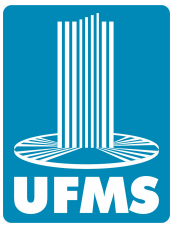
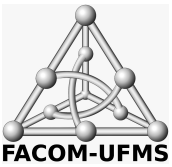


Universidade Federal de Mato Grosso do Sul
Faculdade de Computação



Felipe Machado da Silva

Traçado de raios para superfícies de Bézier em GPU

Campo Grande – MS
2025

Felipe Machado da Silva

Traçado de raios para superfícies de Bézier em GPU

Dissertação apresentada à Faculdade de Computação da Universidade Federal de Mato Grosso do Sul como parte dos requisitos para a obtenção do título de Mestre em Ciência da Computação.

Orientador: Prof. Dr. Paulo Aristarco Pagliosa

Campo Grande – MS
2025

Agradecimentos

Agradeço primeiramente a Deus pela vida que recebi, pelas pessoas que Ele colocou ao meu redor e pela oportunidade de realizar este trabalho.

Agradeço à minha mãe, Carlene Ribeiro da Silva, e ao meu pai, Edson Machado da Silva, por terem cuidado de mim por tanto tempo, por estarem presentes na minha vida e pela oportunidade de poder estudar tranquilamente.

Agradeço à professora Edna Ayako Hoshino por ter tornado a graduação em Ciência da Computação uma experiência sem igual, tornando o conhecimento algo fascinante para mim, além do cuidado que teve por mim e por transmitir tanto ânimo e motivação.

Também agradeço imensamente ao meu orientador, Paulo Aristarco Pagliosa, por toda a sua paciência em me ensinar e orientar, por todo o conhecimento que me transmitiu e me motivar sempre, inclusive nos momentos difíceis.

Também manifesto meu agradecimento pelo apoio e financiamento deste trabalho pela CAPES.

O presente trabalho foi realizado com apoio da Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – Brasil (CAPES) – Código de Financiamento 001.

Conteúdo

Lista de Figuras

Capítulo 1

Introdução

1.1 Motivação e justificativa

A computação gráfica é uma área da computação que estuda técnicas para síntese e manipulação de imagens. Apesar de ser uma área ampla que abrange desde fotografia, armazenamento e compressão de imagens à visualização cientifica e jogos, o produto de todos necessita de alguma forma de visualização. Para que o usuário seja capaz de ver as imagens, são utilizados dispositivos físicos como telas que possuem milhões de pontos capazes de emitir luz em uma cor específica. Estes pontos são conhecidos como *pixels*, são organizados em grade e são utilizados para formar uma imagem visível ao usuário. Para que formas geométricas simples como pontos, linhas e triângulos, posteriormente referidas como *primitivos*, sejam visualizadas em telas, elas precisam ser convertidas em uma série de pixels em um processo chamado de *rasterização*. O uso extensivo de rasterização para sintetizar imagens desde apenas texto até cenas tridimensionais virtuais em jogos levou à criação de coprocessadores especializados para esta tarefa, chamados de *GPUs*, do inglês *Graphics Processing Units*. Atualmente, GPUs são unidades mais flexíveis com estágios programáveis, o que possibilitou a implementação de diversos efeitos visuais em cenas tridimensionais.

Uma alternativa mais lenta e de maior qualidade à rasterização para sintetizar imagens de cenas tridimensionais é o *traçado de raios*. O traçado de raios é um algoritmo que visa criar imagens realistas simulando diversos fenômenos ópticos como a propagação de raios de luz em um espaço tridimensional e a reflexão e refração destas sobre os objetos presentes em uma cena virtual. Usualmente, uma cena é composta de objetos geométricos, fontes de luz e ao menos uma câmera, na qual a imagem é formada pelos raios de luz que incidem nela. Então para cada pixel da imagem, o algoritmo traça o caminho inverso dos raios de luz a partir da câmera simulando todas

as interações físicas com os objetos na cena até chegar na fonte de luz para determinar sua cor. Devido à imensa quantidade de raios de luz traçados, o algoritmo é computacionalmente intensivo. Em contrapartida, a qualidade e realismo das imagens geradas são superiores a técnicas de renderização como a rasterização. Portanto, o traçado de raios tornou-se um algoritmo amplamente utilizado na indústria cinematográfica, em animações, efeitos especiais e, mais recentemente, em jogos.

Um aspecto importante na construção de cenas são as representações dos objetos que compõem a cena. Regularmente, os objetos são representados por malhas de triângulos. Porém, representar modelos geométricos que possuem curvas e secções arredondadas com malhas de triângulos pode requerer uma grande quantidade de pequenos triângulos para representá-los com fidelidade. Além de consumir mais memória, isto dificulta tarefas de modelagem e edição de diversos objetos. *Superfícies paramétricas* resolvem alguns destes obstáculos. Superfícies paramétricas tridimensionais são definidas a partir de uma função paramétrica que opera sobre um conjunto pré-definido de pontos, chamados de *pontos de controle*, e mapeia o domínio paramétrico bidimensional para pontos no espaço euclidiano. Consequentemente, tem-se superfícies que representam curvaturas suaves com fidelidade e são descritas por poucos pontos de controle, tornando-as facilmente manipuláveis. As *superfícies de Bézier*, ou também chamadas de *retalhos de Bézier*, são exemplos de superfícies definidas por funções que interpolam uma *grade de pontos de controle*. Como a modelagem de objetos complexos podem depender de múltiplos retalhos de Bézier, desenvolveram-se tecnologias como superfícies *B-spline* [33] (do inglês *Basis spline*) que são uma generalização de retalhos de Bézier e incorporam múltiplos retalhos numa única grade de pontos de controle utilizando funções de base e sequências de números reais chamados de *nós*. Do mesmo modo, superfícies NURBS [33, 14] (do inglês *Non Uniform Rational B-Spline*) são uma generalização de B-splines que admite sequência não uniforme de nós e pesos nos pontos de controle. Sederberg et al. [40] propuseram as superfícies *T-spline*, uma extensão de superfícies B-spline não uniforme para suportarem uma grade não regular de pontos de controle com *T-junções*, chamada de *T-malha*.

Além das superfícies paramétricas, existem as *superfícies de subdivisão* que são definidas a partir de um conjunto regras de subdivisão aplicada sobre vértices, arestas e faces de uma malha poligonal. Quando o conjunto de regras é aplicado sobre uma malha poligonal, elas tem como resultado uma malha mais refinada. A superfície de subdivisão é baseada na ideia de que, ao aplicar a subdivisão infinitamente, os vértices da malha convergirão para uma superfície e que esta superfície é tida como o limite matemático deste procedimento. Entre as superfícies de subdivisão mais conhecidas tem-se a de *Catmull-Clark* [11] e a de *Loop* [22]. Catmull e Clark [11] generalizaram

o algoritmo de subdivisão inerente a superfícies B-spline para malhas de topologia arbitrária. Como consequência, para malhas retangulares a superfície limite pode ser reduzida para superfícies B-spline, exceto para malhas com pontos extraordinários, os quais surgem de vértices que possuem um número de arestas diferente de quatro. Similarmente, Loop [22] apresentou um esquema de subdivisão que opera em malhas triangulares.

As vantagens de superfícies curvas mencionadas tornaram o uso de superfícies paramétricas como NURBS e T-splines um padrão para ferramentas de CAD e CAM (do inglês *Computer Aided Design* e *Computer Aided Manufacturing*) assim como tornaram superfícies de subdivisão Catmull-Clark também um padrão para a indústria cinematográfica e de animação.

A literatura provê algoritmos de refinamento para decompor NURBS [33] e T-splines [40] em uma coleção de retalhos de Bézier equivalentes. Similarmente, é possível obter retalhos de Bézier e retalhos de Gregory [18], uma extensão dos retalhos de Bézier, a partir de superfícies de subdivisão Catmull-Clark [26, 5]. Não obstante, a Pixar desenvolveu um projeto de código aberto chamado *OpenSubdiv* que implementa diversas ferramentas, inclusive a obtenção de retalhos para superfícies de subdivisão [34]. Alternativamente, no contexto de *análise isogeométrica*, Borden et al. [8] desenvolveram um processo chamado de *extração de Bézier* sobre NURBS e Scott et al. [39] também o fizeram para superfícies T-spline. A extração de Bézier consiste na montagem de um *operador de extração de Bézier* para cada *elemento de Bézier* a ser extraído da superfície. Cada face da malha de controle de uma superfície NURBS ou T-spline corresponde a um trecho da superfície que é influenciada por um conjunto de pontos de controle através das funções de base da superfície. O operador de extração de Bézier consiste numa matriz que transforma as funções de base da superfície em questão para os polinômios base de Bernstein obtendo assim o elemento de Bézier para a região. O elemento de Bézier não é necessariamente um retalho de Bézier mas pode ser visto como um e convertido para um retalho de Bézier.

Consequentemente, um sistema capaz de traçar raios para retalhos de Bézier pode ser naturalmente estendido para operar com superfícies como T-splines e quaisquer superfícies das quais seja possível extrair uma coleção de retalhos. Este é um dos objetivos deste trabalho: estudar o algoritmo traçado de raios sobre retalhos de Bézier, sua extensão para demais superfícies e explorar o paralelismo massivo oferecido por GPUs para executar o algoritmo.

O gargalo do algoritmo de traçado de raios para uma implementação específica é o número e a complexidade do cálculo de intersecção de um raio com uma superfície.

No caso de superfícies paramétricas, o cálculo de intersecção é custoso. Uma maneira de realizar o traçado de raios sobre superfícies paramétricas é realizar a tesselagem prévia da superfície e então realizar o traçado de raios sobre os triângulos gerados pela tesselagem. A tesselagem de uma superfície consiste em coletar uma amostra de pontos pertencentes à superfície e criar uma malha de triângulos sobre estes pontos. Entretanto, o alto número de triângulos necessários para aproximar a superfície com fidelidade eleva o tempo de execução e, principalmente, o uso de memória, mesmo que temporariamente. Para mitigar este problema e diminuir o número de triângulos, estudos foram feitos para aplicar a tesselagem adaptativa sobre os retalhos extraídos de superfícies de subdivisão [26]. Adicionalmente, para evitar o uso exacerbado de memória, estudos recentes envolvem realizar a tesselagem sob demanda, fazendo uso de uma cache distribuída [5] em múltiplos núcleos de uma CPU. Entretanto, uma cache distribuída sob demanda, potencialmente, pode não ser trivial de ser adaptada para GPUs, devido à necessidade de sincronizar o acesso para milhares de núcleos. Além disso, o uso de caches de tesselagem têm como desvantagem a necessidade de reconstrução para cada quadro de uma cena animada. Portanto, assim como Tejima, Fujita e Matsuoka [44] observaram, realizar o traçado de raios diretamente na superfície, sem utilizar tesselagem, é vantajoso em aplicações de animação em tempo real e requer menos memória. Tejima, Fujita e Matsuoka [44] mostraram que o traçado de raios diretamente em superfícies paramétricas tem um desempenho similar ao traçado de raios sobre a tesselagem e, portanto, trata-se de uma abordagem promissora de pesquisa, pois mesmo que em alguns casos o seu desempenho seja levemente pior, a economia de memória é significante e pode ser vantajosa para cenas muito complexas. Desta forma, os próximos parágrafos são dedicados à abordagem de traçado de raios diretamente em superfícies paramétricas.

Neste contexto, diversos estudos apresentam algoritmos para calcular diretamente a intersecção raio-superfície que podem ser categorizadas em ao menos duas abordagens principais: métodos numéricos e subdivisão recursiva.

Os algoritmos baseados em *métodos numéricos* representam o problema em uma ou mais equações polinomiais de maneira que uma de suas raízes seja o ponto de intersecção desejado. Uma vez determinadas as equações, utiliza-se algum método numérico iterativo para encontrar suas raízes e obter o ponto de intersecção. O método de Newton é um exemplo de método bastante utilizado para este fim [43, 50, 2, 1, 24, 47, 23, 30, 46, 27]. Geralmente estes algoritmos sofrem com problemas de convergência, como convergir para a solução errada ou não convergem em alguns casos, mesmo que exista solução. Como consequência, alguns estudos recorrem a técnicas de análise intervalar [45, 1] e subdivisão das superfícies [43, 2] para contornar estes problemas e

então aplicar os métodos numéricos.

Já as abordagens baseadas em *subdivisão recursiva* [48, 49, 3, 4, 7] consistem em intersectar o raio de luz com o envoltório convexo da superfície paramétrica, dada que esta operação é mais simples. Se a intersecção com o envoltório não existir, infere-se que a superfície não é intersectada e então descartada de análises posteriores. Caso contrário, então a superfície é subdividida em duas ou quatro novas superfícies e a operação é repetida recursivamente sobre as partes restantes até que seu envoltório convexo seja pequeno o suficiente para decidir se o raio de luz intersectou ou não a superfície original.

Existe também o algoritmo de recorte de Bézier (em inglês: *Bézier clipping*), proposto por Nishita, Sederberg e Kakimoto [28], considerado um misto entre método numérico e subdivisão recursiva. O algoritmo consiste em projetar o retalho em um plano cartesiano de maneira que o raio de luz seja perpendicular a este plano e o intersecte em sua origem, ou seja, no ponto $(0, 0)$. Em cada iteração, é realizado um cálculo para cortar as extremidades do domínio paramétrico da superfície. Em seguida, cria-se uma caixa limitante para o retalho projetado e é verificado se esta caixa limitante contém a origem do plano. Se sim, é provável uma intersecção entre o raio de luz e o retalho. Se não, a intersecção é descartada. O passos descritos são repetidos até que o retalho projetado se torne pequeno o suficiente para que seja considerado um ponto. Mais detalhes do algoritmo são apresentados na Seção 2.7.

Alguns estudos focaram no traçado de raios sobre retalhos de Bézier em GPU. Entre eles, Löw [23] utiliza o *pipeline* de rasterização para desenhar dois triângulos que cobrem a tela e executar o algoritmo de traçado de raios no *shader* de fragmento utilizando o método de Newton para realizar a intersecção. Mais recentemente, utilizando a API CUDA da NVIDIA para programação paralela de propósito geral em GPU, Valkering [46] e Nigam [27] realizaram o traçado de raios por meio do método de Newton e Binder e Keller [7] por meio da subdivisão recursiva.

1.2 Objetivos

O objetivo geral deste projeto consiste na implementação em GPU da renderização de superfícies paramétricas, utilizando-se de traçado de raios, por meio da extração de retalhos. Em particular, pretende-se aplicar os métodos da subdivisão recursiva e de recorte de Bézier para o cálculo da intersecção de raio com a superfície, conforme [7] e [44].

