André Ferreira Bem Silva

Estudo direcionado para a disciplina INE651100

Estudo direcionado para a disciplina de Engenharia de Sistemas Operacionais com o tema "Renderização interativa baseada em propriedades físicas"

Orientador:
Aldo von Wangenheim

Co-orientador:
Tiago de Holanda Cunha Nobrega

Universidade Federal de Santa Catarina

Departamento de Informática e Estatística

Florianópolis, 17 de maio de 2010

André Ferreira Bem Silva

Estudo direcionado para a disciplina INE651100

Departamento de Informática e Estatística

Florianópolis, 17 de maio de 2010

Estudo dirigido de título “Estudo direcionado para a disciplina INE651100”, feito para a disciplina INE651100, “Engenharia de Sistemas Operacionais”, redigido por André Ferreira Bem Silva no Departamento de Informática e Estatísticada Universidade Federal de Santa Catarina. Avaliado por Tiago de Holanda Cunha Nobrega e aprovado em Florianópolis, 17 de maio de 2010, por:

Prof. Dr. rer. nat. Aldo von Wangenheim
Universidade Federal de Santa Catarina

Tiago de Holanda Cunha Nobrega
Grupo Cyclops

Prof. Dr. Antônio Augusto Fröhlich
Universidade Federal de Santa Catarina

Lista de Algoritmos

1 traceRay . . . . p. 8
2 buildBVH . . . . p. 10
3 traverseAABBTree . . . . p. 13
4 traverseKDTree . . . . p. 13
5 recTraverseA . . . . p. 14
6 Intersecção entre raio e esfera descrita por Ericson (Ericson 2004) . . . . p. 15
7 rayTriIntersect . . . . p. 16
8 cachedRayTriIntersect . . . . p. 17
9 Intersecção raio-caixa . . . . p. 18

Lista de Figuras

1.1 Visão humana e funcionamento de uma câmera . . . . p. 9

1.2 Visão geral do raytracing . . . . p. 9

1.3 Exemplos de BVHs . . . . p. 10

1.4 Cenas e KDTree . . . . p. 12

1.5 Overview de travessia . . . . p. 13

1.6 Casos de intersecção raio-esfera . . . . p. 14

2.1 Organização das bibliotecas C3DE, em diagrama de componentes. . . . . . . p. 20

2.2 Diagrama representativo da situação atual do raytracer. . . . . p. 20

2.3 Comparação entre os algoritmos *naive* e BVH para travessia, em segundos. O ganho de desempenho da BVH em relação a primeira é mostrado pela terceira função. . . . . p. 23

2.4 Tempos, em segundos, de execução para diferentes quantidades de Threads do raytracer implementado. . . . . p. 23

2.5 Comparação de execuções por segundo das versões, normais, SSE e GPU dos referidos algoritmos de intersecção. O teste foi realizado no PC 1. . . . . . . p. 24

Sumário

Organização

1 Introdução p. 7

1.1 Meta de longo prazo . . . . p. 7

1.2 Técnica . . . . p. 7

1.3 BVH . . . . p. 8

1.3.1 Criação . . . . p. 10

1.3.2 Travessia . . . . p. 11

1.4 Intersecções com raio . . . . p. 12

1.4.1 Raio-esfera . . . . p. 14

1.4.2 Raio-triângulo . . . . p. 15

1.4.3 Raio-caixa . . . . p. 17

2 Implementação p. 19

2.1 Implementações com vetorização . . . . p. 19

2.2 Processamento paralelo . . . . p. 21

2.3 Resultados . . . . p. 21

2.4 Discussão conclusiva . . . . p. 22

Referências Bibliográficas p. 25

3 Anexos p. 26

3.1 Implementação de Vec4 e Float4 . . . . p. 26

3.2 Intersecções SSE . . . . p. 30
3.2.1 Raio-caixa . . . . p. 30
3.2.2 Raio-triângulo . . . . p. 31
3.3 Intersecções OpenCL . . . . p. 32
3.3.1 Raio-caixa . . . . p. 32
3.3.2 Raio-triângulo . . . . p. 32

Organização

Esse documento demonstra o estudo direcionado para a disciplina de Engenharia de Sistemas Operacionais, avaliado e co-orientado por Tiago de Holanda Cunha Nobrega, orientado por Aldo von Wangenheim, com o tema "Renderização interativa baseada em propriedades físicas".

