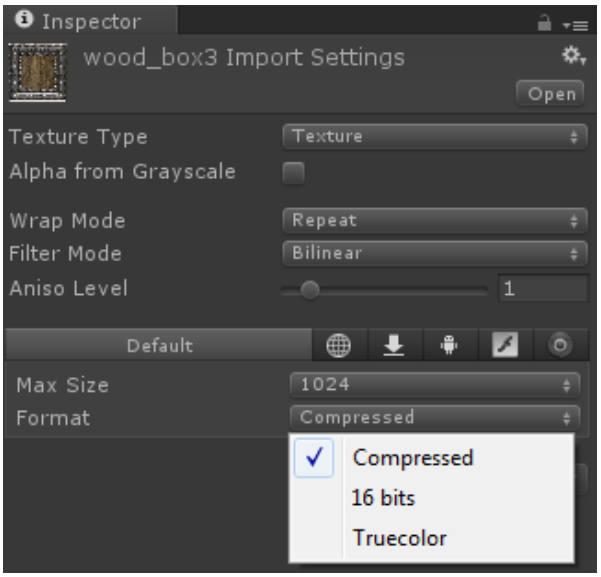


Figura 12 - Configurações padrão nos ajustes de importação de texturas da Unity.

[11] Ao citar texturas comprimidas (ou compactadas), o autor se refere a compressão de imagens realizada pela Unity.

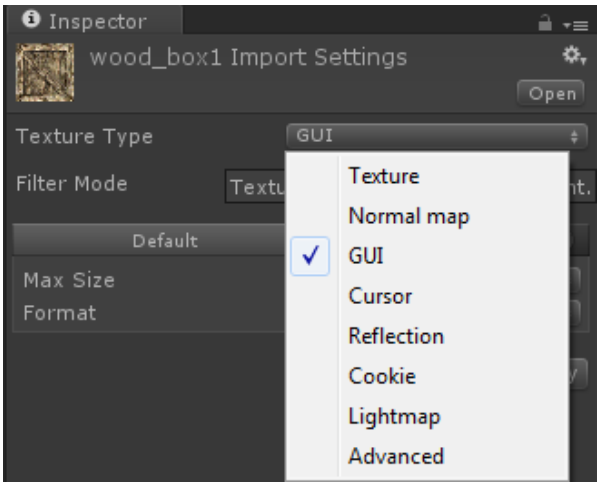


Figura 13 - Configurações de tipo de textura nos ajustes de importação de texturas da Unity.

Para uma configuração manual das propriedades da textura, a opção *Advanced* deve ser selecionada em *Texture Type*, possibilitando assim a configuração e a opção do uso ou do não uso de diversas propriedades, tal como o método de compressão a ser utilizado nessa textura. Para os dispositivos Android, o formato de compressão ETC é o mais recomendado, pois é o formato de compressão padrão desse sistema operacional e é suportado em todos os dispositivos a partir de sua versão 2.0. Porém, o formato ETC não suporta o canal alfa, portanto para texturas que possuirem transparência, a melhor opção é o tipo RGBA 16-bit (UNITY, 2013e).

Para os dispositivos IOS, o formato de compressão recomendado é o PVRTC, pois é o formato de compressão nativo desses dispositivos (UNITY, 2013f). Além disso, as texturas que utilizarem esse formato, obrigatóriamente devem ser quadradas e com dimensões POT, conforme recomenda Apple (2013).

Mcdermott (2012, p.53) reforça a idéia do uso do formato PVRTC em texturas para os dispositivos IOS, pois esse formato consegue reduzir o tamanho da textura em uma boa porcentagem de compressão e manter um bom nível de qualidade. Por exemplo, uma imagem de 256 x 256 e 32 bpp (bits por pixel) com transparência ocupará 256 KB de memória, enquanto que comprimida com o PVRTC 4-bits ocupará 32 KB. A figura 14 compara uma textura de tijolos com 32 bits e 4 bits

PVRTC. A diferença na qualidade é praticamente imperceptível, porém a diferença no tamanho ocupado em memória é significativa.