Nesta seção, listamos alguns objetivos específicos para o projeto:

- Estudar os métodos de intersecção raio/retalho de Bézier;
- Implementar um traçador de raios em GPU para superfícies compostas de retalhos de Bézier;
- Realizar testes de desempenho.

1.3 Organização do texto

O restante do texto está organizado da seguinte forma: o Capítulo 2 introduz os principais conceitos, algoritmos e métodos relacionados a traçado de raios e superfícies paramétricas. No Capítulo 3, são descritas as atividades realizadas para atingir os objetivos do projeto de mestrado. O Capítulo 4 expõe as cenas geradas para testar o traçado de raios sobre as superfícies paramétricas bem como os resultados alcançados durantes os testes. Por fim, o Capítulo 5 delibera sobre possíveis melhorias e a viabilidade de projetos futuros como consequência deste trabalho.

Capítulo 2

Fundamentos e trabalhos relacionados

2.1 Considerações iniciais

Neste capítulo, introduzimos os principais fundamentos matemáticos e computacionais utilizados neste trabalho. Na Seção 2.2, apresentamos a base de curvas paramétricas e, na Seção 2.3, estendemos seus conceitos para superfícies paramétricas. Nas Seções 2.4 e 2.5, apresentamos a versão matricial das curvas e operações necessárias para alguns métodos apresentados posteriormente no capítulo. Na Seção 2.6, revisamos os principais pontos de um algoritmo de traçado de raios e, em seguida, na Seção 2.7, discorremos sobre os métodos de intersecção raio/superfície paramétrica. A Seção 2.8 introduz o conceito de extração de superfícies e, por fim, as considerações finais são apresentadas.

2.2 Curvas paramétricas

Uma curva paramétrica é definida por uma função $\mathbf{C}(u)$ que mapeia um parâmetro real u para um ponto no espaço euclidiano de dimensão arbitrária. Neste trabalho, consideraremos espaços euclidianos tridimensionais $\mathbb{E}^3$, exceto quando especificado. Curvas paramétricas tridimensionais possuem a seguinte forma:

$$\mathbf{C}(u) = \begin{bmatrix} x(u) \\ y(u) \\ z(u) \end{bmatrix}$$

Curva de Bézier É um tipo de curva paramétrica amplamente utilizada e definida por um polinômio de grau n que interpola uma sequência de $n+1$ *pontos de controle* $\mathbf{P}_0, \mathbf{P}_1, \ldots, \mathbf{P}_n$ através da combinação linear de *polinômios base de Bernstein*. A curva é definida para um parâmetro real $u \in [0,1]$ da seguinte forma:

$$\mathbf{C}(u) = \sum_{i=0}^{n} B_{i,n}(u)\mathbf{P}_i, \quad \forall u \in [0,1] \tag{2.1}$$

em que $B_{i,n}(u)$ é o polinômio base de Bernstein:

$$B_{i,n}(u) = \binom{n}{i} u^i (1-u)^{n-i}. \tag{2.2}$$

Cada ponto pertencente a uma curva de Bézier pode ser visto como uma combinação linear de seus pontos de controle. Disto, surge uma propriedade essencial: uma curva de Bézier está contida no *casco convexo* de seus pontos de controle. Também, curvas de Bézier são invariantes a transformações lineares como translação, rotação e escala, ou seja, transformações aplicadas em seus pontos de controle equivalem a aplicar transformações sobre a própria curva [33].

Conforme Piegl e Tiller [33], os polinômios base de Bernstein possuem também uma definição recursiva:

$$B_{i,n}(u) = \begin{cases} (1-u)B_{i,n-1}(u) + uB_{i-1,n-1}(u), & \text{se } 0 \leq i \leq n \\ 0, & \text{caso contrário.} \end{cases} \tag{2.3}$$

Como consequência, torna-se possível definir recursivamente uma curva de Bézier. Para uma curva de Bézier $\mathbf{C}^{(n)}_{\mathbf{P}_0 \cdots \mathbf{P}_n}(u)$ de grau n e de pontos de controle $\mathbf{P}_0 \cdots \mathbf{P}_n$, tem-se que:

$$\mathbf{C}^{(n)}_{\mathbf{P}_0 \cdots \mathbf{P}_n}(u) = (1-u)\mathbf{C}^{(n-1)}_{\mathbf{P}_0 \cdots \mathbf{P}_{n-1}}(u) + u\mathbf{C}^{(n-1)}_{\mathbf{P}_1 \cdots \mathbf{P}_n}(u), \tag{2.4}$$

considerando $\mathbf{C}^{(0)}_{\mathbf{P}_i}(u) = \mathbf{P}_i$. A partir desta definição surge um algoritmo numericamente estável para avaliar pontos sobre a curva: o algoritmo de De Casteljau.

Algoritmo de De Casteljau Da Equação (2.4), conforme explicado em [33], fixando o parâmetro u, representando $\mathbf{P}_i$ como $\mathbf{P}_{i,0}$, pode-se estender a definição de pontos sobre a curva para:

$$\mathbf{P}_{i,k} = (1-u)\mathbf{P}_{i,k-1} + u\mathbf{P}_{i+1,k-1}, \qquad \begin{cases} k = 1, \ldots, n, \\ i = 0, \ldots, n-k. \end{cases} \tag{2.5}$$

Desta forma, a Equação (2.5) expõe um algoritmo recursivo para determinar $\mathbf{C}(u) = \mathbf{P}_{0,n}$. Nota-se uma dependência triangular entre os termos que pode ser arranjada conforme a Tabela 2.1. Os pontos de controle estão na coluna mais à esquerda. A interpolação dois à dois dos pontos de uma coluna gera a coluna imediatamente à sua direita. A avaliação final encontra-se mais à direita, em $\mathbf{P}_{0,n}$. Portanto, desta característica procede o Algoritmo 2.1.

Tabela 2.1: Arranjo dos termos da Equação (2.5).

$\mathbf{P}_{0,0}$	$\mathbf{P}_{0,1}$	$\mathbf{P}_{0,2}$	$\ldots$	$\mathbf{P}_{0,n-2}$	$\mathbf{P}_{0,n-1}$	$\mathbf{P}_{0,n}$
$\mathbf{P}_{1,0}$	$\mathbf{P}_{1,1}$	$\mathbf{P}_{1,2}$	$\ldots$	$\mathbf{P}_{1,n-2}$	$\mathbf{P}_{1,n-1}$	
$\mathbf{P}_{2,0}$	$\mathbf{P}_{2,1}$	$\mathbf{P}_{2,2}$	$\ldots$	$\mathbf{P}_{2,n-2}$		
$\vdots$	$\vdots$	$\vdots$				
$\mathbf{P}_{n-2,0}$	$\mathbf{P}_{n-2,1}$	$\mathbf{P}_{n-2,0}$				
$\mathbf{P}_{n-1,0}$	$\mathbf{P}_{n-1,1}$					
$\mathbf{P}_{n,0}$						

Algoritmo 2.1: De Casteljau para curvas de Bézier

Entrada: Parâmetro u, grau n e vetor de pontos de controle $\mathbf{P}$ (modificado após a execução).
Saída : $\mathbf{C}(u)$.

```
1 função deCasteljau(u, n, P)
2     para k = 1 até n faça
3         para i = 0 até n − k faça
4             P[i] ← (1 − u) ∗ P[i] + u ∗ P_{i+1};
5     retorna P[0];
```

2.3 Superfícies paramétricas

Uma superfície paramétrica é definida por uma função $\mathbf{S}(u, v)$ que mapeia um domínio bidimensional para um ponto no espaço euclidiano. A forma mais comum de definir uma superfície paramétrica, segundo Piegl e Tiller [33], é estendendo a noção de curvas paramétricas para duas dimensões através do esquema de *produto tensorial*.

Superfície de Bézier É definida a partir do esquema de produto tensorial sobre dois espaços vetoriais de polinômios de graus n e m que interpolam uma grade de $(n+1) \times (m+1)$ *pontos de controle* $\mathbf{P}_{0,0}, \mathbf{P}_{0,1}, \ldots, \mathbf{P}_{n,m}$. Para parâmetros reais u e v, a superfície é dada por:

$$\mathbf{S}(u,v) = \sum_{j=0}^{m} \sum_{i=0}^{n} B_{j,m}(v) B_{i,n}(u) \mathbf{P}_{i,j}, \quad \forall u, v \in [0,1]. \tag{2.6}$$

Pontos sobre um retalho de Bézier podem ser avaliados pelo Algoritmo 2.2, uma extensão do algoritmo de De Casteljau para curvas.

Algoritmo 2.2: De Casteljau para superfícies.

Entrada: (u, v), graus (n, m) e matriz de pontos de controle $\mathbf{P}$.
Saída : $\mathbf{S}(u, v)$.

```
1  função S(u, v, n, m, P)
       /* Diminuir número de iterações do laço mais interno.        */
2      se n ≤ m então
3          para j = 0 até m faça
4              para i = 0 até n faça
5                  Q[i] ← P[i, j];
6              R[j] ← deCasteljau(u, n, Q);
7          retorna deCasteljau(v, m, R);
8      senão
9          para i = 0 até n faça
10             para j = 0 até m faça
11                 R[j] ← P[i, j];
12             Q[j] ← deCasteljau(v, m, R);
13         retorna deCasteljau(u, n, Q);
```

Superfícies com grau 3 em ambas as dimensões são chamadas de bicúbicas e superfícies com grau 4 em ambas as dimensões são chamadas de biquárticas. Superfícies de Bézier bicúbicas são as mais comuns em diversas aplicações computacionais.

2.4 Avaliando pontos com matrizes

Dado que na literatura e na indústria, curvas cúbicas ($n = 3$) e superfícies bicúbicas são amplamente utilizadas, as operações descritas nesta seção terão o foco direcionado a elas.

Para uma curva cúbica, tomando a Equação (2.1), pode-se descrever a curva como uma multiplicação de matrizes da seguinte forma:

$$\mathbf{C}(u) = B_{0,3}(u)\mathbf{P}_0 + B_{1,3}(u)\mathbf{P}_1 + B_{2,3}(u)\mathbf{P}_2 + B_{3,3}(u)\mathbf{P}_3$$

$$\mathbf{C}(u) = \begin{bmatrix}\mathbf{P}_0 & \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3\end{bmatrix} \begin{bmatrix}B_{0,3}(u) \\ B_{1,3}(u) \\ B_{2,3}(u) \\ B_{3,3}(u)\end{bmatrix}$$

Por brevidade, a letra **B** em negrito será utilizada para se referir à versão matricial das funções de base de Bernstein para uma curva cúbica. Logo, a partir da definição das funções de base (2.2):

$$\mathbf{B}(u) = \begin{bmatrix}B_{0,3}(u) \\ B_{1,3}(u) \\ B_{2,3}(u) \\ B_{3,3}(u)\end{bmatrix} = \begin{bmatrix}1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1\end{bmatrix} \begin{bmatrix}1 \\ u \\ u^2 \\ u^3\end{bmatrix}$$

Portanto, a versão matricial de uma curva de Bézier cúbica é:

$$\mathbf{C}(u) = \begin{bmatrix}\mathbf{P}_0 & \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3\end{bmatrix} \underbrace{\begin{bmatrix}1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1\end{bmatrix} \begin{bmatrix}1 \\ u \\ u^2 \\ u^3\end{bmatrix}}_{\mathbf{B}(u)} \tag{2.7}$$

As derivadas das funções de base são:

$$\mathbf{B}'(u) = \begin{bmatrix}-3 & 6 & -3 & 0 \\ 3 & -12 & 9 & 0 \\ 0 & 6 & -9 & 0 \\ 0 & 0 & 3 & 0\end{bmatrix} \begin{bmatrix}1 \\ u \\ u^2 \\ u^3\end{bmatrix} \tag{2.8}$$

Para uma curva B-Spline uniforme, por sua vez, as funções de base dão origem às seguintes matrizes [11]:

$$\mathbf{D}(u) = \frac{1}{6}\begin{bmatrix}1 & -3 & 3 & -1 \\ 4 & 0 & -6 & 3 \\ 1 & 3 & 3 & -3 \\ 0 & 0 & 0 & 1\end{bmatrix} \begin{bmatrix}1 \\ u \\ u^2 \\ u^3\end{bmatrix} \tag{2.9}$$

$$\mathbf{D}'(u) = \frac{1}{6}\begin{bmatrix} -3 & 6 & -3 & 0 \\ 0 & -12 & 9 & 0 \\ 3 & 6 & -9 & 0 \\ 0 & 0 & 3 & 0 \end{bmatrix}\begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix} \tag{2.10}$$

2.5 Recortando curvas com matrizes

Para recortar uma curva de Bézier nos pontos a e b, visando obter a subcurva no intervalo $[a, b]$ como uma nova curva independente, pode-se definir outra função de base $\mathbf{B}_{a,b}(u)$ mapeando o intervalo $[a, b]$ para $[0, 1]$:

$$\mathbf{B}_{a,b}(u) = \mathbf{B}((1-u)a + ub)$$

$$\mathbf{B}_{a,b}(u) = \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ ((1-u)a + ub) \\ ((1-u)a + ub)^2 \\ ((1-u)a + ub)^3 \end{bmatrix}$$

$$\mathbf{B}_{a,b}(u) = \begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}\begin{bmatrix} 1 \\ a + (b-a)u \\ a^2 + 2a(b-a)u + (b-a)^2u^2 \\ a^3 + 3a^2(b-a)u + 3a(b-a)^2u^2 + (b-a)^3u^3 \end{bmatrix}$$

$$\mathbf{B}_{a,b}(u) = \underbrace{\begin{bmatrix} 1 & -3 & 3 & -1 \\ 0 & 3 & -6 & 3 \\ 0 & 0 & 3 & -3 \\ 0 & 0 & 0 & 1 \end{bmatrix}}_{M}\underbrace{\begin{bmatrix} 1 & 0 & 0 & 0 \\ a & (b-a) & 0 & 0 \\ a^2 & 2a(b-a) & (b-a)^2 & 0 \\ a^3 & 3a^2(b-a) & 3a(b-a)^2 & (b-a)^3 \end{bmatrix}}_{Q}\begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix}$$

Para manipular a expressão com mais clareza, nomeiam-se a matriz das funções de base de Bernstein de M e a matriz encontrada que mapeia o intervalo de entrada das funções de Q. Visto que a multiplicação de matrizes é associativa e supondo que M seja inversível, tem-se:

$$\mathbf{B}_{a,b}(u) = MQ \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix} = MQ(M^{-1}M) \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix} = (MQM^{-1})M \begin{bmatrix} 1 \\ u \\ u^2 \\ u^3 \end{bmatrix}$$

$$\mathbf{B}_{a,b}(u) = (MQM^{-1})\mathbf{B}(u)$$

Desta maneira, isola-se a matriz capaz de recortar o intervalo $[a, b]$ de uma curva de Bézier. Durante a avaliação dos pontos sobre a curva, pela associatividade da multiplicação de matrizes, a matriz pode ser multiplicada tanto pelo resultado das funções de base quanto pelos pontos de controle.

$$\mathbf{C}_{a,b}(u) = \begin{bmatrix} \mathbf{P}_0 & \mathbf{P}_1 & \mathbf{P}_2 & \mathbf{P}_3 \end{bmatrix} (MQM^{-1})\mathbf{B}(u)$$

Ao ser multiplicada pelos pontos de controle, os pontos resultantes definem uma nova curva de Bézier.

