UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

FELIPE ARAÚJO DE ANDRADE

Aplicação de *decals* sobre superfícies geométricas arbitrárias

Trabalho de Graduação.

Prof. Dr. João Luiz Dihl Comba
Orientador

Porto Alegre, dezembro de 2011.

“What is real? How do you define real? If you're talking about what you can feel, what you can smell, what you can taste and see, then real is simply electrical signals interpreted by your brain.”

– MORPHEUS (THE MATRIX)

SUMÁRIO

LISTA DE ABREVIATURAS E SIGLAS 6

LISTA DE FIGURAS 7

LISTA DE TABELAS 8

RESUMO 9

ABSTRACT 10

1 INTRODUÇÃO 11

1.1 Contexto e Motivação 11

1.2 Organização do Texto 12

2 TRABALHOS RELACIONADOS 13

2.1 Aplicação de *decals* a superfícies arbitrárias 13

2.2 Como projetar *decals* 14

3 IMPLEMENTAÇÃO INICIAL 17

3.1 Visão geral do algoritmo proposto inicialmente 17

3.2 Possíveis melhorias para esta técnica 19

3.3 Observações e conclusões sobre a técnica apresentada 20

4 ALGORITMO PARA APLICAÇÃO DE *DECALS* SOBRE SUPERFÍCIES GEOMÉTRICAS ARBITRÁRIAS 21

4.1 Criação do projetor 21

4.2 Obtenção dos vértices e remoção das faces ocultas 22

4.3 Projeção e recorte em 2D 23

4.4 Coordenadas UV e projeção para 3D 28

4.5 Criação da malha final 29
4.5.1 Vértices 29
4.5.2 Normais 30
4.5.3 Triângulos 31
4.5.4 Aplicando o material e renderizando a malha 31

5 OTIMIZAÇÕES 33

5.1 Criação de um *shader* personalizado 33

5.2 Mudanças nas estruturas de armazenamentos das faces 33

6 RESULTADOS E ANÁLISE 35

6.1 Resultados visuais 35
6.1.1 Aplicação de *decals* sobre primitivas geométricas 35
6.1.2 Aplicação de *decals* sobre superfícies mais complexas 36

6.2 Análise de desempenho 38
6.2.1 Ambiente de medições 38
6.2.2 Análise do desempenho da aplicação de *decals* sobre primitivas geométricas 38
6.2.3 Análise do desempenho da aplicação de *decals* sobre superfícies complexas 40

7 CONCLUSÕES 41

APÊNDICE A ALGORITMO DO SISTEMA DE *DECALS* 43

APÊNDICE B SHADER CUSTOMIZADO 50

LISTA DE ABREVIATURAS E SIGLAS

API	*Application Programming Interface*
Cg	*C for Graphics*
CPU	*Central Processing Unity*
DDR	*Double Data Rate*
FPS	*First Person Shooter*
GB	*Giga Bytes*
GLSL	*OpenGL Shading Language*
GPU	*Graphics Processing Unity*
HD	*Hard Disk*
HLSL	*High Level Shading Language*
OpenGL ARB	*OpenGL Architecture Review Board*
RAM	*Random Access Memory*

LISTA DE FIGURAS

Figura 2.1: Configuração de um decal sobre uma superfície qualquer *13*
Figura 2.2: Verificação de quais triângulos da cena intersectam com o projetor *15*
Figura 2.3: Exemplo de textura (decal) aplicada sobre o objeto atingido *15*
Figura 3.1: Direções de busca para verificar a incidência do decal sobre a superfície atingida *18*
Figura 3.2: Malha resultante após o recorte do decal contra os limites da parede atingida *19*
Figura 3.3: Problema da busca por pontos sobre a superfície em apenas quatro direções *19*
Figura 3.4: Exemplo de malha final de um decal com busca radial a cada 45º *20*
Figura 4.1: Projetor instanciado e orientado sobre a superfície atingida *21*
Figura 4.2: Vértices do objeto atingido após a remoção das faces ocultas *23*
Figura 4.3: Problema da utilização do algoritmo Cohen-Sutherland para recorte de faces *24*
Figura 4.4: Ponto de intersecção com a reta y = 5 ..... *26*
Figura 4.5: Exemplo de caso em que para os 3 vértices iniciais são gerados 6 novos . *26*
Figura 4.6: Novos triângulos gerados utilizando triangle fan *27*
Figura 4.7: Vértices resultantes após o recorte com Sutherland-Hodgman *27*
Figura 4.8: Coordenadas UV de uma imagem *28*
Figura 4.9: Transformações das coordenadas locais em coordenadas UV *29*
Figura 4.10: No ponto P encontram-se 3 vértices, cada um com a sua respectiva normal *30*
Figura 4.11: Malha final renderizada sobre a superfície *31*
Figura 5.1: Modificações feitas na estrutura que armazena as faces *34*
Figura 5.2: Laço principal do método de remoção das faces ocultas antes das alterações *34*
Figura 5.3: Laço principal do método de remoção das faces ocultas após as alterações *34*
Figura 6.1: Resultados da aplicação de decals sobre cubos. *35*
Figura 6.2: Resultados da aplicação de decals sobre cilindros *36*
Figura 6.3: Resultados da aplicação de decals sobre esferas *36*
Figura 6.4: Resultados da aplicação de decals sobre terrenos com malhas irregulares *36*
Figura 6.5: Resultados da aplicação de decals sobre a malha de uma casa com grande número de polígonos *37*
Figura 6.6: Resultados da aplicação de decals sobre um canto da malha da casa *37*
Figura 6.7: Resultados da aplicação de decals sobre o telhado da casa *38*

LISTA DE TABELAS

Tabela 6.1: Medidas de tempo para as primitivas antes da otimização 39
Tabela 6.2: Medidas de tempos para as primitivas após as otimizações 39
Tabela 6.3: Medidas de tempo para a aplicação de decals sobre diferentes terrenos .. 40
Tabela 6.4: Medidas de tempo para a aplicação de decals sobre diferentes partes da malha da casa apresentada na Seção 6.1.2 40

RESUMO

O mapeamento de texturas e – mais especificamente – a utilização de *decals* para aplicação de texturas em tempo de execução, é uma técnica extremamente utilizada na maioria dos jogos 3D atuais. Contudo, a bibliografia referente ao assunto e as ferramentas para a sua aplicação ainda são bastante limitadas. A Unity 3D, líder do mercado de *engines* de jogos, é um exemplo de ferramenta que não possui um sistema nativo para utilização de *decals*. Este trabalho apresentará duas técnicas para aplicação de *decals* em superfícies arbitrárias implementadas dentro do ambiente da Unity 3D, permitindo que este tipo de recurso seja utilizado em jogos e demais aplicativos que venham a ser criados com essa *engine*.

Palavras-Chave: *decals*, mapeamento de texturas, computação gráfica, Unity 3D, jogos

Applying decals to arbitrary geometric surfaces

ABSTRACT

Texture mapping and – more specifically – the use of decals for applying textures during runtime is an extremely used technique in most of the current 3D games. However, the bibliography concerning this subject and the tools for its application are still quite limited. Unity 3D, the market leader in game engines, is an example of a software tool that doesn't have a native system for the utilization of decals. This work presents two different techniques for the application of decals to arbitrary surfaces implemented within the Unity 3D's environment, thus allowing this feature to be used in games and other applications created using this engine.

Keywords: decals, texture mapping, computer graphics, Unity 3D, games.

1 INTRODUÇÃO

A aplicação de texturas para representação de detalhes em superfícies em computação gráfica é uma técnica bastante utilizada desde a primeira vez em que foi mencionada, na tese de doutorado de Edwin Catmull, em 1974 [CATMULL74]. Através do mapeamento de texturas é possível obter superfícies com um alto número de detalhes sem a necessidade de se adicionar uma maior complexidade geométrica. Entre as diversas modalidades de mapeamento de texturas utilizadas hoje em dia, está a de *decals* (ou decalques). Um *decal* é um objeto que coincide com uma superfície existente e exibe uma região com algum padrão diferente sobre ela. Os *decals* costumam ser implementados por meio de uma pequena textura e um conjunto de dados, como: posição, orientação e tamanho. Eles são altamente utilizados em jogos para exibir, por exemplo, buracos de bala sobre uma parede ou mesmo pegadas, pôsters, pixações e outros detalhes sobre as texturas originais, aplicados em tempo de execução.

Em principio, a aplicação de *decals* sobre uma superfície planar é algo relativamente simples, porém o processo é mais complicado quando se deseja aplicar *decals* a superfícies mais complexas, como objetos irregulares e terrenos variados, o que acontece em muitos dos jogos de hoje em dia. Este trabalho apresentará um método para aplicar esses detalhes (*decals*) sobre superfícies geométricas arbitrárias e ao mesmo tempo fazer o recorte do *decal* aos limites da superfície, dentro da *engine* de jogos *Unity 3D*.

1.1 Contexto e Motivação

A *Unity* é uma ferramenta desenvolvida pela empresa dinamarquesa *Unity Technologies*, e está sendo cada vez mais utilizada para a criação de visualizações arquitetônicas, animações 3D em tempo real e – especialmente – para o desenvolvimento de jogos para diversas plataformas.

Ela oferece uma ampla variedade de recursos para a criação dos mais variados tipos de jogos, porém não possui um sistema de criação e aplicação de *decals* nativo. Sendo assim, o objetivo deste trabalho será implementar um método de geração de *decals* otimizado para esta ferramenta, de modo que este recurso possa ser utilizado em jogos que venham a ser criados com ela.

1.2 Organização do Texto

Este trabalho encontra-se dividido em 7 capítulos, distribuídos da seguinte forma:

O Capítulo 2 apresenta dois trabalhos relacionados ao tema da aplicação de *decals* sobre superfícies arbitrárias. As técnicas são brevemente descritas e também comparadas, indicando as diferenças e semelhanças existentes entre elas.

No Capítulo 3, apresenta-se uma implementação inicial (relativamente simples) para a geração de *decals*. Após detalhado o seu passo-a-passo, são discutidos os prós e contras da utilização dessa técnica.

O Capítulo 4 apresenta o algoritmo proposto neste trabalho para a aplicação de *decals* sobre superfícies geográficas arbitrárias. Ele é descrito em detalhes e também são ilustradas várias das etapas do processo de sua implementação.

No Capítulo 5 são sugeridas algumas otimizações para o algoritmo apresentado no capítulo anterior, visando melhorá-lo tanto na questão de desempenho quanto visualmente.

O Capítulo 6 apresenta diversas imagens dos resultados obtidos, além de fazer uma análise de desempenho da aplicação do *decal* em várias superfícies distintas.

Finalmente, no Capítulo 7 são feitas as conclusões e considerações finais com respeito ao trabalho apresentado, além de serem mencionadas algumas sugestões para trabalhos futuros.

2 TRABALHOS RELACIONADOS

Embora o conceito de *decals* já exista há algum tempo, a bibliografia referente a eles é ainda bastante limitada, especialmente se comparada a outros conceitos de computação gráfica. Entre os principais artigos referentes ao assunto, poderiam-se destacar *Applying decals to arbitrary surfaces* [LENGYEL01] e *How to project decals* [ROSEN09]. Apesar desses dois trabalhos citados tratarem do mesmo tema (*decals*), cada um apresenta uma abordagem e uma sugestão de implementação diferente. A escolha da técnica a ser utilizada depende do foco do trabalho e do objetivo proposto, levando-se em consideração as vantagens e desvantagens de cada uma para os fins pretendidos.

2.1 Aplicação de *decals* a superfícies arbitrárias [LENGYEL01]

Neste artigo, presente no livro *Game Programming Gems 2*, Eric Lengyel apresenta um algoritmo geral para a aplicação de um *decal* a uma superfície arbitrária e para a realização de recorte com relação aos limites da superfície.

O algoritmo apresentado para a aplicação do *decal* consiste no seguinte: inicia-se com um ponto P sobre uma superfície existente e um vetor direção normal unitário N, perpendicular à superfície nesse ponto. O ponto P representa o centro do *decal* (podendo ser o ponto em que um projétil atingiu a superfície ou mesmo o ponto central sobre o chão de uma pegada de um jogador, por exemplo). Uma direção tangente unitária T também deve ser escolhida para que possa ser determinada a orientação do *decal*, além de uma direção ortogonal tanto a N quanto a T, que é calculada através do produto vetorial entre N e T (N x T) é chamada de bi-normal e representada pela letra B, conforme ilustrado na Figura 2.1.

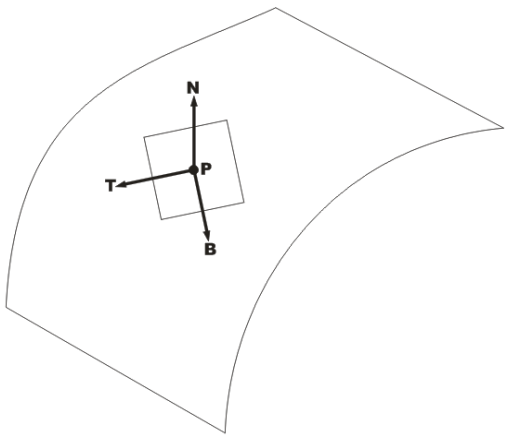

Figura 2.1: Configuração de um *decal* sobre uma superfície qualquer [LENGYEL01]

Como pode ser visto, dado o ponto P e as direções N e T, tem-se um plano orientado que é tangente à geometria da superfície no ponto P. A partir desse plano pode-se "talhar" o retângulo correspondente à área do *decal*, construindo quatro planos limítrofes paralelos à direção normal N. Sendo w e h a largura e a altura, respectivamente, do *decal*; temos então que os vetores 4D correspondentes aos 4 planos limítrofes são dados por:

Esquerdo: $\left(T, \frac{w}{2} - T \bullet P\right)$

Direito: $\left(-T, \frac{w}{2} + T \bullet P\right)$

Inferior: $\left(B, \frac{h}{2} - B \bullet P\right)$

Superior: $\left(-B, \frac{h}{2} + B \bullet P\right)$

Nas equações acima, são definidos quatro planos, cada um através de um ponto e um vetor normal. O próximo passo é a geração da malha de triângulos (um tipo de malha poligonal composta por um conjunto de triângulos conectados por seus lados ou vértices comuns) para o objeto do *decal*, ao realizar o recorte das superfícies próximas com esses 4 planos. Além disso, também é realizado o recorte com um plano frontal e outro traseiro, para evitar o "vazamento" para partes da mesma malha de superfície que podem estar dentro da parte delimitada pelos planos limítrofes, mas muito mais à frente ou atrás do ponto P. Assim como nos planos laterais, são dados os vetores normais e um ponto para definir cada um dos planos:

Plano frontal: $(-N, d + N \bullet P)$

Plano traseiro: $(N, d - N \bullet P)$

Sendo d a distância máxima que qualquer vértice do *decal* poderá estar do plano tangente que passa pelo ponto P.