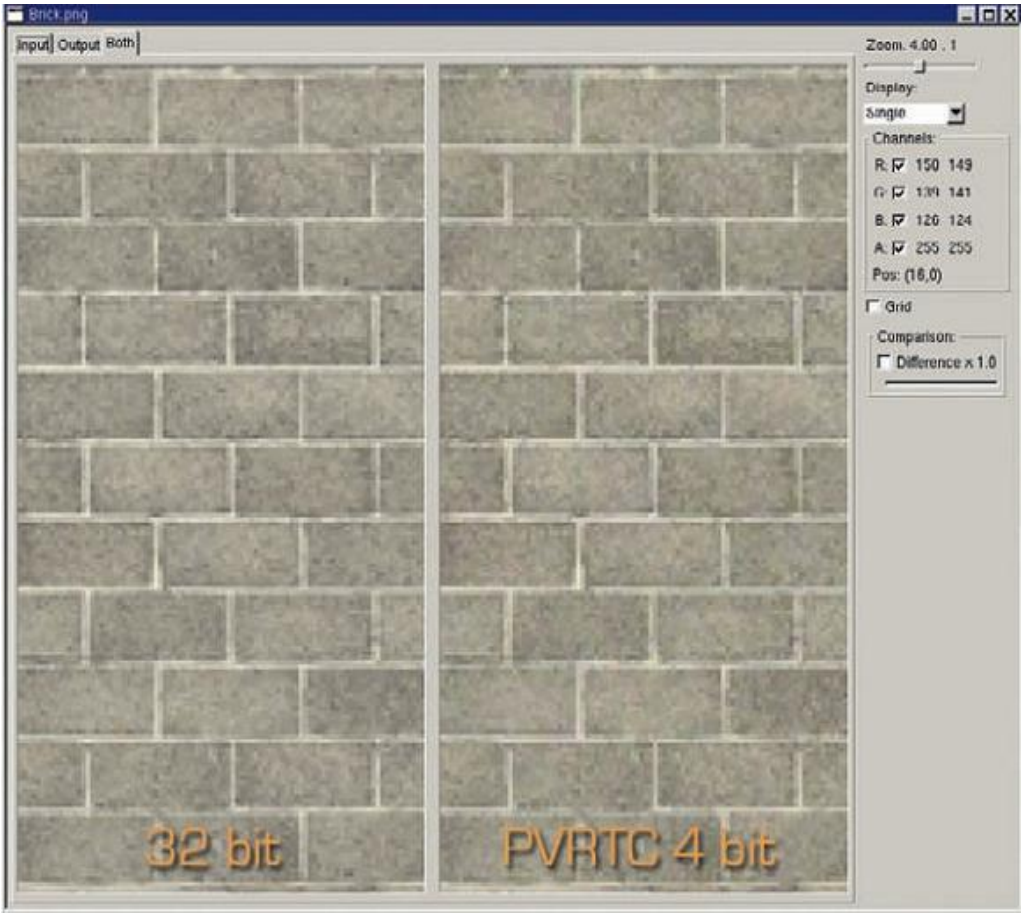


Figura 14 - Comparação entre uma textura de tijolos com 32 bits e 4 bits PVRTC (MCDERMOTT, 2012, p. 53).

Outra propriedade de textura, configurável na Unity (Figura 15), e que pode ser utilizada para a otimização de performance é o uso de mapas Mip. Mapa Mip é uma lista com versões progressivamente menores de uma imagem, utilizado para otimizar o desempenho em motores 3D em tempo real. As texturas dos objetos que estiverem longe da câmera são trocadas em tempo de execução, por versões menores e mais otimizadas. O uso de mapas Mip acresce em 33% a mais o consumo de memória, mas não usá-los pode ser uma grande perda de desempenho. Os únicos tipos de texturas em que não é recomendado o uso de mapas Mip são aquelas que nunca devem ser minificadas, como por exemplo, texturas de GUI (UNITY, 2013d).

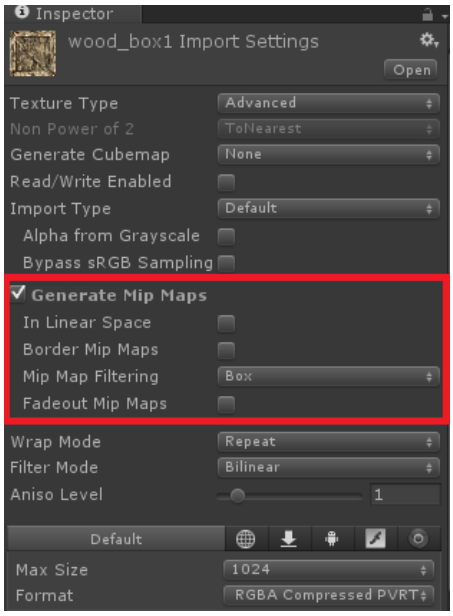


Figura 15 - Caixa de seleção para a geração de Mapas Mip nos ajustes de importação de texturas da Unity.

4.1.4 SHADERS

Shaders são programas com instruções de como a GPU deve renderizar certas imagens na tela. São os *shaders* que definem como um objeto vai aparecer no jogo e como ele vai reagir à iluminação. A Unity traz cerca de oitenta *shaders* prontos para serem usados, o que inclui *shaders* otimizados para dispositivos móveis, e também possibilita a criação ou customização de seus próprios *shaders*.

Na Unity os *shaders* podem ser escritos de três maneiras diferentes: “Shaders de Função Fixa”, “Shaders de Vértice e Fragmento” e “Shaders de Superfície”. O código para esses *shaders* é encapsulado usando uma linguagem chamada ShaderLab, que é utilizada para organizar a estrutura do *shader*. Os *shaders* em si, são tipicamente escritos nas linguagens CG ou HLSL (UNITY, 2013f).

Os “Shaders de Superfície” são normalmente utilizados se o *shader* precisar ser afetado por luzes e sombras, ao contrário dos “Shaders de Vértice e Fragmento”, que podem ser utilizados se o *shader* não precisar interagir com iluminação ou quando houver a necessidade de lidar com algum efeito que não pode ser manipulado com os “Shaders de Superfície”. Já os “Shaders de Função Fixa” são

usados para hardwares antigos que não suportam *shaders* programáveis e são escritos usando a linguagem de *shaders* da Unity, ShaderLab (UNITY, 2013f). A figura 16 exibe um exemplo de script de “Shader de Superfície” Diffuse Simple e o resultado de sua aplicação.

Figura 16 - Exemplo do código de um “Shader de Superfície” e o resultado de sua aplicação.

De acordo com Unity (2013g), da perspectiva de performance, existem duas categorias básicas de *shaders*: Pixel-Lit e Vertex-Lit. Os *shaders* Pixel-Lit calculam a iluminação para cada pixel que é desenhado, por isso o objeto tem que ser desenhado várias vezes, dependendo do número de luzes adicionais que brilham sobre ele. Isso aumenta a carga na CPU para processar e enviar comandos para a GPU, e na GPU para processar os vértices e desenhar os pixels na tela. Portanto, são muito mais caros em termos de processamento do que os Vertex-Lit. Os *shaders* Vertex-Lit calculam a iluminação com base nos vértices da malha, usando todas as luzes ao mesmo tempo, o que significa que a malha só precisará ser desenhada uma única vez (UNITY, 2013g).

Mcdermott (2012, p.220 - 221) ressalta a importância de que nenhum *shader* Pixel-Lit seja usado para as plataformas móveis, pois os *shaders* podem ter um impacto significativo no desempenho do jogo e sugere que os *shaders* otimizados para dispositivos móveis da Unity sejam utilizados o máximo possível.

As figuras 17 e 18 exemplificam a diferença de desempenho no uso de *shaders* otimizados para dispositivos móveis e os não específicos para a plataforma. A cena utilizada para o teste é um nível de amostra do jogo ShadowGun disponibilizado por MadFinger (2012). Os resultados apresentados com o uso de

shaders otimizados para dispositivos móveis demostraram-se superiores, com uma taxa de quadros de aproximadamente 200 FPS (Figura 17). Sem o uso de *shaders* otimizados a quantidade de quadros por segundo é reduzida drasticamente para aproximados 150 FPS (Figura 18).