Esse documento organiza-se da seguinte forma: O capítulo 1 demonstra as idéias respectivas ao raytracing e a sua aceleração; No capítulo 2 demonstra-se as implementações feitas, seus resultados e a discussão desses; Em seguida estão as referências usadas nesse estudo; Por fim o capítulo 3 de códigos anexos, não necessários para a compreensão do intuito e resultados do estudo.

1 Introdução

Com o objetivo de implementar uma “Renderização interativa baseada em propriedades físicas”, esse estudo direcionado foca-se em técnicas de aceleração das estruturas básicas que compõe um raytracing.

1.1 Meta de longo prazo

Traduzir a implementação de primitivas gráficas realizadas em CPU para estruturas que rodem em uma GPU, ao mesmo tempo realizando um estudo de métricas para avaliação de capacidade de renderização CPU vs GPU em determinadas cenas. Dada essa avaliação, criar um escalonador adaptativo que utilize a CPU e GPU para renderizar uma cena. Isso será feito a partir das bibliotecas matemáticas C3DE-CORE e C3DE-AL (Silva et al. 2009), desenvolvidas pelo grupo Cyclops, e o protótipo de renderização baseada em propriedades físicas para criar estruturas que acelerem suas funcionalidades, utilizando GLSL (Rost 2006) ou *Open Computing Language* (OpenCL)(Stone 2009) para acessar a GPU.

1.2 Técnica

Raytracing é uma técnica de renderização que baseia-se em nos fenômenos físicos que compõe a visão humana. Percebemos os raios que originam-se do sol serem refletidos pelos objetos e esses reflexos, por sua vez, serem recebidos por bastonetes que captam frequências relacionadas a vermelho, amarelo e azul. Assim percebemos a imagem invertida, a qual é invertida novamente pelo cérebro. No caso de uma câmera, algo relacionado acontece, porém a luz que reflete nos objetos pode ser lançada também pela câmera na hora de tirar a foto. Ambos processos podem ser observados na figura 1.1.

Para implementar-se em uma unidade de processamento um algoritmo que seja equivalente ao raytracing, faz-se da seguinte forma: para cada pixel de saída, lança-se ao menos um raio,

a partir da posição da câmera, verificando-se com quais geometrias e volumes o raio colide, todos contribuindo para o valor final do pixel. Essa técnica é denominada backward raytracing e encontra-se ilustrada na figura 1.2 e descrita em pseudocódigo no algoritmo 1. Assim, a complexidade geral do raytracing torna-se dependente da complexidade do algoritmo de intersecção (*intersects*) com as primitivas, do número de luzes presentes no ambiente para o cálculo da cor final (*shadeObject*), do número de recursões máximas do raytracing e tudo isso multiplicado pelo número de pixeis. Isso demonstra o quão custoso computacionalmente um raytracer pode-se facilmente tornar-se, principalmente em implementações que utilizem algoritmos de força bruta.

Algoritmo 1: traceRay

```
input  : ray, depth, maxDepth
output: pixel color
color ← BLACK;
hit ← intersects(object, objects, ray);
if hit exists then
    color ← shadeObject(object, ray, hit);        Shade and texture that object
    if depth < maxDepth then
        reflectedColor ← traceRay(reflectedRay, depth + 1);      Reflected ray
        color = reflectedColor * object.reflectivity;
    if depth < maxDepth then
        refractedColour ← traceRay(refractedRay, depth + 1);     Refracted ray
        color = color + refractedColor * objectRefractivity;
return color;
```

1.3 BVH

Para tornar as computações complexas do raytracing mais aceleradas, são utilizadas algumas estruturas de aceleração que tornem mais eficazes a travessia pelas geometrias do mundo. As estruturas mais comuns de serem usadas são as *Bounding Volume Hierarchies* (BVHs). Especificamente, a KDTree e a AABBTree apresentam interessante para cenas diversas e são estruturas altamente paralelizáveis (Lauterbach et al. 2009, Günther et al. 2007). Exemplos de ambas são ilustradas em 1.3a e 1.3b.

