DETECÇÃO DE COLISÃO ENTRE SUPERFÍCIES DE SUBDIVISÃO

Thiago Ferronatto
Orientador: Paulo Aristarco Pagliosa

2024

Resumo

A detecção de colisão entre objetos virtuais é de suma importância para vários tipos de simulações. Este trabalho visa apresentar uma proposta de processo de detecção de colisão entre tipos específicos de objetos: superfícies paramétricas comumente resultantes de processos de subdivisão de malhas poligonais; além disso, também é discorrida a tesselação acelerada por *hardware* de superfícies de tal natureza com o auxílio da API gráfica OpenGL. O método empregado envolve a conversão das superfícies em malhas de triângulos, além do uso de algumas estruturas de aceleração; nominalmente, árvores de volumes limitantes. Os resultados obtidos mostram que a tesselação em GPU funcionou como esperado e, também, que o método de detecção de colisões adotado é viável; no entanto, não foram obtidos dados de desempenho suficientes para que seja feita uma declaração definitiva sobre esse aspecto.

Palavras-chave: Detecção de colisão, superfícies de subdivisão, tesselação por *hardware*, rasterização.

1 Introdução

No contexto da Computação Gráfica, há inúmeras maneiras de se representar um objeto; a escolha de uma entre elas é feita com base em quão vantajosa tal representação é para a tarefa em questão. Para aplicações em tempo real, como jogos eletrônicos, é comum que malhas de triângulos sejam empregadas, devido a otimizações criadas ao longo das últimas décadas voltadas à exibição de triângulos com o auxílio de placas gráficas (*Graphics Processing Units*, ou GPUs).

Tais malhas, no entanto, possuem limitações quando utilizadas para representar superfícies curvas, devido ao aspecto planar dos triângulos; nesses casos, essas malhas não passam de aproximações e, tendo em mente essa natureza, consequentemente apresentam imperfeições que as tornam inadequadas para aplicações nas quais um maior grau de precisão é requerido. Um exemplo se dá na forma de *softwares* de *design* e fabricação (*Computer Aided Design/Manufacturing*, ou CAD/CAM). Essas ferramentas são utilizadas para a confecção de produtos reais, desde *smartphones* a aeronaves e maquinários utilizados em grandes cadeias de produção.

Por consequência, aplicações CAD/CAM frequentemente trabalham com superfícies que possuem curvatura, possivelmente demandando o cálculo de propriedades intrinsecamente atreladas à natureza não-planar da superfície, tornando inadequadas aproximações como as malhas poligonais. Nesses casos, superfícies suaves são representações comumente aplicadas. Em particular, superfícies de B-Spline racionais não-uniformes (*Non-Uniform Rational B-Splines*, ou NURBS) [15] encontram, aí, um forte nicho.

Finalmente, aplicações que se encontram entre esses dois extremos podem variar em suas escolhas de representações de superfícies. Um exemplo seria a indústria cinematográfica; seja para efeitos especiais ou mesmo para obras inteiramente animadas, alguma representação deve ser empregada. Como a geração das imagens não será em tempo real, não há necessidade de se utilizar malhas de triângulos como aproximações; porém, também não é requerida a precisão oferecida pelas superfícies suaves.

Levando isso em conta, uma abordagem híbrida é comumente adotada em tais cenários: superfícies de subdivisão [1]. Malhas poligonais de baixa complexidade podem ser subdivididas, gerando malhas mais detalhadas; esse processo, se repetido indefinidamente, geraria superfícies suaves. De acordo com [1], pode-se obter representações de tais superfícies sem que haja a necessidade de efetivamente subdividir uma malha ao infinito. Essa estratégia permite que *designers* armazenem e trabalhem com malhas de baixa densidade poligonal, que são eventualmente refinadas e geram representações de alta fidelidade visual.

Em tais ambientes, pode emergir uma situação na qual simulações físicas, como interações entre corpos rígidos, são convenientes. Pode-se tomar como exemplo algum cenário em que objetos caem de uma certa altura e colidem entre si e com o solo. Os objetos podem ser manualmente inseridos na cena e animados por animadores, porém, conforme o número

de objetos aumenta, o esforço manual necessário cresce na mesma medida. Assim, seria vantajoso simplesmente aplicar uma simulação de colisão nos objetos e deixá-los cair naturalmente. Se tais objetos forem superfícies de subdivisão, então um método eficiente para simular tais colisões seria pertinente.

Com tal contexto em mente, este trabalho propõe uma metodologia de detecção de colisões entre superfícies de subdivisão baseada em malhas de triângulos decorrentes de processos de tesselação, além de prover a exibição via rasterização de tais superfícies por meio de tesselação acelerada por *hardware*.

1.1 Motivação

Apesar da aparente quebra da Lei de Moore [10], *hardwares* ainda estão evoluindo a um rápido passo. Limitações técnicas se encontram cada vez mais maleáveis. Um exemplo seria o traçado de raios em tempo real; outrora impedido pelo *hardware* disponível, agora uma realidade devido ao advento de placas gráficas com capacidade de aceleração por *hardware*.

Assim, não é absurdo esperar que aplicações em tempo real também passem a adotar, em virtude de *hardwares* mais poderosos, representações gráficas de objetos que oferecem níveis de detalhes superiores às malhas de triângulos, como superfícies de subdivisão.

Tomemos, novamente, jogos como exemplos de tais aplicações. A grande maioria dos motores de jogos fornecem um motor de física aos desenvolvedores; simulações de colisões são executadas dezenas ou centenas de vezes por segundo para que um jogo mantenha uma performance suave.

É evidente que, se tais *engines* fizerem uso de superfícies de subdivisão, então qualquer melhoria no desempenho de detecções de colisões entre tais representações refletiria diretamente em uma melhoria na taxa de quadros da aplicação, desde que ela não estivesse limitada por outros fatores.

Ademais, se tal avanço de *hardwares* de fato ocorrer, os atuais usos de superfícies de subdivisão em ambientes com simulações de colisão também seriam beneficiados por uma implementação de tal comportamento. Consideremos o cenário de corpos deformáveis, ou seja, representações de objetos que podem ter sua forma alterada em caso de aplicação de força sobre ela; tais deformações podem ser elásticas – nas quais o corpo é deformado mas, quando as forças diminuem, volta a seu formato original – ou plásticas – quando o corpo é deformado permanentemente.

Atualmente, corpos deformáveis representados por superfícies paramétricas são simulados, no campo da física e engenharia, por meio de abordagens que são computacionalmente intensivas e inadequadas para aplicações em tempo real, como jogos; nesses casos, são empregados outros métodos que sacrificam a precisão da simulação a favor da performance. No entanto, como ocorreu com o traçado de raios, é plausível que, com o avanço do *hardware*, se possa aproveitar as mesmas técnicas utilizadas em aplicações de engenharia em *apps* que funcionam em tempo real.

Com isso em mente, espera-se que a importância de um processo de detecção de colisões entre superfícies de subdivisão tenha sido devidamente enfatizada.

1.2 Objetivos

O objetivo geral deste trabalho é detectar colisões entre superfícies de subdivisão. Para tal, é necessário adotar uma abordagem de detecção; embora computacionalmente intensivo, é possível calcular de forma exata a intersecção de duas superfícies paramétricas. No entanto, a estratégia escolhida, aqui, envolve gerar uma malha de triângulos a partir da superfície – processo chamado de tesselação – e, utilizando algoritmos altamente eficientes já disponíveis na literatura para intersecções de triângulos, determinar se duas malhas tesseladas estão colidindo em dado momento.

Além disso, é desejável que as superfícies possam ser exibidas em uma aplicação gráfica; há, então, duas abordagens a se considerar: traçado de raios, que envolve calcular de forma exata a intersecção de raios com superfícies paramétricas, ou rasterização, que gira em torno da tesselação da superfície. Devido ao processo de detecção das colisões, a tesselação é uma etapa necessária de qualquer maneira, portanto o método utilizado neste trabalho é voltado à rasterização, o que permite que a simulação seja executada em tempo real sem que haja dependência em *hardwares* para aceleração de traçado de raios.

Por fim, é natural que seja almejada uma forma de comprovar de maneira visual que as colisões estão, de fato, sendo detectadas corretamente; um tipo de "resposta" à colisão. Um exemplo de resposta é encontrado em motores de física de corpos rígidos, em que colisões são detectadas e utilizadas para aplicar forças em corpos, que reagem de maneira apropriada. Essa é a abordagem adotada, aqui, para verificar as colisões. Seguem, então, os objetivos específicos para que o objetivo geral seja atingido.

Tesselar superfícies em CPU e GPU. Para detectar colisões, a superfície deve ser tesselada uma só vez em CPU com baixa granularidade, isto é, a densidade poligonal da malha resultante será a menor possível, a fim de diminuir o número de testes de colisão de primitivos; a malha resultante deve ser armazenada em memória e utilizada pelo algoritmo de detecção. Para exibir as superfícies em tela, elas deverão ser tesseladas em tempo real na GPU com alta granularidade; a malha resultante deve seguir seu caminho na *pipeline* gráfica e não deve ser armazenada.

Detectar colisões. As malhas resultantes das tesselações em CPU devem ser armazenadas e, a cada *frame*, testadas entre si para determinar se há pares de malhas colidindo em determinado instante. Devem ser utilizados procedimentos e estruturas de modo a acelerar o processo de detecção de colisão entre as superfícies.