A matriz das funções de base é inversível de forma que:

$$M^{-1} = \begin{bmatrix} 1 & 1 & 1 & 1 \\ 0 & \frac{1}{3} & \frac{2}{3} & 1 \\ 0 & 0 & \frac{1}{3} & 1 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{2.11}$$

Seja $\mathbf{Y}(a, b) = MQM^{-1}$ a função que determina a matriz que recorta uma curva de Bézier no intervalo $[a, b]$. Após as devidas expansões e fatorações, encontra-se:

$$\mathbf{Y}(a,b) = \begin{bmatrix} (1-a)^3 & (1-a)^2(1-b) & (1-a)(1-b)^2 & (1-b)^3 \\ 3(1-a)^2a & (1-a)(2a+b-3ab) & (1-b)(a+2b-3ab) & 3(1-b)^2b \\ 3(1-a)a^2 & a(a+2b-3ab) & b(2a+b-3ab) & 3(1-b)b^2 \\ a^3 & a^2b & ab^3 & b^3 \end{bmatrix} \tag{2.12}$$

O mesmo método pode ser usado para recortar outros tipos de curvas paramétricas, como B-Splines uniformes, desde que a matriz formada a partir das funções de base sejam inversíveis.

2.6 Traçado de raios

O traçado de raios é um algoritmo derivado do funcionamento de elementos reais como os olhos humanos e câmeras. A ideia geral é similar a uma caixa fechada que possui um orifício pequeno pelo qual os raios de luz passam e projetam uma imagem invertida na face da caixa oposta à face que contém o orifício. A face na qual a imagem foi projetada pode ser chamada de filme. Dentro de uma cena virtual pré-estabelecida, posiciona-se um observador ou uma câmera virtual com as características descritas para que a imagem seja gerada. Na prática, o filme é situado à frente do observador. Neste contexto, o traçado de raios *progressivo* é um algoritmo que simula os caminhos percorridos por raios de luz, partindo de uma série de fontes de luz e refletindo ou refratando em objetos até chegar no observador. Já no traçado de raios *regressivo*, percorre-se o caminho inverso dos raios de luz que de fato alcançam o observador e, a partir do observador, avaliam-se quais fontes de luz deram origem aos raios de luz. Desta forma, o algoritmo não calcula raios de luz que não afetarão a imagem final.

Dito isto, pode-se descrever uma versão bem simples do algoritmo de traçado de raios regressivo com os seguintes passos:

Passo 1 Para cada *pixel* (i, j) da imagem, $0 \leq i < m$, $0 \leq j < n$, trace um *raio de pixel* R_p. A cor do raio R_p será a cor do *pixel* (i, j).

Passo 2 Se o raio R_p não interceptar nenhum objeto da cena, sua cor será a cor de fundo de cena. Se o raio R_p interceptar um objeto, denote O_p como o objeto interceptado mais próximo da origem de R_p e determine o ponto de intersecção $\mathbf{I}_p$ na superfície de O_p.

Passo 3 Para cada fonte de luz l, trace um *raio secundário* R_l com origem no ponto $\mathbf{I}_p$ em direção à fonte de luz l. Se o raio R_l não interceptar algum objeto, têm-se que a fonte de luz l ilumina diretamente o ponto $\mathbf{I}_p$ e R_l é um *raio de luz*. Caso contrário, se houver um objeto interceptado por R_l, então R_l é um *raio de sombra* de cor preta. A cor de R_l é determinada por um modelo de iluminação local e adicionada à cor de R_p.

Passo 4 Se o material do objeto O_p no ponto $\mathbf{I}_p$ for reflexivo, traça-se um raio R_r com origem em $\mathbf{I}_p$ na direção de reflexão. A direção de reflexão é determinada em função da direção de incidência do raio R_p em $\mathbf{I}_p$ e do *vetor normal* à superfície no ponto I_p. A cor de R_r é adicionada à R_p. Para determinar a cor de R_r, executa-se o algoritmo recursivamente tendo o raio R_r como um raio de pixel R_p.

Nesta versão, traça-se somente um raio por pixel, na direção do centro do pixel. Ainda, consideram-se somente materiais opacos, sem texturização.

2.7 Intersecção raio/superfície paramétrica

Uma das principais operações do algoritmo de traçado de raios consiste na intersecção de um raio com a superfície de um objeto. No caso de superfícies de Bézier, a literatura destaca três procedimentos resumidos nas seções a seguir: métodos numéricos, subdivisão recursiva e recorte de Bézier (em inglês: *Bézier clipping*).

2.7.1 Métodos numéricos

Um dos primeiros estudos sobre traçado de raios sobre superfícies paramétricas polinomiais foi feito por Kajiya [20], o qual representa um raio de luz como a intersecção entre dois planos. Kajiya transforma o problema de intersectar um raio com uma superfície polinomial bicúbica em encontrar as raízes de um polinômio de grau 18 utilizando o método numérico de Laguerre que possui convergência bicúbica. Ele estima que para computar cada raio são gastas aproximadamente 6000 operações de ponto flutuante.

Posteriormente, Toth [45] apresenta um algoritmo para realizar a intersecção entre um raio de luz e uma superfície paramétrica utilizando técnicas de *análise intervalar* [25]. Assim como ocorre em abordagens por métodos numéricos, os pontos de intersecção são representados como as raízes de um sistema de equações. Ele usa uma extensão intervalar do método de Newton para calcular uma margem de erro para a solução atual do sistema e propõe uma maneira de sondar regiões da superfície usando um teste conhecido como *teste de Krawczyk–Moore* [45]. Dado um intervalo do domínio paramétrico da superfície, este teste é capaz de determinar a existência de uma única solução. Logo, em seu algoritmo, Toth [45] utiliza um esquema de busca binária sobre o domínio paramétrico para encontrar uma região adequada e uma solução inicial para então iniciar o método de Newton escalar. Desta forma, ele evita os problemas de convergência recorrentes ao utilizar o método de Newton.

Enger [13] também utiliza análise intervalar para traçar *clusters* de raios em algoritmo baseado no paradigma e divisão e conquista. Ademais, Barth, Lieger e Schindler [1] dividem a superfície em partes pequenas e emprega análise intervalar para construir volumes limitantes justos para estas partes e conseguir soluções iniciais para o método de Newton.

Joy e Bhetanabhotla [19] estudam a aplicação de métodos numéricos Quasi-Newton para o traçado de raios. Estes métodos necessitam de uma solução inicial para executar e, a cada iteração, melhoram a qualidade da solução. Em alguns casos, mesmo que haja solução, o método pode não convergir ou a solução encontrada pode não ser a desejada, o ponto de intersecção mais próximo. Logo, a qualidade da solução inicial é essencial para a convergência do método para a solução desejada. Para isso, Joy e Bhetanabhotla [19] empregam métodos de subdivisão de espaço para delimitar regiões da superfície paramétrica e encontrar uma solução inicial adequada. Além disso, eles empregam o conceito de *coerência de raios*. Baseado na premissa de que raios de pixel próximos intersectarão a mesma superfície em pontos próximos, a coerência de raios trata de utilizar o resultado da intersecção de um raio anterior como ponto de partida para encontrar o ponto de intersecção do próximo raio de luz.

Sweeney e Bartels [43] traçam raios sobre superfícies B-spline. Para tal, aplicam algoritmos de refinamento sobre os pontos de controle da superfície e gerar uma árvore de volumes limitantes referente a superfície. A partir disso, o traçado de raios é feito sobre a árvore e o resultado da intersecção com o volume limitante é utilizado para derivar uma solução inicial para o método de Newton. Diversos artigos utilizam a intersecção com o volume limitante como maneira de determinar a solução inicial, incluindo Yang [50], Barth e Stürzlinger [2], Qin et al. [35], Martin et al. [24], Geimer e Abert [17], Parker et al. [31], Valkering [46] e Sloup e Havran [42].

Lischinski e Gonczarowski [21] deliberam sobre a abordagem de intersecção proposta por Joy e Bhetanabhotla [19] e argumentam que, para os métodos Quasi-Newton usados nessa abordagem, mesmo que as técnicas para determinar uma solução inicial funcionem para os casos apresentados, não há garantia que o método de Newton convergirá para a solução correta. Portanto, Lischinski e Gonczarowski [21], em seu trabalho, sem utilizar as demais ferramentas da análise intervalar, subdividem o domínio paramétrico da superfície recursivamente e aplicam o teste de Krawczyk–Moore sobre a região para determinar de antemão a existência de uma única solução e então inicia a versão escalar do método de Newton. Assim como Joy e Bhetanabhotla [19], eles exploram a coerência de raios. Para tal, eles guardam a árvore de subdivisão das superfícies criadas durante o procedimento de intersecção com um raio em uma memória de *cache*, para que seja reutilizada na intersecção com os próximos raios.

O uso de coerência de raios, seja como reutilização das coordenadas de intersecção encontradas como solução inicial do próximo raio [19, 47] ou como reutilização da árvore de subdivisão entre a intersecção de diferentes raios [21], tem resultado em melhoras no desempenho do algoritmo.

Nos casos de teste de Lischinski e Gonczarowski [21], é citado que muitos raios intersectam a caixa limitante da superfície mas não intersectam a superfície propriamente dita. Em suma, na literatura, em sua maioria, vemos esforços para encontrar caixas limitantes mais justas para evitar o número alto de raios que intersectam a caixa limitante mas não intersectam a superfície e esforços para subdividir a superfície e isolar os múltiplos pontos de intersecção existentes. Os tipos de volumes limitantes variam desde simples esferas [48] à caixas alinhadas com eixos [46] (do inglês *Axis-Aligned Bounding Boxes*), paralelepípedos orientados [43] e paralelepípedos oblíquos orientados [1]. Benthin et al. [5] tesselam os retalhos temporariamente para criar volumes limitantes justos e então utiliza uma *heurística de área de superfície*, ou SAH (do inglês *Surface Area Heuristic*), para criar uma estrutura contendo uma *hierarquia de volumes limitantes*, ou BVH (do inglês *Bounding Volume Hierarchy*).Selgrad et al. [41] apresentam uma técnica para comprimir os nós da BVH utilizando as propriedades das superfícies paramétricas. Nos dois artigos mencionados, os volumes limitantes são construídos para traçar raios sobre os triângulos oriundos da tesselação.

Wang, Shih, Chang et al. [47] apresentaram uma abordagem híbrida empregando o método de Newton e o recorte de Bézier – algoritmo a ser apresentado posteriormente – no caso de o método divergir. O método de Newton demonstrou-se computacionalmente mais barato, principalmente com boas soluções iniciais. Entretanto, a necessidade de garantir a convergência do método para a solução correta é a principal desvantagem. Quando sucessivas subdivisões da superfície são feitas para assegurar a solução correta, são gerados diversos sub-retalhos com vários pontos de controles intermediários calculados, culminando num custo computacional maior.

Adicionalmente, Qin et al. [35] propuseram o uso de um método para acelerar a convergência do método de Newton em traçado de raios chamado *extrapolação de polinômio* e utilizaram prismas trapezoidais como volumes limitantes. Geimer e Abert [17] exploraram o paralelismo ao executar os métodos numéricos utilizando instruções SIMD (do inglês *Single Instruction Multiple Data*) em CPU. Parker et al. [31] também exploram o paralelismo ao traçar raios em um servidor distribuído. Na mesma linha, Valkering [46] utilizou centenas de *threads* SIMT (do inglês *Single Instruction Multiple Threads*) disponíveis em placas de vídeo da NVIDIA através da API CUDA para traçar raios sobre retalhos de Bézier extraídos de superfícies NURBS.

2.7.2 Subdivisão recursiva

Whitted [48] realizou o traçado de raios sobre diversos objetos, incluindo superfícies paramétricas, para as quais a intersecção raio-superfície foi calculada a partir

do algoritmo de subdivisão recursiva. Woodward [49] propôs aplicar uma transformação ortogonal dos pontos de controle da superfície e projetá-los sobre um plano de tal forma que o raio de luz seja perpendicular a este plano e intersecte o seu ponto de origem. Feito isto, ele reduz o número de coordenadas dos pontos de controle de três para dois e aplica o algoritmo de subdivisão recursiva sobre a superfície projetada. A cada iteração, o algoritmo de subdivisão recursiva verifica se o raio de luz intersecta a caixa limitante dos pontos de controle do retalho. Se não houve intersecção com a caixa limitante, o retalho pode ser descartado. Caso contrário, o algoritmo escolhe uma das dimensões paramétricas e subdivide o retalho no meio em relação à dimensão escolhida. Como resultado, tem-se dois retalhos menores que, por sua vez, são avaliados recursivamente. O procedimento continua até que o retalho seja pequeno o suficiente para que qualquer ponto dentro dele seja considerado um ponto de intersecção. A Figura 2.1 ilustra as iterações iniciais do algoritmo descrito.