Uma vez definidos esses 6 planos, o passo seguinte é identificar quais superfícies na cena podem ser afetadas pelo *decal*. Para cada uma dessas superfícies potencialmente afetadas deve-se analisar individualmente cada um dos triângulos em sua malha, para que se possa realizar o recorte desses triângulos em relação aos planos definidos acima. As coordenadas de mapeamento de textura são aplicadas à malha de triângulos resultante ao medir-se a distância dos planos passando pelo ponto P e tendo as direções normais T e B.

2.2 Como projetar *decals* [ROSEN09]

Ao contrário do trabalho apresentado anteriormente, este não se encontra em uma publicação oficial ou artigo científico, mas sim no *blog* de seu autor, David Rosen. Porém, trata-se de uma técnica para a aplicação de *decals* sobre superfícies arbitrárias extremamente otimizada, produzindo bons resultados com um processamento relativamente pequeno.

A principal diferença desta técnica com a apresentada em [LENGYEL01] é que aqui o autor transporta os triângulos do objeto afetado pelo *decal* ao espaço de projeção (2D) para realizar o recorte. Isso otimiza significativamente o processamento visto que realizar o recorte de faces contra retas em 2D é muito mais simples do que realizar o mesmo processo em um ambiente 3D, no qual tem-se a necessidade de definir 6 planos e realizar todo o recorte das faces no espaço tridimensional.

O algoritmo inicia definindo o chamado "projetor do *decal*" (que nada mais é do que um prisma retangular 3D que possui três propriedades: tamanho, posição e orientação) e verificando quais triângulos da cena intersectam este projetor.

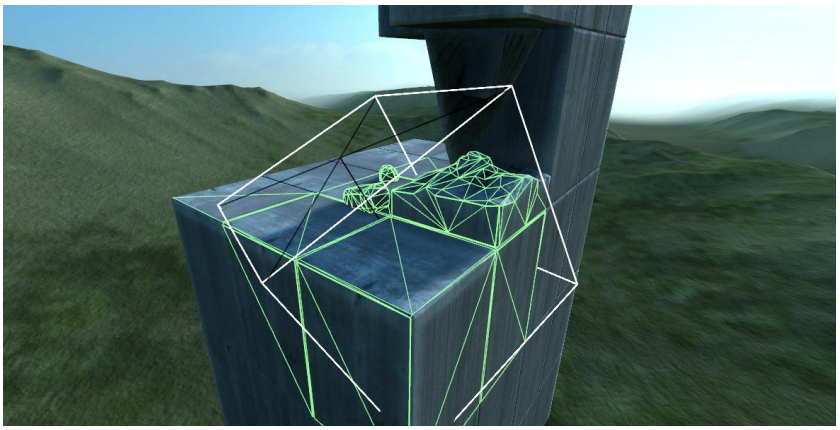

Figura 2.2: Verificação de quais triângulos da cena intersectam com o projetor [ROSEN09]

O passo seguinte consiste justamente em levar esses triângulos do espaço do universo para o espaço de projeção, no qual é realizado o recorte contra as linhas do projetor antes de transformar os triângulos novamente ao espaço do universo. Feito isso, tem-se já os vértices definitivos para a malha do *decal*. Além disso, ao passar os triângulos para o espaço de projeção e realizar o recorte tendo como referência o projetor com coordenadas locais que vão de (0,0) a (1,1), já se tem também as coordenadas UV correspondentes para cada vértice, bastando aplicar a textura sobre a malha criada para ter o *decal* devidamente aplicado sobre a superfície desejada.

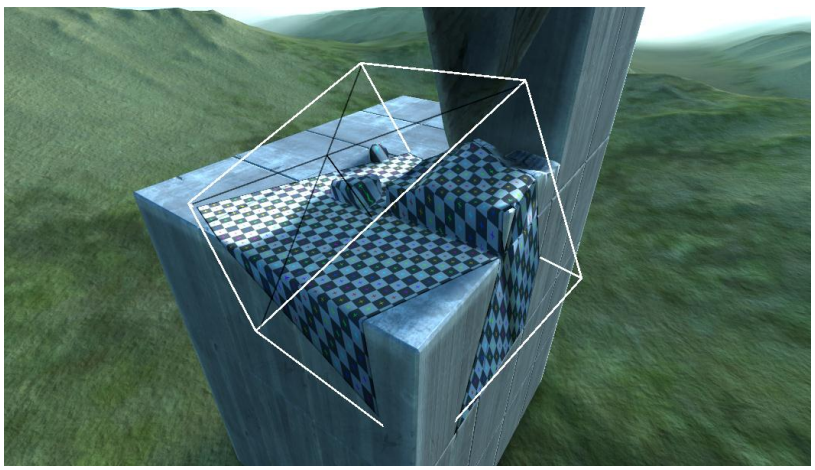

Figura 2.3: Exemplo de textura (*decal*) aplicada sobre o objeto atingido [ROSEN09]

3 IMPLEMENTAÇÃO INICIAL

O algoritmo de aplicação de *decals* desenvolvido para este trabalho baseia-se principalmente em [ROSEN09], utilizando um projetor para definir o tamanho, a posição e a orientação do *decal*, levando os triângulos afetados da cena para realizar o recorte no espaço de projeção, para depois transformá-los novamente ao espaço do universo.

Antes, porém, realizou-se uma implementação mais básica, muito útil e otimizada para casos em que o *decal* é relativamente pequeno (como buracos de bala, por exemplo), porém não viável para quando se deseje aplicar uma textura maior ou ter uma maior precisão sobre a superfície atingida. De qualquer forma, visto que há casos em que este tipo de implementação pode ser mais vantajosa (visando-se ganho em velocidade à custa de uma maior precisão) ela também será descrita aqui.

3.1 Visão geral do algoritmo proposto inicialmente

A ideia central desta abordagem do problema é a geração de um plano (que, dentro da *Unity*, consiste em um retângulo cujas únicas faces visíveis são aquelas com a mesma orientação de sua normal), redimensionado ao tamanho desejado para o *decal*, que é aplicado no ponto exato em que a superfície foi atingida e com a mesma orientação da face afetada. Uma vez aplicado esse plano, é feita uma verificação em quatro direções previamente definidas, para ver se os extremos da textura encontram-se sobre a superfície. No caso de ao menos um ponto não estar, é calculado um novo ponto de recorte para posterior geração de uma malha, a qual substituirá o plano inicial e conterá o *decal*.

O algoritmo inicia encontrando o ponto do espaço em que o *decal* deverá ser aplicado. Este ponto geralmente corresponde ao centro da textura final do *decal*. Dado esse ponto (e sua normal), passa-se à criação do plano que conterá inicialmente essa textura. Esse plano é então posicionado nesse ponto com a referida normal, porém com um deslocamento mínimo (e visualmente imperceptível) no sentido dessa normal, para evitar problemas como *z-fighting* sobre a superfície em que foi aplicado.

O passo seguinte é a aplicação do material do *decal* sobre esse plano, material esse que contém a textura correspondente ao *decal* e o *shader* implementado para ele. Uma vez atribuído o material apropriado, é redimensionado o plano que o contém para o tamanho adequado (de acordo com a textura em questão).

O cerne da aplicação de *decals* sobre superfícies arbitrárias está na realização do recorte naquelas partes da textura que excederem os limites da superfície atingida. Para este algoritmo, é feita uma verificação em quatro direções perpendiculares, conforme ilustrado na Figura 3.1.

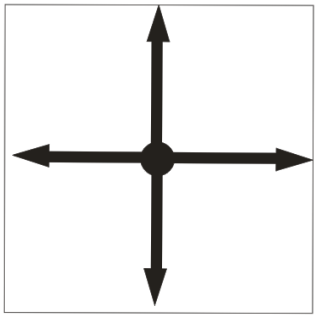

Figura 3.1: Direções de busca para verificar a incidência do *decal* sobre a superfície atingida

Em cada uma dessas direções, é realizada uma busca binária para verificar se o ponto está sobre a superfície em que o *decal* foi aplicado. Essa busca consiste no seguinte: passa-se um vetor direção *v* e uma determinada distância *d*, que inicialmente corresponde à metade da distância do centro do decal até a sua extremidade. Se esse ponto não estiver sobre a superfície, inverte-se a direção de busca e usa-se *d/2*. Caso esteja, a direção de busca permanece a mesma, e é feita uma nova verificação a *d/2* unidades desse ponto. É especificado um número máximo de passos de busca para não gerar sobrecarga de processamento. Esse número máximo de passos deve ser grande o suficiente para produzir um resultado razoável (de aproximação com os limites da superfície), porém pequeno o suficiente para não gerar muito *overhead*.

Realizada a busca nas quatro direções previamente definidas, tem-se duas possíveis situações: todos os quatro extremos da textura encontravam-se sobre a superfície desejada (caso ótimo) ou em ao menos em uma dessas quatro direções houve a necessidade de se aproximar mais do limite da superfície (visto que a textura estava excedendo a área em que deveria ser aplicada). No primeiro caso, temos o caso ótimo, em que não há necessidade de se realizar qualquer outro tipo de processamento, visto que o *decal* já está completamente sobre a superfície em que deveria ser aplicado. Contudo, caso tenha ocorrido a necessidade de recalcular ao menos um ponto de recorte para o *decal*, é necessário ajustar a textura para a nova área a ser ocupada.

Ao invés de realizar um recorte sobre o plano já existente, a solução adotada nesta implementação foi a de gerar uma nova malha, limitada pelos quatro pontos obtidos (isto é, os extremos do *decal* nas direções especificadas ou o novo ponto calculado através da busca binária). Para a criação de uma nova malha basta especificar os seus vértices, suas normais, seus triângulos e as suas coordenadas UV. Os vértices correspondem aos quatro pontos obtidos no passo anterior, os triângulos são definidos a partir desses vértices (no sentido horário, para ter as normais das faces corretas) e as normais de cada um desses vértices são iguais à do ponto de impacto do *decal*, que já foi armazenada previamente. Para a definição das coordenadas UV, é preciso calcular a distância dos novos pontos para o centro da textura do *decal*, para assim poder definir o deslocamento proporcional adequado. Definidos esses atributos, a nova malha está pronta, e basta destruir o plano criado anteriormente para termos um *decal* que se ajusta (aproximadamente) à superfície em que foi aplicado.

Esta é uma solução relativamente simples para o problema, e que produz um resultado final satisfatório na maioria dos casos (ver Figura 3.2). Porém, caso deseje-se aplicar o *decal* sobre uma superfície cujos extremos não são paralelos aos planos

cartesianos, começam a ser percebidas as principais falhas desta técnica. Visto que a busca é feita apenas naquelas quatro direções (para cima, para baixo, para esquerda e para a direita), para casos em que a superfície não possui limites paralelos aos eixos de referência poderíamos ter efeitos não muito agradáveis, como o ilustrado na Figura 3.3.

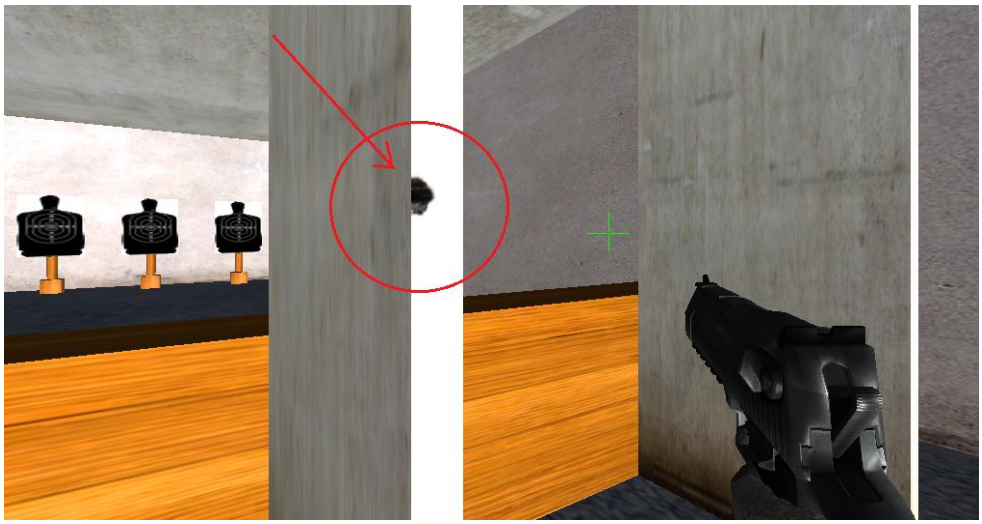

Figura 3.2: Malha resultante após o recorte do *decal* contra os limites da parede atingida

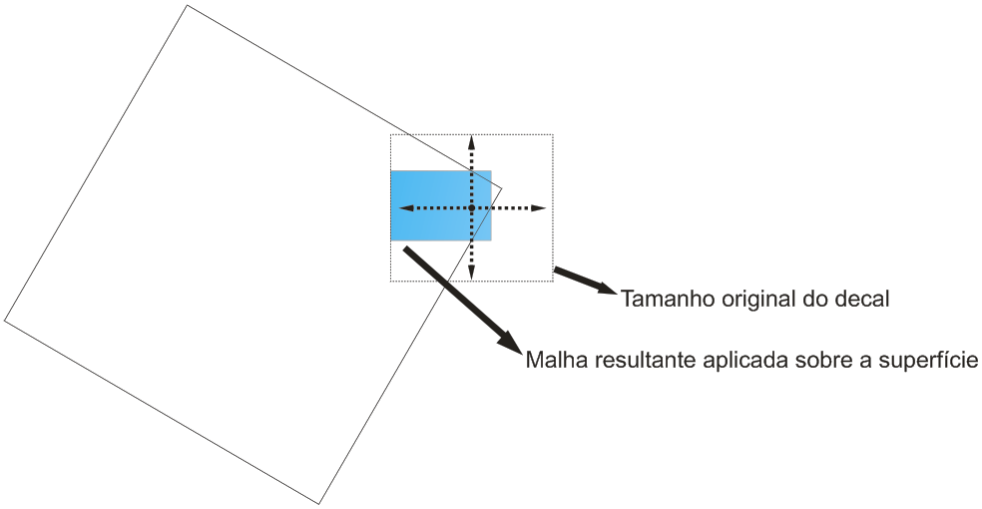

Figura 3.3: Problema da busca por pontos sobre a superfície em apenas quatro direções

3.2 Possíveis melhorias para esta técnica

Como citado anteriormente, um dos principais problemas deste algoritmo é que, ao fazer a busca em apenas quatro direções, podem-se obter resultados não muito precisos no momento de realizar o recorte do *decal*. Uma possível melhoria para esta técnica seria a de não limitar em quatro direções previamente definidas, mas sim especificar (da mesma forma que é especificado o número máximo de passos para a busca binária) um determinado número de direções de busca, para a realização de buscas radiais em determinadas direções sobre a superfície. Por exemplo, especificando o número de direções de busca para 8, teria-se o cálculo de contato da superfície do *decal* a cada 45º.