Aplicar mecânica de corpos rígidos. A fim de verificar a exatidão da detecção de colisões, uma simulação de física de corpos rígidos, baseada em impulsos, deve ser executada sobre os objetos.

É desenvolvido, então, um ferramental que leva grande parte da carga atrelada à tesselação da CPU para a GPU – mais especificamente, por meio da API gráfica OpenGL –, já que a tesselação das superfícies é executada inteiramente pela placa gráfica. A arquitetura SIMD de uma GPU é bastante propícia para essa tarefa e, portanto, provê um aumento de performance em cenários em que a CPU é uma limitação.

1.3 Organização do texto

Primeiramente, na seção 2, são abordados os conceitos e as definições dos construtos e procedimentos geométricos e matemáticos utilizados no decorrer do texto, como descrições formais das superfícies suaves resultantes dos processos de subdivisão, seguidas dos procedimentos de geração de malhas de triângulos decorrentes de tais superfícies. Isso é seguido pelo detalhamento, na seção 3, do processo de detecção de colisões entre superfícies de subdivisão. Por fim, na sequência, são discutidos na seção 4 os resultados obtidos e como eles se alinham com os objetivos propostos.

2 Tesselação

Uma tesselação (do Inglês *tessellation*) é "uma cobertura de um plano geométrico infinito sem buracos ou sobreposições por figuras planares congruentes de um ou mais tipos". No contexto da Computação Gráfica, algumas restrições dessa definição são relaxadas, resultando em "uma cobertura sem buracos ou sobreposições de uma superfície planar ou curva, finita, por triângulos não necessariamente congruentes". Tesselações são úteis tanto quando aplicadas sobre superfícies genéricas, com o propósito de exibi-las, quanto quando sobre malhas de triângulos, com o propósito de adicionar detalhes a suas geometrias.

Primeiramente, é importante que seja definido o que são superfícies paramétricas e quais tipos são utilizados neste trabalho; com esse fim, a seção 2.1 contém uma explicação sucinta. Em seguida, para que os resultados da detecção de colisões sejam devidamente comprovados, é necessária uma maneira de visualizá-los. É aí que entra a exibição das superfícies, detalhada na seção 2.2; mais especificamente, são abordados os desafios apresentados pela tesselação de tais superfícies, tanto em CPU na seção 2.3 quanto em GPU na seção 2.4. Ao fim, na seção 2.5, são apresentados os resultados do processo de tesselação.

2.1 Superfícies paramétricas

O conteúdo desta seção é abordado de forma tangencial, já que a teoria da construção de superfícies paramétricas e de subdivisão não é o foco deste texto. Para mais detalhes, ver [11], onde os conceitos são formalmente definidos em detalhe.

2.1.1 Definição

Uma superfície paramétrica é um mapeamento de um espaço paramétrico bidimensional para o espaço Euclidiano tridimensional e é definida como

$$\mathbf{S}(u,v) = (x(u,v), y(u,v), z(u,v)),$$

em que x, y e z são funções que podem tomar qualquer forma, porém são geralmente polinomiais. As representações de superfícies nas quais este trabalho se interessa são comumente descritas por meio de pontos de controle, que são pontos quaisquer no espaço Euclidiano, e funções de base que ponderam os pontos de controle com base nas coordenadas uv do domínio paramétrico. Matematicamente,

$$\mathbf{S}(u,v) = \sum_i f_i(u,v)\mathbf{P}_i,\ 0 \leq u,v \leq 1, \tag{1}$$

em que $\mathbf{S}(u,v)$ é um ponto na superfície, f_i são funções de base quaisquer, e $\mathbf{P}_i$ são pontos de controle. O conjunto dos pontos de controle de uma superfície faz parte da topologia da malha ou poliedro de controle, juntamente a arestas e faces orientadas.

Há vários tipos de representações paramétricas de superfícies, que distinguem-se pelas escolhas de funções de base. Superfícies Bézier, por exemplo, utilizam polinômios de Bernstein, enquanto superfícies B-spline fazem uso de, como o nome sugere, funções B-spline.

Em algumas representações, como as Bézier, devido às escolhas de funções de base, uma alteração de um vértice qualquer da malha de controle provoca alterações na superfície inteira; portanto, diz-se que não há controle local. Representações B-spline, pelo outro lado, subdividem o domínio paramétrico em intervalos, e utilizam funções de base B-spline que, por natureza, fornecem maior controle local, dando a artistas maior flexibilidade ao esculpir superfícies.

Até este ponto, foi utilizado somente o termo "representação de superfície", porém, mais precisamente, a estrutura descrita por (1) é uma representação de um "retalho" de superfície. A representação da superfície em si é um conjunto de retalhos adjacentes.

Os retalhos B-spline são de especial importância para este trabalho, juntamente a outro tipo de retalho introduzido na seção seguinte. Assim, é válido que suas estruturas sejam brevemente descritas, já que o processo de tesselação envolve manipulá-las diretamente de modo a gerar as malhas de triângulos desejadas.

Um retalho B-spline é uma representação de superfície por meio de produto tensorial, isto é, as funções f_i de duas coordenadas paramétricas presentes em (1) são obtidas pelo produto de duas outras funções de uma só coordenada; especificamente no caso de retalhos B-spline bicúbicos – ou seja, que faz uso de funções de base cúbicas –, (1) se torna

$$\mathbf{S}(u,v) = \sum_{i=0}^{3}\sum_{j=0}^{3} B_{i,3}(u)B_{j,3}(v)\mathbf{P}_{i,j},$$

em que $B_{i,3}$ são as funções de base B-spline de ordem três – introduzidas em detalhe e discutidas por [11] – e $\mathbf{P}_{i,j}$ são dezesseis pontos de controle dispostos em uma grade bidimensional como ilustrado pela Figura 1.

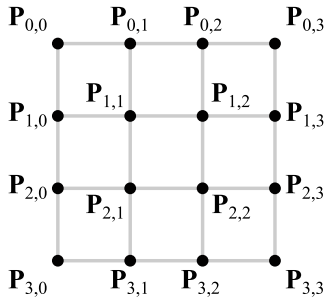

Figura 1: Pontos de controle de um retalho B-spline.

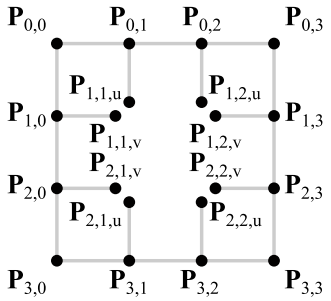

Figura 2: Pontos de controle de um retalho de Gregory.

2.1.2 Superfícies de subdivisão

Dada uma malha de controle, pode-se aplicar um processo iterativo denominado "subdivisão". Tal procedimento refina a malha, adicionando novos elementos – vértices, arestas e faces – e modificando os vértices já existentes. A cada passo de subdivisão, os vértices da malha se aproximam mais e mais de uma superfície limítrofe. A depender do método de subdivisão, tal superfície pode ter características diferentes; neste trabalho, consideramos apenas o método de subdivisão de Catmull-Clark, detalhado em [11].

Esse método gera malhas refinadas; se a malha de controle sobre a qual é aplicado o processo de subdivisão contiver vértices extraordinários, então a malha resultante também os conterá. Um vértice extraordinário é aquele cuja valência – número de arestas incidentes – é diferente de quatro. Isso resulta em faces – e retalhos – que são chamadas de irregulares e, consequentemente, têm sua avaliação afetada de forma a não permitir que retalhos B-spline sejam suficientes para devidamente representá-los. Assim, [11] introduz os retalhos de Gregory.

Retalhos de Gregory são retalhos especiais gerados a partir de retalhos B-spline; há a necessidade de, nesse processo de geração de tais retalhos, se utilizar pontos "fantasmas", que não pertencem, de fato, à malha de controle. Esses pontos podem ser gerados por meio de uma combinação linear dos pontos que já estão na malha; tal combinação pode ser representada de forma matricial, como descrito por [11], e é parte do processo utilizado pela API de código aberto OpenSubDiv [13], da qual o trabalho de [11] faz uso.

Assim como foi feito para retalhos B-spline, segue uma breve definição dos retalhos de Gregory, como em [17]:

$$\mathbf{S}(u,v) = \sum_{i=1}^{2}\sum_{j=1}^{2}\sum_{k\in\{\mathrm{u,v}\}} G_{i,j,k}(u,v)\mathbf{P}_{i,j,k} + \sum_{i=0}^{3}\sum_{j\in J_i} G_{i,j}(u,v)\mathbf{P}_{i,j},$$

em que $G_{i,j}$ e $G_{i,j,k}$ são as funções de base de Gregory – que são rigorosamente definidas e podem ser encontradas nas obras de [11] e [17] –, $\mathbf{P}_{i,j}$ e $\mathbf{P}_{i,j,k}$ são vinte pontos de controle dispostos como a Figura 2 ilustra, e

$$J_i = \begin{cases} \{0,1,2,3\} & \text{se } i \equiv 0 \pmod 3, \\ \{0,3\} & \text{caso contrário.} \end{cases}$$

Há, também, o caso de fronteiras e vincos, onde a continuidade das superfícies é quebrada – por exemplo, as quinas de um cubo, ou o gume de uma lâmina –. Há, novamente, a necessidade de gerar pontos extras a partir dos já existentes na malha; [13] e, consequentemente, [11] utilizam para a geração desses pontos o mesmo conceito matricial aplicado aos retalhos de Gregory.