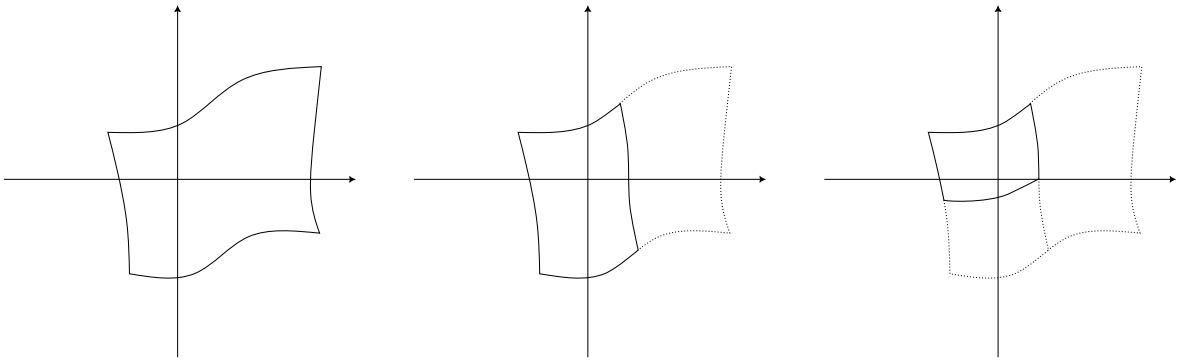


Figura 2.1: Exemplo de iterações iniciais do algoritmo de subdivisão recursiva. O algoritmo projeta o retalho sobre projetado em um plano bidimensional perpendicular ao raio de luz. O raio de luz, por sua vez, intersecta o plano em sua origem. Para verificar a intersecção com o raio de luz, basta verificar-se a caixa limitante do retalho contém a origem do plano de projeção. A cada iteração, o algoritmo escolhe uma das dimensões paramétricas e subdivide o retalho no meio. A dimensão paramétrica escolhida é alternada a cada iteração.

Na literatura, com foco em aplicações interativas, Benthin, Wald e Slusallek [4] empregou o algoritmo de subdivisão recursiva com instruções SIMD em CPU e utilizou *kd-trees* como estrutura de aceleração. Mais recentemente, Binder e Keller [7] adaptou o algoritmo de subdivisão recursiva para execução paralela em GPUs utilizando a API CUDA e gerou imagens traçando raios sobre retalhos de Bézier e de Gregory extraídos de superfícies de subdivisão Catmull-Clark.

2.7.3 Recorte de Bézier

O recorte de Bézier (em inglês: *Bézier clipping*) é um algoritmo considerado um misto entre subdivisão recursiva e método numérico e proposto por Nishita, Sederberg e Kakimoto [28]. No lugar de apenas dividir o domínio paramétrico pela metade a

cada iteração, o algoritmo corta ambas as extremidades do domínio paramétrico com base na posição dos pontos de controle, assim como exemplificado na Figura 2.2. A região restante é chamada de *região de interesse*. Novos pontos de controle para um sub-retalho são calculados e a iteração é repetida até que seu envoltório convexo seja pequeno suficiente e intersectado pelo raio de luz. Sua vantagem em relação ao método de Newton é não precisar de uma solução inicial. Entretanto, quando houver mais de uma intersecção, o algoritmo realizará cortes mais modestos e, neste caso, é necessário subdividir o retalho pela metade e avaliá-los separadamente para isolar os pontos de intersecção distintos. A Figura 2.3 ilustra um caso de raio que intersecta o retalho em dois pontos distintos, para o qual uma etapa de subdivisão é necessária para isolar os pontos.

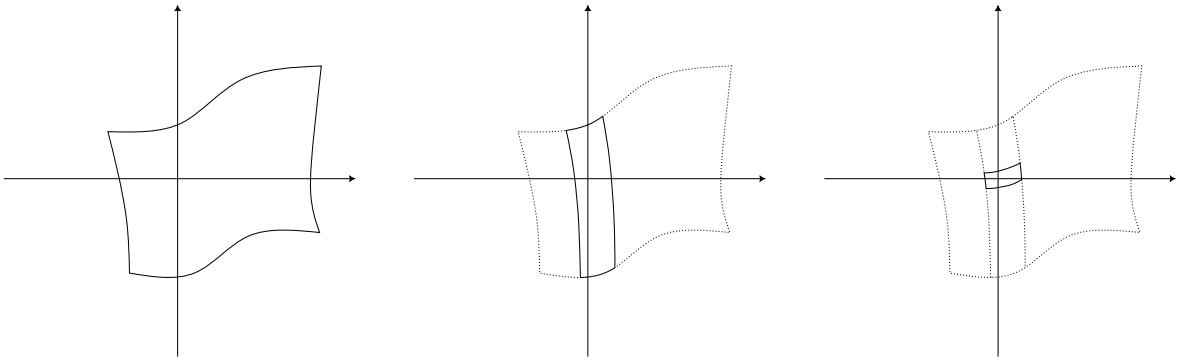


Figura 2.2: Exemplo de iterações iniciais do algoritmo de recorte de Bézier. Assim como na subdivisão recursiva, o recorte de Bézier opera sobre um retalho projetado em um plano bidimensional perpendicular ao raio de luz que intersecta o plano em sua origem. A cada iteração, o algoritmo escolhe uma das dimensões paramétricas e realiza um corte de ambas as extremidades do retalho com base em um cálculo feito a partir dos pontos de controle do retalho. A dimensão paramétrica escolhida é alternada a cada iteração.

Campagna e Slusallek [9] aprimoraram o recorte de Bézier utilizando uma representação alternativa da superfície de Bézier com *polinômios de Chebyshev* proposta por Fournier e Buchanan [16]. A partir da representação alternativa eles criam um volume limitante, chamado de *Chebyshev boxing*, para a superfície cuja intersecção com um raio pode ser calculada mais rapidamente. Entretanto, a técnica não pode ser estendida para retalhos racionais [10].

O recorte de Bézier possui alguns problemas, não especificados em seu artigo original, como a determinação de falsas intersecções, descrito por Campagna e Slusallek [9], que é resolvido com um aumento sutil da região de interesse a cada iteração. Outro problema, chamado de *problema das múltiplas intersecções equivalentes* e detectado por Efremov, Havran e Seidel [12], deve-se ao uso de *retalhos degenerados*, ou seja, com pontos de controle coincidentes. Quando existem dois pontos de controle coincidentes, tem-se uma região do domínio paramétrico que colapsa em um único ponto no espaço euclidiano. Desta maneira se um raio intersecta o retalho neste ponto, a intersecção

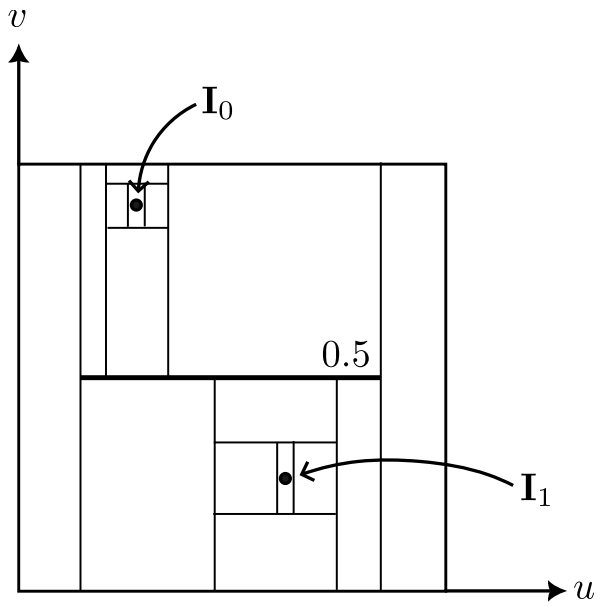


Figura 2.3: Domínio paramétrico de um retalho que é intersectado em dois pontos distintos, $\mathbf{I}_0$ e $\mathbf{I}_1$, por um raio de luz. O primeiro corte na dimensão u é modesto. Portanto, para isolar os pontos, o retalho é subdivido em $v = 0.5$ e, então, o algoritmo prossegue com os cortes em dois retalhos separados.

será atribuída a inúmeras coordenadas do domínio paramétrico que mapeiam para este mesmo ponto. A Figura 2.4 ilustra este caso com retalhos que possuem até quatro pontos de controle coincidentes.

Roth, Diezi e Gross [38] aplicaram o recorte de Bézier em retalhos triangulares. Reshetov [37] adaptou o algoritmo para traçar raios sobre fios de cabelo descritos por curvas de Bézier.

Tejima, Fujita e Matsuoka [44] utilizaram o recorte de Bézier para traçar raios em retalhos extraídos superfícies de subdivisão. Adicionalmente, no final do algoritmo, eles utilizam os pontos de controle para aproximar o retalho com uma superfície bilinear [36] e então realizar a intersecção do raio de luz diretamente com a aproximação bilinear. Seus resultados demonstraram-se competitivos com traçado de raios a partir da tesselação das superfícies e possuem uma considerável economia de memória.

Fougerolle et al. [15] propuseram um algoritmo similar ao recorte de Bézier baseada na construção de uma malha aproximada da superfície com uma margem de erro ε garantida. Assim como no recorte de Bézier, a malha aproximada é projetada num plano perpendicular ao raio de luz e, se o cilindro de raio ε centrado na origem do plano intersectar a malha aproximada, as extremidades do retalho são cortadas e prossegue-se para a próxima iteração até que seja determinado que o raio não intersectou o retalho ou que a margem de erro ε tenha atingindo uma precisão aceitável.

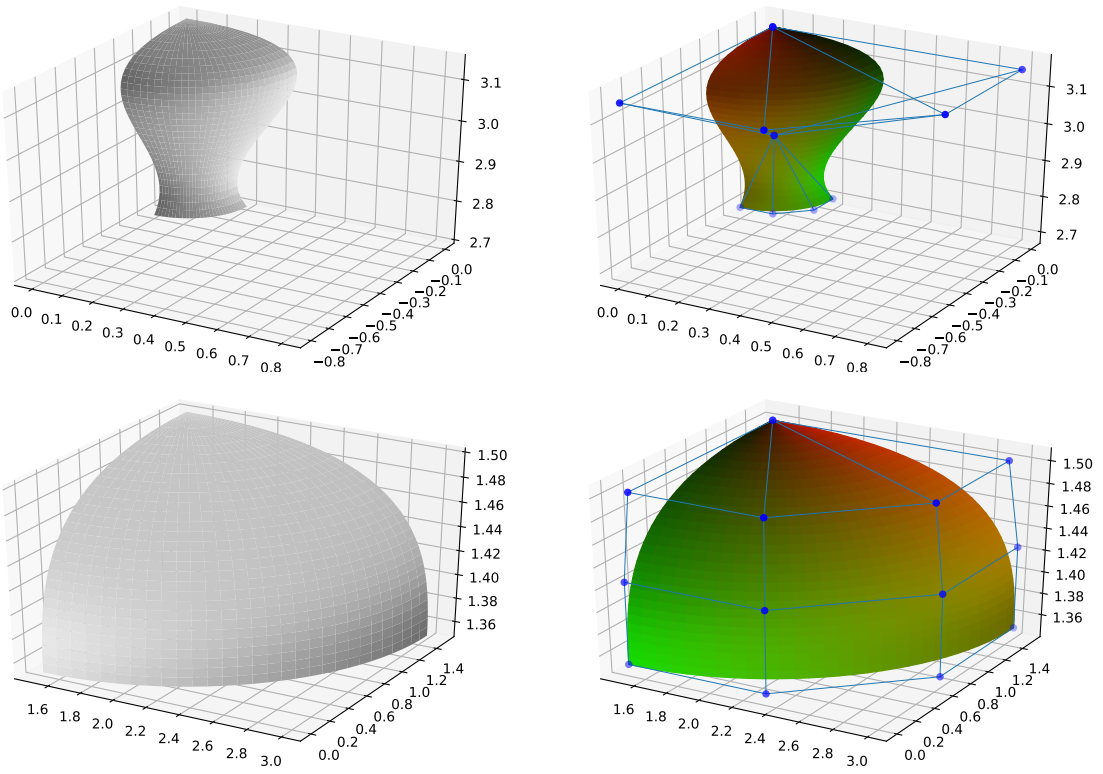


Figura 2.4: Ilustração de retalhos com pontos de controle coincidentes. Na coluna mais à direita, o domínio paramétrico com as cores vermelho e verde para cada dimensão, a malha de controle e seus pontos estão em evidência. Em ambos exemplos, quatro pontos de controle coincidem em um único ponto mais ao topo.

2.8 Extração de retalhos de Bézier

Até então apresentamos métodos utilizados na literatura para realizar a intersecção entre um raio e um retalho de Bézier. Para estender o traçado de raios a outros tipos de superfícies — mais especificamente, superfícies de subdivisão Catmull-Clark [11] e T-splines [40], ambas bicúbicas — admitiremos que estas podem ser decompostas em retalhos cujas funções de base podem ser definidas em termos das funções de base de Bézier, isto é, os polinômios de Bernstein, Equação (2.3).

De fato, pode-se provar que a superfície limite resultante do processo de subdivisão de Catmull-Clark — considerando-se malhas de controle regulares, isto é, em cujos vértices incidem quatro arestas (exceto nas bordas) — é uma coleção de retalhos de B-splines bicúbicas. A conversão de retalhos de B-splines em retalhos de Bézier não é difícil e pode ser encontrada, por exemplo, em [33]. Similarmente, cada face da malha de controle de uma T-spline, ou T-malha, gera um retalho da superfície que pode ser associado a um retalho de Bézier. Em ambos os casos, pontos sobre um retalho de superfície, p, para o qual contribuem n_p pontos de controle da malha de controle,

podem ser expressos matricialmente por

$$\mathbf{S}_p(u,v) = \mathbf{C}_p\,\mathbf{B}(u,v)\,\mathbf{P}_p = \mathbf{N}_p(u,v)\,\mathbf{P}_p, \tag{2.13}$$

em que os valores das funções de base de Bézier são organizadas em

$$\mathbf{B}(u,v) = \begin{bmatrix} B_{00}(u,v) & B_{01}(u,v) & \dots & B_{32}(u,v) & B_{33}(u,v) \end{bmatrix},$$

$B_{ij}(u,v) = B_{i,3}(u)B_{j,3}(v)$, $i,j = 0,\dots,3$, e $\mathbf{P}_p$ é o vetor com a sequência dos n_p pontos de controle do retalho. A matriz $\mathbf{C}_p$, de dimensão $n_p \times 16$, é chamada de *operador de extração de Bézier* do retalho p. A função de base do retalho é $\mathbf{N}_p(u,v) = \mathbf{C}_p\mathbf{B}(u,v)$. A extração de Bézier consiste na determinação de n_p, do vetor $\mathbf{P}_p$ e, claro, no operador $\mathbf{C}_p$. Para B-splines bicúbicas (derivadas da subdivisão de Catmull-Clark), $n_p = 16$, sendo os pontos de controle e o operador de extração dados pelo algoritmo de conversão de B-splines em retalhos de Bézier (veja [33]). Para T-splines, detalhes da extração de Bézier podem ser encontrados em [39]. Ambos os procedimentos foram implementados em outros trabalhos do nosso grupo de pesquisa (veja [32]).