Como pode ser observado na Figura 3.4, com 8 direções de busca já se tem uma malha bastante mais ajustada à superfície atingida do que com as 4 iniciais, e essa precisão pode ser ainda maior aumentando-se o número de direções de busca.

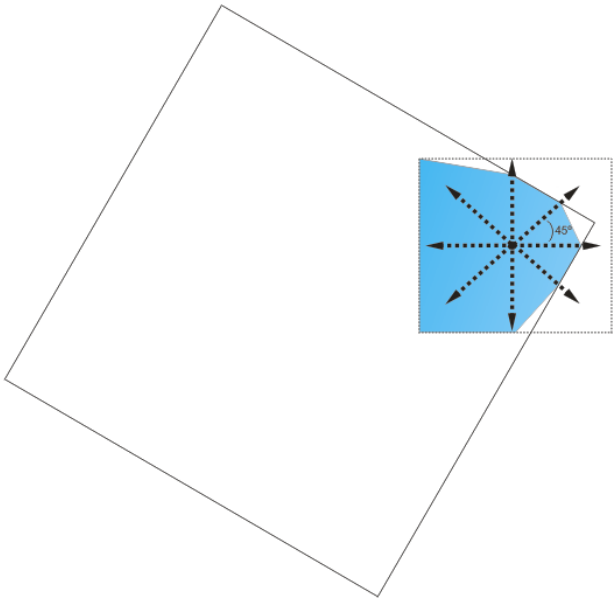

Figura 3.4: Exemplo de malha final de um *decal* com busca radial a cada 45º

Uma outra possível melhoria para esta técnica seria a de não apenas verificar se as extremidades estão sobre alguma superfície qualquer (como é feito), mas sim se estão sobre uma superfície que compartilhe o mesmo material daquela em que está o ponto atingido. Isso é útil para evitar exibir um *decal* de determinado material sobre um outro diferente, como uma textura de concreto sobre um objeto de madeira, por exemplo.

3.3 Observações e conclusões sobre a técnica apresentada

Conforme já dito e visto, o algoritmo apresentado acima representa uma solução válida para a aplicação de *decals* sobre superfícies arbitrárias, embora não seja ideal para todas as situações. Entre suas principais vantagens destacam-se a sua velocidade e a sua facilidade de implementação. Além disso, este algoritmo gera uma malha que não ficará necessariamente apenas sobre o objeto atingido, mas poderá ficar sobre algum outro que esteja suficientemente próximo a este, o que não ocorre com outras técnicas.

Para problemas simples (como a exibição de buracos de balas, por exemplo, que são texturas pequenas e que não precisam estar totalmente "coladas" à superfície atingida) esse algoritmo é altamente viável. Porém, na medida em que se deseje ter um sistema mais robusto, que realmente "molde" os *decals* ao objeto atingido (independentemente de sua geometria e do tamanho do *decal*) se faz necessária uma técnica mais elaborada, como a que foi desenvolvida neste projeto e será apresentada a seguir.

De qualquer forma, mesmo não sendo a solução ideal para a maioria dos casos, uma versão levemente modificada deste algoritmo foi utilizada para o jogo *Quantum Conflict*, um FPS *multiplayer* feito pela empresa porto-alegrense de desenvolvimento de jogos *Aquiris Game Studio*, e que possui previsão de lançamento oficial para o primeiro trimestre de 2012.

4 ALGORITMO PARA APLICAÇÃO DE *DECALS* SOBRE SUPERFÍCIES GEOMÉTRICAS ARBITRÁRIAS

Para solucionar o problema da geração e aplicação de *decals* sobre superfícies geométricas arbitrárias dentro da *engine* Unity 3D, optou-se por gerar um algoritmo seguindo a ideia básica apresentada por David Rosen em [ROSEN09] (ver Apêndice A). Conforme já descrito na Seção 2.2, parte-se da utilização de um objeto chamado "projetor" (que definirá o tamanho, a posição e a orientação do *decal*) para verificar quais faces do objeto atingido precisam ser recortadas para a geração da nova malha. Para realizar este recorte, são levados todos os vértices para o espaço de projeção (2D) e retornados ao espaço 3D após realizado o cálculo do recorte e das coordenadas UV que serão utilizadas para essa nova malha (do *decal*).

4.1 Criação do projetor

A classe criada para a implementação do sistema de *decals* deve possuir um método base que permita gerar um *decal* partindo apenas de um determinado conjunto limitado de dados (ponto de aplicação, orientação, objeto atingido, material e escala), de modo que outras informações (tais como a forma com que a orientação é definida ou como é obtido o objeto atingido) estão fora de seu escopo. Sendo assim, o primeiro passo deste algoritmo é gerar o projetor do *decal* a partir dos parâmetros recebidos.

O projetor nada mais é do que um cubo cuja escala em X e Y equivale à desejada para o *decal* e que é instanciado a uma certa distância do ponto de impacto, com a orientação (ângulo em que está em relação ao objeto atingido) especificada previamente. A Figura 4.1 ilustra um projetor instanciado sobre a superfície em que se deseja aplicar um novo *decal*.

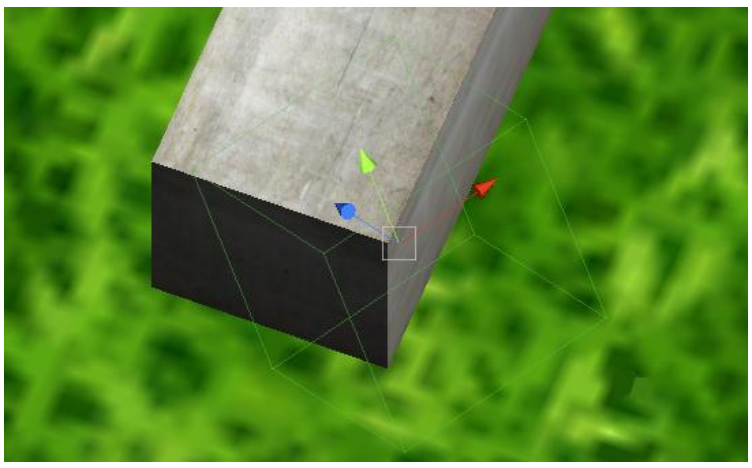

Figura 4.1: Projetor instanciado e orientado sobre a superfície atingida

4.2 Obtenção dos vértices e remoção das faces ocultas

O objeto sobre o qual deseja-se aplicar o *decal* é passado como parâmetro pois a partir dele obtêm-se as coordenadas dos vértices de todas as suas faces. Essas coordenadas estão inicialmente como coordenadas locais ao objeto, e precisam ser transformadas para coordenadas do universo antes de prosseguir.

A transformação entre diferentes sistemas de coordenadas é uma operação bastante comum em computação gráfica e nada mais é do que uma multiplicação de matrizes [SHIRLEY05]. Como é sabido, um ponto P do espaço $\Re^3$ representado pelas coordenadas (x_p, y_p, z_p) pode também ser representado através da forma:

$$P = (x_p, y_p, z_p) \equiv O + x_p x + y_p y + z_p z$$

No sistema de coordenadas canônico, o ponto O corresponde a (0, 0, 0) e os vetores **x**, **y** e **z** são respectivamente representados pelas componentes (1, 0, 0), (0, 1, 0) e (0, 0, 1). Dessa forma, uma transformação do sistema de coordenadas canônico para um sistema arbitrário cuja origem está no ponto E, e cujos eixos de coordenadas são representados pelos vetores **u**, **v** e **w** poderia ser representada pela seguinte multiplicação de matrizes:

$$\begin{bmatrix} u_p \\ v_p \\ w_p \\ 1 \end{bmatrix} = \begin{bmatrix} x_u & y_u & z_u & 0 \\ x_v & y_v & z_v & 0 \\ x_w & y_w & z_w & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_e \\ 0 & 1 & 0 & -y_e \\ 0 & 0 & 1 & -z_e \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_p \\ y_p \\ z_p \\ 1 \end{bmatrix}$$

Após transformar as coordenadas de cada um dos vértices do objeto atingido do sistema de coordenadas local para o global, passa-se à remoção daquelas faces que não são visíveis (ou seja: cujo ângulo entre a normal da face analisada e a normal da face correspondente ao ponto de impacto é maior do que 90º).

Partindo da fórmula do produto escalar entre dois vetores A e B:

$$A \bullet B = |A||B| \cos\theta$$

podemos facilmente obter a fórmula para calcular o ângulo θ entre dois vetores, dada por:

$$\theta = a\cos\left(\frac{[AB]}{[|A||B|]}\right)$$

e visto que está-se trabalhando com normais (que são, por definição, já normalizadas) basta utilizar a seguinte fórmula:

$$\theta = a\cos(A \bullet B)$$

Calcula-se então o ângulo entre a normal de cada uma das faces do objeto atingido e a do ponto de impacto, armazenando-se apenas aquelas cujo ângulo é menor ou igual a 90º.

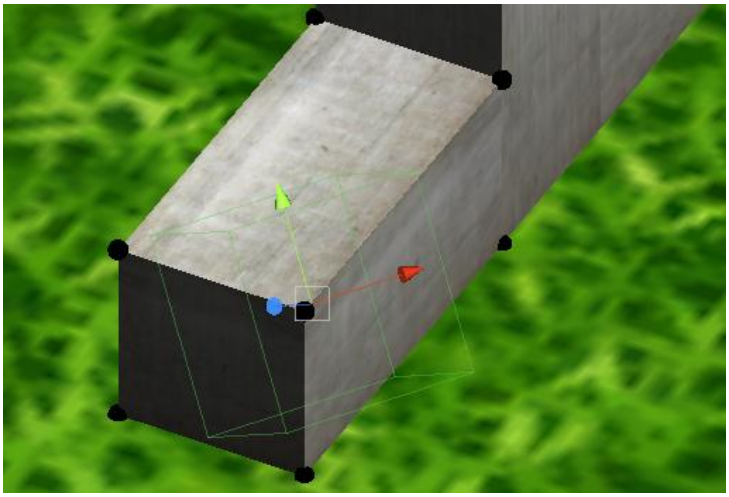

Figura 4.2: Vértices do objeto atingido após a remoção das faces ocultas

4.3 Projeção e recorte em 2D

Embora seja possível realizar o recorte das faces afetadas em 3D (como é feito na técnica apresentada por Lengyel em [LENGYEL01]), há também a possibilidade de levar os vértices ao plano de projeção (2D) deixando suas coordenadas Z temporariamente como 0 para que seja realizado um recorte contra as 4 linhas correspondentes aos extremos do projetor (ao invés de fazer o mesmo contra 6 planos, como ocorreria em 3D), ganhando-se assim em velocidade de processamento.

Conforme já explicado anteriormente, a transformação entre diferentes sistemas de coordenadas dá-se através de uma multiplicação de matrizes. Para este passo do algoritmo, são multiplicados cada um dos vértices remanescentes do objeto atingido pela matriz local do projetor. A componente Z de cada vértice pode então ser considerada como 0, para que se passe à fase de recorte contra a *viewport* definida pelas arestas do cubo projetor.

No espaço local, estas arestas vão do canto inferior esquerdo de coordenadas (x, y) dadas por (-0.5, -0.5) ao canto superior direito, cujas coordenadas são (0.5, 0.5). Sendo assim, o recorte das faces visíveis do objeto será feito contra as quatro retas definidas pelas seguintes equações:

Esquerda: $x = -0{,}5$

Direita: $x = 0{,}5$

Acima: $x = 0{,}5$

Abaixo: $x = -0{,}5$

Existem diversos algoritmos para recorte, tanto para retas quanto para polígonos, círculos e *B-Splines*. Entre os algoritmos de recorte mais populares, está o de Cohen-Sutherland, que divide o espaço 2D em 9 regiões, das quais a região central corresponde à área visível (*viewport*) [SN73]. Embora seja relativamente simples e fácil de ser implementado, por ser um algoritmo de recorte de linhas pode gerar resultados não desejados ao ser aplicado às faces, como o ilustrado na Figura 4.3.

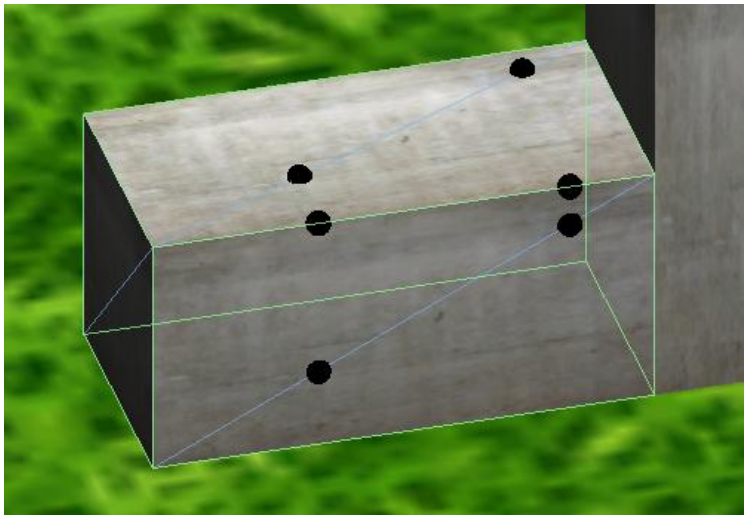

Figura 4.3: Problema da utilização do algoritmo Cohen-Sutherland para recorte de faces

Logo, para este passo deve-se utilizar um algoritmo apropriado para o recorte de polígonos, como é o caso do Sutherland-Hodgman, desenvolvido por Ivan Sutherland e Gary W. Hodgman [SH74]. O algoritmo de recorte Sutherland-Hodgman utiliza uma estratégia de "dividir para conquistar" para abordar o problema. Ele inicia realizando o recorte contra um determinado extremo da *viewport*, e para o conjunto de novos vértices obtido realiza um novo recorte contra outro extremo, e assim sucessivamente para cada uma das 4 arestas.