2.2 Exibindo superfícies paramétricas

Placas gráficas são fortemente otimizadas para exibir primitivos gráficos – pontos, linhas e triângulos – em uma tela, por meio de um processo denominado rasterização (do Inglês *rasterization*) [6]. Como mencionado na seção 2.1, é evidente que superfícies suaves não são compostas por tais primitivos, consequentemente requerendo um processo de conversão antes que possam ser devidamente exibidas. Tal conversão consiste em gerar uma malha de primitivos – neste caso, triângulos – que representa a superfície com um grau arbitrário de precisão; esta malha é uma tesselação da superfície.

2.3 Tesselação em CPU

O processo de tesselação em CPU consiste em sequencialmente percorrer o domínio paramétrico dos retalhos da superfície, em ambas as direções u e v, em incrementos regulares Δu e Δv tais que $0 < \Delta u, \Delta v < 1$. A cada incremento, são calculadas as propriedades – nominalmente, a posição e o vetor normal – do retalho naquele ponto utilizando suas respectivas funções de base e pontos de controle. Quanto menores os valores de Δu e Δv, maior a densidade poligonal da malha de triângulos resultante da tesselação; dá-se o nome de "níveis de tesselação" aos números $1/\Delta u$ e $1/\Delta v$, que representam a quantidade de incrementos em cada direção.

Há três tipos de retalhos a serem considerados no processo de tesselação utilizado neste trabalho: retalhos B-spline regulares e de fronteira e retalhos de Gregory. Enquanto retalhos B-spline regulares podem seguir o processo diretamente, retalhos de fronteira e de Gregory fazem uso de pontos fantasmas, ou seja, há uma etapa intermediária que envolve utilizar as matrizes mencionadas na seção 2.1.2 para gerar os pontos extras e, então, encaminhá-los ao processo de tesselação juntamente ao retalho a que pertencem. Note que as matrizes são armazenadas na estrutura que representa a superfície; este trabalho não se encarrega de construir as superfícies e, tampouco, as matrizes.

2.4 Tesselação em OpenGL

Como mencionado na seção 1.2, é utilizada neste trabalho a API gráfica OpenGL. Nessa API, o *pipeline* gráfico é composto por vários estágios, sendo alguns deles programáveis e chamados de *shaders*. De forma simplificada, tais estágios são organizados como mostra a Figura 3. Cada estágio recebe dados do estágio anterior e produz novos dados que são passados ao estágio seguinte. Para uma aplicação gráfica funcional, não é necessário que todos os *shaders* sejam fornecidos; são comumente empregados os *shaders* de vértices – cujo propósito é determinar a posição de cada vértice da geometria da cena no espaço – e de fragmentos – cujo objetivo é determinar a cor de um fragmento de geometria que pode, eventualmente, se tornar um *pixel* em tela –. No presente caso há, adicionalmente, o emprego dos *shaders* de controle e avaliação de tesselação.

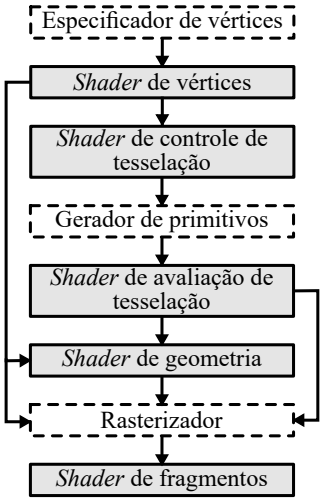

Figura 3: *Pipeline* gráfico simplificado da OpenGL com os estágios programáveis destacados.

Assim como muitos processos em Computação Gráfica, a tesselação pode ser acelerada por *hardware*; GPUs modernas possuem unidades especializadas para tesselar superfícies arbitrárias de maneira extremamente eficiente.

A maneira pela qual isso é exposto ao programador se dá na forma de *shaders* de tesselação fornecidos pela API gráfica escolhida. No caso da OpenGL, como mencionado, há os *shaders* de controle de tesselação (TCS, do Inglês *Tessellation Control Shader*) e de avaliação de tesselação (TES, do Inglês *Tessellation Evaluation Shader*). Ambos cumprem papéis cruciais para a transformação das superfícies de entrada, sejam elas quais forem, em malhas de triângulos.

2.4.1 TCS

O TCS é responsável por gerar a topologia da malha resultante da tesselação de uma superfície. Na prática, isso consiste em tomar como entrada um retalho de superfície – que pode ser tanto paramétrica quanto um simples triângulo –, e níveis de tesselação – que determinam a granularidade com que a superfície será subdividida em primitivos gráficos – e fornecer como saída um conjunto de pares de coordenadas *uv* correspondentes a pontos intermediários do retalho de entrada. Esses pontos são puramente simbólicos a essa altura da *pipeline*, correspondendo a simples pares *uv*, mas se tornam efetivamente concretos no estágio de avaliação. A conectividade desses pontos – ou seja, como eles formam triângulos – é denominada "geração de primitivos" e é determinada por uma unidade de *hardware* que é empregada logo após a execução do TCS, de forma transparente ao desenvolvedor.

O TCS é invocado n vezes para cada retalho, sendo n seu número de vértices; no caso de retalhos paramétricos, n corresponde ao número de pontos de controle. Em cada invocação, acontece uma de duas coisas. Se o retalho for regular, então o TCS é um *shader* "de passagem", ou seja, nele só são definidos os níveis de tesselação – que são passados via variável de entrada uniforme – e encaminhadas ao TES as posições dos pontos de controle[1]. Se o retalho for irregular – ou seja, se for um retalho B-spline de fronteira ou um retalho de Gregory –, o TCS faz o pré-processamento do retalho por meio da multiplicação da matriz correspondente, mencionada na seção 2.1.2, pelos pontos de controle, gerando os pontos fantasmas, também discutidos na seção 2.1.2. As matrizes dos retalhos são acessadas por meio de um único *buffer* contíguo em memória; cada retalho conhece o índice correspondente ao início de sua matriz nesse *buffer*. Ao fim da multiplicação, os novos dados são encaminhados ao estágio seguinte da *pipeline*: o TES. A fim de acelerar a geração dos pontos fantasma, foi utilizado o esquema de invocação do TCS para paralelizar a multiplicação matricial.

2.4.2 TES

O TES toma como entrada os pares *uv* resultantes da execução do TCS e atribui a eles propriedades concretas, como posições e vetores normais, assim gerando, de fato, uma malha de triângulos pronta para a rasterização. É de responsabilidade do desenvolvedor calcular tais grandezas.

Há, na aplicação, dois *shaders* de avaliação de tesselação: um para retalhos B-spline e outro para retalhos de Gregory. O TES é invocado uma vez para cada par *uv* gerado pelo estágio anterior da *pipeline*. Em cada invocação, basta adaptar (1) para obter a posição no espaço Euclidiano do ponto (u,v) do domínio paramétrico, e utilizar as primeiras derivadas parciais de $f_i(u,v)$ nas direções de u e v para obter dois vetores tangentes à superfície em (u,v) e, utilizando o produto vetorial, obter o vetor normal ao retalho naquele ponto.

Ao fim da execução do TES, a *pipeline* segue seu caminho, passando pelo rasterizador e pelo *shader* de fragmentos, completando o processo que permite a exibição de uma superfície paramétrica em tela com grau arbitrário de precisão.

2.5 Resultados

A tesselação em CPU é efetuada somente uma vez ao início da exibição da cena, sendo a malha resultante armazenada e utilizada em *frames* subsequentes. A tesselação em GPU,

[1] Vale notar que a geração de primitivos ainda é efetuada, mas sem que haja um pré-processamento extra como ocorre no caso de retalhos irregulares.

por outro lado, ocorre inerentemente em tempo real, ou seja, a superfície é tesselada a todo quadro da simulação. Isso significa que não há possibilidade de comparar a performance dos dois métodos. Entretanto, é possível comparar a triangulação das malhas.

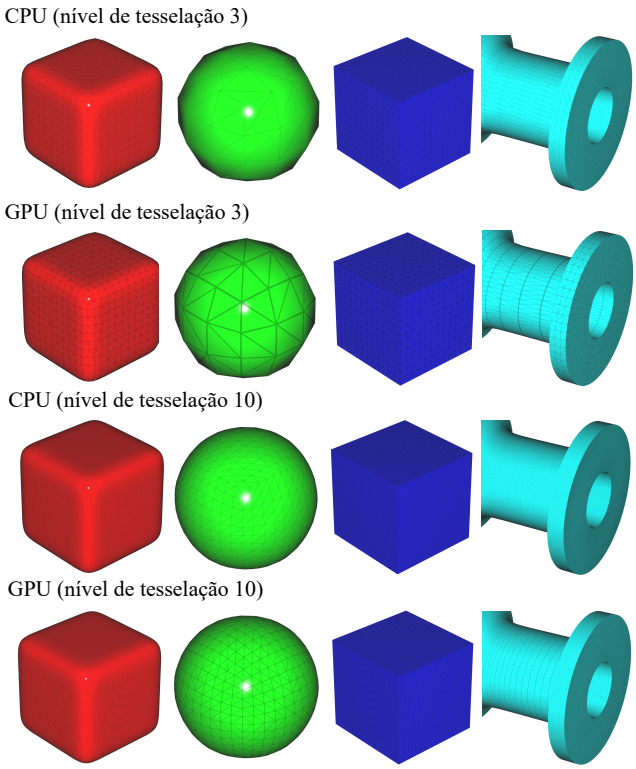

Figura 4: Tesselação de várias superfícies em CPU e GPU com dois níveis de tesselação.