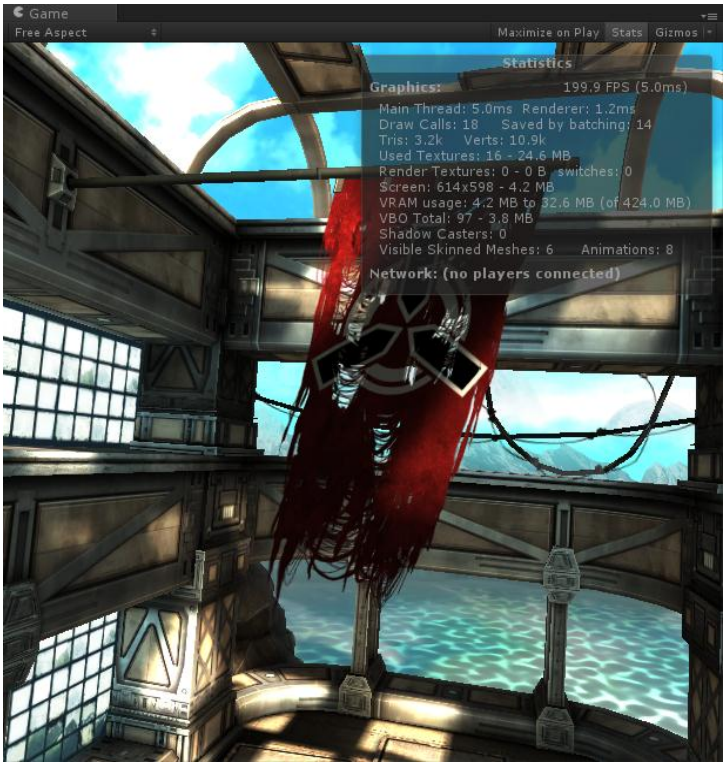


Figura 17 - Resultados da renderização de uma cena utilizando *shaders* otimizados para dispositivos móveis.

Figura 18 - Resultados da renderização de uma cena sem o uso de *shaders* otimizados para dispositivos móveis.

4.2 OTIMIZANDO A PROGRAMAÇÃO

Na Unity, os scripts responsáveis pela parte lógica do jogo, podem ser escritos em até três linguagens diferentes: Javascript, C# ou Boo. Não existem indícios de que uma linguagem, na Unity, seja mais performática do que outra, passando a ser apenas uma questão de opção do desenvolvedor. Para este estudo em questão, a linguagem escolhida foi o Javascript, porém as técnicas de programação descritas a seguir, podem ser utilizadas nas outras linguagens permitidas pela *engine*, desde que sejam adaptadas.

A principal otimização de scripts neste contexto trata-se basicamente de evitar a sobrecarga de alocação de memória. Quanto mais a alocação de memória for evitada, menor será o trabalho do coletor de lixo e menores serão as chances da criação ou destruição de um objeto ocasionar uma queda de desempenho, seja pela alocação ou seja pela demasiada coleção do lixo de memória gerado (UNITY, 2013h).

4.2.1 EVITANDO ALOCAÇÃO DE MEMÓRIA

Na Unity, a cada vez que um objeto é criado, a memória necessária para guardá-lo é alocada de um *pool* central chamado *heap*. Quando este objeto não estiver mais em uso, a memória que ele ocupava pode ser reclamada e usada para outros fins. Em sistemas antigos, cabia ao programador alocar e liberar blocos de memória *heap*, fazendo chamadas explícitas no código, porém, sistemas atuais como o Mono[12], utilizado pela Unity, gerenciam a memória automaticamente para o programador, requerindo menor esforço de programação e minimizando as possibilidades de memória ser alocada e subsequentemente nunca ser liberada (UNITY, 2013i).

A alocação de memória é um processo inevitável, porém, o problema se encontra quando ela é realizada de maneira desnecessária. Frequentemente no código, objetos são criados até mesmo sem o conhecimento do desenvolvedor. Para tal, existem algumas boas práticas de codificação e a técnica do *pool* de objetos, que contribuem para a redução de alocação de memória e otimizam a coleta de lixo de memória feita pelo coletor de lixo da Unity. Esses serão os assuntos abordados nos subtópicos seguintes.

4.2.1.1 OTIMIZANDO O COLETOR DE LIXO DA UNITY

O processo de localização e liberação de memória não utilizada é conhecido como coleta de lixo. Na Unity, a coleta de lixo é automática e invisível para o programador, porém o processo de coleta exige tempo de CPU significativo em segundo plano. Por tanto é fundamental que o programador evite erros que irão acionar o coletor mais vezes do que o necessário e introduzir pausas na execução. Para isso, alguns algoritmos devem ser utilizados de maneira adequada, como os de concatenação de strings e os de funções que retornam arrays (UNITY, 2013i).

A figura 19 demonstra uma maneira comum de se concatenar strings e outra maneira otimizada e mais adequada, na linguagem Javascript. O script Concat.js demonstra uma maneira comum de se concatenar strings, já o script

[12] http://www.mono-project.com/

Concat_Builder.js apresenta a concatenação de strings de uma forma otimizada, utilizando a classe *System.Text.StringBuilder* da biblioteca Mono. Ambos os scripts foram desenvolvidos para avaliar as diferenças entre os dois métodos de concatenação de strings, por isso o uso da concatenação dentro da função *Update*, para que os resultados fossem perceptíveis e pudessem ser identificados através do *Profiler*. A função *Start* funciona apenas para popular o array com uma carga inicial, para que então a cada atualização de quadro, o array seja percorrido e a string inicial seja aumentada. Em uma situação normal, o ideal seria a variável do tipo string, á ser concatenada, ser declarada fora do escopo da função *Update*, para que ela não seja instânciada a cada atualização de quadro.

```
//Concat.js

#pragma strict

var intArray : int[] = new int[50];

function Start() {
      for (var i = 0; i < intArray.Length; i++) {
                intArray[i] = i;
      }
}

function Update () {

      var line = intArray[0].ToString();

      for (var i = 1; i < intArray.Length; i++) {
            line += ", " + intArray[i].ToString();
      }

}
```