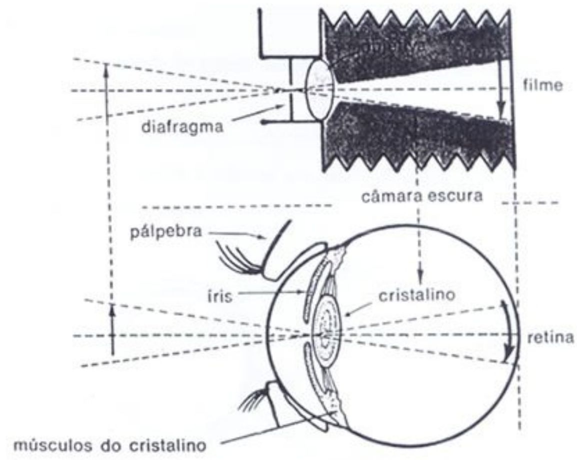


Figura 1.1: Visão humana e funcionamento de uma câmera

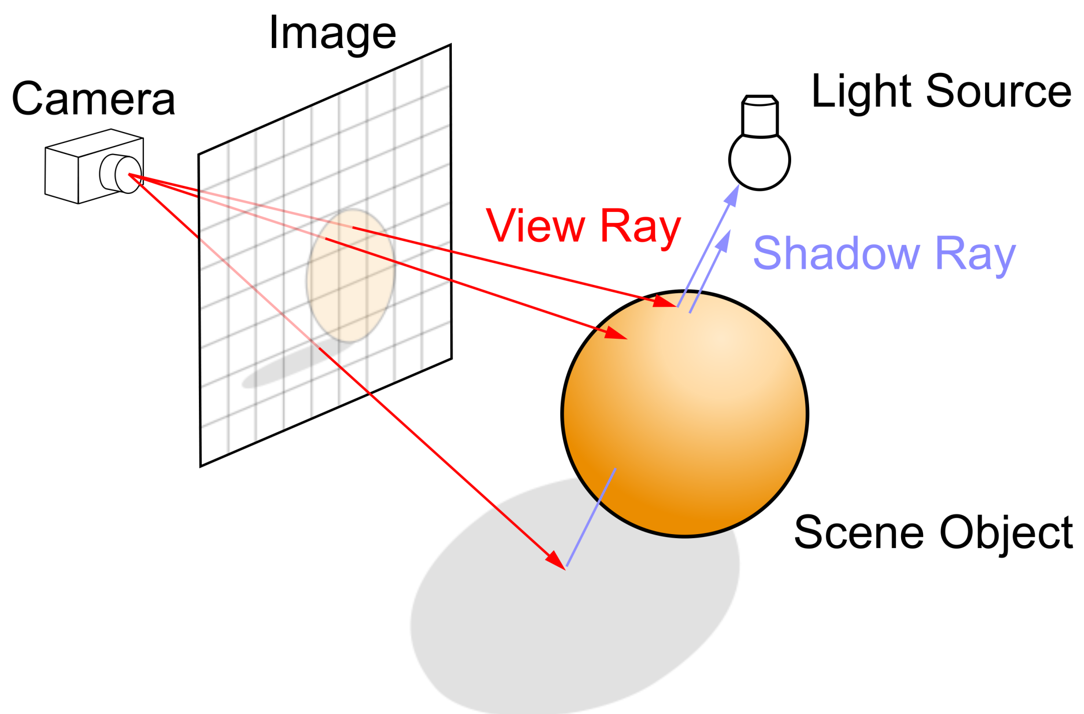
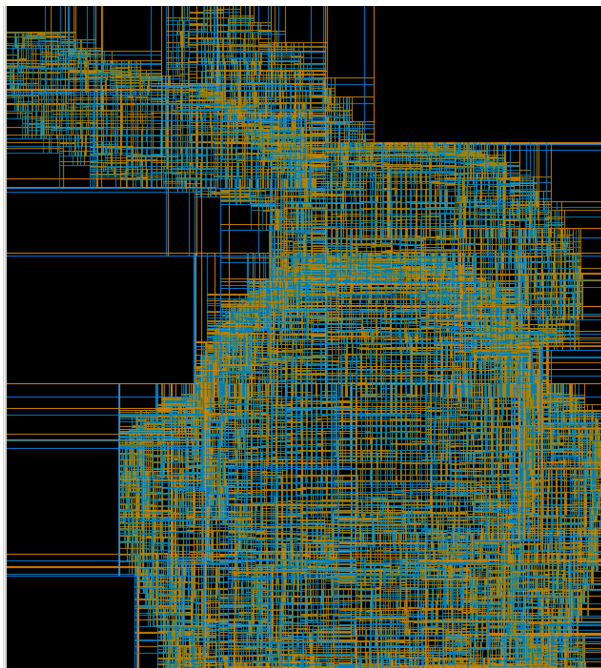
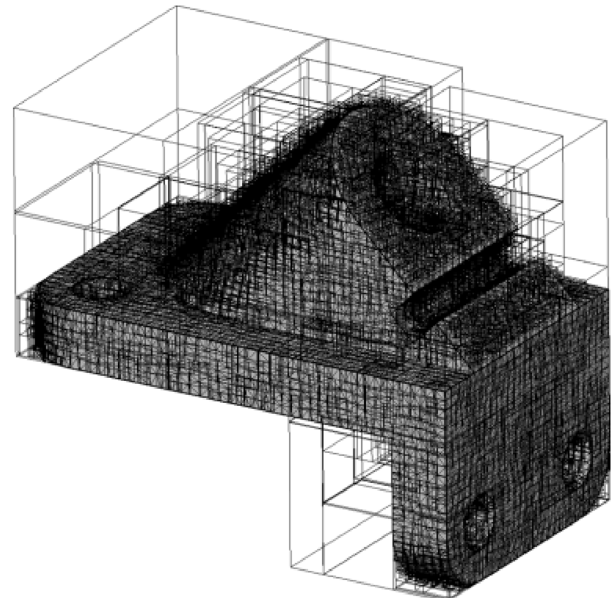