Pode-se reparar, na Figura 4, que a tesselação em GPU gera primitivos de forma levemente irregular, já que não há controle algum do desenvolvedor sobre o padrão de acesso em paralelo da *pipeline* OpenGL em GPU, diferentemente do que ocorre na tesselação em CPU. De qualquer maneira, o resultado é satisfatório, já que o único requisito para esta etapa era simplesmente mover a tesselação do processador para a placa gráfica.

3 Detecção de colisão

Para determinar se dois corpos estão colidindo – isto é, se há pontos em suas superfícies nos quais há contato mútuo ou se têm partes de seus volumes em intersecção –, é necessário conhecer qual a representação utilizada para descrever tais corpos; afinal, uma abordagem desenvolvida para detectar intersecções entre NURBS é bastante diferente de uma utilizada com malhas de triângulos. Tendo isso em vista, sempre que houver menção à detecção de colisões neste documento deste ponto em diante, trata-se especificamente de colisões entre malhas de triângulos, a não ser que o contrário seja explicitamente mencionado.

Em primeiro lugar, é importante reforçar que, enquanto há métodos para se detectar colisões de forma analítica entre duas superfícies de subdivisão representadas de forma paramétrica, o método escolhido para este trabalho consiste em utilizar as malhas resultantes do processo de tesselação juntamente a algoritmos conhecidos de intersecções entre primitivos – nominalmente, triângulos – para efetuar a detecção de colisão de forma aproximada. Assim, é apresentado na seção 3.1 o conceito de intersecções entre triângulos. Isso é seguido, na seção 3.2, por uma argumentação a favor de estruturas e processos que aceleram a detecção de intersecções entre malhas de triângulos. A seção 3.2.1 se adentra nos procedimentos utilizados na aceleração, enquanto as seções 3.2.2 e 3.2.3 detalham algumas estruturas espaciais empregadas na aceleração. Em seguida, as seções 3.2.4, 3.2.5 e 3.2.6 detalham a metodologia adotada por este trabalho, que faz uso dos procedimentos e estruturas mencionadas. A seção 3.3 descreve a simulação de corpos rígidos extremamente básica utilizada para demonstrar o funcionamento da detecção de colisões. Finalmente, a seção 3.4 une todas as etapas de modo a descrever o algoritmo completo de detecção de colisões e a resposta física às colisões detectadas.

3.1 Intersecção de triângulos

Há múltiplas estratégias para determinar se dois triângulos se intersectam. É adotado, aqui, o método proposto por Devillers e Guigue [4], que é referenciado pela obra de Ericson voltada à detecção de colisões em tempo real [5]. Segue uma breve descrição textual do algoritmo, adaptada diretamente de [4].

Primeiramente, dados dois triângulos $T_1 = (\mathbf{P}_1, \mathbf{Q}_1, \mathbf{R}_1)$ *e* $T_2 = (\mathbf{P}_2, \mathbf{Q}_2, \mathbf{R}_2)$*, checa-se a intersecção de um triângulo com o plano do outro. A priori, o triângulo* T_1 *é testado contra o plano* π_2 *de* T_2*. Para tal, o algoritmo classifica os vértices de* T_1 *em relação a* π_2 *ao comparar os sinais das três determinantes* $[\mathbf{P}_2, \mathbf{Q}_2, \mathbf{R}_2, \mathbf{P}_1]$*,* $[\mathbf{P}_2, \mathbf{Q}_2, \mathbf{R}_2, \mathbf{Q}_1]$ *e* $[\mathbf{P}_2, \mathbf{Q}_2, \mathbf{R}_2, \mathbf{R}_1]$*. Três situações distintas são possíveis: a) todas as três determinantes têm o mesmo sinal e nenhuma é zero, b) todas as três determinantes são zero, e c) as determinantes têm sinais diferentes. O caso a) ocorre quando todos os três vértices de* T_1 *se encontram no mesmo subespaço criado por* π_2*. Nesse caso, não pode haver intersecção. O caso b) ocorre quando os triângulos são coplanares e, consequentemente, definem o mesmo plano; esse caso especial é tratado utilizando um algoritmo de intersecção bidimensional. Por fim, o caso c) ocorre quando os vértices de* T_1 *estão em lados diferentes de* π_2 *e* T_1 *certamente intersecta* π_2*; nesse caso, é checado se* T_2 *intersecta o plano* π_1 *de* T_1 *da mesma maneira.*

Se o par de triângulos passa nesses testes e se for assumido que estão em "posição geral" (isto é, se nenhum vértice de um triângulo é coplanar ao outro triângulo), cada triângulo tem exatamente um vértice em um lado do plano do outro triângulo, e dois vértices do outro lado. O algoritmo aplica, então, uma permutação circular aos vértices de cada triângulo tal que $\mathbf{P}_1$ *(e, respectivamente,* $\mathbf{P}_2$*) é o único vértice de seu triângulo que se encontra em um dos lados do plano do outro triângulo. Uma operação de transposição adicional é efetuada concomitantemente nos vértices* $\mathbf{Q}_2$ *e* $\mathbf{R}_2$ *(e, respectivamente,* $\mathbf{Q}_1$ *e* $\mathbf{R}_1$*) de forma a mapear* $\mathbf{P}_1$ *(e,*

respectivamente, $\mathbf{P}_2$) ao subespaço positivo induzido por π_2 (e, respectivamente, π_1). Nota-se que tal reorganização existe para todas as possíveis configurações exceto quando dois dos vértices do triângulo estão em um lado do plano e o terceiro está no plano em si. Tais casos são tratados ao permutar os vértices do triângulo de tal forma que $\mathbf{P}_1$ (e, respectivamente, $\mathbf{P}_2$) é o único vértice que não se encontra no subespaço negativo induzido por π_2 (e, respectivamente, π_1).

Devido a todas essas rejeições e permutações, é garantido que cada aresta incidente nos vértices $\mathbf{P}_1$ e $\mathbf{P}_2$ intersecta L em um ponto único, com L sendo a reta correspondente à intersecção de π_1 e π_2. Sejam i, j, k e l as coordenadas paramétricas que definem os pontos de intersecção de L com as arestas $\mathbf{P}_1\mathbf{R}_1$, $\mathbf{P}_1\mathbf{Q}_1$, $\mathbf{P}_2\mathbf{Q}_2$ e $\mathbf{P}_2\mathbf{R}_2$, respectivamente. Esses pontos de intersecção formam intervalos fechados I_1 e I_2 em L que correspondem à intersecção dos dois triângulos e L. Ademais, a essa altura, há informação suficiente para conhecer uma ordem consistente dos limites de cada intervalo. Precisamente, $I_1 = [i, j]$ e $I_2 = [k, l]$ se L estiver orientada na direção de $\mathbf{n} = \mathbf{n}_1 \times \mathbf{n}_2$, em que $\mathbf{n}_1 = (\mathbf{Q}_1 - \mathbf{P}_1) \times (\mathbf{R}_1 - \mathbf{P}_1)$ e $\mathbf{n}_2 = (\mathbf{Q}_2 - \mathbf{P}_2) \times (\mathbf{R}_2 - \mathbf{P}_2)$. Assim, só é necessário checar uma condição mín./máx. para determinar se os dois intervalos se intersectam; com "condição mín./máx.", quer-se dizer que o mínimo de cada intervalo deve ser menor que o máximo do outro. Nominalmente, $k \leq j$ e $i \leq l$.

Já que as arestas $\mathbf{P}_1\mathbf{R}_1$, $\mathbf{P}_1\mathbf{Q}_1$, $\mathbf{P}_2\mathbf{Q}_2$ e $\mathbf{P}_2\mathbf{R}_2$ intersectam L, então testar essa condição se reduz a checar o predicado

$$[\mathbf{P}_1, \mathbf{Q}_1, \mathbf{P}_2, \mathbf{Q}_2] \leq 0 \wedge [\mathbf{P}_1, \mathbf{R}_1, \mathbf{R}_2, \mathbf{P}_2] \leq 0.$$

Ao fim da execução do algoritmo, sabe-se que o segmento de intersecção entre dois triângulos T_1 e T_2 é definido pela sobreposição de dois intervalos paramétricos I_1 e I_2; basta substituir os limites dessa sobreposição na equação paramétrica da reta de intersecção L para que sejam obtidos dois pontos $\mathbf{A}$ e $\mathbf{B}$ no espaço Euclidiano que definem o segmento de intersecção entre T_1 e T_2. Neste trabalho, adotou-se o ponto $(\mathbf{A}+\mathbf{B})/2$ como um ponto único de intersecção, ao invés de levar em conta o segmento inteiro. Além disso, o vetor normal considerado, aqui, é $\mathbf{n}_2$, correspondendo a T_2.

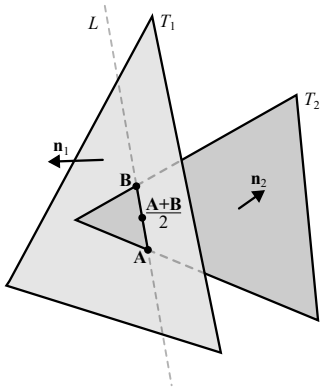

Figura 5: Intersecção de dois triângulos via [4].