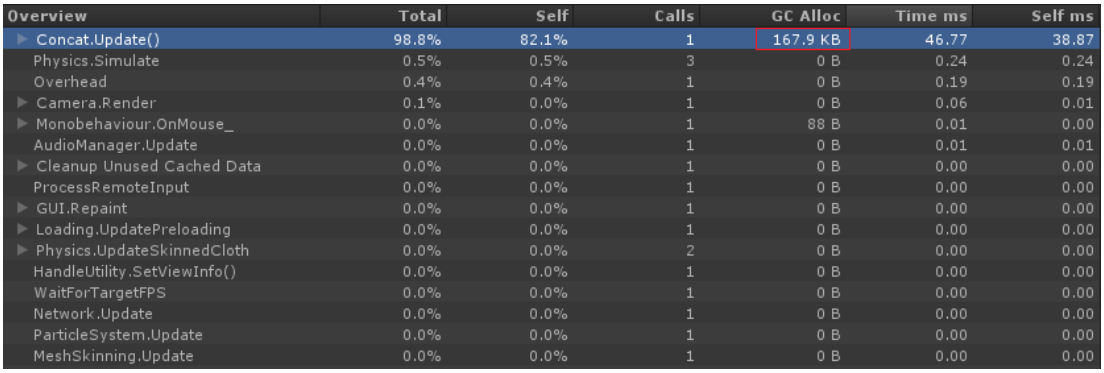
```
//Concat_Builder.js

#pragma strict

var intArray : int[] = new int[50];

function Start() {
      for (var i = 0; i < intArray.Length; i++) {
                intArray[i] = i;
      }
}

function Update () {

      var builder : System.Text.StringBuilder =
                  new System.Text.StringBuilder();

      for (var i = 1; i < intArray.Length; i++) {
            builder.Append(", " + intArray[i].ToString());
      }

}
```

Figura 19 – Comparação de algoritmos de concatenação de strings na linguagem Javascript.

A grande diferença encontrada entre os scripts Concat.js e Concat_Builder.js é justamente na maneira como as strings são concatenadas. No script Concat.js, a cada iteração do loop dentro da função *Update*, o conteúdo anterior da variável line se torna morto: uma string completamente nova é alocada para conter a parte original mais a nova parte em seu fim. Por tanto, conforme a string se torna maior com o aumento dos valores da variável i, a quantidade de espaço de memória *heap* sendo consumida também aumenta. Devido á isso, para um melhor desempenho, Unity (2013i) recomenda o uso da classe *System.Text.StringBuilder*, cujo o valor é uma sequência de caracteres mutáveis, sendo que todas e quaisquers alterações

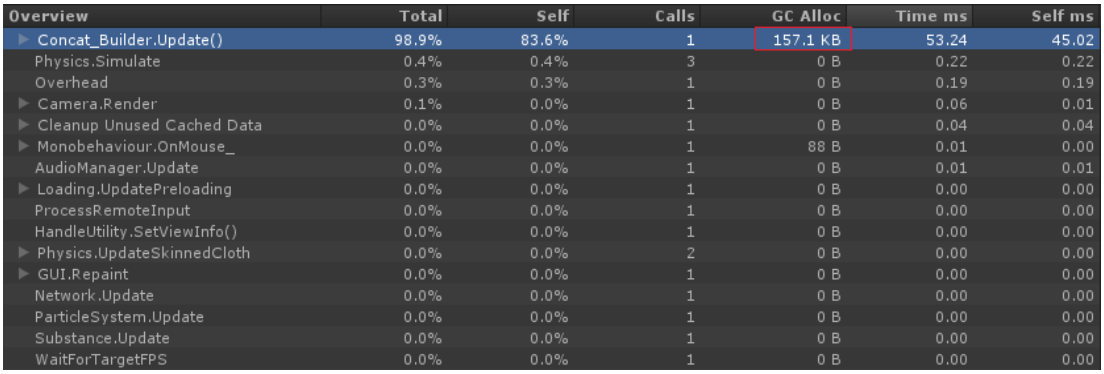
realizadas em alguma instância dessa classe, são retornadas como referência para a mesma instância (MICROSOFT, 2013).

As figuras 20 e 21 demonstram as diferenças de performance encontradas no uso dos scripts Concat.js e Concat_Builder.js. Com o uso do script Concat.js a cada atualização de quadro, um total de 167.9 KB de memória é enviado para o coletor de lixo, já utilizando o script Concat_Builder.js, é enviado apenas 157.1 KB, totalizando uma diferença de 10.8 KB. Claro que, com um único script sendo executado, essa quantidade de memória pode parecer insignificante, porém quanto maior for o seu uso, maior também será o ganho de performance.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
Concat.Update()	98.8%	82.1%	1	167.9 KB	46.77	38.87
Physics.Simulate	0.5%	0.5%	3	0 B	0.24	0.24
Overhead	0.4%	0.4%	1	0 B	0.19	0.19
Camera.Render	0.1%	0.0%	1	0 B	0.06	0.01
Monobehaviour.OnMouse_	0.0%	0.0%	1	88 B	0.01	0.00
AudioManager.Update	0.0%	0.0%	1	0 B	0.01	0.01
Cleanup Unused Cached Data	0.0%	0.0%	1	0 B	0.00	0.00
ProcessRemoteInput	0.0%	0.0%	1	0 B	0.00	0.00
GUI.Repaint	0.0%	0.0%	1	0 B	0.00	0.00
Loading.UpdatePreloading	0.0%	0.0%	1	0 B	0.00	0.00
Physics.UpdateSkinnedCloth	0.0%	0.0%	2	0 B	0.00	0.00
HandleUtility.SetViewInfo()	0.0%	0.0%	1	0 B	0.00	0.00
WaitForTargetFPS	0.0%	0.0%	1	0 B	0.00	0.00
Network.Update	0.0%	0.0%	1	0 B	0.00	0.00
ParticleSystem.Update	0.0%	0.0%	1	0 B	0.00	0.00
MeshSkinning.Update	0.0%	0.0%	1	0 B	0.00	0.00

Figura 20 – Resultados da aplicação do script Concat.js.



Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
Concat_Builder.Update()	98.9%	83.6%	1	157.1 KB	53.24	45.02
Physics.Simulate	0.4%	0.4%	3	0 B	0.22	0.22
Overhead	0.3%	0.3%	1	0 B	0.19	0.19
Camera.Render	0.1%	0.0%	1	0 B	0.06	0.01
Cleanup Unused Cached Data	0.0%	0.0%	1	0 B	0.04	0.04
Monobehaviour.OnMouse_	0.0%	0.0%	1	88 B	0.01	0.00
AudioManager.Update	0.0%	0.0%	1	0 B	0.01	0.01
Loading.UpdatePreloading	0.0%	0.0%	1	0 B	0.00	0.00
ProcessRemoteInput	0.0%	0.0%	1	0 B	0.00	0.00
HandleUtility.SetViewInfo()	0.0%	0.0%	1	0 B	0.00	0.00
Physics.UpdateSkinnedCloth	0.0%	0.0%	2	0 B	0.00	0.00
GUI.Repaint	0.0%	0.0%	1	0 B	0.00	0.00
Network.Update	0.0%	0.0%	1	0 B	0.00	0.00
ParticleSystem.Update	0.0%	0.0%	1	0 B	0.00	0.00
Substance.Update	0.0%	0.0%	1	0 B	0.00	0.00
WaitForTargetFPS	0.0%	0.0%	1	0 B	0.00	0.00

Figura 21 - Resultados da aplicação do script Concat_Builder.js.

Outra boa prática de programação para a otimização do desempenho da coleta de lixo é no uso de funções que retornam arrays por referência. A figura 22 demonstra dois scripts na linguagem Javascript que possuem algoritmos que

retornam valores para um array. A função *RandomList* no script ArrayFunc.js funciona de maneira genérica, pois sempre que é chamada, irá criar um novo array com os valores preenchidos. No entanto, se essa função for chamada repetidamente e dependendo do tamanho do array, o espaço livre de memória *heap* pode se esgotar rapidamente, resultando em frequentes coletas de lixo. Para evitar esse problema, Unity (2013i) recomenda o uso de funções que recebem argumentos passados por referência, como mostra o script ArrayFunc_Ref.js. Na linguagem Javascript, todo argumento do tipo primitivo é passado por valor para a função que o recebe, já os argumentos do tipo objeto são passados por referência[13], ou seja, a função que recebe o objeto irá trabalhar o próprio objeto passado como parâmetro, alterando o seu conteúdo original (MDN, 2013). Dessa forma, ao se passar um array por parâmetro, a função nunca irá gerar coleta de lixo, pois estará alterando o array criado anteriormente, necessitando apenas de sua alocação inicial.

```
//ArrayFunc.js

#pragma strict

var numElements : int = 50;
var array : float[];

function Update () {
      array = RandomList(numElements);

}


function RandomList(numElements: int) {
      var result = new float[numElements];

      for (var i = 0; i < numElements; i++) {
            result[i] = Random.value;
      }

      return result;
}
```

```
//ArrayFunc_Ref.js

#pragma strict

var array : float[] = new float[50];

function Update () {
      RandomList(array);

}


function RandomList(arrayToFill: float[]) {
      for (var i = 0; i < arrayToFill.Length; i++) {
            arrayToFill[i] = Random.value;
      }
}
```

Figura 22 - Comparação de algoritmos que populam um array na linguagem Javascript.

As figuras 23 e 24 demonstram na prática os resultados obtidos na aplicação dos scripts ArrayFunc.js e ArrayFunc_Ref.js. Como se pode observar, utilizando o script ArrayFunc.js, a quantidade de memória enviada para o coletor de lixo é de 224 Bytes, já o script ArrayFunc_Ref.js, como dito anteriormente, não envia memória alguma para o coletor de lixo devido á passagem de parâmetro por referência da

[13] Na Unity, o tipo struct da linguagem Javascript, também é um argumento passado por valor.

função *RandomList*, que altera os valores do mesmo array recebido como parâmetro.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
Overhead	30.6%	30.6%	1	0 B	0.20	0.20
Physics.Simulate	28.7%	28.7%	2	0 B	0.19	0.19
▶ Cleanup Unused Cached Data	13.1%	5.6%	1	0 B	0.08	0.03
▶ Camera.Render	12.2%	3.7%	1	0 B	0.08	0.02
AudioManager.Update	6.5%	6.5%	1	0 B	0.04	0.04
▶ Monobehaviour.OnMouse_	2.9%	0.2%	1	88 B	0.02	0.00
ArrayFunc.Update()	0.8%	0.8%	1	224 B	0.00	0.00
ParticleSystem.Update	0.8%	0.8%	1	0 B	0.00	0.00
▶ GUI.Repaint	0.7%	0.2%	1	0 B	0.00	0.00
Network.Update	0.7%	0.7%	1	0 B	0.00	0.00
HandleUtility.SetViewInfo()	0.5%	0.5%	1	0 B	0.00	0.00
▶ Physics.UpdateSkinnedCloth	0.5%	0.1%	2	0 B	0.00	0.00
▶ Loading.UpdatePreloading	0.4%	0.1%	1	0 B	0.00	0.00
ProcessRemoteInput	0.2%	0.2%	1	0 B	0.00	0.00
MeshSkinning.Update	0.1%	0.1%	1	0 B	0.00	0.00
Substance.Update	0.1%	0.1%	1	0 B	0.00	0.00
WaitForTargetFPS	0.1%	0.1%	1	0 B	0.00	0.00
ParticleSystem.WaitForUpdateThreads	0.0%	0.0%	1	0 B	0.00	0.00
Physics.Interpolation	0.0%	0.0%	2	0 B	0.00	0.00