2.9 Considerações finais

Neste capítulo, efetuou-se um resumo de superfícies paramétricas, em especial, aquelas representadas por retalhos de Bézier. Outros tipos importantes de representação, tais como superfícies de subdivisão e T-splines, podem ter a forma de seus retalhos relacionada às mesmas funções de base de Bézier, ou seja, polinômios de Bernstein, através de um processo de conversão denominado extração de Bézier. Este consiste na determinação de um operador, dado por uma matriz, que aplicado às funções de base de Bézier resulta na função de base específica de retalhos de B-spline e T-spline. Em adição, o capítulo descreve o algoritmo de traçado de raios básico a ser utilizado neste trabalho, bem como uma revisão da literatura dos principais métodos de intersecção de um raio com retalhos de Bézier, com destaque para métodos numéricos, subdivisão recursiva e recorte de Bézier. Destes, os algoritmos baseados em métodos numéricos consistem em escolher um ponto de intersecção aproximado inicial e aprimorar sua precisão através de sucessivas iterações. Já a subdivisão recursiva não necessita de solução inicial e a cada iteração, subdivide o retalho em duas ou quatro partes recursivamente, verificando a intersecção através de volumes limitantes até que o retalho seja pequeno o suficiente para determinar um ponto. Foi visto que da mesma forma que a subdivisão recursiva, o recorte de Bézier verifica a intersecção através de volumes limitantes,

porém, a cada iteração, exceto quanto houver a possibilidade de mais de uma solução, o retalho não é subdivido, mas tem suas extremidades podadas até que seja pequeno o suficiente para determinar um ponto de intersecção. Levando-se em conta que os métodos numéricos dependem de um valor inicial das coordenadas paramétricas do pontos de intersecção, tal que o cálculo da intersecção é sensível a esses valores iniciais, optou-se neste trabalho pela utilização dos métodos de subdivisão e recorte de Bézier para a intersecção raio/retalho.

Capítulo 3

Desenvolvimento

Nesta seção, estão descritos apenas os passos fundamentais e as peculiaridades da implementação dos algoritmos de intersecção em GPU. Os algoritmos utilizam com frequência os conceitos apresentados no Capítulo 2. Em particular, a Seção 2.5 é essencial para os processos de subdivisão e recorte dos retalhos. Toda a implementação encontra-se disponível em um repositório público no *GitHub* através do link `https://github.com/MachSilva/cgspline`.

Tanto em CPU como em GPU, foi implementada a versão mais simples do algoritmo descrito na Seção 2.6, isto é, com o traçado de um único raio ao centro de cada pixel de imagem. Para a aceleração do cálculo de intersecção de um raio com a superfície paramétrica, foi utilizada uma estrutura contendo uma hierarquia de volumes limitantes, ou BVH (do inglês *Bounding Volume Hierarchy*), de retalhos de todas as superfícies que compõem a cena. Como o volume limitante de cada retalho, foi considerada a caixa limitante dos pontos de controle que influenciam a forma do retalho. Para a intersecção de um raio com um retalho de Bézier, foi implementada a técnica de recorte de Bézier descrita na Seção 2.7.

A Seção 3.1 discorre brevemente sobre a implementação do traçado de raios em CPU. A Seção 3.2 detalha todas as adaptações necessárias feitas para realizar o traçado de raios em GPU bem como a travessia na BVH na Seção 3.2.1, a intersecção raio/retalho por meio da subdivisão recursiva na Seção 3.2.2 e por meio do recorte de Bézier na Seção 3.2.3.

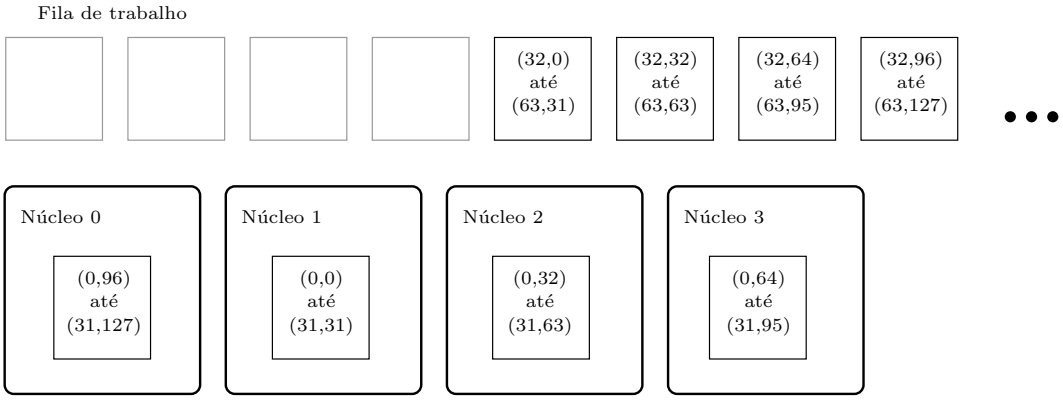


Figura 3.1: Ilustração do modelo de execução do traçador de raios em CPU. Cada *núcleo* retira um item por vez da fila de trabalho para determinar qual parte da imagem renderizar.

3.1 Traçado de raios em CPU

Em CPU, o modelo de execução utilizado foi o de *"worker thread pool"*, isto é, foi lançado um número fixo de *threads* que formam um grupo de trabalho com uma fila de tarefas compartilhada. Enquanto houver tarefas agendadas na fila, cada *thread* ficou responsável por retirar e realizar a próxima tarefa da fila. No contexto do traçado de raios, a imagem a ser renderizada foi divida em vários ladrilhos de 32×32 *pixels* e uma tarefa foi criada para cada ladrilho contendo sua posição e tamanho na imagem final. Em outras palavras, cada *thread* renderizava um ladrilho por vez até que a fila estivesse vazia e a imagem final estivesse completa. A depender da cena sendo traçada, algumas regiões da imagem final demandam mais trabalho computacional do que outras. A Figura 3.1 fornece uma visão geral deste processo. O traçado de raios em CPU utiliza o recorte de Bézier como método de intersecção raio/retalho.

3.2 Traçado de raios em GPU utilizando CUDA

Em GPU, a API de programação paralela CUDA da NVIDIA foi utilizada. Foram escritos *kernels* específicos para o cálculo de intersecção de um raio com retalhos de superfícies paramétricas. O modelo de execução de CUDA e de GPUs em geral é baseada na arquitetura *Single Instruction Multiple Threads*, na qual diversas *threads* são capazes de realizar operações lógico-aritméticas em diversos registradores ao mesmo tempo e de ler e escrever diversas posições de memória com uma única instrução despachada. Conforme ilustrada pela Figura 3.2, cada *thread* é agrupada em blocos e os blocos são agrupados em uma grade. As coordenadas de uma *thread* em um bloco e em uma grade são utilizadas para determinar qual *pixel* da imagem final será renderizado. Um raio de *pixel* foi alocado para cada *thread*.

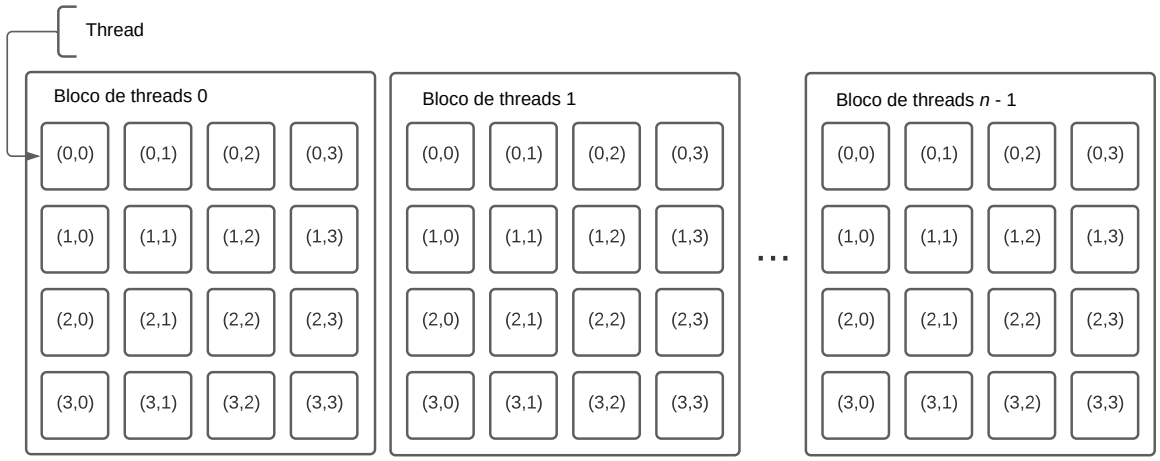


Figura 3.2: Modelo de programação CUDA, no qual as *threads* estão agrupadas em blocos de *threads* que por sua vez foram uma grade de *threads*. Este é o modelo lógico, no qual cada *thread* é identificada por uma coordenada na grade e uma coordenada num bloco e não representa o que ocorre de fato no *hardware*.

3.2.1 Bounding Volume Hierarchy

A abordagem padrão para a travessia da BVH consiste em uma busca por profundidade pelos nós que são intersectados pelo raio de luz. Isto exige manter uma pilha de nós da BVH em memória. Em virtude das limitações da unidade gráfica, o espaço necessário para armazenar a pilha torna-se considerável, visto que cada *thread* dentre centenas precisaria armazenar a própria pilha, aumentando o uso de registradores da unidade gráfica e o número de requisições a memória local, o que pode resultar em tempos de espera maiores, fenômeno conhecido como *stall*, até que o valor requisitado da memória esteja disponível. Um uso maior de registradores por *thread* diminui a quantidade de *threads* que a unidade gráfica consegue executar simultaneamente. De maneira similar ao que foi feito em [7] e descrito por Binder et al., o algoritmo de travessia da BVH foi implementado com uma adaptação para eliminar a necessidade de uma pilha contendo os endereços dos nós superiores na hierarquia, diminuindo o uso de memória.

Em linhas gerais, partindo de uma árvore binária como BVH, cada nó é rotulado com um número inteiro, que posteriormente será referido como *chave*, de maneira que seja possível identificar os nós relativos apenas a partir de sua chave. Seja n um número inteiro que identifica um nó. Os nós filhos esquerdo e direito são definidos respectivamente pelas funções $\texttt{left}(n) = 2n$ e $\texttt{right}(n) = 2n + 1$. Consequentemente, infere-se que o nó pai é dado pela função $\texttt{parent}(n) = \lfloor n/2 \rfloor$ e que o nó irmão, o esquerdo a partir do direito ou o direto a partir do esquerdo, é dado pela inversão do bit menos significativo do número inteiro por meio de sua representação em binário: $\texttt{sibling}(n) = n \oplus 1$. Por simplicidade, tomou-se o valor *um* (1) como a chave do nó

raiz. Desta maneira, resta apenas uma maneira de acessar as informações do nó através de sua respectiva chave. Assim como em [6], após a construção da BVH, gera-se uma função de *hash* perfeita para mapear a chave do nó para um índice referente a um vetor de nós da BVH.

Função de *hash* perfeita É uma função $H : \mathbb{Z} \to \mathbb{N}$ tal que, dado um conjunto pré-determinado de números inteiros S, $\forall p, q \in S : p \neq q \implies H(p) \neq H(q)$.

Dada a BVH completa, coletam-se as chaves dos nós existentes que devem ser mapeadas. Seja S o conjunto de chaves coletadas. Sejam a e b dois inteiros positivos aleatórios, sendo a e $|S|$ primos entre si. Para uma chave arbitrária $k \in S$, a expressão $ak+b \mod |S|$ serve como uma função de *hash*. Entretanto, a probabilidade de colisões é alta. Em busca de uma função de *hash* perfeita, adiciona-se à expressão um vetor de deslocamento com a finalidade de resolver colisões existentes na expressão original. Dito isto, define-se uma função de *hash*

$$H(k) = (ak + b + D_{k \mod |D|}) \mod |S|$$

na qual D é o vetor de deslocamento referido que possui um conjunto de chaves associadas a ele através da expressão $k \mod |D|$.

O tamanho escolhido do vetor D que demonstrou-se satisfatório foi a maior potência de 2 menor que a metade de $|S|$, para permitir usar operação `AND` *bit-a-bit* no lugar de uma divisão completa. Logo, $|D| = 2^m$ para um $m \in \mathbb{N}$. Com os coeficientes a e b escolhidos, resta resolver os conflitos preenchendo os valores do vetor de deslocamento da seguinte maneira:

- agrupe as chaves que compartilharão o mesmo valor de deslocamento num mesmo conjunto $B_i = \{k \mid k \in S,\ k \mod |D| = i\}$;
- ordene os conjuntos de chaves agrupados anteriormente, do maior para o menor;
- inicialize um vetor temporário T com zero para marcar quais índices do vetor de nós foram alocados;
- para cada conjunto B_i, na ordem especificada, aplique a função de *hash* em cada chave para obter seu índice:
  - se houver colisão entre os índices das chaves e os índices já marcados no vetor T, altere o valor de D_i no vetor de deslocamento e repita esta operação; (se esta operação falhar diversas vezes, reinicie a abordagem com novos valores aleatórios para a e b)

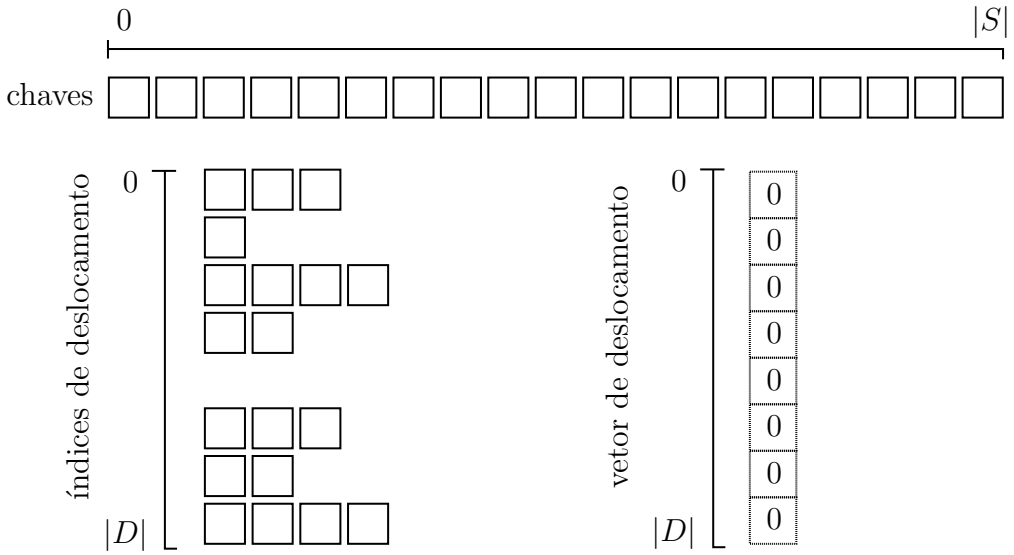


Figura 3.3: Ilustração do conjunto de chaves de cardinalidade $|S|$ utilizando pequenos quadrados para representar cada chave de valor de inteiro sendo agrupadas a partir do seu índice no vetor de deslocamento. Para cada índice de deslocamento i, haverá um conjunto B_i de chaves correspondentes.