A Figura 5 contém uma representação visual desses aspectos. Vale notar que o vetor normal correto não necessariamente corresponde a $\mathbf{n}_2$ em todos os casos, visto que devem ser levadas em conta várias características da intersecção para que o vetor normal fisicamente correto seja determinado; assim, a simulação de corpos rígidos detalhada na seção 3.3 não é capaz de replicar fielmente o comportamento dos objetos, já que é requerido, ali, o vetor normal certo. De qualquer forma, isso não afeta o processo da detecção da colisão em si, mantendo o trabalho alinhado com os objetivos propostos.

3.2 Acelerando a detecção

Dado um conjunto C de malhas de triângulos, com a i-ésima malha tendo n_i triângulos, determinar se há colisão entre duas delas envolveria testar todas as malhas, duas a duas, a fim de encontrar um par cujos triângulos se interceptam. Isso implica em $|C|(|C|-1)/2$ comparações entre malhas; cada uma dessas comparações demandaria testar todos os triângulos das duas malhas, i-ésima e j-ésima, envolvidas, dois a dois, para descobrir se algum par de triângulos se intercepta, resultando em $n_i n_j$ comparações para cada par de malhas. Assim, nessa cena, teríamos $|C|(|C|-1)n_i n_j/2$ testes de intersecção entre triângulos. É evidente que isso se torna inviável conforme o número de malhas e/ou suas densidades poligonais crescem. Assim, faz-se necessário o emprego de uma técnica para descartar candidatos a colisões cujas intersecções não podem, de fato, ocorrer. Um exemplo se daria na forma de malhas que estão bem longe uma da outra, não permitindo possibilidade alguma de haver uma intersecção entre elas.

3.2.1 Fases da detecção de colisões

Na maioria dos motores de física, detecções de colisões são implementadas em duas fases: ampla e estreita, do inglês *broad and narrow phases*. A fase ampla consiste no descarte supracitado de malhas candidatas que efetivamente não podem colidir. Isso reduz a quantidade de pares de malhas cujos triângulos serão testados, resultando em uma melhora perceptível de desempenho na maioria dos casos. A fase estreita, por sua vez, é composta pelos testes de primitivos detalhados na seção 3.1.

3.2.2 Caixa limitante

Uma caixa limitante (do Inglês *bounding box*) é, como o nome sugere, uma caixa que envolve determinado objeto; no presente caso, malhas de triângulos. O processo de determinar intersecções de caixas limitantes é relativamente barato computacionalmente, então, envolver cada malha em uma dessas caixas resultaria em um descarte bem mais rápido de candidatos improváveis; basta testar todas as caixas limitantes, duas a duas, a fim de determinar se algum par se intercepta. Se sim, então efetua-se os testes com os triângulos das malhas cujas caixas possuem intersecção, já que há chance de haver colisão entre as malhas em si. Se duas caixas limitantes não se interceptam, então não há possibilidade de os objetos nelas contidos o fazer.

Há alguns tipos de estruturas limitantes (do Inglês *bounding structures*), das quais caixas limitantes compõem so-

mente um subconjunto. Já as caixas limitantes, por sua vez, possuem duas variações: caixas limitantes alinhadas aos eixos (AABBs, do Inglês *axis-aligned bounding boxes*) e caixas limitantes orientadas (OBBs, do Inglês *oriented bounding boxes*). Dado um sistema de coordenadas ortogonal, AABBs são prismas retangulares retos que envolvem determinada malha e cujos eixos se alinham com os vetores da base do sistema. OBBs, no entanto, não possuem restrição alguma sobre seus eixos; o prisma pode estar orientado em qualquer direção, permitindo que melhor aproxime a malha em si contida.

Há benefícios que AABBs possuem sobre OBBs, e vice-versa. AABBs precisam ser computadas novamente se a malha contida for rotacionada, mas não se for escalada ou transladada. OBBs, no entanto, não precisam ser recalculadas ao transladar ou girar a malha, mas sim ao escalá-la. Testes com AABBs são simples, enquanto testes com OBBs são computacionalmente intensivos. Além disso, é importante fixar que a computação de uma OBB a partir de uma malha é mais complexa que a de uma AABB. Porém, como OBBs são, em geral, mais compactas que AABBs, seu uso possivelmente descartaria mais candidatos de colisões que um processo equivalente utilizando AABBs. Neste trabalho, optou-se pelo uso de somente AABBs.

3.2.3 Hierarquia de volumes

Enquanto o uso de caixas limitantes por si só já provê, teoricamente, uma aceleração a um algoritmo de detecção de colisão, os problemas mencionados no início da seção 3.2 se mantêm: conforme argumentado, em uma cena com n objetos, será necessária uma quantidade de testes entre AABBs proporcional a n^2. Assim, é benfazejo que seja adotada uma estratégia para reduzir o número de comparações na fase ampla. Há várias abordagens que satisfazem essa necessidade: partição binária do espaço [16], *Octrees* [9], árvores de volumes limitantes (BVHs, do Inglês *Bounding Volume Hierarchy*), entre outros.

Neste trabalho, optou-se por utilizar árvores de volumes limitantes. Essas estruturas podem ser dinâmicas (DBVHs, do Inglês *Dynamic BVHs*) – com "dinâmicas", quer-se dizer que os objetos nela contidos podem ter suas posições e orientações modificadas após a inicialização da árvore, e ela se adapta de acordo – e estáticas; ambos os tipos foram empregados, para estágios distintos da detecção de colisão. As árvores dinâmicas foram definidas por Catto [2].

3.2.4 Pré-processamento

Como detalhado no início da seção 3.2, testar duas malhas para determinar intersecções resulta em um número de comparações entre triângulos que cresce de forma quadrática com a quantidade de polígonos em ambas as malhas. Para evitar a perda de desempenho que esse fato pode proporcionar, a depender da densidade poligonal das malhas envolvidas, é benéfico descartar no processo da fase ampla não somente malhas inteiras, mas também partes – doravante denominadas *meshlets* – das malhas que estão longe demais uma da outra e que, consequentemente, não podem colidir.

Essa "quebra" de uma malha em *meshlets* é executada para cada malha utilizada por pelo menos um objeto de cena no momento em que um corpo rígido é acoplado a ele, antes do início da simulação. O processo de quebra resulta em uma BVH para cada malha, cujas folhas contêm, cada uma, informações sobre um *meshlet* específico. O procedimento de geração de *meshlets* é ilustrado na Figura 6. Na prática, a malha é subdividida em *meshlets* utilizando heurísticas para tal descritas por [14]. As BVHs resultantes da quebra são estáticas, já que representam corpos rígidos cujas geometrias permanecerão constantes durante o decorrer da simulação; caso fosse desejado simular corpos deformáveis que, por definição, têm uma estrutura modificável, então essa abordagem não seria adequada.

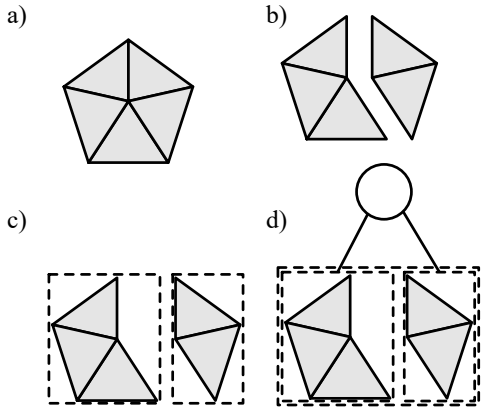

Figura 6: Quebra de malhas. a) malha de entrada, b) separação em *meshlets*, c) geração de AABBs para cada *meshlet*, d) construção da BVH.

No passo seguinte do pré-processamento, as BVHs de cada malha são encapsuladas em corpos rígidos que são adicionados, como folhas, a uma DBVH única que representa a cena como um todo; a construção da DBVH é baseada na heurística de área de superfície (SAH, do Inglês *Surface Area Heuristic*) [8]. A estrutura de aceleração final é, então, uma DBVH de BVHs. É importante enfatizar que há só uma BVH para cada malha da cena; mesmo que haja várias instâncias – ou seja, atores/objetos de cena – que utilizem uma mesma malha, existirá somente uma BVH correspondente, que será mantida constante e inalterável durante todo o decorrer da simulação. Essa abordagem adiciona uma complicação: dois objetos diferentes cujas geometrias de colisão são determinadas pela mesma malha podem, naturalmente, estar em locais diferentes no espaço e possuir orientações distintas, isto é, podem conter matrizes de transformação diferentes. Se a mesma BVH é utilizada para ambos, então a BVH também deve ser transformada pelas matrizes de transformação; enquanto translações não invalidam AABBs, rotações em torno dos eixos do sistema de coordenadas por ângulos que não são múltiplos inteiros de $\pi/2$ convertem AABBs em OBBs, já que o alinhamento aos três eixos é quebrado. Assim, pode-se adotar uma de três abordagens para lidar com essa situação:

1. cessar o uso de AABBs, as substituindo por OBBs em todas as ocasiões, permitindo que BVHs sejam giradas sem invalidar as caixas limitantes nela contidas;

2. no momento em que um teste entre duas BVHs for necessário, recomputar ambas as BVHs após aplicar as matrizes de transformação distintas às duas malhas, levando em conta as novas posições e orientações; ou

3. no momento em que um teste entre duas BVHs for necessário, aplicar as matrizes de transformação nas BVHs, convertendo as AABBs em OBBs, gerando em seguida, em tempo real, novas AABBs, cada uma contendo uma OBB resultante da transformação.