Para realizar o recorte contra uma das extremidades, o algoritmo percorre todos os vértices do polígono e em cada passo analisa dois deles, chamados "atual" e "próximo". Para cada vértice deste par, ele determina se estão dentro ou fora da área visível, em relação à aresta analisada no momento. Essa determinação é facilmente feita verificando-se as coordenadas x e y do ponto em relação à reta de referência atual. Em função da situação dos vértices analisados (se estão dentro ou fora da área visível) o algoritmo aplica as seguintes regras:

- Se ambos estiverem dentro: armazena o ponto "próximo"
- Se o "atual" estiver dentro e o "próximo" fora: armazena o ponto de intersecção entre a reta que une os dois pontos e a aresta da *viewport*
- Se ambos estiverem fora: não faz nada
- Se o "atual" estiver fora e o "próximo" dentro: armazena o ponto de intersecção com a *viewport* e o ponto "próximo"

A verificação de se um ponto está "dentro" ou "fora" é trivial, bastando analisar a componente horizontal ou vertical (dependendo do caso) do ponto em relação à aresta atual. Já a intersecção com a *viewport* é obtida seguindo a equação da reta que une os dois pontos até que esta cruze a aresta limítrofe atual.

A equação de uma reta que passa por um ponto pode ser expressa na forma:

$$y - y_1 = m(x - x_1)$$

na qual x_1 e y_1 são as coordenadas x e y (respectivamente) desse ponto e m representa o coeficiente angular da reta, também chamado de constante de

proporcionalidade, visto que representa a proporção entre a diferença na coordenada y (isto é, $y - y_1$) e a diferença na coordenada x (ou seja, $x - x_1$). Caso se deseje a equação da reta que passa por dois pontos, pode-se modificar a equação acima deixando-a na seguinte forma:

$$y - y_1 = \frac{y_2 - y_1}{x_2 - x_1} * (x - x_1)$$

sendo (x_1, y_1) e (x_2, y_2) dois pontos da reta (com $x_1 \neq x_2$), e como pode ser percebido, a diferença com relação à equação anterior é que aqui a constante de proporcionalidade (m) é dada por $\frac{y_2 - y_1}{x_2 - x_1}$.

Para obter os novos valores das componentes x e y em função da intersecção com os limites da *viewport*, basta realizar uma interpolação linear, verificando-se a variação em cada eixo (em função do novo ponto desejado) e somando-se essa diferença à coordenada inicial, de modo que temos:

$$x = x_1 + (x_2 - x_1) * \left(\frac{(NOVO\_Y - y_1)}{(y_2 - y_1)} \right)$$

$$y = y_1 + (y_2 - y_1) * \left(\frac{(NOVO\_X - x_1)}{(x_2 - x_1)} \right)$$

considerando a variação do ponto (x_1, y_1) em relação ao ponto (x_2, y_2) e sendo $NOVO\_X$ e $NOVO\_Y$ as novas coordenadas x e y (respectivamente) definidas em função da aresta da *viewport* contra a qual está sendo realizado o recorte, no momento.

Por exemplo, conforme ilustrado na Figura 4.4, para calcular o ponto P(x_p, y_p) correspondente à intersecção da reta que passa por P1 (-0,3; -0,8) e P2 (0,1; -0,3) com a aresta inferior do projetor (y = -0,5), realiza-se o seguinte cálculo:

$$x_p = -0{,}3 + (0{,}1 - (-0{,}3)) * \left(\frac{-0{,}5 - (-0{,}8)}{-0{,}3 - (-0{,}8)} \right)$$

$$= -0{,}3 + 0{,}4 * \frac{0{,}3}{0{,}5}$$

$$= -0{,}3 + 0{,}4 * 0{,}6$$

$$= -0{,}3 + 0{,}24$$

$$= -0{,}06$$

$$y_p = -0{,}5$$

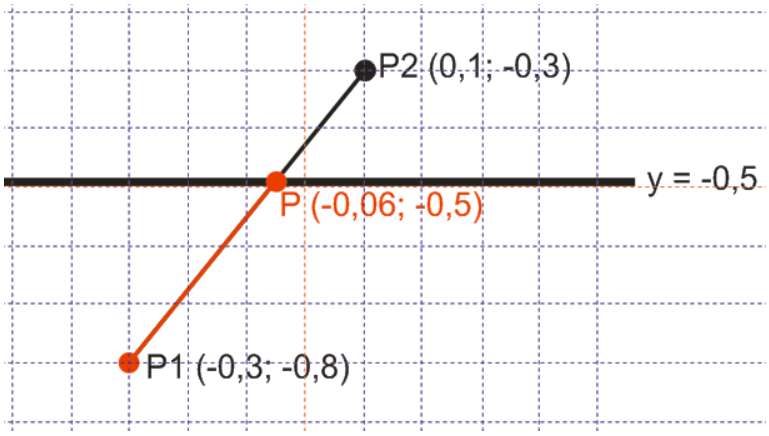

Figura 4.4: Ponto de intersecção com a reta $y = -0,5$

Outra coisa que é importante ser lembrada neste passo é que, embora para a realização do recorte só sejam necessárias as coordenadas x e y, para os casos em que um novo vértice é calculado é preciso realizar também uma interpolação linear (de forma análoga à feita para x e y) para a coordenada z deste novo vértice, para que quando os vértices sejam levados novamente ao espaço 3D ele esteja na posição correta.

Após fazer o recorte de cada face contra cada uma das 4 arestas da viewport (atualizando a lista de vértices a cada passo), para uma dada face que inicialmente possuía um conjunto de vértices V= {v0, v1, v2}, pode-se ter 3, 4, 5 ou até 6 novos vértices.

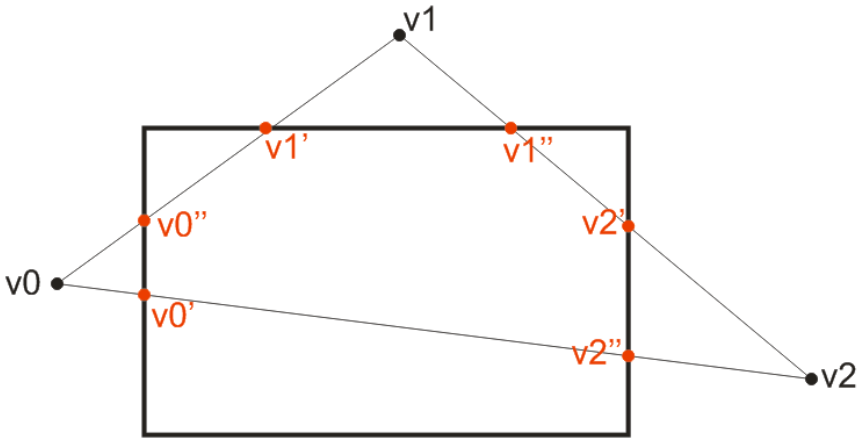

Figura 4.5: Exemplo de caso em que para os 3 vértices iniciais são gerados 6 novos

O caso em que após o recorte tem-se também 3 vértices é o caso ótimo, pois basta armazenar a nova face. Porém, para os casos em que se obtém mais de 3 vértices, é preciso redefinir as novas faces que serão criadas a partir deles, mantendo o valor original da normal da face para cada uma destas novas.

Para a definição de quais vértices irão compor cada uma destas novas faces, utilizou-se um *triangle fan*, que não apenas facilita a geração destas faces como também otimiza o resultado final, já que para um número N de triângulos no *triangle fan*, precisa-se apenas de N + 2 vértices, ao contrário dos 3N que são necessários para descrever triângulos separadamente. No caso da Figura 4.5, por exemplo (que corresponde ao número máximo possível de vértices após o recorte da face) tem-se:

N + 2 = 6
N = 6 – 2
N = 4

Ou seja, no pior caso (6 novos vértices) para uma dada face inicial serão geradas outras 4. O vértice que corresponde ao centro do *triangle fan* é decidido arbitrariamente, como sendo o primeiro vértice da lista final gerada. No exemplo acima, o conjunto de novos triângulos T = {t0, t1, t2, t3} é dado por:

t0 = v0', v0'', v1'
t1 = v0', v1', v1''
t2 = v0', v1'', v2'
t3 = v0', v2', v2''

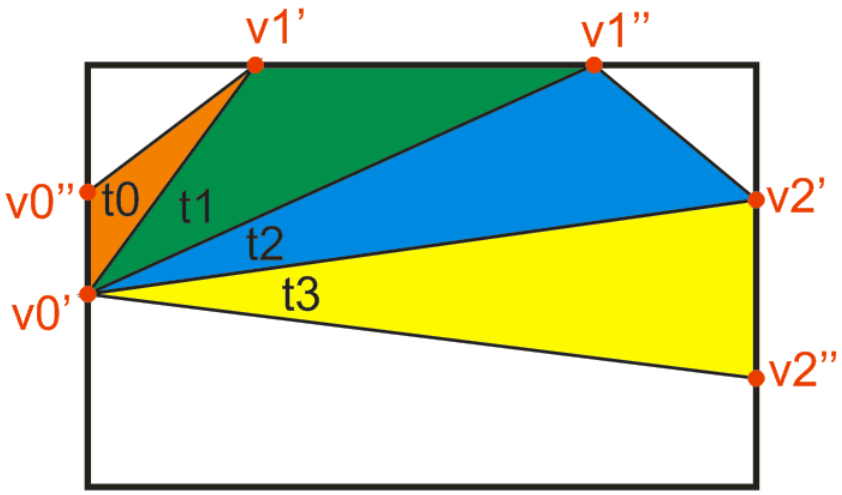

Figura 4.6: Novos triângulos gerados utilizando *triangle fan*

Cada um dos novos triângulos gerados é então armazenado, com a normal correspondente à da face que foi recortada inicialmente.

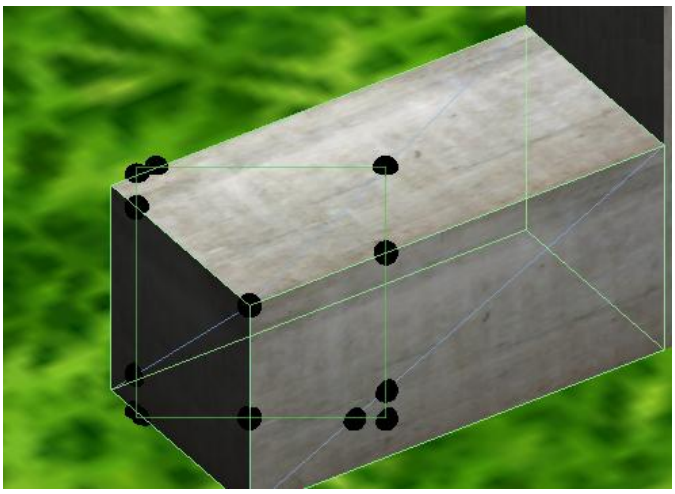

Figura 4.7: Vértices resultantes após o recorte com Sutherland-Hodgman

4.4 Coordenadas UV e projeção para 3D

As malhas de triângulos dentro da Unity 3D são compostas por quatro informações básicas: um conjunto de vértices do $\Re^3$, um conjunto de coordenadas UV do $\Re^2$, um conjunto de normais do $\Re^3$ e um conjunto de inteiros representando os índices dos vértices que correspondem a cada triângulo da malha.

O mapeamento UV é o processo no qual se realiza uma representação em uma imagem 2D de um modelo 3D. Ele consiste em projetar o mapa de textura em um objeto 3D. As letras "U" e "V" são usadas para descrever a malha 2D visto que as letras "X", "Y" e "Z" já são usadas para descrever o objeto 3D no espaço de modelagem. A texturização UV permite que os polígonos que compõem um objeto 3D sejam pintados com determinadas cores de uma imagem. A imagem é chamada de mapa de textura UV, mas nada mais é do que uma imagem 2D comum. No momento da criação de uma malha poligonal em 3D, as coordenadas UV podem ser geradas para cada vértice da malha. Por convenção, o sistema de coordenadas na imagem é definido como sendo o quadrado unitário $(u,v) \in [0,1]^2$, conforme ilustrado na Figura 4.8.

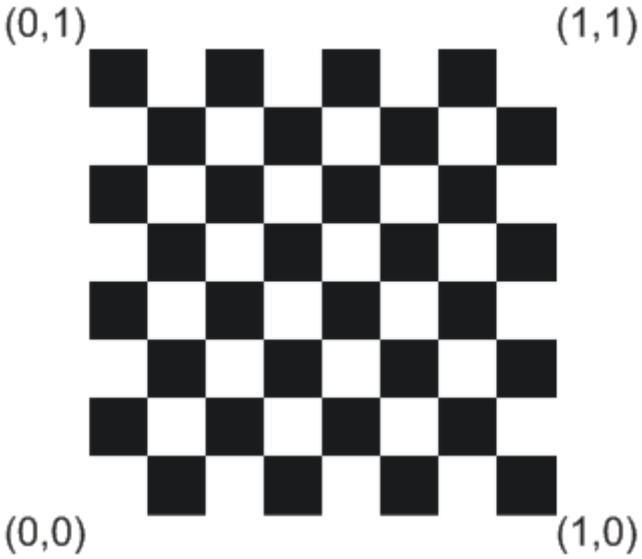

Figura 4.8: Coordenadas UV de uma imagem

Visto que neste passo estamos no sistema de coordenadas locais do objeto, e levando em consideração que estas coordenadas vão de (-0,5; -0,5) no extremo inferior esquerdo, até (0,5; 0,5) no extremo superior direito, bastaria então somar 0,5 unidades às componentes x e y de cada vértice (ver Figura 4.9), para ter-se assim o equivalente em coordenadas $(u,v) \in [0,1]^2$.

Uma vez armazenadas as coordenadas UV correspondentes a cada um dos vértices da nova malha, pode-se então voltar a transformar eles para o espaço do universo, multiplicando-os pela matriz correspondente para levar do espaço local do objeto ao sistema de coordenadas global.