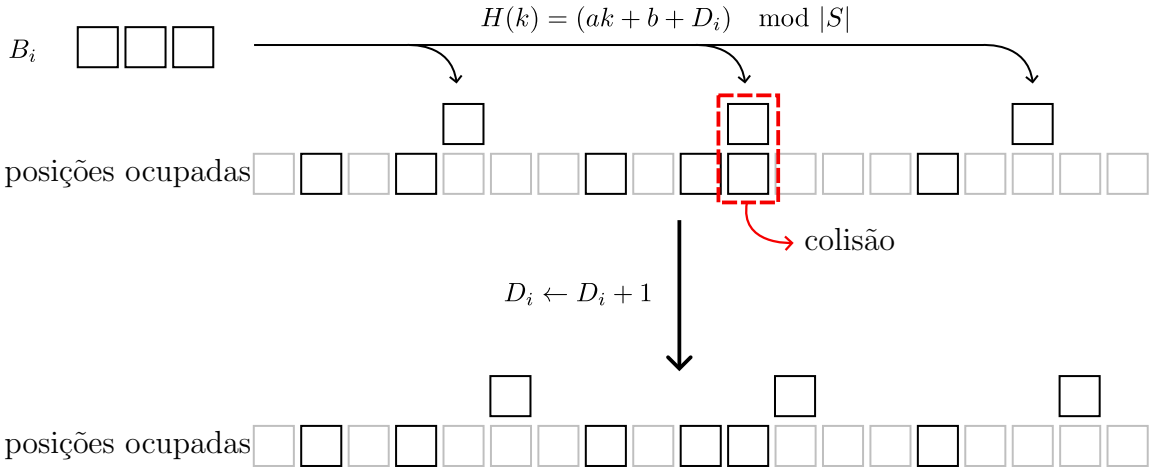


Figura 3.4: Ilustração de como o algoritmo resolve colisões ao alterar o valor de deslocamento D_i durante a inserção de chaves do conjunto B_i.

– caso contrário, marque os índices das chaves no vetor T como alocados.

A Figura 3.3 ilustra o primeiro passo de agrupar as chaves com o mesmo valor de deslocamento em conjuntos B_i. A Figura 3.4 ilustra último passo: o processo de inserção das chaves e a correção do valor de deslocamento relativo ao conjunto de chaves. Desta maneira, implementou-se um algoritmo para preencher os valores do vetor de deslocamento.

Por fim, o Algoritmo 3.1 descreve como a travessia da BVH é realizada durante a intersecção de um objeto constituído de retalhos. A estrutura de um nó da BVH possui quatro campos nomeados *caixaEsq*, *caixaDir*, *índiceEsq* (índice do nó esquerdo no vetor de nós), *índiceDir* (índice do nó direito). Um nó folha é codificado com *índiceEsq* $= \varnothing$ e, conforme ocorre na linha 33, *índiceDir* passa a possuir o índice do primeiro primitivo

de uma sequência de primitivos (ou retalhos) que estão contidos pela caixa limitante do nó folha. A função `ctz`, do inglês *count trailing zeros*, conta a quantidade de *bits* zeros consecutivos a partir do *bit* menos significativo da representação binária de um valor inteiro. Os símbolos $\ll, \gg, \oplus, \wedge$ e $\vee$ representam respectivamente as operações binárias *shift left*, *shift right* e operações lógicas ***ou*** *exclusivo bit a bit*, ***e*** *bit a bit* e ***ou*** *bit a bit*.

3.2.2 Subdivisão recursiva

A implementação da intersecção raio/retalho por meio da subdivisão recursiva é feita com uma busca em profundidade e ela precisou de uma alteração significativa: a remoção dos sub-retalhos da pilha de busca. Visto que, de acordo com a documentação oficial [29, Seção 5.3.2], vetores que não são indexados por valores inteiros constantes, como uma pilha, muito provavelmente são alocados na memória local que possui uma latência de acesso considerável. Sendo assim, guardar um sub-retalho para cada item da pilha colocaria muita pressão sobre a memória local de cada *thread*. A pilha de busca foi substituída por uma representação com um inteiro sem sinal e um inteiro contando a profundidade atingida. A variável de tipo inteiro, chamada de *trilha*, codifica o percurso percorrido na árvore de profundidade como uma pilha de *bits*.

Esta abordagem é similar ao realizado por [7], porém com duas diferenças fundamentais. A primeira diferença é, que de forma similar a [49], o retalho foi projetado em um plano bidimensional perpendicular ao raio de luz e com o raio de luz intersectando a origem do plano. Isto diminui em um terço a quantidade de componentes de cada ponto de controle do retalho e quantidade de operações requeridas para subdividí-lo. Em contrapartida, com esta projeção não é possível determinar a distância das caixas limitantes dos sub-retalhos em relação à origem do raio. Consequentemente, o algoritmo não decide qual dos dois ramos da árvore de busca será percorrido primeiro com base nesta distância. Quando o sub-retalho é divido em duas partes, o algoritmo realiza a busca na primeira parte, empilhando o *bit* zero na variável *trilha* através da operação $\mathit{trilha} \leftarrow 2 \cdot \mathit{trilha}$. Posteriormente, ao realizar a busca na segunda parte, o algoritmo empilha o *bit* um através da operação $\mathit{trilha} \leftarrow 2 \cdot \mathit{trilha} + 1$. A cada *bit* empilhado, a dimensão paramétrica em que ocorre o corte é trocada.

A cada iteração da busca, o sub-retalho correspondente é reconstruído a partir do histórico de subdivisões armazenado na variável *trilha*. Para facilitar a reconstrução, duas variáveis $\mathit{trilha}U$ e $\mathit{trilha}V$ representando o histórico de subdivisões de cada dimensão são atualizadas juntamente com *trilha*. Detalhes desta implementação estão no Algoritmo 3.2.

3.2.3 Recorte de Bézier

A implementação em GPU do recorte de Bézier precisou de algumas adaptações, principalmente pela quantidade limitada de registradores disponíveis. Se a quantidade de registradores não for suficiente para armazenar os dados da computação, cada *thread* precisará salvar os dados na memória local. Com centenas de *threads* acessando frequente e simultaneamente a memória local para realizar os seus cálculos, a memória e o *cache L1* ficarão saturados e deixarão as *threads* em espera até que as requisições sejam concluídas. Em outras palavras, para evitar a degradação do desempenho, é necessário diminuir a quantidade de variáveis locais o tanto quanto possível, diminuindo então o que a documentação chama de *register pressure* [29, Seção 9.2.7].

Uma adaptação foi diminuir o tamanho da estrutura que mantém o estado da busca. Como mencionado na Seção 2.7.3, o algoritmo mantém uma pilha de estados para lidar com casos em que o mesmo raio intersecta o retalho em pontos distintos. O retalho então é divido pela metade e o algoritmo precisa encontrar uma intersecção em uma metade, recarregar da pilha o estado guardado relativo à outra metade e executar novamente a procura da próxima intersecção. Na versão em CPU, cada estado de busca possui os campos: *patch*, sub-retalho com as extremidades podadas nas iterações anteriores do algoritmo, que é equivalente à região de interesse; *min*, menor valor das coordenadas paramétricas delimitando a região de interesse, ou seja, a região do retalho original que ainda não foi podada; *max*, maior valor das coordenadas paramétricas que delimitam a região de interesse; *size*, a área de região de interesse, calculada como $max - min$; e *cutside*, a dimensão do domínio paramétrico na qual o algoritmo deverá realizar o recorte na próxima iteração. Enquanto na versão em CPU seja vantajoso guardar estas informações e evitar calculá-las novamente a cada iteração, em GPU, apenas os campos *min* e *max* foram guardados na pilha de estados e o campo *cutside*, por registrar apenas uma das duas dimensões paramétricas u e v possíveis, foi armazenado numa variável local que operou como uma pilha de *bits*. Operações de baixo nível como *shift left* e *shift right* foram utilizadas para empilhar e remover da pilha o valor do campo *cutside*. Assim como mencionado anteriormente, visto que estrutura de dados como uma pilha são muito provavelmente alocados na memória local, cuja latência de acesso é considerável, esta abordagem na prática mostrou-se mais apropriada. O restante da implementação assemelha-se ao artigo original e ao que foi descrito no Capítulo 2.

Algoritmo 3.1: Travessia de BVH sem pilha.

Entrada: Raio de luz $\mathbf{R}$, vetor de nós V e vetor de elementos (retalhos) a serem intersectados E.

Saída : Elemento intersectado e e distância t do ponto de intersecção até a origem do raio $\mathbf{R}$.

```
1  função IntersectBVH(R,V,E)
2      e ← ∅;                          /* elemento intersectado mais próximo */
3      t ← ∞;                        /* distância da intersecção mais próxima */
4      k ← 1;                      /* a chave do nó atual começando pela raiz */
5      atual ← V[H(k)];                                      /* nó atual */
6      postergado ← 0;
7      enquanto verdadeiro faça
8          se atual não for folha então
9              (hitL, L0, L1) ← IntersectAABB(atual.caixaEsq, R);
10             (hitR, R0, R1) ← IntersectAABB(atual.caixaDir, R);
11             hitL ← hitL ∧ (L0 < t);
12             hitR ← hitR ∧ (R0 < t);
13             se hitL ∨ hitR então
14                 postergado ← postergado ≪ 1;
15             se hitL então
16                 se hitR então
17                     postergado ← postergado ∨ 1;
18                     se L0 < R0 então
19                         k ← k ≪ 1;                        /* left(k) = 2k */
20                         atual ← V[atual.índiceEsq];
21                     senão
22                         k ← (k ≪ 1) ∨ 1;                /* right(k) = 2k +1 */
23                         atual ← V[atual.índiceDir];
24                 senão
25                     k ← k ≪ 1;
26                     atual ← V[atual.índiceEsq];
27                 pule para a próxima iteração;
28             se hitR então
29                 k ← (k ≪ 1) ∨ 1;
30                 atual ← V[atual.índiceDir];
31                 pule para a próxima iteração;
32         senão
               /* se for um nó folha                                        */
33             i ← atual.índiceDir;
34             enquanto P[i] ≠ ∅ faça
35                 distância ← Intersect(E[i], R);
36                 se distância < t então t ← distância, e ← E[i];
37         se postergado ≠ 0 então retorna (e, t);
38         n ← ctz(postergado);                    /* count trailing zeros */
39         postergado ← (postergado ≫ n) ⊕ 1;
40         k ← (k ≫ n) ⊕ 1;
41         atual ← V[H(k)];
```

Algoritmo 3.2: Intersecção raio/retalho a partir da subdivisão recursiva.

Entrada: Raio de luz $\mathbf{R}$ e retalho $\mathbf{P}$ projetado num plano bidimensional perpendicular à direção do raio de luz.

```
1  função IntersectarRetalho(R,P)
2      recarregar ← falso;
3      profundidade ← −1;
4      (trilha, trilhaU, trilhaV) ← (0, 0, 0);
5      Q ← P;                                              /* sub-retalho */
6      enquanto profundidade < 0 faça
7          s_0 ← 2^{−⌊trilhaU/2⌋};        /* comprimento da região de interesse */
8          s_1 ← 2^{−⌊(trilhaV−1)/2⌋};      /* largura da região de interesse */
9          r_min ← (trilhaU · s_0, trilhaV · s_1);
10         r_max ← r_min + (s_0, s_1);
11         se recarregar então Q ← SubRetalho(P, r_min, r_max);
12         se a origem (0, 0) está contida em AABB(Q) então
13             se o tamanho de Q estiver dentro da tolerância então
14                 I ← IntersectarQuad(P, r_min, r_max);
15                 se I ≠ ∅ então ReportarIntersecção(I);
16         senão
17             se profundidade for ímpar então
18                 SubRetalhoEmU(Q, 0, 1/2);
19                 trilhaU ← trilhaU ≪ 1;
20             senão
21                 SubRetalhoEmV(Q, 0, 1/2);
22                 trilhaV ← trilhaV ≪ 1;
23             recarregar ← falso;
24             trilha ← trilha ≪ 1;
25             profundidade ← profundidade − 1;
26             pule para a próxima iteração;
27         recarregar ← verdadeiro;
           /* contar bits uns consecutivos menos significativos      */
28         uns ← ctz(¬trilha);
29         se profundidade for ímpar então
30             trilhaU ← trilhaU ≪ ⌊uns/2⌋;                    /* piso */
31             trilhaV ← trilhaV ≪ ⌈uns/2⌉;                    /* teto */
32             se uns for ímpar então trilhaU ← trilhaU + 1;
33             senão trilhaV ← trilhaV + 1;
34         senão
35             trilhaV ← trilhaV ≪ ⌊uns/2⌋;                    /* piso */
36             trilhaU ← trilhaU ≪ ⌈uns/2⌉;                    /* teto */
37             se uns for ímpar então trilhaV ← trilhaV + 1;
38             senão trilhaU ← trilhaU + 1;
39         trilha ← (trilha ≫ uns) + 1;
40         profundidade ← profundidade + uns;
```

Capítulo 4

Resultados

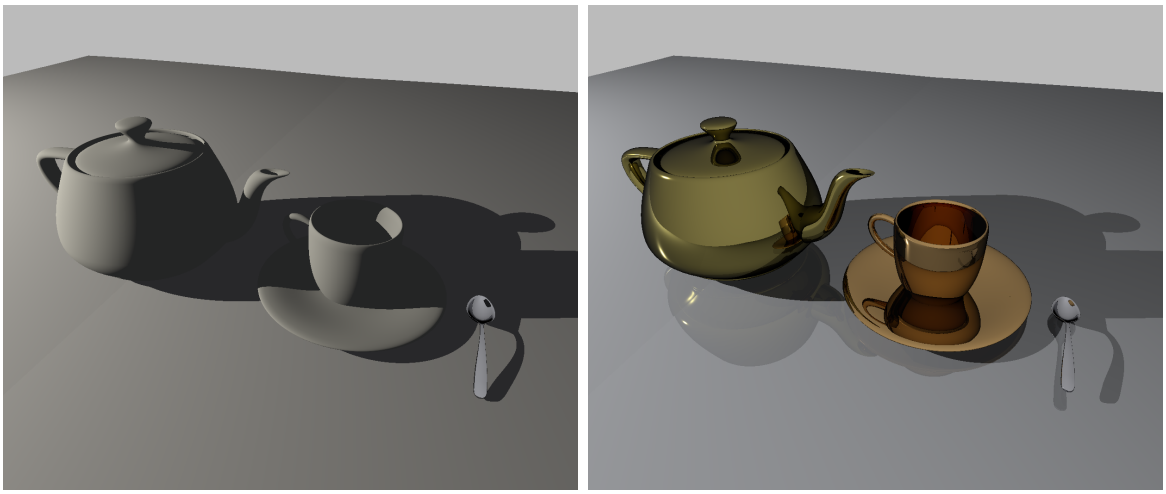
Este capítulo contém a avaliação do traçador de raios com algoritmos de subdivisão recursiva e recorte de Bézier implementados. A Seção 4.1 mostra as cenas criadas para testar o traçado de raios e descreve suas principais características. A Seção 4.2 registra o ambiente computacional de teste utilizado bem como os tempos de execução alcançados pelo traçador de raios em cada cena. Por fim, a Seção 4.3 delibera sobre aspectos dos resultados obtidos.