É adotada, aqui, a terceira opção. Isso é ilustrado na Figura 7: um objeto qualquer, representado ali por uma elipse, está contido em uma AABB. Quando o objeto é transformado, pode-se recomputar a AABB do zero – ou seja, utilizando a opção 2 –, resultando na AABB da porção inferior-esquerda da figura; no entanto, pode-se optar por transformar a AABB inicial, obtendo uma OBB, posteriormente gerando uma nova AABB a partir dessa OBB, como é mostrado na porção inferior-direita da figura.

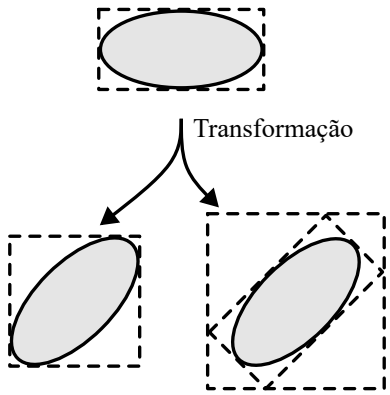

Figura 7: Rotação de AABBs.

É evidente que a abordagem 3 resulta em AABBs levemente infladas quando comparadas às geradas pela opção 2, já que elas contêm não somente os objetos em questão, mas também as OBBs resultantes da rotação das AABBs que encobrem tais objetos. No que se refere à detecção de colisões, caixas limitantes menos justas resultam em mais falsos positivos, ou seja, casos em que as caixas se interceptam sem que os objetos aos quais elas pertencem o façam; isso segue do fato de haver mais espaço vazio dentro das caixas. Essa consequência afeta diretamente a qualidade da fase ampla de forma negativa, já que o número de pares descartados será menor. Espera-se, no entanto, que o detrimento decorrente de AABBs maiores seja amenizado tanto pela quebra das malhas em *meshlets* quanto pela evitação da recomputação da BVH sempre que o corpo ao qual ela pertence for transformado. Computar uma BVH, nesse contexto, envolve calcular várias AABBs para uma malha de triângulos; gerar a AABB de uma malha poligonal é uma operação de complexidade proporcional ao número de vértices da malha – algo que pode ser custoso dependendo de sua densidade –, enquanto gerar uma AABB a partir de uma OBB leva tempo constante, já que há somente oito vértices por OBB.

3.2.5 Fase ampla

A fase ampla utilizada é adaptada diretamente de uma das abordagens empregadas no motor de física Bullet [3]. Nela, a implementação da fase ampla envolve percorrer uma DBVH, testando os nós de forma recursiva – ou, como foi feito aqui, utilizando uma pilha explícita – até que as caixas limitantes de duas folhas se interceptem; quando isso acontece, um procedimento de *callback* é executado para lidar com a colisão entre as caixas. A descrição desse processo é apresentada no Algoritmo 1.

Algoritmo 1 Colisão entre duas árvores de volumes limitantes t_1 e t_2, com *callback* f de detecção de contato

```
1:  função ColidirÁrvores(t1, t2, f)
2:      se t1 ou t2 estiver vazia então
3:          retorne
4:      fim se
5:      r1 ← raiz de t1
6:      r2 ← raiz de t2
7:      declare uma pilha p
8:      empilhe (r1, r2) em p
9:      enquanto p não estiver vazia faça
10:         desempilhe (n1, n2) de p
11:         se n1 = n2 então
12:             se n1 não for uma folha então
13:                 c11 ← primeiro filho de n1
14:                 c12 ← segundo filho de n1
15:                 empilhe (c11, c11) em p
16:                 empilhe (c12, c12) em p
17:                 empilhe (c11, c12) em p
18:             fim se
19:         senão se n1 e n2 se sobrepuserem então
20:             se n1 não for uma folha então
21:                 c11 ← primeiro filho de n1
22:                 c12 ← segundo filho de n1
23:                 se n2 não for uma folha então
24:                     c21 ← primeiro filho de n2
25:                     c22 ← segundo filho de n2
26:                     empilhe (c11, c21) em p
27:                     empilhe (c12, c21) em p
28:                     empilhe (c11, c22) em p
29:                     empilhe (c12, c22) em p
30:                 senão
31:                     empilhe (c11, n2) em p
32:                     empilhe (c12, n2) em p
33:                 fim se
34:             senão
35:                 se n2 não for uma folha então
36:                     c21 ← primeiro filho de n2
37:                     c22 ← segundo filho de n2
38:                     empilhe (n1, c21) em p
39:                     empilhe (n1, c22) em p
40:                 senão
41:                     f(n1, n2)
42:                 fim se
43:             fim se
44:         fim se
45:     fim enquanto
```

46: **fim função**

Vale reparar que a função COLIDIRÁRVORES toma como entrada duas árvores ao invés de somente uma; isso é porque ela pode, se necessário, ser utilizada para detectar colisões entre duas BVHs diferentes. Tal comportamento permite que ela seja utilizada quando tanto t_1 quanto t_2 se referem à mesma árvore: a DBVH da cena, e também quando t_1 e t_2 correspondem a duas BVHs diferentes; o que muda, além das árvores, é o parâmetro f que corresponde à *callback* de detecção.

Após determinar que, em um instante da simulação, dois objetos de cena têm suas AABBs em intersecção, é necessário percorrer as BVHs de ambos a fim de determinar se há intersecção entre as AABBs de algum par de *meshlets* pertencentes às BVHs. Isso é concretizado por meio do uso do mesmo Algoritmo 1, agora empregado a fim de percorrer duas BVHs ao invés de somente uma DBVH. Nesse percurso, caso seja determinado que a caixa limitante de uma folha de uma das BVHs se intercepta com a caixa de uma folha da outra – tais folhas correspondendo, cada uma, a um *meshlet* – então o processo de detecção de colisão é encaminhado à ultima etapa: os testes de primitivos que compõem a fase estreita, realizados sobre os triângulos dos dois *meshlets*.

3.2.6 Fase estreita

Quando as AABBs de dois *meshlets* de objetos diferentes se interceptam, o último passo para determinar se há, de fato, uma colisão entre os dois objetos aos quais os *meshlets* pertencem é efetuar testes de intersecção entre cada par de triângulos dos dois *meshlets*. Primeiramente, percorre-se todos os triângulos dos dois *meshlets*, checando, via força bruta, par por par; para cada um desses pares, é executado o algoritmo de [4], descrito na seção 3.1. Caso algum teste seja positivo, são obtidas as informações da intersecção entre os dois triângulos envolvidos, que são encaminhadas à simulação de corpos rígidos. O efeito de utilizar uma BVH para cada malha no algoritmo de detecção se dá na forma da drástica redução no número de testes entre primitivos que precisam ser realizados; enquanto um objeto de cena pode ter sua geometria composta por milhões de triângulos, um *meshlet* não terá mais de sessenta e quatro triângulos, como determinado pela aplicação. Ou seja, o processamento máximo a ser realizado pela *narrow-phase* é $64^2 = 4096$ testes de primitivos; tais testes são definidos pelo algoritmo de [4] e estão descritos na seção 3.1. Por fim, se uma intersecção entre dois triângulos for encontrada, então os dados da intersecção – nominalmente o ponto, a normal e o par de corpos aos quais os triângulos pertencem – serão marcados por meio de uma *flag* booleana para que, ao fim do passo de simulação, tenham suas propriedades atualizadas de acordo com algum tipo de resposta à colisão.

3.3 Aplicando física de corpos rígidos

Após detectar uma colisão, a fim de demonstrar que os resultados obtidos estão corretos, é necessário que haja uma resposta correspondente. Foi decidido que, para servir tal propósito, seria implementado um motor de física de corpos rígidos elementar.

O modelo de resposta adotado é descrito por [7], e consiste na aplicação de impulsos de modo a alterar o momento linear e angular de corpos rígidos. Primeiramente, são atreladas a cada objeto de cena propriedades que representam seu estado na simulação física: massa, velocidade linear, velocidade angular e tensor de inércia; tais grandezas compõem, juntamente à geometria do objeto, o que é chamado de um "corpo rígido".