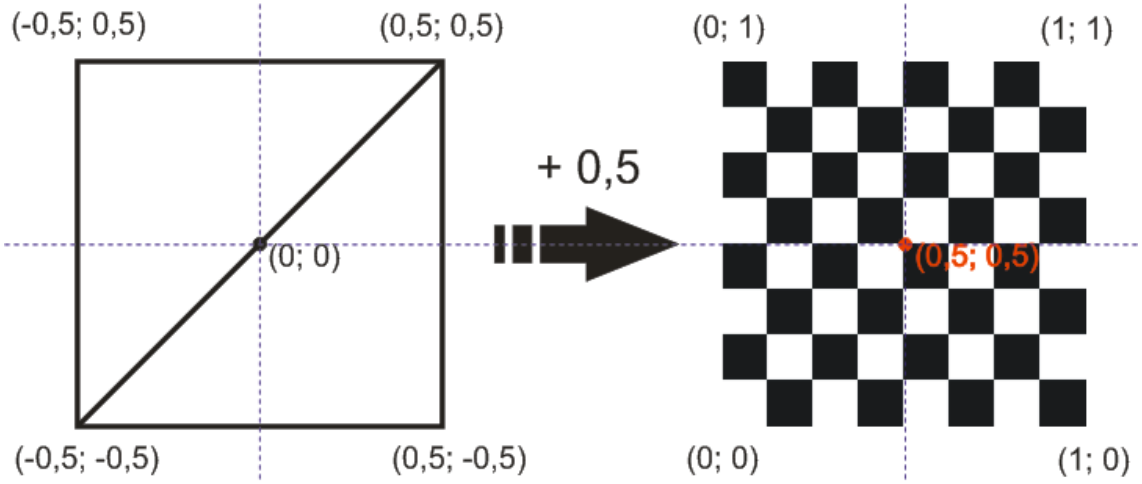

Figura 4.9: Transformações das coordenadas locais em coordenadas UV

4.5 Criação da malha final

Com as coordenadas UV já previamente armazenadas no passo anterior, passa-se agora à definição das demais componentes da nova malha para que ela possa ser instanciada. Conforme já dito anteriormente, estes atributos restantes são: os vértices, as normais e os triângulos.

4.5.1 Vértices

Os vértices estão neste momento no sistema de coordenadas canônico, e precisarão ser convertidos ao do novo objeto (vazio) que será criado para conter a malha. Com este objeto já instanciado na sua posição correspondente (o ponto de aplicação inicial do *decal*), pode-se utilizar a sua matriz local para converter cada um dos vértices para este sistema de coordenadas. Para evitar problemas de *z-fighting* com a textura do objeto sobre o qual esta malha será aplicada, pode-se aplicar um pequeno deslocamento (visualmente imperceptível) na posição de cada vértice. Visto que dependendo da face em que se esteja a normal pode ser diferente da do ponto atingido, o melhor é utilizar sempre a direção da própria normal da face cujos vértices estão sendo armazenados neste momento. Para realizar este deslocamento, são feitas duas operações básicas de vetores: a multiplicação por um escalar e a soma de vetores [LAY97].

A multiplicação por um escalar consiste no seguinte: dados um vetor $\vec{v}$ = (x, y, z) e um número real c, o múltiplo escalar de $\vec{v}$ por c é o próprio vetor $c\vec{v}$ obtido multiplicando cada componente de $\vec{v}$ por c. Ou seja:

$$c * \begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} cx \\ cy \\ cz \end{bmatrix}$$

Por exemplo, se $\vec{v}$ = (7, 4, -13) e c = 3 então

$$c\vec{v} = 3 * \begin{bmatrix} 7 \\ 4 \\ -13 \end{bmatrix} = \begin{bmatrix} 21 \\ 12 \\ -39 \end{bmatrix}$$

No caso da soma, dados dois vetores $\vec{v}$ e $\vec{w}$ do $\Re^2$, sua soma é o vetor $\vec{v} + \vec{w}$ obtido somando as componentes correspondentes de $\vec{v}$ e $\vec{w}$, ou seja, para $\vec{v} = (x_v, y_v, z_v)$ e $\vec{w} = (x_w, y_w, z_w)$ tem-se:

$$\vec{v} + \vec{w} = \begin{bmatrix} x_v + x_w \\ y_v + y_w \\ z_v + z_w \end{bmatrix}$$

Por exemplo, se $\vec{v} = (1, 9, 85)$ e $\vec{w} = (13, 0, 8)$, o vetor $\vec{v} + \vec{w}$ é dado por:

$$\begin{bmatrix} 1+13 \\ 9+0 \\ 85+8 \end{bmatrix} = \begin{bmatrix} 14 \\ 9 \\ 93 \end{bmatrix}$$

Logo, o que é feito para cada vértice é dar um pequeno deslocamento (d), desde sua posição original e no sentido de sua normal, isto é, para obter a nova posição (P') de um ponto $P = (x_p, y_p, z_p)$ cuja normal é dada pelo vetor $N = (x_n, y_n, z_n)$ faz-se:

$$P' = \begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} + d * \begin{bmatrix} x_n \\ y_n \\ z_n \end{bmatrix} = \begin{bmatrix} x_p \\ y_p \\ z_p \end{bmatrix} + \begin{bmatrix} x_n d \\ y_n d \\ z_n d \end{bmatrix} = \begin{bmatrix} x_p + x_n d \\ y_p + y_n d \\ z_p + z_n d \end{bmatrix}$$

4.5.2 Normais

As normais da malha são especificadas para cada vértice, e não por face. Dessa forma, tem-se um vetor com o mesmo número de posições que o vetor de vértices, e uma dada normal N[i] deve corresponder exatamente ao vértice V[i], para que o mapeamento seja feito corretamente.

Outro detalhe importante, é que nesta implementação os vértices não são compartilhados entre as faces, ou seja, em uma mesma posição P pode haver diversos vértices, cada um correspondente a uma face diferente da malha (ver Figura 4.10). O lado negativo disso é, obviamente, a existência de uma maior quantidade de vértices do que seria necessário para cada objeto. Por outro lado, dessa forma pode-se definir as normais individualmente (por vértice) sem a necessidade de se realizar uma interpolação entre as normais de todas as faces que compartilham o mesmo vértice.

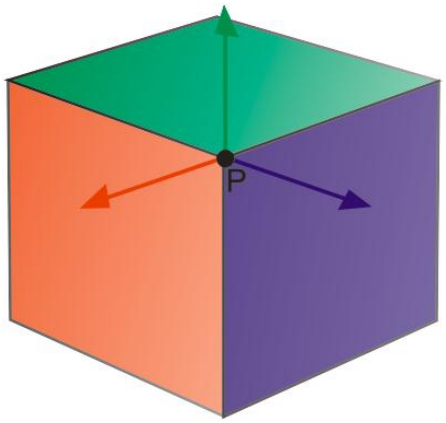

Figura 4.10: No ponto P encontram-se 3 vértices, cada um com a sua respectiva normal

4.5.3 Triângulos

A definição dos triângulos que irão compor a malha é feita através de um vetor de números inteiros, cada um representando uma posição no vetor de vértices, logo, o número de posições do vetor de triângulos deve ser sempre um múltiplo de 3. Para o cálculo dos triângulos da malha, o que a Unity faz é ir pegando em grupos de 3, e gerando um triângulo a partir deles.

Por exemplo, supondo um vetor de vértices V = [v0, v1, v2, v3, v4, v5], poderia ser criado o seguinte vetor de triângulos T: T[0] = 0; T[1] = 1; T[2] = 2; T[3] = 0; T[4] = 2; T[5] = 3; T[6] = 0; T[7] = 3; T[8] = 4, T[9] = 0; T[10] = 4, T[11] = 5, gerando assim os triângulos para a malha através de um *triangle fan*.

4.5.4 Aplicando o material e renderizando a malha

Neste ponto já temos todas as configurações necessárias definidas para a malha que representará o *decal* no espaço 3D. Tudo que resta ser feito é atribuir o material (passado inicialmente como parâmetro para o método de geração do *decal*), e renderizar a malha.

Dentro da *Unity*, para que a malha seja renderizada (após atribuí-la para o objeto apropriado) deve-se adicionar um *MeshFilter* e um *MeshRenderer*. A função do *MeshFilter* é obter a malha que foi configurada previamente e passá-la ao *MeshRenerer* para que ele seja renderizada na tela. A função do *MeshRenderer* é justamente pegar a geometria especificada no *MeshFilter* e renderizá-la na posição definida pelo objeto em que se está aplicando a malha.

A Figura 4.11 ilustra como ficou a malha aplicada sobre uma superfície e já renderizada.

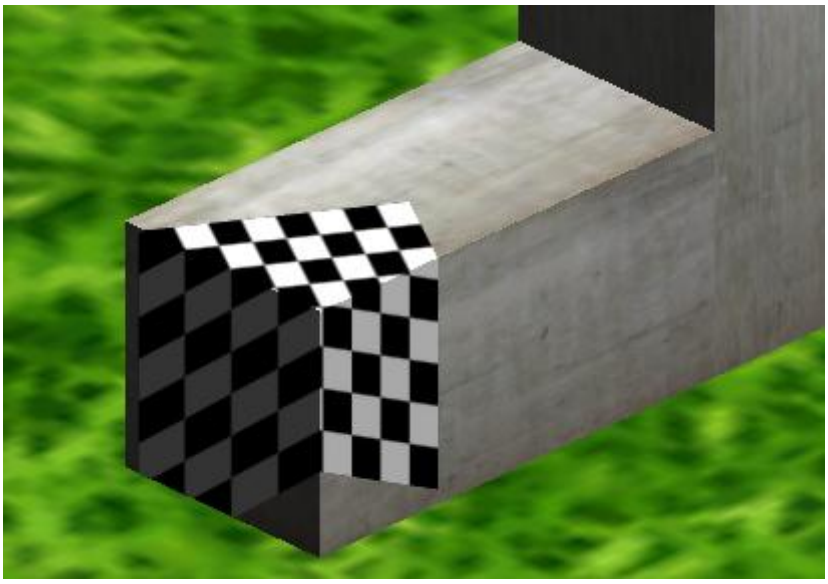

Figura 4.11: Malha final renderizada sobre a superfície

5 OTIMIZAÇÕES

Visando melhorar ainda mais os resultados obtidos, tanto visualmente como em termos de desempenho, foram implementadas algumas modificações e acréscimos à técnica apresentada no capítulo anterior.

5.1 Criação de um *shader* personalizado

Shaders são conjuntos de instruções usados, principalmente, para calcular efeitos de renderização no hardware gráfico com um alto grau de flexibilidade. Eles são utilizados para configurar o *pipeline* de renderização programável da GPU, permitindo a criação de diversos efeitos personalizados.

Entre as principais linguagens para programação de *shaders* disponíveis hoje em dia, destacam-se a *HLSL (High Level Shading Language)* da Microsoft, a *Cg (C for Graphics),* desenvolvida em conjunto pela nVIDIA e a Microsoft, e a *GLSL (OpenGL Shading Language)* desenvolvida pelo *OpenGL ARB (OpenGL Architecture Review Board).* Dentro da Unity, é possível programar os *shaders* em Cg, HLSL e *ShaderLab*, que é similar aos arquivos .fx da Microsoft ou ao CgFX da nVIDIA.

Por serem texturas aplicadas em tempo de execução sobre outras texturas pré-existentes, os *decals* requerem certos cuidados e tratamentos especiais de modo a gerar um efeito visualmente agradável, que não deixe transparecer como foram feitos ou eventuais falhas que possam vir a ser inseridas sobre a superfície no momento de suas aplicações. Sendo assim, eles requerem um tipo de tratamento específico para o momento de serem renderizados, o que foi feito através da criação de um *shader* customizado, por meio de algumas edições em um shader do tipo *diffuse*, para permitir *alpha-blending* (combinação convexa de duas cores para permitir efeitos de transparência) e para garantir que os pixels desse objeto não sejam escritos no *depth buffer* (para evitar problemas de *z-fighting*).

Para maiores informações sobre o shader implementado, recomenda-se conferir seu código fonte disponível no Apêndice B.

5.2 Mudanças nas estruturas de armazenamentos das faces

Durante as medições de desempenho que foram realizadas (e que serão descritas com mais detalhes no próximo capítulo) foi percebido que o método de remoção de faces ocultas estava levando um tempo considerável em ser executado para objetos com muitas faces (como esferas, por exemplo). Ao analisar o método, perceberam-se algumas alterações que poderiam ser feitas para otimizá-lo.

A primeira alteração foi na estrutura que armazena os dados das faces no programa (ver Figura 5.1), na qual trocou-se o vetor de vértices (que em cada passo em que se criava uma face, era inicializado e especificado seu tamanho) por uma sequência de 3 vértices (já que sabe-se de antemão que este é o número de vértices que cada face possuirá).

```
private struct Face
{
    public Vector3[] vertices;
    public Vector3 normal;
}
```

```
private struct Face
{
    public Vector3 v0;
    public Vector3 v1;
    public Vector3 v2;
    public Vector3 normal;
}
```

Figura 5.1: Modificações feitas na estrutura que armazena as faces

Além dessa modificação, alterou-se também a lista de faces que armazena todas aquelas faces visíveis do objeto, de modo que o laço principal do método de remoção de faces ocultas passou da forma apresentada na Figura 5.2 para a forma da Figura 5.3. Os resultados em desempenho dessas alterações serão vistos no próximo capítulo.

```
for(int i = 0; i < numFaces; i++)
{
    faces[i].vertices = new Vector3[3];
    for(int j = 0; j < faces[i].vertices.Length; j++)
    {
        int index = (i * 3) + j;
        faces[i].vertices[j] = _meshVertices[_mesh.triangles[index]];
        faces[i].normal = _mesh.normals[_mesh.triangles[index]];
    }

    _visibleFaces = new List<Face>();
    float angle = Vector3.Angle(faces[i].normal, -p_orientation);
    if(angle <= 90f)
    {
        _visibleFaces.Add(faces[i]);
    }
}
```

Figura 5.2: Laço principal do método de remoção das faces ocultas antes das alterações

```
for(int i = 0; i < numFaces; i++)
{
    int index = i * 3;
    Vector3 normal = _mesh.normals[_mesh.triangles[index]];
    float angle = Vector3.Angle(normal, -p_orientation);
    if(angle <= 90f)
    {
        Face newFace = new Face();

        newFace.v0 = _meshVertices[_mesh.triangles[index++]];
        newFace.v1 = _meshVertices[_mesh.triangles[index++]];
        newFace.v2 = _meshVertices[_mesh.triangles[index]];
        newFace.normal = normal;
        _visibleFaces[_visibleFacesCount++] = newFace;
    }
}
```

Figura 5.3: Laço principal do método de remoção das faces ocultas após as alterações

6 RESULTADOS E ANÁLISE

Neste capítulo serão apresentados alguns resultados obtidos utilizando a técnica implementada anteriormente. Além disso, será feita uma análise desses resultados bem como do desempenho do algoritmo, ao ser aplicado sobre diferentes tipos de superfícies.