Figura 1.2: Visão geral do raytracing



(a) KDTree de uma mesh de coelho



(b) ABBTree de uma peça naval (âncora)

Figura 1.3: Exemplos de BVHs

1.3.1 Criação

O algoritmo de criação básico para as BVHs pode ser visto em 2. Ele segue basicamente da mesma forma para todas as estruturas, modificando-se no que se refere a critérios de parada e a como cria-se os filhos. Implementações mais eficientes para criação de estruturas hierárquicas podem ter complexidade $O(n \log n)$, mesmo em GPU (Lauterbach et al. 2009).

Algoritmo 2: buildBVH

```
input  : node
output: The complete BVH
if stopCriterionIsMet() then
    return;
initSplit(node.left, node.right);
for x ∈ node.objects do
    if node.left.intersects(x) then
        node.left.add(x);
    if node.right.intersects(x) then
        node.left.add(x);
buildBVH(node.left);
buildBVH(node.right);
```

Surface Area Heuristic (SAH)

A KDTree é um caso genérico de BSPTrees, onde os *splits* encontram-se implementados de acordo com alguma política de divisão. Assim, pode-se definir, por exemplo uma heurística de acordo com a área e o número de geometrias num determinado nodo:

$$cost = TC + costLeft + costRight \tag{1.1}$$

$$costRight = rightArea * rightObjs * IC * (1 - bonus) \tag{1.2}$$

$$costLeft = leftArea * leftObjs * IC * (1 - bonus) \tag{1.3}$$

$$bonus = \begin{cases} K, & rightObjs = 0 \vee leftObjs = 0 \\ 0, & otherwise \end{cases} \tag{1.4}$$