Figura 23 - Resultados da aplicação do script ArrayFunc.js.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	93.8%	93.8%	1	0 B	14.65	14.65
Physics.Simulate	2.4%	2.4%	1	0 B	0.37	0.37
Overhead	1.1%	1.1%	1	0 B	0.17	0.17
▶ Camera.Render	0.7%	0.2%	1	0 B	0.11	0.03
▶ Physics.UpdateSkinnedCloth	0.6%	0.0%	2	0 B	0.10	0.00
▶ Cleanup Unused Cached Data	0.4%	0.4%	1	0 B	0.06	0.06
▶ Monobehaviour.OnMouse_	0.3%	0.0%	1	88 B	0.04	0.00
AudioManager.Update	0.1%	0.1%	1	0 B	0.02	0.02
HandleUtility.SetViewInfo()	0.1%	0.1%	1	0 B	0.01	0.01
ArrayFunc_Ref.Update()	0.0%	0.0%	1	0 B	0.01	0.01
▶ GUI.Repaint	0.0%	0.0%	1	0 B	0.00	0.00
▶ Loading.UpdatePreloading	0.0%	0.0%	1	0 B	0.00	0.00
ProcessRemoteInput	0.0%	0.0%	1	0 B	0.00	0.00
MeshSkinning.Update	0.0%	0.0%	1	0 B	0.00	0.00
Network.Update	0.0%	0.0%	1	0 B	0.00	0.00
ParticleSystem.Update	0.0%	0.0%	1	0 B	0.00	0.00
Physics.Interpolation	0.0%	0.0%	2	0 B	0.00	0.00
ParticleSystem.WaitForUpdateThreads	0.0%	0.0%	1	0 B	0.00	0.00
Substance.Update	0.0%	0.0%	1	0 B	0.00	0.00

Figura 24 - Resultados da aplicação do script ArrayFunc_Ref.js.

Ainda existe a possibilidade da chamada manual do coletor de lixo através do comando *System.GC.Collect()*. Unity (2013i) indica para alguns casos de jogos o uso da chamada manual do coletor, em um período regular de intervalos de quadros, porém afirma também que se as boas práticas de programação forem seguidas, a coleta automática irá funcionar tão bem e até melhor do que sendo realizada manualmente.

4.2.1.2 CRIANDO UM POOL DE OBJETOS

Pool de objetos é uma técnica utilizada para simplificar o gerenciamento de memória, fazendo os programas serem executados de maneira suave, ou seja, sem pausas ou oscilações na taxa de quadros, causadas por instanciações ou destruições de objetos em tempo de execução[14]. Essa técnica consiste em instanciar um grupo de objetos no momento em que a cena é carregada e deixá-los em algum lugar na cena onde eles nunca serão visíveis, para que então, quando houver a necessidade de usar algum desses objetos, eles não precisarão ser instanciados ou destruídos após seu uso, mas apenas ter suas posições alteradas para a localização desejada e após, movidos novamente para a posição de origem, onde mais uma vez não serão visíveis e ficarão aguardando por uma nova chamada para serem usados de novo. Isso elimina a sobrecarga de alocação de memória dinâmica e a coleta de lixo (UNITY, 2013i).

Porém, a técnica do *pool* de objetos deve ser utilizada com cautela, pois a criação de um *pool* reduz a quantidade de memória *heap* disponível para outros propósitos, o que pode acabar ativando o coletor de lixo até mais vezes do que o necessário. Devido a esse fator, torna-se evidente que, caso seja alocado *pools* demasiadamente grandes ou mantê-los ativos quando os objetos que eles contêm não serão mais necessários por algum tempo, haverá perda de desempenho.

As figuras 25 e 26 comparam scripts na linguagem Javascript, a implementação de um simples projétil, um usando instanciação e o outro usando um *pool* de objetos.

[14] É interessante notar que, a técnica do *pool* de objetos já é utilizada há muito tempo atrás, desde as primeiras gerações de videogames, como por exemplo, nos jogos de Atari onde a quantidade de memória era mínima, o jogador não podia disparar vários tiros de uma vez, mas deveria aguardar o projétil acertar algum inimigo ou desaparecer de seu campo de visão, para conseguir atirar novamente.

```
// GunWithInstantiate.js

#pragma strict

var prefab : ProjectileWithInstantiate;

var power = 10.0;

function Update () {
  if(Input.GetButtonDown("Fire1")) {
    var instance : ProjectileWithInstantiate =
      Instantiate(prefab,
                  transform.position,
                  transform.rotation);

    instance.velocity = transform.forward *
                                        power;
  }
}
```

```
// GunWithObjectPooling.js

#pragma strict

var prefab : ProjectileWithObjectPooling;
var maximumInstanceCount = 10;
var power = 10.0;

private var instances : ProjectileWithObjectPooling[];

static var stackPosition = Vector3(-9999, -9999, -9999);

function Start () {
  instances = new ProjectileWithObjectPooling[
    maximumInstanceCount];
  for(var i = 0; i < maximumInstanceCount; i++) {
    instances[i] = Instantiate(prefab,
                               stackPosition,
                               Quaternion.identity);
    instances[i].enabled = false;
  }
}

function Update () {
  if(Input.GetButtonDown("Fire1")) {
    var instance : ProjectileWithObjectPooling =
                               GetNextAvailiableInstance();
    if(instance != null) {
      instance.Initialize(transform, power);
    }
  }
}

function GetNextAvailiableInstance () :
                                ProjectileWithObjectPooling {
  for(var i = 0; i < maximumInstanceCount; i++) {
    if(!instances[i].enabled) return instances[i];
  }
  return null;
}
```

Figura 25 - Comparação do script GunWithInstantiate.js que utiliza instanciação e o script GunWithObjectPooling.js que faz o uso de um *pool* de objetos (UNITY, 2013i).

```
// ProjectileWithInstantiate.js

#pragma strict

var gravity = 10.0;
var drag = 0.01;
var lifetime = 10.0;

var velocity : Vector3;

private var timer = 0.0;

function Update () {
  velocity -= velocity *
                drag * Time.deltaTime;

  velocity -= Vector3.up *
                gravity * Time.deltaTime;

  transform.position += velocity *
                        Time.deltaTime;

  timer += Time.deltaTime;

  if(timer > lifetime) {
     Destroy(gameObject);
  }
}
```

```
// ProjectileWithObjectPooling.js

#pragma strict

var gravity = 10.0;
var drag = 0.01;
var lifetime = 10.0;

var velocity : Vector3;

private var timer = 0.0;

function Initialize(parent : Transform, speed : float) {
  transform.position = parent.position;
  transform.rotation = parent.rotation;
  velocity = parent.forward * speed;
  timer = 0;
  enabled = true;
}

function Update () {
  velocity -= velocity * drag * Time.deltaTime;
  velocity -= Vector3.up * gravity * Time.deltaTime;
  transform.position += velocity * Time.deltaTime;

  timer += Time.deltaTime;
  if(timer > lifetime) {
     transform.position = GunWithObjectPooling.stackPosition;
     enabled = false;
  }
}
```

Figura 26 - Comparação do script ProjectileWithInstantiate.js que utiliza instanciação e o script ProjectileWithObjectPooling.js que faz o uso de um *pool* de objetos (UNITY, 2013i).