6.1 Resultados visuais

Aqui serão mostrados os resultados da aplicação de uma textura de teste sobre diferentes tipos de superfícies, desde primitivas mais simples (Figuras 6.1, 6.2, e 6.3) até malhas mais complexas (Figuras 6.4, 6.5, 6.6 e 6.7).

6.1.1 Aplicação de *decals* sobre primitivas geométricas

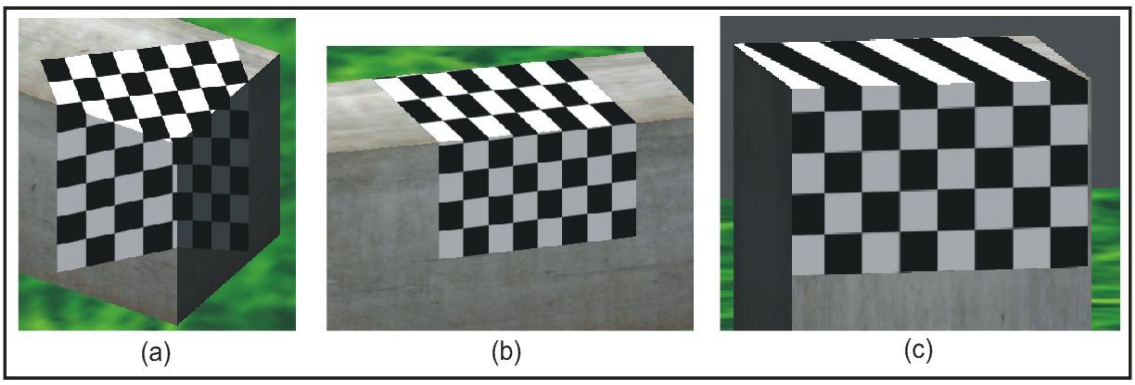

Figura 6.1: **Resultados da aplicação de *decals* sobre cubos.** A mesma textura de *decal* aplicada sobre cubos de diferentes escalas e com orientações variadas: Desde um ângulo superior e abrangendo 3 faces (a), também de um ângulo superior mas em apenas 2 faces (b) e desde um ângulo menor, fazendo com que o *decal* fique "espichado" na face superior.

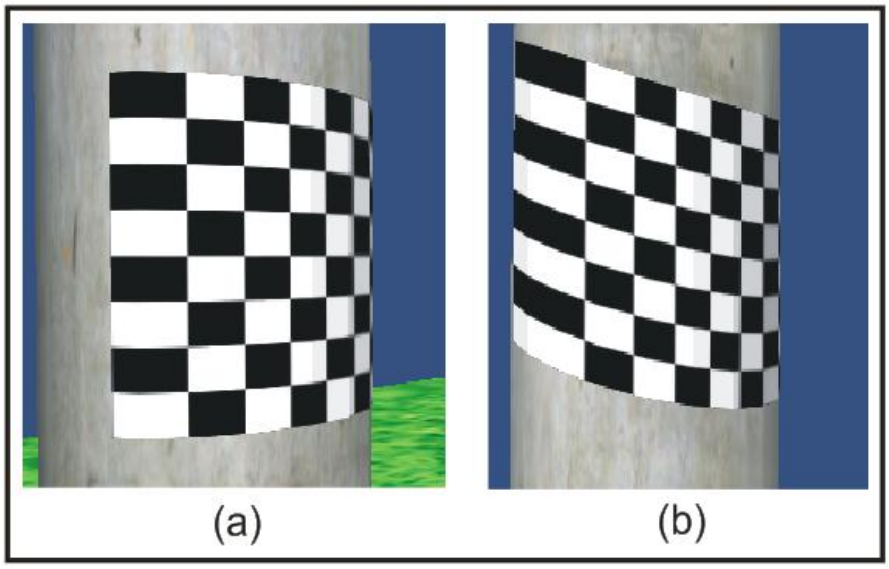

Figura 6.2: **Resultados da aplicação de *decals* sobre cilindros.** Note que em (a) a orientação de aplicação foi reta enquanto que em (b) foi feita desde baixo, provocando o efeito de "textura estirada" nas extremidades.

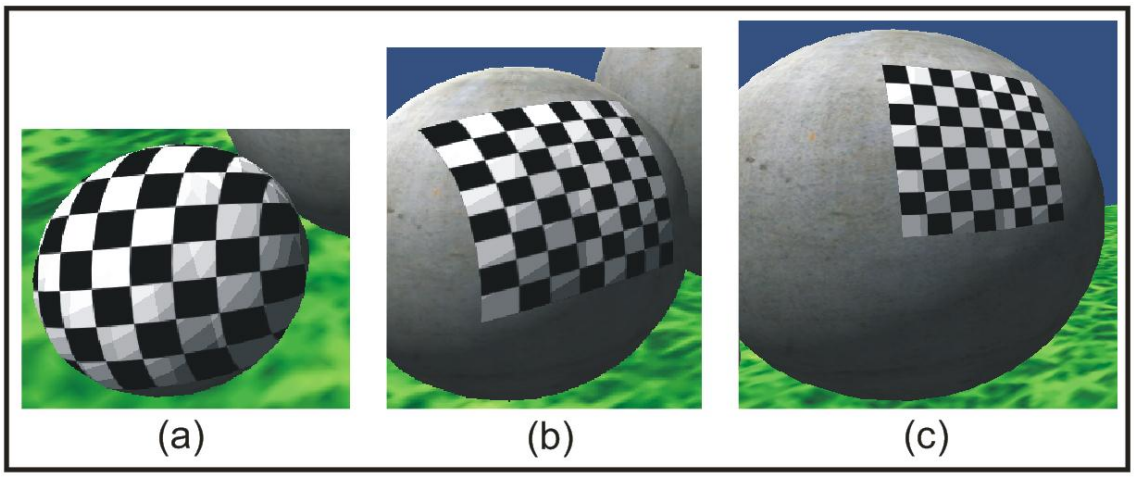

Figura 6.3: **Resultados da aplicação de *decals* sobre esferas.** Desde uma esfera com escala (1, 1, 1), a qual é coberta quase que totalmente pela textura, até em esferas com o dobro (b) e o triplo (c) do seu tamanho.

6.1.2 Aplicação de *decals* sobre superfícies mais complexas

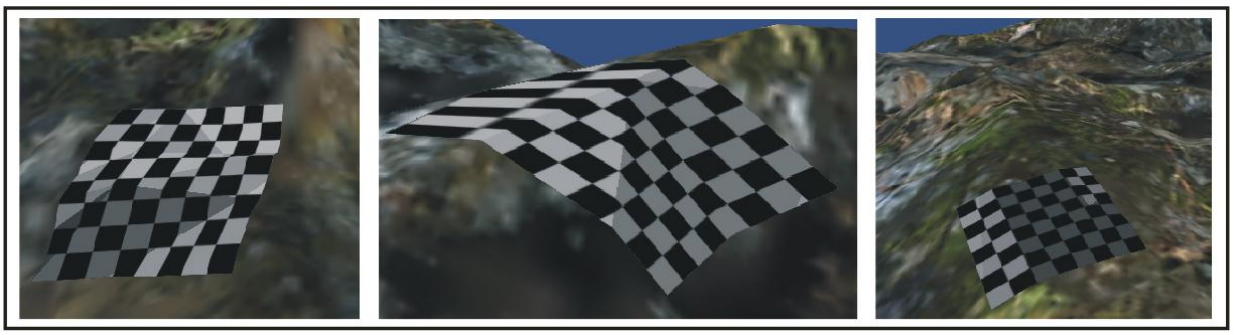

Figura 6.4: Resultados da aplicação de *decals* sobre terrenos com malhas irregulares

Figura 6.5: **Resultados da aplicação de *decals* sobre a malha de uma casa com grande número de polígonos.** Note como na imagem da direita é possível perceber que o recorte acabou gerando uma malha côncava, que se adequou às faces sobre as quais foi aplicada.

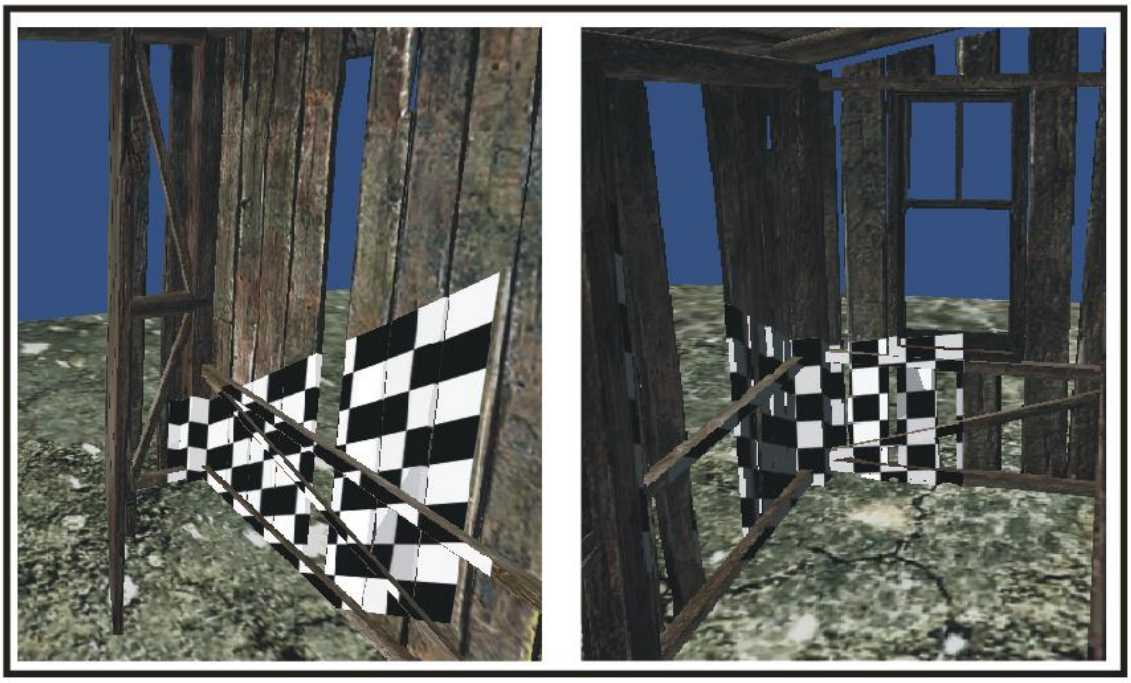

Figura 6.6: **Resultados da aplicação de *decals* sobre um canto da malha da casa.** Repare que o *decal* não ficou aplicado sobre o chão também, por este ser um objeto diferente.

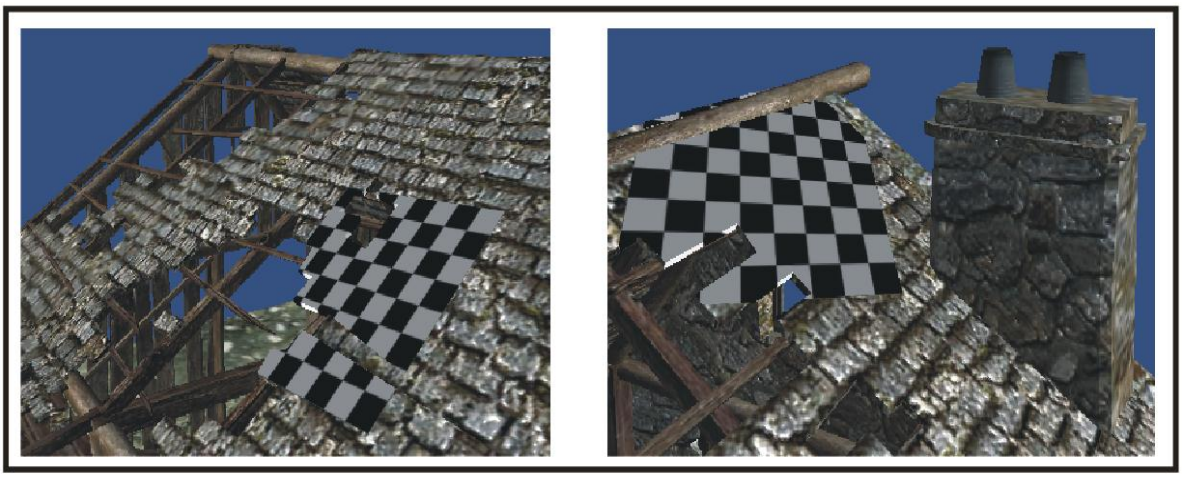

Figura 6.7: **Resultados da aplicação de *decals* sobre o telhado da casa.** Note que assim como ocorreu nas imagens anteriores, o decal fica entrecortado nas partes em que não há a malha do objeto atingido embaixo, conforme era esperado.

6.2 Análise de desempenho

Nesta seção será avaliado o desempenho do algoritmo criado ao ser aplicado a diferentes tipos de superfícies, desde primitivas geométricas até malhas mais complexas.

6.2.1 Ambiente de medições

O projeto foi todo desenvolvido dentro da *engine* Unity 3D utilizando C# como linguagem de programação. Todos os testes e medições aqui citados foram realizados em um notebook com processador *Intel® CORE™* i3 330M, 4GB de memória RAM DDR3, HD de 500GB, placa de vídeo *Intel® Graphics Media Accelerator HD*, rodando Windows® 7.

6.2.2 Análise do desempenho da aplicação de *decals* sobre primitivas geométricas

Conforme comentado na Seção 5.3, foi percebido um certo problema de desempenho no método de remoção das faces, de modo que certas estruturas de dados do jogo e partes do algoritmo acabaram sendo mudadas. Isso foi percebido através de uma análise do tempo consumido por cada um dos principais métodos do jogo. Eles são análogos às seções do capítulo 4, tendo a seguinte relação: Seção 4.1 (*InstantiateProjector*), Seção 4.2 (*GetMeshVertices* e *RemoveHiddenFaces*), Seção 4.3 (*ProjectVerticesTo2D* e *ClipFaces*), Seção 4.4 (*GetDecalMeshData* e *ProjectVerticesTo3D*) e Seção 4.5 (*CreateDecalMesh*).