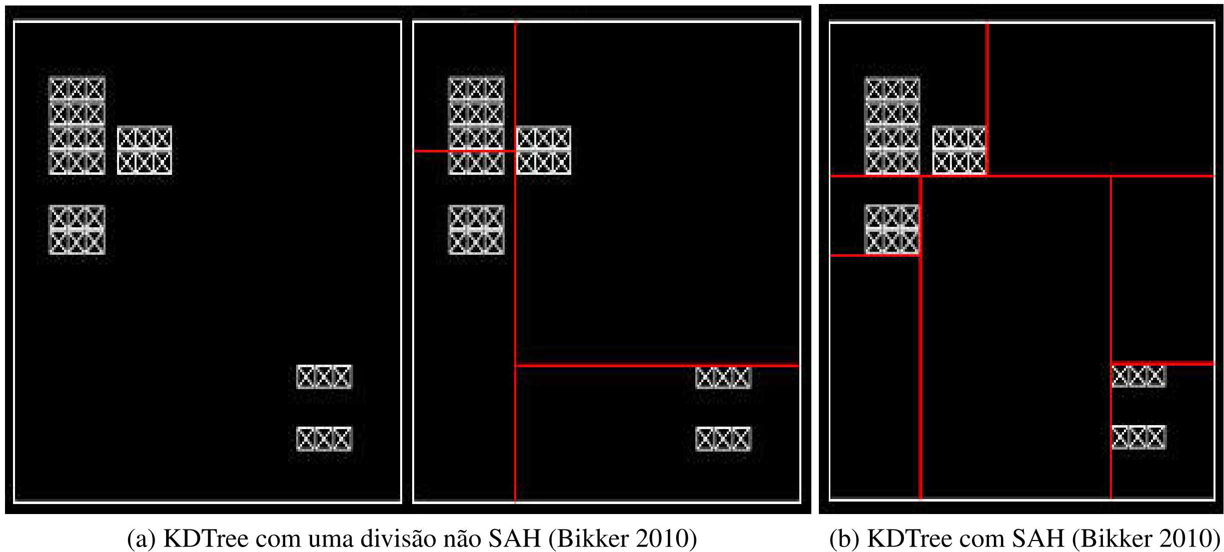
$$K \in R, \quad 0 \leq K \leq 1 \tag{1.5}$$

Onde TC é o custo constante de travessia por qualquer nodo, IC é a constante de custo para o cálculo de uma intersecção. O custo para a fórmula geral 1.1 inclui o custo dos dois possíveis nodos gerados, em1.2 e 1.3. No custo de cada nodo é levado em conta uma constante de bônus, a qual leva em consideração se algum dos dois nodo é vazio na equação 1.4. A constante deve ser entre 0 e 1, conforme está em 1.5, já que ela representa uma porcentagem de bonificação para os casos onde há pelo menos um nodo dentre os dois vazios. Isso faz com que a travessia possa percorrer nodos vazios. O efeito do SAH pode ser observado na figura 1.4. Observa-se que na ilustração 1.4a, um raio que atravesse a cena testaria intersecção com pelo menos uma geometria em cada nodo. Já na 1.4b, um raio que atravesse diagonalmente a cena testaria intersecção com provavelmente nenhuma geometria.

1.3.2 Travessia

Para descobrir com quais objetos os raios provenientes da câmera intersectam existem alguns algoritmos de travessia existem para as estruturas de aceleração.

Nas BVHs, os nodos são atravessados até chegar-se em nodos folhas, nos quais o raio é intersectado com as primitivas contidas naquele nodo. Numa AABBTree, basta checar-se, a partir da folha, recursivamente, com quais nodos filhos o raio colide e continuar a recursão até atingir os nodos folhas e testar-se a colisão do raio com os objetos contidos em si. Esse processo está descrito no algoritmo 3.



(a) KDTree com uma divisão não SAH (Bikker 2010) (b) KDTree com SAH (Bikker 2010)

Figura 1.4: Cenas e KDTree

As KDTrees, apresentam uma recursão mais peculiar, dependente da heurística de travessia adotada. Isso é, o algoritmo de travessia dela sempre checa os nodos filhos seguindo algum tipo de heurística, muitas vezes envolvendo as fórmulas do raio e do plano representado pelo valor de *split.* Os algoritmos 4 e 5 ilustram o TA^{A}_{Rec}. Os treze casos de travessia de um raio por um nodo da KDTree são ilustrados na figura 1.5. Havran (Havran 2000), propôs uma versão heurística para travessia de KDTrees.