4.1 Cenas

As cenas foram construídas utilizando o jogo de chá de Newell[1] (em inglês: *Newell teaset*), composto pelos modelos *Teapot* (também conhecido como *Utah teapot*), *Teacup* e *Teaspoon* (ver Figura 4.1).

As cenas destinadas para teste foram nomeadas com o prefixo "Teste" e o número de objetos que ela contém. Cada uma delas possui o modelo *Teapot* com material reflexivo no centro e um número arbitrário de conjuntos de um *Teacup* e dois *Teaspoons* ao redor conforme a Figura 4.2. Isto foi feito para verificar como o número de retalhos impactaria no tempo de execução.

[1]Os modelos e programas de leitura relacionados estão publicamente disponíveis no repositório `https://github.com/rm-hull/newell-teapot`. Último acesso em: 07/06/2022.



(a) Teaset01 (b) Teaset02

Figura 4.1: Jogo de chá de Newell contendo três objetos: a colher com 16 retalhos bicúbicos; a xícara com 26 retalhos e chaleira com 32 retalhos. Adicionalmente, as cenas contém um piso levemente deformado modelado com 4 retalhos. Na Figura 4.1(a), todos os objetos, com exceção da colher, são tonalizados com um material não-metálico com foco na reflexão difusa. A Figura 4.1(b) mostra o jogo de chá tonalizado com materiais metálicos e reflexivos e o piso refletindo levemente o jogo de chá.

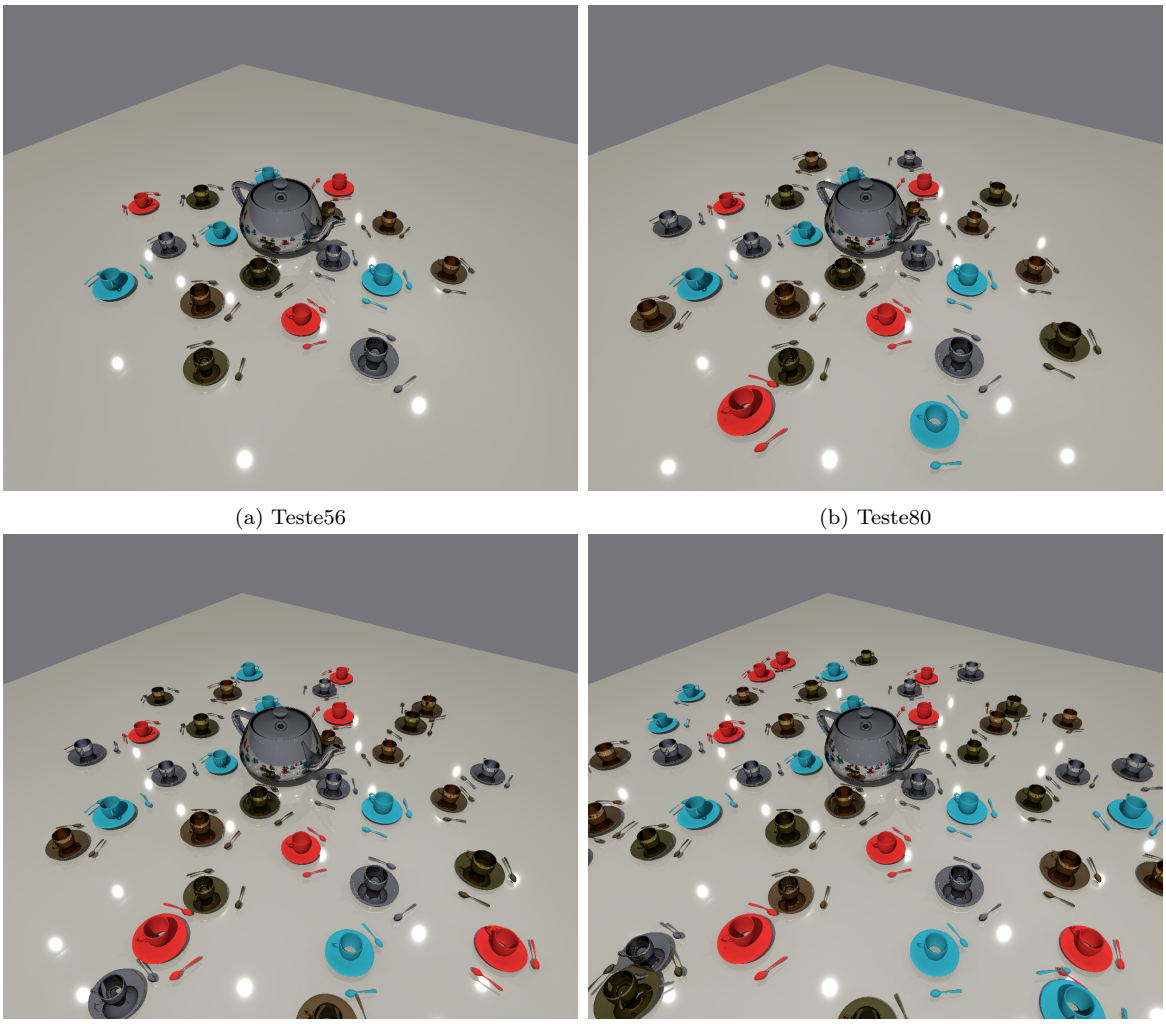
Tabela 4.1: Descrição das cenas de teste, ordenadas pela quantidade de objetos. Todas as cenas possuem um piso levemente curvo formado por 4 retalhos.

Cena	Luzes	Objetos	Teapot	Teacup	Teaspoon	Retalhos
Teaset01	1	4	1	1	1	78
Teaset02	1	4	1	1	1	78
Teste32	11	32	1	10	20	616
Teste56	19	56	1	18	36	1080
Teste80	27	80	1	26	52	1544
Teste104	35	104	1	34	68	2008
Teste128	43	128	1	42	84	2472
Teste152	51	152	1	50	100	2936

4.2 Desempenho

Os testes de desempenho do algoritmo de traçado de raios foram realizados em um *laptop* executando uma distribuição *Linux*[2] com um processador Intel Core i5-10300H a 4.5 GHz com 4 núcleos físicos e 8 núcleos lógicos e uma placa de vídeo NVIDIA GTX 1650 Mobile com *compute capability* 7.5 e 14 SMs (SM: *simultaneous multiprocessor*), 64 KB de *cache L1* por SM. Todos os *kernels* foram compilados com a opção "`-maxrregcount=168`". Todas as imagens foram geradas com uma resolução de 1242×1048 *pixels* durante a realização dos testes. Para cada cena, os testes foram realizados em séries de nove execuções, tanto em CPU e quanto em GPU. As Tabelas 4.2 e 4.3 expõem o tempo médio de execução em milissegundos e o desvio padrão de cada série como também a quantidade de raios processados por segundo.

[2]Versão do *kernel*: 6.13. Versão do módulo gráfico proprietário: 550.144.03.



(a) Teste56 (b) Teste80

(c) Teste104 (d) Teste152

Figura 4.2: Cenas geradas para os testes. Cada cena possui um bule de chá no centro e um número variável de objetos ao redor. A diferença entre uma cena e outra é a quantidade de objetos e de luzes, visíveis pela reflexão especular do piso. A posição da câmera é a mesma em todas.

Tabela 4.2: Resultado dos testes de desempenho em CPU, com o tempo médio de execução em milissegundos e a velocidade de processamento em milhões de raios por segundo.

CPU			
Cena	Tempo médio (ms)	Desvio padrão	MRaios/s
Teaset01	814.27	39.32	2.89
Teaset02	1578.1	43.62	2.52
Teste32	4730.28	121.85	3.01
Teste56	9226.9	168.04	2.52
Teste80	15856.97	215.28	2.06
Teste104	23079.12	279.72	1.84
Teste128	31000.83	61.32	1.68
Teste152	43135.23	371.89	1.46

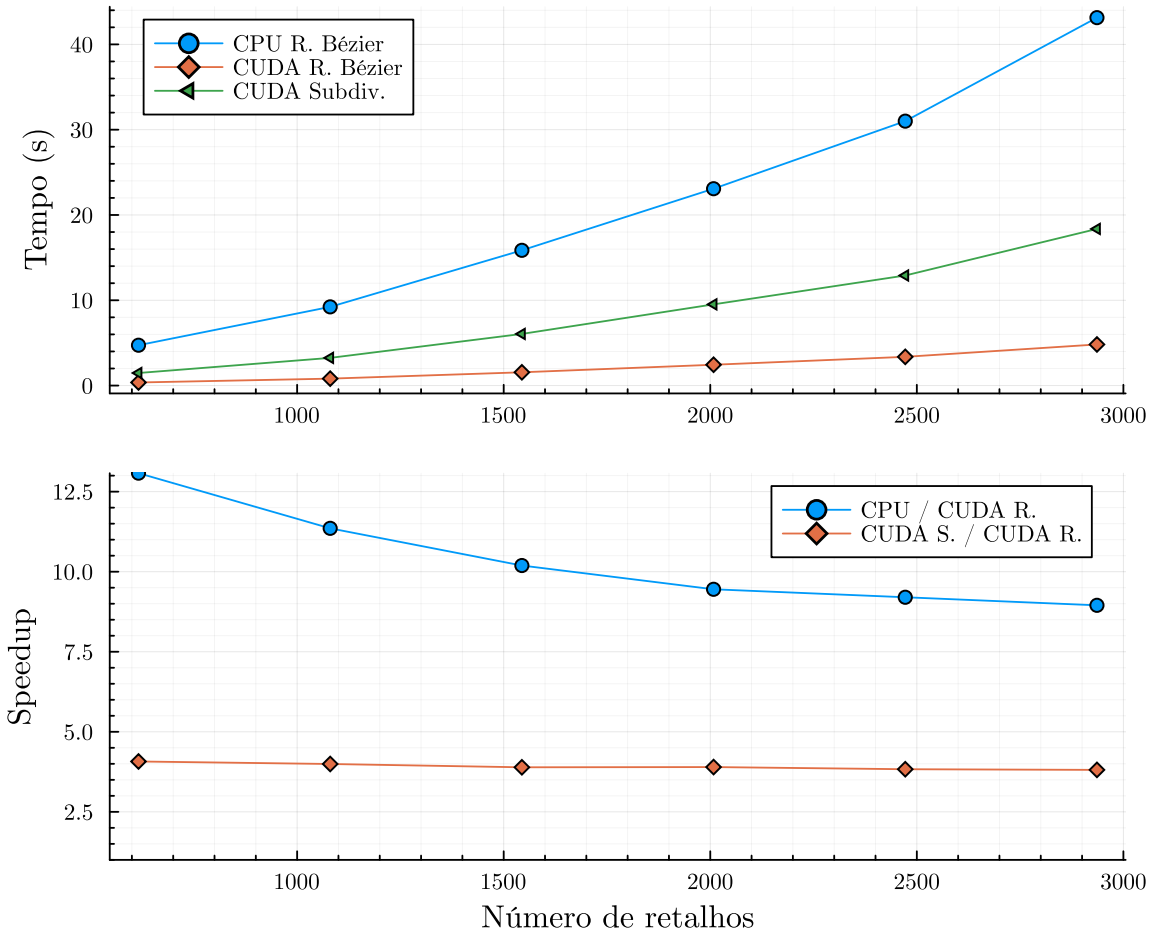


Figura 4.3: Gráfico da evolução dos tempos de execução, em segundos, conforme a quantidade de retalhos renderizados dentro da cena cresce para cenas nomeadas com o prefixo “Teste”. O segundo gráfico mostra os ganhos de desempenho dados pela razão do tempo de execução da versão em CPU sobre a versão em GPU/CUDA e pela razão entre as duas versões em CUDA: versão com subdivisão sobre versão com recorte de Bézier.

Tabela 4.3: Resultado dos testes de desempenho em GPU/CUDA para cada método, com o ganho relativo à CPU calculado como *Tempo CPU/ Tempo GPU*.

CUDA / Recorte de Bézier				
Cena	Tempo médio (ms)	Desvio padrão	MRaios/s	Ganho relativo à CPU
Teaset01	42.32	0.78	55.69	19.24
Teaset02	99.59	9.29	40.0	15.85
Teste32	361.71	14.6	39.37	13.08
Teste56	812.62	15.76	28.64	11.35
Teste80	1555.72	15.01	21.00	10.19
Teste104	2442.04	7.24	17.41	9.45
Teste128	3368.91	18.20	15.47	9.20
Teste152	4819.40	14.68	13.06	8.95
CUDA / Subdivisão recursiva				
Cena	Tempo médio (ms)	Desvio padrão	MRaios/s	Ganho relativo à CPU
Teaset01	157.12	9.50	15.00	5.18
Teaset02	419.19	15.10	9.50	3.76
Teste32	1473.38	16.11	9.67	3.21
Teste56	3246.18	26.45	7.17	2.84
Teste80	6054.25	25.35	5.40	2.62
Teste104	9520.80	46.34	4.47	2.42
Teste128	12906.09	49.72	4.04	2.40
Teste152	18365.01	56.12	3.43	2.35

4.3 Análise

Ambos os algoritmos performaram melhor em GPU do que em CPU. A versão que utiliza recorte de Bézier manteve-se em média $3,9$x mais rápida que a versão que utiliza subdivisão. O ganho de desempenho alcançado da versão de recorte de Bézier foi até 19x mais rápida em cenas mais simples sem muitos objetos com superfícies reflexivas. A cena "Teste152" é a cena que possui o maior número de retalhos com material reflexivo. Os raios traçados nesta cena ricochetearam em diversos retalhos e a versão em GPU com recorte de Bézier foi pelo menos $8,95$x mais rápida que a versão em CPU.