Esses nomes serão usados nas próximas tabelas e abaixo deles estará o tempo médio consumido pelo método, para os diferentes tipos de superfícies sobre as quais os *decals* são aplicados. Além disso, são calculados também os tempos totais levados para gerar um *decal* para uma dada superfície e as médias de tempos consumidos por cada método (independentemente do tipo de primitiva). Todos os tempos estão expressos em segundos.

A Tabela 6.1 apresenta as medidas de tempo para as primitivas antes de serem feitas as otimizações descritas na Seção 5.3. Como pode ser observado, o tempo total que se destaca é o da esfera, que chega a quase 1 segundo (muito superior relativamente aos

demais, que estão nas casas dos centésimos). Observando-se os tempos médios de cada método, percebeu-se justamente que o gargalo encontrava-se no *RemoveHiddenFaces*, que posteriormente sofreu as alterações já apresentadas visando corrigir isso.

A Tabela 6.2 exibe os tempos médios das mesmas primitivas, porém após terem sido feitas as otimizações. Além disso, nesta tabela estão presentes variações dos mesmos tipos primitivos, nas quais foi alterada a escala. As escalas em (x, y, z) são as seguintes: Cubo 1 = (1, 2, 1), Cubo 2 = (2, 1, 1), Cubo 3 = (2, 2, 2), Cilindro 1 = (1, 2, 1), Cilindro 2 = (1, 2, 1), Cilindro 3 = (2, 2, 2), Esfera 1 = (1, 1, 1), Esfera 2 = (2, 2, 2) e Esfera 3 = (3, 3, 3). O número de faces e vértices é constante entre as primitivas do mesmo tipo, e é o seguinte: 12 faces e 36 vértices para os cubos, 80 faces e 240 vértices para os cilindros, 760 faces e 2280 vértices para as esferas e 200 faces e 600 vértices para o plano.

PRIMITIVA	Instantiate Projector	Get Mesh Vertices	Remove Hidden Faces	Project Vertices To 2D	Clip Faces	Get Decal Mesh Data	Project Vertices To 3D	Create Decal Mesh	TOTAL
Cubo	0,0013604	0,000398	0,00443	0,00075	0,004022	9,7E-05	4,01E-05	0,0329	0,044
Cilindro	0,0002441	0,000288	0,01628	4,67E-05	0,000592	1,81E-05	3,15E-05	0,0002	0,0177
Esfera	0,0002734	0,001016	0,76008	0,00027	0,011317	5,59E-05	0,00011	0,0003	0,7735
Plano	0,000248	0,000198	0,06911	0,00015	0,001465	3,81E-06	1,53E-05	0,0002	0,0714
MÉDIAS	0,0005315	0,000475	0,21247	0,0003	0,004349	4,4E-05	5E-05	0,0084	0,2266

Tabela 6.1: Medidas de tempo para as primitivas antes da otimização

PRIMITIVA	Instantiate Projector	Get Mesh Vertices	Remove Hidden Faces	Project Vertices To 2D	Clip Faces	Get Decal Mesh Data	Project Vertices To 3D	Create Decal Mesh	TOTAL
Cubo 1	0,000247955	0,0002	0,00701	1,4E-05	3,5E-05	6,68E-06	1,05E-05	0,000144	0,007669
Cubo 2	0,000244141	0,00035	0,00555	1,3E-05	4,8E-05	6,68E-06	1,72E-05	0,000224	0,006453
Cubo 3	0,000216484	0,00016	0,00488	1,1E-05	3,3E-05	3,81E-06	9,06E-06	0,000122	0,005442
Cilindro 1	0,000252724	0,00025	0,00983	5E-05	0,00015	1,24E-05	2,86E-05	0,000196	0,010762
Cilindro 2	0,000644684	0,00029	0,01252	4E-05	0,00014	1,34E-05	3,62E-05	0,000185	0,013866
Cilindro 3	0,000230789	0,00017	0,0068	5,1E-05	0,00012	7,63E-06	1,72E-05	0,000172	0,007568
Esfera 1	0,000238419	0,001	0,25088	0,00029	0,00076	5,53E-05	0,00032	0,000551	0,254093
Esfera 2	0,000230789	0,00092	0,23794	0,00027	0,00057	1,72E-05	8,77E-05	0,00029	0,240324
Esfera 3	0,000230789	0,0008	0,29976	0,00027	0,00054	1,53E-05	4,58E-05	0,000229	0,301899
Plano	0,000244141	0,00019	0,03751	0,00014	0,00027	1,53E-05	2,29E-05	0,000191	0,038589
MÉDIAS	0,000278091	0,00043	0,08727	0,00012	0,00027	1,54E-05	5,96E-05	0,00023	0,088667

Tabela 6.2: Medidas de tempos para as primitivas após as otimizações

A primeira análise que pode ser feita é com relação ao método de remoção das faces ocultas, que nas esferas caiu de 0,7 para 0,2 segundos, como era o objetivo. Uma segunda conclusão com relação aos dados apresentados na tabela é que o tempo médio total levado para projetar um *decal* sobre determinada superfície não depende da escala do objeto, mas sim da quantidade de faces que este possui. Isso ocorre pois, embora as

faces não visíveis sejam descartadas durante o processo do algoritmo, todas elas são analisadas inicialmente para poder ser armazenadas, o que gera um certo *overhead*.

6.2.3 Análise do desempenho da aplicação de *decals* sobre superfícies complexas

Aqui serão apresentadas tabelas com as médias de tempos para as superfícies complexas ilustradas na seção 6.1.2.

A Tabela 6.3 mostra o desempenho do algoritmo quando aplicados a três diferentes terrenos: Terreno 1 (com 699 faces e 2097 vértices), Terreno 2 (com 850 faces e 2550 vértices) e Terreno 3 (com 218 faces e 654 vértices). Já na Tabela 6.4 estão as médias de tempo para as diferentes partes da casa apresentada na seção 6.1.2, sendo a Malha 1 correspondente às paredes de madeira da casa (5632 faces e 18896 vértices), a Malha 2 à parede de concreto lateral (486 faces e 1458 vértices) e a Malha 3 correspondente ao telhado da casa (com 1782 faces e 5346 vértices).

PRIMITIVA	Instantiate Projector	Get Mesh Vertices	Remove Hidden Faces	Project Vertices To 2D	Clip Faces	Get Decal Mesh Data	Project Vertices To 3D	Create Decal Mesh	TOTAL
Terreno 1	0,0018888	0,001325	0,16906	0,001043	0,001573	3,34E-06	1,5E-05	0,00813	0,18304
Terreno 2	0,0002313	0,000492	0,27994	3,39E-05	3,43E-05	2,38E-06	5,7E-06	0,00015	0,28089
Terreno 3	0,000253	0,000171	0,03822	9,38E-05	0,000505	8,58E-06	2,1E-05	0,00019	0,03946
MÉDIAS	0,000791	0,000663	0,16241	0,00039	0,000704	4,77E-06	1,4E-05	0,00282	0,1678

Tabela 6.3: Medidas de tempo para a aplicação de *decals* sobre diferentes terrenos

PRIMITIVA	Instantiate Projector	Get Mesh Vertices	Remove Hidden Faces	Project Vertices To 2D	Clip Faces	Get Decal Mesh Data	Project Vertices To 3D	Create Decal Mesh	TOTAL
Malha 1	0,000682	0,004463	11,67673	0,002305	0,004699	3,31E-05	0,00017	0,003019	11,69211
Malha 2	0,0002314	0,000295	0,135142	0,000165	0,000313	5,09E-06	1,27E-05	0,00017	0,136335
Malha 3	0,0002543	0,001569	1,290369	0,000605	0,004013	1,27E-05	5,85E-05	0,000237	1,297119
MÉDIAS	0,0003893	0,002109	4,367415	0,001025	0,003008	1,7E-05	8,05E-05	0,001142	4,375186

Tabela 6.4: Medidas de tempo para a aplicação de *decals* sobre diferentes partes da malha da casa apresentada na Seção 6.1.2

Embora os tempos de execução do algoritmo estejam bastante aceitáveis para os terrenos e parte da malha da casa, é justamente para a Malha 1 que percebe-se o já mencionado problema da quantidade de faces. Neste caso tem-se 5632 faces, o que acabou gerando um tempo médio total de execução de mais de 11 segundos.

7 CONCLUSÕES

Este trabalho apresentou duas técnicas diferentes para a aplicação de *decals* sobre superfícies geométricas arbitrárias. A primeira delas, embora mais simples, mostrou-se com um excelente desempenho quando aplicada a texturas pequenas (como buracos de bala, por exemplo). Para o caso de texturas maiores, são percebidas certas falhas do algoritmo, que deixa a desejar. Já a segunda técnica descrita apresentou resultados satisfatórios conforme era o desejado inicialmente, independentemente do tamanho definido para o *decal*, porém acabou pecando no tempo de execução para casos em que o número de faces do objeto atingido é muito grande.

Ao longo deste texto, foram explicados diversos conceitos que são utilizados para a implementação do algoritmo de geração de *decals* apresentado, visando auxiliar àqueles que venham a implementar essa técnica e tomem este trabalho como referência.

Pode-se dizer que o objetivo deste trabalho foi atingido na medida em que desenvolveram-se métodos para aplicação de *decals* que podem ser utilizados dentro da *engine Unity* 3D, ferramenta que não possui suporte nativo para este tipo de recurso, bastando para isso utilizar o código-fonte apresentado neste trabalho e realizar a chamada ao sistema de *decals*. Contudo, para que a segunda técnica apresentada possa ser utilizada em tempo real dentro de jogos, ainda é necessário realizar certas otimizações para evitar a sobrecarga de processamento para objetos com um grande número de faces. Por outro lado, uma vez otimizado o algoritmo poderia ser utilizado não apenas para *feedback* visual para certas ações do jogador, mas também como uma ferramenta de edição para adicionar novos detalhes aos cenários em tempo real.

REFERÊNCIAS

[CATMULL74] CATMULL, Edwin E. **A subdivision algorithm for computer display of curved surfaces**, Ph. D. Thesis. Utah: University of Utah, 1974.

[JING06] JING, YingHui et al. **A post-processing decal texture mapping algorithm on graphics hardware**. Disponível em: ACM's VRCIA, p. 99-104, 2006.

[LAY97] LAY, David C. **Linear Algebra and its applications**, 2nd ed. Addison Wesley Longman, Inc., 1997.

[LENGYEL01] LENGYEL, Eric. **Applying decals to arbitrary surfaces**. Disponível em: Game Programming Gems 2, Charles River Media, M. DeLoura, Ed., p. 497-509, 2001.

[NVIDIA04] NVIDIA, **Improve batching using texture atlases**. Disponível em: SDK White Paper, 2004.

[PB11] PONS, Eduardo; BALDI, Raphael. **Criação de um sistema de *decals***. Notas pessoais. 2011.

[PRIESTER05] PRIESTER, Sjaak. **Polygon clipping**. Disponível em: http://www.codeguru.com/cpp/misc/misc/graphics/article.php/c8965__1/Polygon-Clipping.htm. Acessado em: nov. 2011.

[ROSEN09] ROSEN, David. **How to project decals**. Disponível em: http://blog.wolfire.com/2009/06/how-to-project-decals/. Acessado em: jul. 2011.

[SHIRLEY05] SHIRLEY, Peter. **Fundamentals of computer graphics**, 2nd ed. Wellesley: A. K. Peters, 2005.

[SN73] SPROULL, Bob; NEWMAN, William M. **Principles of interactive computer graphics**. p. 124 e p. 252. McGraw–Hill Education, International Ed., 1973.

[SH74] SUTHERLAND, Ivan; HODGMAN, Gary W. **Reentrant polygon clipping**. Comunications of the ACM, vol. 17, pp. 32-42, 1974.

[UNITY11] **Unity Documentation**. Disponível em: http://unity3d.com/support/documentation/. Acessado em: out. 2011