Em seguida, no *loop* da simulação, caso haja alguma colisão, então é executada uma sub-rotina responsável por modificar as propriedades de corpos rígidos dos dois objetos colidentes de forma a respeitar a conservação de momento linear, ou seja,

$$\begin{aligned} \mathbf{v}_1' &= \mathbf{v}_1 + Jm_1^{-1}\mathbf{n}, \text{ e} \\ \mathbf{v}_2' &= \mathbf{v}_2 - Jm_2^{-1}\mathbf{n}, \end{aligned} \tag{2}$$

em que $\mathbf{v}_i'$ é a velocidade do i-ésimo objeto colidente após a colisão, $\mathbf{v}_i$ é a velocidade do i-ésimo objeto colidente antes da colisão, m_i é a massa do i-ésimo objeto colidente, $\mathbf{n}$ é o vetor normal à colisão, Δt é o passo de tempo da simulação, e J é o impulso resultante da colisão dado por

$$J = \frac{(1+e)(\mathbf{v}_2 - \mathbf{v}_1)\cdot\mathbf{n}}{m_1^{-1} + m_2^{-1} + \mathbf{w}\cdot\mathbf{n}}, \tag{3}$$

com $e \in [0,1]$ sendo um número chamado de "coeficiente de restituição", que determina a elasticidade da colisão, sendo que $e = 0$ descreve uma interação totalmente inelástica, enquanto $e = 1$ retrata uma completamente elástica, e

$$\mathbf{w} = \mathbf{I}_1^{-1}(\mathbf{r}_1 \times \mathbf{n}) \times \mathbf{r}_1 + \mathbf{I}_2^{-1}(\mathbf{r}_2 \times \mathbf{n}) \times \mathbf{r}_2,$$

em que $\mathbf{I}_i$ é o tensor de inércia do i-ésimo objeto colidente e $\mathbf{r}_i$ é vetor do centro de massa do i-ésimo objeto colidente até o ponto de colisão. A velocidade angular ω dos corpos rígidos também é atualizada, seguindo os mesmos princípios:

$$\begin{aligned} \omega_1' &= \omega_1 + \mathbf{I}_1^{-1}(\mathbf{r}_1 \times J\mathbf{n}), \text{ e} \\ \omega_2' &= \omega_2 - \mathbf{I}_2^{-1}(\mathbf{r}_2 \times J\mathbf{n}). \end{aligned} \tag{4}$$

Vale reparar que, em (2), a velocidade do segundo objeto é modificada na direção oposta ao vetor $\mathbf{n}$. Na seção 3.1, foi mencionado que o segundo vetor normal é utilizado, e não o primeiro; esse é o motivo: se $\mathbf{n}_1$ fosse o vetor escolhido, então (2), (3) e (4) teriam de ser adaptadas.

3.4 Resumo

Segue um breve passo-a-passo descrevendo como as diferentes etapas do processo se interligam.

1. No início, uma cena é criada.
2. Quando um ator é adicionado à cena, sua geometria deve ser oriunda da descrição de uma superfície paramétrica composta de retalhos B-spline, B-spline de fronteira e de Gregory; essa superfície é, então, tesselada em CPU como mencionado na seção 2.3, sendo o resultado dessa tesselação armazenado em memória.

3. Quando um corpo rígido é acoplado a um ator, algumas propriedades são calculadas – nominalmente, seu centro de massa e seu tensor de inércia – para que sejam utilizadas na simulação de corpos rígidos; além disso, é gerada e armazenada em memória uma BVH, como descrito na seção 3.2.4, para a malha resultante da tesselação da superfície paramétrica que representa a geometria do ator – se já houver uma BVH correspondente à malha em questão neste momento, não é gerada outra BVH.
4. Quando a simulação de corpos rígidos é iniciada, cada corpo rígido tem seu estado inicial armazenado; em sequência, a DBVH da cena é construída com as informações das BVHs, correspondentes às malhas que definem a geometria de cada corpo rígido, transformadas – como descrito na seção 3.2.4 – pelas matrizes de transformação dos atores a que pertencem. Por fim, as seguintes etapas são executadas a cada passo de tempo até o fim da simulação.
   (a) A fase ampla é executada, seguindo o Algoritmo 1 sobre a DBVH da cena.
   (b) Se a fase ampla determinar que há um par de *meshlets* cujas caixas limitantes estão em contato, então a fase estreita é imediatamente executada.
   (c) Se a fase estreita determinar que há um par de triângulos em intersecção, então são calculados impulsos – conforme explicado na seção 3.3 – que serão aplicados a ambos os corpos rígidos cujas geometrias são representadas pelas malhas que contêm os triângulos em questão, de modo a fornecer uma resposta visual à colisão.
   (d) A DBVH é reconstruída do zero, levando em conta as novas posições e orientações dos corpos rígidos, como feito no passo 4; as BVHs, no entanto, continuam inalteradas. O sistema está, então, pronto para o início do próximo passo.
5. Ao fim da simulação, os corpos rígidos têm seus estados iniciais restaurados.

3.5 Resultados

Com o auxílio dos movimentos resultantes da aplicação do motor de física extremamente simples descrito na seção 3.3, pode-se comprovar que, enquanto o comportamento dos objetos de cena não obedece completamente às leis da Física devido à extrema simplicidade dos cálculos utilizados, a colisão é detectada de forma correta. Para fins ilustrativos, a Figura 8 contém *snapshots* de uma cena composta por vários objetos representados por superfícies paramétricas, que são tesselados em tempo real e exibidos em tela e, além disso, sofrem efeitos oriundos de colisões com outros corpos da cena. A cena contém

- 2 planos, cada um com 800 triângulos divididos em 31 *meshlets*, resultando em aproximadamente 26 triângulos por *meshlet*;
- 6 cubos arredondados, cada um com 4.800 triângulos divididos em 255 *meshlets*, resultando em aproximadamente 19 triângulos por *meshlet*;
- 5 *quadballs*, cada uma com 1.200 triângulos divididos em 63 *meshlets*, resultando em aproximadamente 19 triângulos por *meshlet*;
- 2 cubos, cada um com 1.728 triângulos divididos em 63 *meshlets* resultando em aproximadamente 27 triângulos por *meshlet*.

Isso totaliza, na cena, 8520 triângulos armazenados em memória; se houvesse replicação de malhas, então seriam 39856 triângulos, no entanto, a abordagem aqui utilizada tessela uma superfície em CPU somente uma vez e reutiliza a malha resultante para múltiplas instâncias, ou seja, todas as *quadballs*, por exemplo, utilizam a mesma malha, porém com *transforms* diferentes.

Pode-se inferir, com tais informações em mente, que o uso de uma BVH por malha reduz dramaticamente a quantidade de testes de primitivos a serem realizados; sem elas, caso as AABBs de um cubo arredondado e de uma *quadball* se interceptassem, então seriam realizados $4.800 \times 1.200 = 5.760.000$ testes entre triângulos em um único passo de simulação, no pior caso. Em comparação, com as BVHs em efeito, isso se reduz a aproximadamente $19^2 = 361$ testes em média e $64^2 = 4096$ no pior caso; no entanto, é válido reforçar que a inclusão das BVHs adiciona alguns testes extras entre AABBs, porém, eles são de baixo custo computacional quando comparados a testes entre triângulos e vêm, também, em menor número.

4 Conclusão

Este trabalho se propôs a cumprir três objetivos: a) tesselar superfícies de subdivisão em CPU e GPU para que pudessem ser exibidas, b) detectar colisões entre tais superfícies, e c) demonstrar, por meio de uma simulação rudimentar de corpos rígidos, que a detecção de colisões cumpre seu papel.

A tesselação de superfícies foi abordada na seção 2, tanto em CPU na seção 2.3 quanto em GPU na seção 2.4, e, como aparente na Figura 4, o projeto e a implementação da solução em OpenGL visando atingir o objetivo a) foram bem-sucedidos. Por sua vez, o processo de detecção de colisões foi detalhado na seção 3 e teve seu funcionamento comprovado pela aplicação da simulação de corpos rígidos descrita na seção 3.3, cuja execução pode ser observada, de certa forma, na Figura 8, resultando no cumprimento dos objetivos b) e c) e na finalização do trabalho.

No entanto, enquanto todos os objetivos foram atingidos, é importante reparar que os dados obtidos da intersecção entre duas superfícies de subdivisão pelo método aqui apresentado são consideravelmente limitados, visto que o algoritmo de [4] não explicita uma forma de acessar algumas grandezas como, por exemplo, a normal correta de intersecção. Assim, pode-se dizer que os resultados não são satisfatórios para propósitos que requerem tais informações de intersecção. Coincidentemente, a simulação de corpos rígidos é um de

tais propósitos, ou seja, parte do comportamento eventualmente errático dos corpos na simulação pode ser atribuída à normal errônea.

Levando isso em consideração, uma possível continuação do que foi concretizado neste texto em trabalhos futuros se daria na forma da adoção de um algoritmo de intersecção entre triângulos capaz de fornecer mais dados de intersecção de modo a permitir que aplicações tenham acesso às informações necessárias para que a implementação aqui apresentada tenha mais casos de uso. Posteriormente, um motor de Física devidamente robusto – com um *solver* iterativo para simulações fisicamente precisas – poderia ser, então, implementado com os vetores normais corretos. Por fim, enquanto as colisões são detectadas em tempo real, não houve tempo suficiente para que testes de performance fossem executados e documentados, deixando o desempenho da solução apresentada em termos abstratos; assim, uma análise desse aspecto da implementação seria uma continuação viável.