O script GunWithInstantiate.js instancia um novo projétil a cada vez que o jogador pressiona o botão de ação. O projétil instanciado é movimentado e destruído após um período de tempo pelo script ProjectileWithInstantiate.js. Já os scripts GunWithObjectPooling.js e ProjectileWithObjectPooling.js possuem a mesma finalidade, porém fazem uso de um *pool* de objetos, instanciando todos os projéteis necessários no momento em que a cena é carregada e desabilitando-os quando não estiverem em uso. Apenas os scripts GunWithInstantiate.js e GunWithObjectPooling.js são responsáveis pela alocação de memória no momento em que eles instanciam os projéteis, de acordo com cada algoritmo, portanto somente seus resultados serão analisados. Os scripts ProjectileWithInstantiate.js e ProjectileWithObjectPooling.js se encarregam apenas da movimentação dos projéteis e o gerenciamento dos mesmos na cena.

Na prática, o script GunWithInstatiate.js envia um total de 48 Bytes para o coletor de lixo a cada vez que um projétil é instanciado, quando o jogador pressiona o botão de ação, assim como demonstra a figura 27. Já o script GunWithObjectPooling.js possui apenas um esforço de alocação inicial, quando a função *Start* é executada, alocando um total de 480 Bytes, referente a quantidade máxima de objetos a serem enviados para o *pool*, e nada mais após isso, como demonstra a figura 28.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	57.7%	57.7%	1	0 B	7.39	7.39
▶ ProjectileWithInstantiate.Update()	18.9%	10.2%	581	40 B	2.42	1.31
Physics.Simulate	7.8%	7.8%	1	0 B	1.00	1.00
Overhead	6.6%	6.6%	1	0 B	0.85	0.85
▶ Camera.Render	6.1%	0.1%	1	0 B	0.79	0.02
▼ GunWithInstantiate.Update()	1.4%	0.3%	1	48 B	0.18	0.04
▶ Instantiate	0.5%	0.5%	1	48 B	0.07	0.07
Static Collider.Create (Expensive dela	0.5%	0.5%	1	0 B	0.06	0.06

Figura 27 - Resultados da aplicação do script GunWithInstantiate.js.

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	57.4%	57.4%	2	0 B	10.88	10.88
▶ Monobehaviour.OnMouse_	11.0%	0.0%	2	0.7 KB	2.09	0.00
▼ GunWithObjectPooling.Update()	6.4%	6.3%	2	40 B	1.23	1.19
Static Collider.Move (Expensive delayed	0.1%	0.1%	4	0 B	0.03	0.03
▼ GunWithObjectPooling.Start()	6.2%	3.7%	1	0.6 KB	1.18	0.71
▶ Instantiate	1.4%	1.4%	10	480 B	0.27	0.27
Static Collider.Create (Expensive delaye	1.0%	1.0%	10	0 B	0.19	0.19

Overview	Total	Self	Calls	GC Alloc	Time ms	Self ms
WaitForTargetFPS	96.9%	96.9%	1	0 B	14.65	14.65
Physics.Simulate	0.8%	0.8%	1	0 B	0.12	0.12
Overhead	0.8%	0.8%	1	0 B	0.12	0.12
▶ Camera.Render	0.5%	0.1%	1	0 B	0.08	0.01
▶ ProjectileWithObjectPooling.Update()	0.4%	0.2%	10	0 B	0.06	0.03
▶ Monobehaviour.OnMouse_	0.1%	0.0%	1	88 B	0.02	0.00
AudioManager.Update	0.0%	0.0%	1	0 B	0.01	0.01
HandleUtility.SetViewInfo()	0.0%	0.0%	1	0 B	0.00	0.00
▶ Cleanup Unused Cached Data	0.0%	0.0%	1	0 B	0.00	0.00
GunWithObjectPooling.Update()	0.0%	0.0%	1	0 B	0.00	0.00

Figura 28 - Resultados da aplicação do script GunWithObjectPooling.js.

5 CONCLUSÃO

A partir da pesquisa realizada, é possível identificar os aspectos que mais pesam no desempenho de um jogo, assim como a elaboração de um plano para a otimização dos mesmos, aplicando-se as técnicas necessárias para tal finalidade. O trabalho não funciona como uma receita ou tutorial para otimizações, pois o desenvolvimento de um jogo é algo extremamente complexo e cada tipo de jogo possui suas particularidades e necessidades específicas, mas funciona como um guia do que pode ser feito e de como isso pode ser feito para a obtenção de uma ótima performance em jogos desenvolvidos ou em desenvolvimento para dispositivos móveis com a Unity3D.

É notável a quantidade de áreas do conhecimento humano que um jogo envolve. Muitos dos temas tratados nesse trabalho dariam por si só, uma única pesquisa científica, tamanha grandeza e complexidade desses assuntos. Em sua maioria, os assuntos abordados nessa pesquisa se referem á conceitos relacionados a computação gráfica, servindo também como uma introdução á essa área.

Com base nas pesquisas torna-se nítido o fato de que todas as técnicas aqui apresentadas irão contribuir para uma melhora no desempenho de um jogo, porém nem todas elas podem ser necessárias ou suficientes. Tudo irá depender de diversos fatores como o tipo do jogo que se deseja criar e o hardware para o qual se deseja exportar.