APÊNDICE A ALGORITMO DO SISTEMA DE *DECALS*

Abaixo encontra-se o código da classe *DecalSystem* utilizada para aplicar os *decals* neste trabalho. Para utilizá-lo, basta criar a classe e realizar a chamada desde qualquer ponto de um script na *Unity*.

```
using System;
using System.Linq;
using UnityEngine;
using System.Collections;
using System.Collections.Generic;

public class DecalSystem : MonoBehaviour
{
	private struct Face
	{
		public Vector3 v0;
		public Vector3 v1;
		public Vector3 v2;
		public Vector3 normal;
	}

	static public GameObject projector;
	private const float PROJECTOR_DISTANCE = 0.1f;
	private const float DECAL_DISTANCE = 0.01f;

	private const float MIN_X = -0.5f;
	private const float MAX_X = 0.5f;
	private const float MIN_Y= -0.5f;
	private const float MAX_Y = 0.5f;

	private const byte ID_TOPEDGE = 0;
	private const byte ID_BOTTOMEDGE = 1;
	private const byte ID_RIGHTEDGE = 2;
	private const byte ID_LEFTEDGE = 3;
	static private Mesh _mesh;
	static private Vector3[] _meshVertices;
	static private Face[] _visibleFaces;
	static private int _visibleFacesCount;
	static private Face[] _clippedFaces;
	static private int _clippedFacesCount;

	static private Mesh _decalMesh;
	static private Vector3[] _decalMeshVertices;
	static private Vector3[] _decalMeshNormals;
	static private Vector2[] _decalMeshUV;
	static private int[] _decalMeshTriangles;
	static private GameObject _decalContainer;
	static private int _decalCounter;
```

```
	static public void ProjectDecal(Vector3 p_hitPoint, Vector3
p_orientation, Transform p_hitTransform, Material p_decalMaterial, Vector2
p_scale)
	{
		InstantiateProjector(p_hitPoint, p_orientation, p_scale);
		GetMeshVertices(p_hitTransform);
		RemoveHiddenFaces(p_orientation);
		ProjectVerticesTo2D();
		ClipFaces();
		GetDecalMeshData();
		ProjectVerticesTo3D();
		CreateDecalMesh(p_hitPoint, -p_orientation, p_decalMaterial);
	}

	static private void InstantiateProjector(Vector3 p_hitPoint, Vector3
p_orientation, Vector2 p_scale)
	{
		projector = GameObject.CreatePrimitive(PrimitiveType.Cube);
		projector.transform.position = p_hitPoint + (PROJECTOR_DISTANCE *
-p_orientation);
		projector.transform.forward = p_orientation;
		projector.transform.localScale = new Vector3(p_scale.x,
p_scale.y, 1f);
		projector.transform.name = "projector";
		projector.collider.enabled = false;
	}

	static private void GetMeshVertices(Transform p_hitTransform)
	{
		//Getting the mesh vertices of the hit object
		_mesh = p_hitTransform.GetComponent<MeshFilter>().mesh;
		_meshVertices = _mesh.vertices;

		//Converting the vertices position from Local to World
		for(int i = 0; i < _meshVertices.Length; i++)
		{
			Vector3 worldVertex =
p_hitTransform.TransformPoint(_meshVertices[i]);
			_meshVertices[i] = worldVertex;
		}
	}

	static private void RemoveHiddenFaces(Vector3 p_orientation)
	{
		int numFaces = _mesh.triangles.Length / 3;

		_visibleFaces = new Face[numFaces];
		_visibleFacesCount = 0;

		//Filling the face array according to the vertices
		for(int i = 0; i < numFaces; i++)
		{
			int index = i * 3;
			Vector3 normal = _mesh.normals[_mesh.triangles[index]];
			float angle = Vector3.Angle(normal, -p_orientation);
			if(angle <= 90f)
			{
				Face newFace = new Face();

				newFace.v0 =_meshVertices[_mesh.triangles[index++]];
				newFace.v1 =_meshVertices[_mesh.triangles[index++]];
				newFace.v2 = _meshVertices[_mesh.triangles[index]];
				newFace.normal = normal;
				_visibleFaces[_visibleFacesCount++] = newFace;
			}
		}
	}
```

```
	static private void ProjectVerticesTo2D()
	{
		Matrix4x4 projectorLocalMatrix = Matrix4x4.identity *
projector.transform.worldToLocalMatrix;

		for(int i = 0; i < _visibleFacesCount; i++)
		{
			Face newFace = _visibleFaces[i];

			Vector4 v0 = newFace.v0;
			v0.w = 1;
			_visibleFaces[i].v0 = projectorLocalMatrix * v0;

			Vector4 v1 = newFace.v1;
			v1.w = 1;
			_visibleFaces[i].v1 = projectorLocalMatrix * v1;

			Vector4 v2 = newFace.v2;
			v2.w = 1;
			_visibleFaces[i].v2 = projectorLocalMatrix * v2;
		}
	}

	static private void ClipFaces()
	{
		//Initializing the list of the clipped faces
		_clippedFaces = new Face[_visibleFacesCount * 4];
		_clippedFacesCount = 0;

		Vector3[] tempVertices = new Vector3[6];
		int tempVerticesCount;

		for(int i = 0; i < _visibleFacesCount; i++)
		{
			tempVerticesCount = SutherlandHodgman(_visibleFaces[i],
tempVertices);

			//Recalculating the faces for the new vertices list
			RecalculateFaces(tempVertices, tempVerticesCount,
_visibleFaces[i].normal);
		}
	}

	static private Vector2 IntersectionPoint(byte p_currentEdge, Vector2
p_point1, Vector2 p_point2)
	{
		if (p_currentEdge == ID_TOPEDGE)
		return new Vector2(p_point1.x + (MAX_Y - p_point1.y) * (p_point2.x
- p_point1.x) / (p_point2.y - p_point1.y), MAX_Y);
		else if (p_currentEdge == ID_BOTTOMEDGE)
		return new Vector2(p_point1.x + (MIN_Y - p_point1.y) * (p_point2.x
- p_point1.x) / (p_point2.y - p_point1.y), MIN_Y);
		else if (p_currentEdge == ID_RIGHTEDGE)
		return new Vector2(MAX_X, p_point1.y + (MAX_X - p_point1.x) *
(p_point2.y - p_point1.y) / (p_point2.x - p_point1.x));
		else if (p_currentEdge == ID_LEFTEDGE)
		return new Vector2(MIN_X, p_point1.y + (MIN_X - p_point1.x) *
(p_point2.y - p_point1.y) / (p_point2.x - p_point1.x));

		return Vector2.zero;
	}
```

```
       static private bool IsInsideClipWindow(Vector2 p_point, byte
p_currentEdge)
       {
             if (p_currentEdge == ID_TOPEDGE)
            return (p_point.y <= MAX_Y);
         else if (p_currentEdge == ID_BOTTOMEDGE)
            return (p_point.y >= MIN_Y);
         else if (p_currentEdge == ID_LEFTEDGE)
            return (p_point.x >= MIN_X);
         else if (p_currentEdge == ID_RIGHTEDGE)
            return (p_point.x <= MAX_X);
         else
            return false;
       }

       static private int SutherlandHodgman(Face p_face, Vector3[]
p_tempVertices)
       {
             Vector3 currentPoint;
         Vector3 nextPoint;

             Vector3[] faceVertices = p_tempVertices;
             faceVertices[0] = p_face.v0;
             faceVertices[1] = p_face.v1;
             faceVertices[2] = p_face.v2;
             int faceVerticesCount = 3;

             Vector3[] clippedVertices = new Vector3[6];

         //Looping through all four clipping edges
         for (byte currentEdge = 0; currentEdge < 4; currentEdge++)
         {
                  //Resseting the current clipped vertices' counter
                  int clippedVerticesCount = 0;

            for (int i = 0; i < faceVerticesCount; i++)
            {
                        currentPoint = faceVertices[i];
                        nextPoint = (i == (faceVerticesCount - 1)) ?
faceVertices[0] : faceVertices[i + 1];

               if (IsInsideClipWindow(currentPoint, currentEdge))
               {
                   if (IsInsideClipWindow(nextPoint, currentEdge))
                   {
                       //Both points are inside: Add nextPoint to the list
                       clippedVertices[clippedVerticesCount++] = nextPoint;
                   }
                   else
                   {
                       //currentPoint is inside but nextPoint is outside
                       Vector3 newPoint = IntersectionPoint(currentEdge,
currentPoint, nextPoint);
                       newPoint.z = RecalculateZ(nextPoint.z, currentPoint.z,
nextPoint, newPoint, currentPoint);

                       clippedVertices[clippedVerticesCount++] = newPoint;
                   }
               }
               else
               {
                   if (IsInsideClipWindow(nextPoint, currentEdge))
                   {
                       //currentPoint is outside but nextPoint is inside
                       Vector3 newPoint = IntersectionPoint(currentEdge,
currentPoint, nextPoint);
```

```
                        newPoint.z = RecalculateZ(currentPoint.z, nextPoint.z,
currentPoint, newPoint, nextPoint);
                        clippedVertices[clippedVerticesCount++] = newPoint;
                        clippedVertices[clippedVerticesCount++] = nextPoint;
                    }
                }
            }

            //Updating the current vertices list
            for(int j = 0; j < clippedVerticesCount; j++)
            {
                faceVertices[j] = clippedVertices[j];
            }
            faceVerticesCount = clippedVerticesCount;
        }

        return faceVerticesCount;
    }

    static private float RecalculateZ(float p_originalZ1, float
p_originalZ2, Vector2 p_point1, Vector2 p_newPoint1, Vector2 p_point2)
    {
        //Checking the new Z value
        if(p_point1 == p_newPoint1)
        {
            return p_originalZ1;
        }
        else
        {
            //Recalculating the Z value
            float originalDistance = Vector2.Distance(p_point1, p_point2);
            float newDistance = Vector2.Distance(p_newPoint1, p_point2);
            float distanceOffset = newDistance / originalDistance;
            float zVariation = p_originalZ1 - p_originalZ2;
            float zOffset = zVariation * distanceOffset;
            return (p_originalZ2 + zOffset);
        }
    }

    static private void RecalculateFaces(Vector3[] p_vertices, int
p_verticesCount, Vector3 p_normal)
    {
        if(p_verticesCount < 3) return;

        Face newFace = new Face();
        newFace.normal = p_normal;

        if(p_verticesCount == 3)
        {
            //Optimum case: The 3 vertices correspond to the face's vertices
            newFace.v0 = p_vertices[0];
            newFace.v1 = p_vertices[1];
            newFace.v2 = p_vertices[2];
            _clippedFaces[_clippedFacesCount++] = newFace;
        }
        else
        {
            //More than 3 vertices: Recalculate using a Triangle Fan
            Vector3 center = p_vertices[0];
            for(int i = 1; i < (p_verticesCount - 1); i++)
            {
                if(i != 1)
                {
                    //Resseting the newFace variable
                    newFace = new Face();
                    newFace.normal = p_normal;
                }
```

```
                        //Assigning the other vertices of the triangle fan
                        Vector3 vertex1 = p_vertices[i];
                        Vector3 vertex2 = p_vertices[i + 1];

                        //Avoiding "in-line faces"
                        if((center.x == vertex1.x && center.x == vertex2.x)
|| (center.y == vertex1.y && center.y == vertex2.y))
                        {
                              continue;
                        }

                        //Adding the vertices to the new face
                        newFace.v0 = center;
                        newFace.v1 = vertex1;
                        newFace.v2 = vertex2;
                        _clippedFaces[_clippedFacesCount++] = newFace;
                  }
            }
      }

      static private void GetDecalMeshData()
      {
            int totalVertices = _clippedFacesCount * 3;

            _decalMeshVertices = new Vector3[totalVertices];
            _decalMeshNormals = new Vector3[totalVertices];
            _decalMeshUV = new Vector2[totalVertices];
            _decalMeshTriangles = new int[totalVertices];

            for(int i = 0; i < _clippedFacesCount; i++)
            {
                  int index = i * 3;
                  float uvOffset = 0.5f; //To change from [(-0.5, -0.5) (0.5,
0.5)] to [(0, 0) (1,1)]
                  Vector3 v;

                  v = _clippedFaces[i].v0;
                  _decalMeshUV[index] = new Vector2(v.x + uvOffset, v.y +
uvOffset);
                  _decalMeshNormals[index] = _clippedFaces[i].normal;
                  _decalMeshTriangles[index] = index;
                  index++;

                  v = _clippedFaces[i].v1;
                  _decalMeshUV[index] = new Vector2(v.x + uvOffset, v.y +
uvOffset);
                  _decalMeshNormals[index] = _clippedFaces[i].normal;
                  _decalMeshTriangles[index] = index;
                  index++;

                  v = _clippedFaces[i].v2;
                  _decalMeshUV[index] = new Vector2(v.x + uvOffset, v.y +
uvOffset);
                  _decalMeshNormals[index] = _clippedFaces[i].normal;
                  _decalMeshTriangles[index] = index;
            }
      }

      static private void ProjectVerticesTo3D()
      {
            Matrix4x4 projectorWorldMatrix= Matrix4x4.identity *
projector.transform.localToWorldMatrix;

            for(int i = 0; i < _clippedFacesCount; i++)
            {
                  Face newFace = _clippedFaces[i];
```

```
                    Vector4 v0 = newFace.v0;
                    v0.w = 1;
                    _clippedFaces[i].v0 = projectorWorldMatrix * v0;

                    Vector4 v1 = newFace.v1;
                    v1.w = 1;
                    _clippedFaces[i].v1 = projectorWorldMatrix * v1;

                    Vector4 v2 = newFace.v2;
                    v2.w = 1;
                    _clippedFaces[i].v2 = projectorWorldMatrix * v2;
             }
       }

       static private void CreateDecalMesh(Vector3 p_position, Vector3
p_normal, Material p_material)
       {
             GameObject decal = new GameObject();
             decal.transform.name = "decal";
             decal.transform.position = p_position;

             for(int i = 0; i < _clippedFacesCount; i++)
             {
                    int index = i * 3;
                    Vector3 v;

                    v = _clippedFaces[i].v0;
                    _decalMeshVertices[index] =
decal.transform.InverseTransformPoint(v);
                    _decalMeshVertices[index++] += DECAL_DISTANCE * p_normal;

                    v = _clippedFaces[i].v1;
                    _decalMeshVertices[index] =
decal.transform.InverseTransformPoint(v);
                    _decalMeshVertices[index++] += DECAL_DISTANCE * p_normal;

                    v = _clippedFaces[i].v2;
                    _decalMeshVertices[index] =
decal.transform.InverseTransformPoint(v);
                    _decalMeshVertices[index] += DECAL_DISTANCE * p_normal;
             }

             Mesh decalMesh = new Mesh();
             decalMesh.vertices = _decalMeshVertices;
             decalMesh.normals = _decalMeshNormals;
             decalMesh.uv = _decalMeshUV;
             decalMesh.triangles = _decalMeshTriangles;

             //Creating the MeshFilter and Renderer for the new decal mesh
             decal.AddComponent<MeshFilter>().mesh = decalMesh;
             decal.AddComponent<MeshRenderer>();
             decal.renderer.material = p_material;
             Destroy(projector.gameObject);

             //Adding the decal to the decal container
             if(_decalContainer == null)
             {
                    _decalContainer = new GameObject();
                    _decalContainer.transform.name = "_DecalContainer";
                    _decalCounter = 0;
             }
             decal.transform.parent = _decalContainer.transform;
             decal.renderer.material.renderQueue =
decal.renderer.material.shader.renderQueue + _decalCounter;
             _decalCounter++;
       }
}
```

APÊNDICE B SHADER CUSTOMIZADO

O código abaixo corresponde ao *shader* implementado para os materiais dos *decals*. Trata-se de uma variação de um *shader* do tipo *diffuse*, para permitir *alpha-blending* e para garantir que os pixels desse objeto não sejam escritos no *depth buffer* (para evitar problemas de *z-fighting*).

```
Shader "Andrade/Decal-Andrade"
{
Properties
	{
		_Color ("Main Color", Color) = (1,1,1,1)
		_MainTex ("Base (RGB) Trans (A)", 2D) = "white" {}
	}

	SubShader
	{
		Tags {"Queue"="Transparent" "IgnoreProjector"="True"
		"RenderType"="Transparent"}
		LOD 200
		ZWrite Off
		Blend SrcAlpha OneMinusSrcAlpha

		CGPROGRAM
		#pragma surface surf Lambert alpha

		sampler2D _MainTex;
		fixed4 _Color;

		struct Input
		{ float2 uv_MainTex; };

		void surf (Input IN, inout SurfaceOutput o)
		{
			fixed4 c = tex2D(_MainTex, IN.uv_MainTex) * _Color;
			o.Albedo = c.rgb;
			o.Alpha = c.a;
		}
		ENDCG
		}
		Fallback "Transparent/VertexLit"
	}
```