Para uma análise mais aprofundada, foi utilizado o programa *Nsight Compute* da NVIDIA para coletar amostras da utilização da placa de vídeo durante o traçado de raios utilizando a cena "Teaset02". Os dados obtidos para a versão de subdivisão foram uso: de $64,83\%$ do *pipeline* da FMA (do inglês: *Fused Multiply Add*) e $17,01\%$ do *pipeline* da ALU (do inglês: *Arithmetic Logic Unit*), com uma média de $1,83$ instruções executadas por ciclo e uma média de $8,78$ *threads* ativas por *warp*, grupo de 32 *threads* que executam simultaneamente cada instrução despachada, o que indica alta divergência. Já para versão com recorte de Bézier, os dados obtidos foram: uso de $22,8\%$ do *pipeline* da FMA e $8,83\%$ do pipeline da ALU, com uma média de $0,84$

instruções por ciclo e uma média de $15,79$ de *threads* ativas por *warp*, o que indica uma divergência menor que o algoritmo anterior.

A GPU passou mais tempo ociosa executando a versão do traçado de raios com recorte de Bézier do a versão com subdivisão visto que o uso dos *pipelines* de computação são bem menos utilizados, mesmo com uma divergência menor. Amostras adicionais do estado de cada *warp* coletados pelo *Nsight Compute* fortalecem essa informação ao mostrar que, para cada instrução despachada para o grupo de *threads*, o grupo gastou em média $4,77$ ciclos esperando uma instrução estar disponível (métrica correspondente: *stall no instruction*) e $2,35$ ciclos esperando dados serem carregados a partir da memória (métrica correspondente: *stall long scoreboard*).

Estes dados evidenciam que a versão do traçado de raios com recorte de Bézier é limitado pelo acesso à memória, enquanto a versão com subdivisão é limitado pela quantidade de cálculos realizados.

Mesmo não operando em condições ideais, o traçado de raios com o algoritmo de recorte de Bézier obteve um ganho considerável de desempenho quando executado em GPU. Por não operar em condições ideais, ainda há margem para o algoritmo que obteve melhor desempenho em GPU.

Capítulo 5

Conclusão

5.1 Revisão dos objetivos propostos

Os objetivos propostos no início deste trabalho foram:

- Estudo dos métodos de intersecção raio/retalho de Bézier: objetivo alcançado após uma extensa revisão da literatura acadêmica, na qual foram destacados três métodos promissores descritos no Capítulo 2 juntamente com todos fundamentos necessários para entendê-los;
- Implementação em GPU de dois dos três métodos mencionados anteriormente em um traçador de raios para superfícies compostas de retalhos de Bézier: objetivo concluído, cujos frutos observam-se reunidos em um repositório público[1] acompanhados de detalhes de implementação descritos no Capítulo 3;
- Realização de testes de desempenho do traçador de raios implementado: objetivo alcançado com resultados documentados no Capítulo 4 juntamente com as cenas geradas.

Verificou-se até aqui que a GPU é superior à CPU no traçado de raios em diversos tipos de cenas e, mesmo nas mais difíceis, o melhor algoritmo alcançou um *speedup* de ao menos 8x e é capaz de alcançar até 19x a depender da cena e da posição da câmera. Assim sendo, pode-se considerar que todos os objetivos do trabalho foram alcançados com sucesso.

[1] `https://github.com/MachSilva/cgspline`

5.2 Trabalhos futuros

Apesar de os algoritmos terem sido implementados usando a API CUDA, eles também podem ser implementados em *shaders* de computação para OpenGL e Vulkan, permitindo que o traçado de raios seja feito em uma variedade maior de dispositivos. Uma melhoria que soa promissora para remover gargalos e fazer o traçado de raios realizado neste trabalho utilizar toda capacidade da placa de vídeo possível é dividir o cálculo de um raio entre dois ou quatro *threads* usando funções existentes de comunicação entre si quando necessário para diminuir a pressão sobre os registradores e a memória.

Conforme foi dito em outra seção do texto, existem várias representações de superfícies que, conforme foi verificado na literatura, são compatíveis com retalhos de Bézier, isto é, é possível determinar um operador linear que transforma as funções de base de uma B-Spline e T-Spline para Bézier permitindo que os algoritmos mencionados aqui sejam utilizados para traçar raios sobre outros tipos de superfícies, incluindo superfícies de subdivisão Catmull-Clark. Portanto, gostaríamos de estender este trabalho para demais superfícies em trabalhos futuros.

Referências Bibliográficas

[1] Wihelm Barth, Roland Lieger e Michael Schindler. "Ray tracing general parametric surfaces using interval arithmetic". Em: *The Visual Computer* 10.7 (1994), pp. 363–371.

[2] Wilhelm Barth e Wolfgang Stürzlinger. "Efficient ray tracing for Bezier and B-spline surfaces". Em: *Computers & Graphics* 17.4 (1993), pp. 423–430.

[3] Carsten Benthin, Ingo Wald e Philipp Slusallek. "Interactive ray tracing of free-form surfaces". Em: *Proceedings of the 3rd international conference on Computer graphics, virtual reality, visualisation and interaction in Africa*. 2004, pp. 99–106.

[4] Carsten Benthin, Ingo Wald e Philipp Slusallek. "Techniques for interactive ray tracing of Bézier surfaces". Em: *Journal of Graphics Tools* 11.2 (2006), pp. 1–16.

[5] Carsten Benthin et al. "Efficient ray tracing of subdivision surfaces using tessellation caching". Em: *Proceedings of the 7th Conference on High-Performance Graphics*. 2015, pp. 5–12.

[6] Nikolaus Binder e Alexander Keller. "Efficient stackless hierarchy traversal on GPUs with backtracking in constant time." Em: *High Performance Graphics*. 2016, pp. 41–50.

[7] Nikolaus Binder e Alexander Keller. "Massively Parallel Stackless Ray Tracing of Catmull-Clark Subdivision Surfaces". Em: *arXiv preprint arXiv:1811.03510* (2018).

[8] Michael J. Borden et al. "Isogeometric finite element data structures based on Bézier extraction of NURBS". Em: *International Journal for Numerical Methods in Engineering* 87.1-5 (2011), pp. 15–47.

[9] Swen Campagna e Philipp Slusallek. "Improving bézier clipping and chebyshev boxing for ray tracing parametric surfaces". Em: *Proceedings of 3D Image Analysis and Synthesis*. Vol. 96. 1996, pp. 95–102.

[10] Swen Campagna, Philipp Slusallek e Hans-Peter Seidel. "Ray tracing of spline surfaces: Bézier clipping, Chebyshev boxing, and bounding volume hierarchy a critical comparison with new results". Em: *The Visual Computer* 6.13 (1997), pp. 265–282.

[11] Edwin Catmull e James Clark. "Recursively generated B-spline surfaces on arbitrary topological meshes". Em: *Computer-aided design* 10.6 (1978), pp. 350–355.

[12] Alexander Efremov, Vlastimil Havran e Hans-Peter Seidel. "Robust and numerically stable Bézier clipping method for ray tracing NURBS surfaces". Em: *Proceedings of the 21st spring conference on Computer graphics*. 2005, pp. 127–135.

[13] Wolfgang Enger. "Interval ray tracing—a divide and conquer strategy for realistic computer graphics". Em: *The Visual Computer* 9.2 (1992), pp. 91–104.

[14] Gerald E. Farin. *Curves and surfaces for CAGD: a practical guide*. Morgan Kaufmann, 2002.

[15] Yohan D. Fougerolle et al. "A geometric algorithm for ray/Bézier surfaces intersection using quasi-interpolating control net". Em: *2008 IEEE International Conference on Signal Image Technology and Internet Based Systems*. IEEE. 2008, pp. 451–457.

[16] Alain Fournier e John Buchanan. "Chebyshev polynomials for boxing and intersections of parametric curves and surfaces". Em: *Computer Graphics Forum*. Vol. 13. 3. Wiley Online Library. 1994, pp. 127–142.

[17] Markus Geimer e Oliver Abert. "Interactive ray tracing of trimmed bicubic Bézier surfaces without triangulation". Em: (2005).

[18] John A. Gregory. "Smooth interpolation without twist constraints". Em: *Computer aided geometric design*. Elsevier, 1974, pp. 71–87.

[19] Kenneth I. Joy e Murthy N. Bhetanabhotla. "Ray tracing parametric surface patches utilizing numerical techniques and ray coherence". Em: *ACM SIGGRAPH Computer Graphics* 20.4 (1986), pp. 279–285.

[20] James T. Kajiya. "Ray tracing parametric patches". Em: *ACM SIGGRAPH Computer Graphics* 16.3 (1982), pp. 245–254.

[21] Daniel Lischinski e Jakob Gonczarowski. "Improved techniques for ray tracing parametric surfaces". Em: *The visual computer* 6.3 (1990), pp. 134–152.

[22] Charles Loop. "Smooth Subdivision Surfaces Based on Triangles". Diss. de mestr. Jan. de 1987. URL: https://www.microsoft.com/en-us/research/publication/smooth-subdivision-surfaces-based-on-triangles/.

[23] Joakim Löw. *Ray Tracing Bézier Surfaces on GPU*. 2006.

[24] William Martin et al. "Practical ray tracing of trimmed NURBS surfaces". Em: *Journal of Graphics Tools* 5.1 (2000), pp. 27–52.

[25] Ramon E. Moore e C. T. Yang. *Interval analysis*. Vol. 2. Prentice-Hall Englewood Cliffs, N. J., 1996.

[26] Matthias Nießner et al. "Feature-adaptive GPU rendering of Catmull-Clark subdivision surfaces". Em: *ACM Transactions on Graphics (TOG)* 31.1 (2012), pp. 1–11.

[27] Rohit Nigam. "Efficient Ray Tracing of Parametric Surfaces for Advanced Effects". Tese de doutorado. International Institute of Information Technology Hyderabad, 2015.

[28] Tomoyuki Nishita, Thomas W. Sederberg e Masanori Kakimoto. "Ray tracing trimmed rational surface patches". Em: *Proceedings of the 17th annual conference on Computer graphics and interactive techniques*. 1990, pp. 337–345.

[29] NVIDIA Corporation. *CUDA C++ Programming Guide*. 2024. URL: `https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html` (acesso em 22/11/2024).

[30] Hans-friedrich Pabst et al. "Ray casting of trimmed NURBS surfaces on the GPU". Em: *2006 IEEE Symposium on Interactive Ray Tracing*. IEEE. 2006, pp. 151–160.

[31] Steven Parker et al. "Interactive ray tracing". Em: *ACM SIGGRAPH 2005 Courses*. 2005, 12–es.

[32] Marcio Artacho Peres. "Elementos de Contorno para Análise Isogeométrica de Sólidos Elásticos com Vincos". Tese de doutorado. 2021.

[33] Les Piegl e Wayne Tiller. *The NURBS book*. Springer Science & Business Media, 1996.

[34] Pixar. *OpenSubdiv*. 2012. URL: `https://graphics.pixar.com/opensubdiv/docs/intro.html` (acesso em 03/02/2022).

[35] Kaihuai Qin et al. "A new method for speeding up ray tracing NURBS surfaces". Em: *Computers & Graphics* 21.5 (1997), pp. 577–586.

[36] Shaun D. Ramsey, Kristin Potter e Charles Hansen. "Ray bilinear patch intersections". Em: *Journal of Graphics Tools* 9.3 (2004), pp. 41–47.

[37] Alexander Reshetov. "Exploiting Budan-Fourier and Vincent's theorems for ray tracing 3D Bézier curves". Em: *Proceedings of High Performance Graphics*. 2017, pp. 1–11.

[38] S. H. Martin Roth, Patrick Diezi e Markus H. Gross. "Ray tracing triangular Bézier patches". Em: *Computer graphics forum*. Vol. 20. 3. Wiley Online Library. 2001, pp. 422–432.

[39] Michael A. Scott et al. "Isogeometric finite element data structures based on Bézier extraction of T-splines". Em: *International Journal for Numerical Methods in Engineering* 88.2 (2011), pp. 126–156.

[40] Thomas W. Sederberg et al. "T-splines and T-NURCCs". Em: *ACM transactions on graphics (TOG)* 22.3 (2003), pp. 477–484.

[41] Kai Selgrad et al. "A compressed representation for ray tracing parametric surfaces". Em: *ACM Transactions on Graphics (TOG)* 36.1 (2016), pp. 1–13.

[42] Jaroslav Sloup e Vlastimil Havran. "Optimizing Ray Tracing of Trimmed NURBS Surfaces on the GPU". Em: *Computer Graphics Forum*. Vol. 40. 7. Wiley Online Library. 2021, pp. 161–172.

[43] Michael A. J. Sweeney e Richard H. Bartels. "Ray tracing free-form B-spline surfaces". Em: *IEEE Computer Graphics and Applications* 6.2 (1986), pp. 41–49.

[44] Takahito Tejima, Masahiro Fujita e Toru Matsuoka. "Direct ray tracing of full-featured subdivision surfaces with bezier clipping". Em: *Journal of Computer Graphics Techniques (JCGT)* 4.1 (2015), pp. 69–83.

[45] Daniel L. Toth. "On ray tracing parametric surfaces". Em: *ACM SIGGRAPH Computer Graphics* 19.3 (1985), pp. 171–179.

[46] Erik Valkering. "Ray Tracing NURBS Surfaces using CUDA". Diss. de mestr. 2010.

[47] Shyue-Wu Wang, Zen-Chung Shih, Ruei-Chuan Chang et al. "An efficient and stable ray tracing algorithm for parametric surfaces". Em: *Journal of Information Science and Engineering* 18.4 (2002), pp. 541–561.

[48] Turner Whitted. "An improved illumination model for shaded display". Em: *Proceedings of the 6th annual conference on Computer graphics and interactive techniques*. 1979, p. 14.

[49] Charles Woodward. "Ray tracing parametric surfaces by subdivision in viewing plane". Em: *Theory and practice of geometric modeling*. Springer, 1989, pp. 273–287.

[50] Chang-Gui Yang. “On speeding up ray tracing of B-spline surfaces”. Em: *Computer-Aided Design* 19.3 (1987), pp. 122–130.