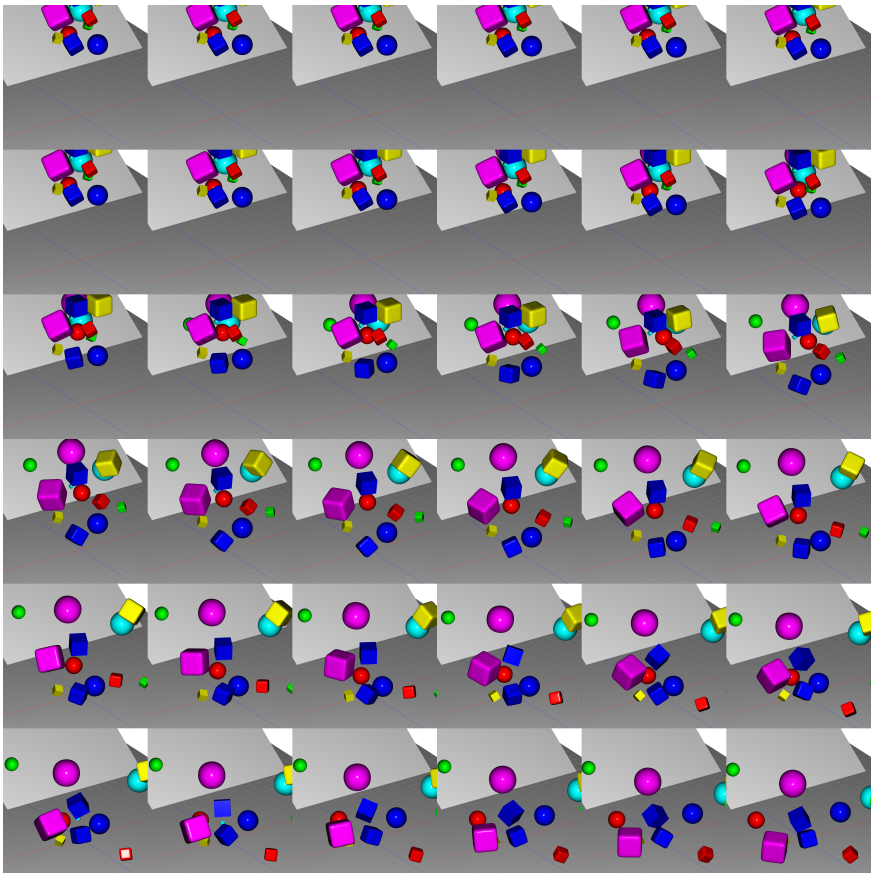

Figura 8: Física básica aplicada a superfícies de subdivisão.

Referências

[1] Lars-Erik Andersson and Neil F. Stewart. *Introduction to the Mathematics of Subdivision Surfaces*. Society for Industrial and Applied Mathematics, 2010.

[2] Erin Catto. Dynamic bounding volume hierarchies. Disponível em `https://box2d.org/files/ErinCatto_DynamicBVH_Full.pdf`. Acessado em: 9 de abril de 2024.

[3] Erwin Coumans. Bullet physics engine. Disponível em `https://github.com/bulletphysics/bullet3`. Acessado em: 9 de abril de 2024.

[4] Olivier Devillers and Philippe Guigue. Faster Triangle-Triangle Intersection Tests. Technical Report RR-4488, INRIA, June 2002. Disponível em `https://inria.hal.science/inria-00072100/file/RR-4488.pdf`. Acessado em: 9 de abril de 2024.

[5] Christer Ericson. *Real-Time Collision Detection*. CRC Press, Inc., USA, 2004.

[6] N. Gharachorloo, S. Gupta, R. F. Sproull, and I. E. Sutherland. A characterization of ten rasterization techniques. In *Proceedings of the 16th Annual Conference on Computer Graphics and Interactive Techniques*, SIGGRAPH '89, page 355–368, New York, NY, USA, 1989. Association for Computing Machinery.

[7] Newcastle University's Game Technology Group. Physics - collision response. Disponível em `https://research.ncl.ac.uk/game/mastersdegree/gametechnologies/physicstutorials/5collisionresponse/Physics%20-%20Collision%20Response.pdf`. Acessado em: 9 de abril de 2024.

[8] J. David MacDonald and Kellogg S. Booth. Heuristics for ray tracing using space subdivision. *The Visual Computer*, 6:153–166, 1990.

[9] Donald Meagher. Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer, 10 1980.

[10] Gordon E. Moore. Cramming more components onto integrated circuits, reprinted from electronics, volume 38, number 8, april 19, 1965, pp.114 ff. *IEEE Solid-State Circuits Society Newsletter*, 11(3):33–35, 2006.

[11] Bárbara Santos Munhão. Análise isogeométrica com elementos de contorno e superfícies de subdivisão. Master's thesis, Universidade Federal de Mato Grosso do Sul, 2023. Indexado em `https://repositorio.ufms.br/handle/123456789/5835`. Acessado em: 9 de abril de 2024.

[12] Matthias Nießner, Christian Siegl, Henry Schäfer, and Charles T Loop. Real-time collision detection for dynamic hardware tessellated objects. In *Eurographics (Short Papers)*, pages 33–36. Citeseer, 2013.

[13] Animation Studios of Pixar. Opensubdiv. Disponível em `https://github.com/PixarAnimationStudios/OpenSubdiv`. Acessado em: 10 de abril de 2024.

[14] Matt Pharr, Wenzel Jakob, and Greg Humphreys. *Physically Based Rendering: From Theory to Implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2016.

[15] Les Piegl and Wayne Tiller. *The NURBS Book*. Springer-Verlag, New York, NY, USA, second edition, 1996.

[16] R. Schumacher and Air Force Human Resources Laboratory. Training Research Division. *Study for Applying Computer-generated Images to Visual Simulation*. AFHRL-TR. Air Force Human Resources Laboratory, Air Force Systems Command, 1969.

[17] Jun Zhou, Pieter J. Barendrecht, Michael Bartoň, and Jiří Kosinka. Numerical quadrature for gregory quads. *Applied Mathematics and Computation*, 453:128051, 2023.

Apêndice – Estudo preliminar

A tese de doutorado de Nießner [12], publicada em 2013, descreve uma metodologia para detectar colisões entre superfícies de subdivisão de forma eficiente fazendo uso direto da pipeline gráfica empregada. Tal monografia foi utilizada como base para uma abordagem inicial, porém, foi eventualmente descartada devido a lacunas no algoritmo que não puderam ser preenchidas. Apesar disso, uma parcela de tempo considerável foi dedicada à compreensão deste material, portanto é pertinente que ele seja documentado; segue, então, uma uma breve descrição do processo de Nießner.

A fase ampla de Nießner

Especificamente nesta seção e na seguinte, são mencionadas caixas limitantes de objetos que não são malhas de triângulos, mas sim superfícies suaves representadas por malhas de retalhos Bézier. Nießner envolve seus objetos em OBBs. A fase ampla de sua detecção de colisões consiste em utilizar Análise de Componentes Principais (PCA, do Inglês *Principal Component Analysis*) e algumas propriedades de retalhos Bézier para computar as OBBs de cada superfície. Em seguida, as OBBs das superfícies são comparadas duas a duas; se algum par resultar em uma intersecção, o volume poliedral representado por essa intersecção tem sua OBB computada utilizando o mesmo ferramental mencionado acima. Essa nova OBB é, então, encaminhada à próxima fase do processo de detecção, e os retalhos das superfícies que podem estar contidos nessa OBB são marcados.

A fase estreita de Nießner

Para cada OBB resultante da fase ampla de Nießner, é construída uma matriz de projeção ortogonal que transforma a OBB no cubo do NDC[2]. É efetuada, então, uma *draw call* com *shaders* que tesselam e posteriormente efetuam uma voxelização binária nos retalhos envolvidos pela OBB.

Presumindo que os retalhos já foram tesselados, tem-se como resultado malhas de triângulos contidas no NDC. Também será assumido que há um *buffer* de armazenamento já alocado para guardar o resultado da voxelização; esse *buffer* deve ser visualizado como uma grade tridimensional de valores binários. Por simetria, utiliza-se aqui um *buffer* de inteiros de tamanho 32×32 para armazenamento do resultado da voxelização, mas vale notar que a resolução é arbitrária, e um *buffer* maior resultaria em uma voxelização com mais granularidade nos eixos x e y.

De acordo com Nießner, desde a OpenGL 4.0 pode-se acessar e modificar de maneira avulsa o conteúdo do *buffer* no *shader* de fragmentos; o processo de voxelização faz uso dessa funcionalidade, consistindo em utilizar a posição (x,y,z) do fragmento atual em NDC para determinar índices i,j,k do voxel na grade:

$$\begin{aligned} i &= \lfloor 16(1-y) \rfloor, \\ j &= \lfloor 16(1+x) \rfloor, \\ k &= \lfloor 16(1+z) \rfloor. \end{aligned} \tag{5}$$

Assim, sabe-se que o k-ésimo bit da $(32i+j)$-ésima posição do *buffer* deve ser 1, já que por aquele voxel passa um primitivo. Isso pode ser feito via operações *bitwise* – executadas de forma atômica, já que várias instâncias do *shader* podem estar operando sobre uma mesma posição do *buffer* simultaneamente. Mais especificamente:

```
atomicOr(buffer[32 * i + j], 1 « k).
```

Essa voxelização deve ser executada duas vezes; uma para cada superfície cujos patches estão contidos na OBB. O resultado disso se dá na forma de dois *buffers* de tamanho idêntico contendo duas grades de voxels. Resta, então, comparar ambas as grades para descobrir em que voxels uma superfície intercepta a outra.

Com esse fim, um *compute kernel* é executado, com uma *thread* para cada posição (i,j) dos *buffers*. Como os *buffers* guardam inteiros que representam, cada um, 32 voxels binários, pode-se obter os índices das colisões utilizando uma simples operação *bitwise and* entre cada elemento das grades; isso resulta em 32×32 inteiros cujos bits iguais a 1 determinam a existência de uma colisão naquela célula da grade. Utilizando a matriz de projeção anteriormente mencionada, que deve ser mantida em memória até este ponto, pode-se calcular a posição das intersecções utilizando o inverso das equações em (5). As normais, no entanto, devem ser computadas utilizando o conceito de vizinhança de *voxels*.

[2] Do Inglês *Normalized Device Coordinates*, é um sistema de coordenadas que descreve um cubo com vértice mínimo em $(-1,-1,-1)$ e máximo em $(1,1,1)$ e origem em $(0,0,0)$; é o volume de rasterização utilizado pela API gráfica OpenGL.