Como trabalhos futuros, evidencio a necessidade de pesquisas sobre quais técnicas podem ser utilizadas na otimização de jogos para outras plataformas com a Unity3D e até mesmo na otimização de jogos para dispositivos móveis utilizando outras *engines* como a Unreal.

6 REFERÊNCIAS BIBLIOGRÁFICAS

APPLE. Using texturetool to Compress Textures. 2013. Disponível em: <http://developer.apple.com/library/ios/#documentation/3DDrawing/Conceptual/OpenGLES_ProgrammingGuide/TextureTool/TextureTool.html> Acesso em: 03 Maio 2013.

BRATCHER, Daniel. Unity games sweep App Store awards, 2013. Disponível em: <http://blogs.unity3d.com/2013/02/19/unity-games-sweep-app-store-awards/> Acesso em: 10 Março 2013.

DALMAZO, Luiza; FERRARI, Bruno. Os jogos online passam de fase com os smartphones. 2013. Disponível em: <http://exame.abril.com.br/revista-exame/edicoes/1038/noticias/os-jogos-online-passam-de-fase> Acesso em: 30 Maio 2013.

GDMAG. Mobile game developer survey leans heavily toward iOS, Unity, 2012. Disponível em: <http://www.gamasutra.com/view/news/169846/Mobile_game_developer_survey_leans_heavily_toward_iOS_Unity.php#.UTvvPBxwps4> Acesso em: 09 Março 2013.

HAAPASALO, Jaakko. Unraveling the unending oink that is Rovio's Bad Piggies with the studio's Jaakko Haapasalo, 2012. Disponível em <http://unity3d.com/gallery/made-with-unity/profiles/profile#rovio-badpiggies> Acesso em: 16 Março 2013.

HEINEMAN, Becky. Sponsored Feature: Common Performance Issues in Game Programming, 2008. Disponível em <http://www.gamasutra.com/view/feature/3687/sponsored_feature_common_.php> Acesso em: 17 Março 2013.

IDC. IDC Forecasts Worldwide Tablet Shipments to Surpass Portable PC Shipments in 2013, Total PC Shipments in 2015. 2013. Disponível em: <http://blogs.estadao.com.br/link/tablets-continuam-crescendo-e-pcs-caindo/> Acesso em: 31 Maio 2013.

LUZ, Mairlo Hideyoshi Guibo Carneiro da. Desenvolvimento de Jogos de Computador, 2004. Disponível em: <http://www.programadoresdejogos.com/trab_academicos/marlo_luiz.pdf> Acesso em 07 Abril 2013.

MADFINGER. ShadowGun: Optimizing for Mobile Sample Level, 2012. Disponível em: <http://blogs.unity3d.com/2012/03/23/shadowgun-optimizing-for-mobile-sample-level/> Acesso em: 05 Maio 2013.

MCDERMOTT, Wes. Criando Arte de Jogos 3D para Iphone com Unity: usando modo e blender na linha de produção. Rio de Janeiro: Elsevier, 2012.

MDN. Mozilla Developer Network: Javascript Glossary, 2013. Disponível em: < https://developer.mozilla.org/en-US/docs/Web/JavaScript/Glossary> Acesso em: 22 Junho 2013.

MICROSOFT. StringBuilder Class. 2013. Disponível em: < http://msdn.microsoft.com/en-gb/library/system.text.stringbuilder.aspx> Acesso em: 25 Maio 2013.

PRESSEISEN, Josh. Crescent Moon Games ups the iOS ante with its open-world RPG, Ravensword: Shadowlands, 2013. Disponível em <http://unity3d.com/gallery/made-with-unity/profiles/profile#crescent-ravensword> Acesso em: 16 Março 2013.

RABAS, Marek. Czech developer MADFINGER Games stretches the limits of mobile development, 2011. Disponível em: <http://unity3d.com/gallery/made-with-unity/profiles/profile#madfinger-shadowgun> Acesso em: 17 Março 2013.

TELES, Giovana. Governo diminui impostos para baratear o custo dos smartphones. 2013. Disponível em: <http://g1.globo.com/jornal-hoje/noticia/2013/04/governo-diminui-impostos-para-baratear-o-custo-dos-smartphones.html> Acesso em: 30 Maio 2013.

UNITY. Create the games you love with Unity. 2013. Disponível em: <http://unity3d.com/unity/> Acesso em: 09 Março 2013.

UNITY. Vision: Democratize game development and enable everyone to create rich interactive 3D content. 2013a. Disponível em: <http://unity3d.com/company/public-relations/> Acesso em: 09 Março 2013.

UNITY. Measuring Performance with the Built-in Profiler. 2013b. Disponível em: <http://docs.unity3d.com/Documentation/Manual/iphone-InternalProfiler.html> Acesso em: 31 Março 2013.

UNITY. Optimizing Graphics Performance. 2013c. Disponível em: <http://docs.unity3d.com/Documentation/Manual/OptimizingGraphicsPerformance.html> Acesso em: 20 Abril 2013.

UNITY. Texture 2D. 2013d. Disponível em: <http://docs.unity3d.com/Documentation/Manual/Textures.html> Acesso em: 03 Maio 2013.

UNITY. Getting Started with Android Development. 2013e. Disponível em: <http://docs.unity3d.com/Documentation/Manual/android-GettingStarted.html> Acesso em: 03 Maio 2013.

UNITY. Shaders. 2013f. Disponível em: <http://docs.unity3d.com/Documentation/Manual/Shaders.html> Acesso em: 04 Maio 2013.

UNITY. Performance of Unity shaders. 2013g. Disponível em: <http://docs.unity3d.com/Documentation/Components/shader-Performance.html> Acesso em: 04 Maio 2013.

UNITY, Practical Guide to Optimization for Mobiles - Optimizing Scripts. 2013h. Disponível em: <http://docs.unity3d.com/Documentation/Manual/iphone-PracticalScriptingOptimizations.html> Acesso em: 19 Maio 2013.

UNITY, Understanding Automatic Memory Management. 2013i. Disponível em: <http://docs.unity3d.com/Documentation/Manual/UnderstandingAutomaticMemoryManagement.html> Acesso em: 19 Maio 2013.