Algoritmo TA^{A}_{Rec} para KDTree

O algoritmo 5, descreve a implementação recursiva para o algoritmo de travessia A. Nele, quando um raio entre dentro de um nodo interior da KDTree, o qual tem dois nodos, o algoritmo decide deve-se visitar ambos e em qual ordem. Ele classifica os nodos interiores de acordo com a posição da origem do raio em relação ao plano de *split* como sendo *near* e *far.* Quando o raio atravessa somente o nodo *near,* ele continua a recursão somente nesse nodo. Quando visita-se ambos os nodos, ele salva a informação de recursão do *far,* fazendo a recursão primeiramente no *near* e depois no *far.* Assim, quando nenhum objeto é intersectado dentro do nodo *near,* então o nodo *far* é resgatado e a recursão continua por ele.

1.4 Intersecções com raio

Um raio qualquer pode ser definido como um ponto e uma direção, $R(t) = O + D * t$, onde $|D| = 1$ e $t \geq 0$. Dada a simplicidade do raio, e de suas intersecções com outras geometrias, eles

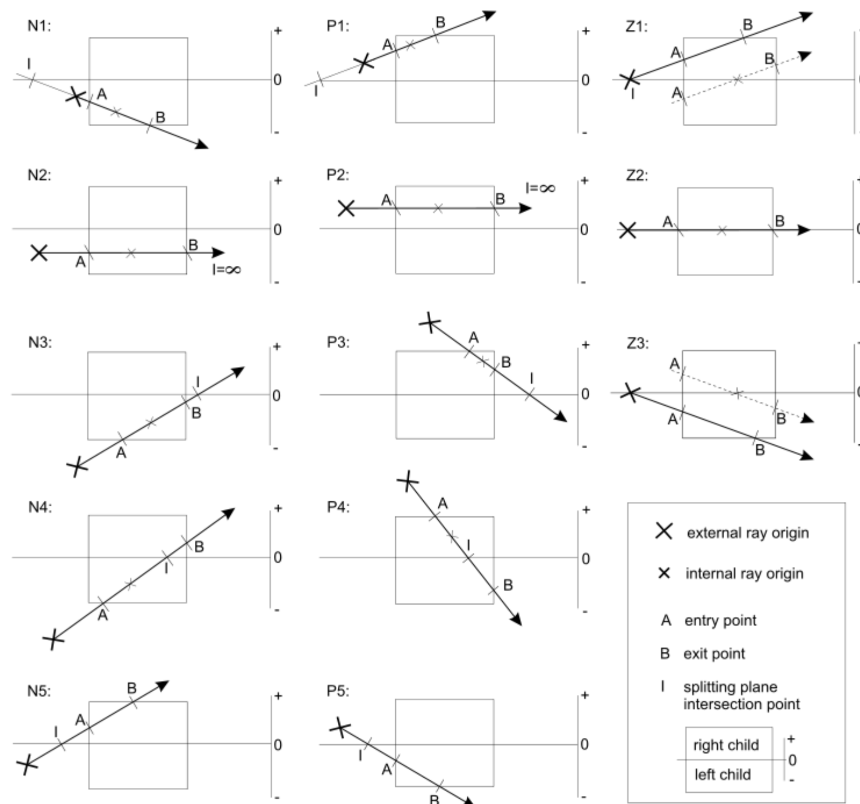


Figura 1.5: Overview de travessia

Algoritmo 3: traverseAABBTree

input : node, ray, hits
output: intersection with geometry or none

if !(*node.box*.`intersects`(*ray*)) **then**
  return;
if *node*.`isLeaf()` **then**
  for $x \in node.geometries$ **do**
    `x.intersect`(*ray, hits*);
else
  `traverseAABBTree`(*node.lowerChild*, *ray*, *hits*);
  `traverseAABBTree`(*node.upperChild*, *ray*, *hits*);

Algoritmo 4: traverseKDTree

input : tree, tnear, tfar, ray, hit
output: hit info
$tnear \leftarrow epsilon$;
$tfar \leftarrow ray.tMax$;
`clipToBox`(*tnear*,*tfar*);
if $tnear > tfar$ **then**
  return;
$hit \leftarrow$ `recTraverseA`(*tree.root, tnear, tfar, ray*);

Algoritmo 5: recTraverseA

```
input  : node, tnear, tfar, ray
output: intersection with geometry or none
if node.isLeaf() then
    closestHit ← MAX;
    for x ∈ node.geometries do
        x.intersect(ray, hit);
        if dist < closestHit then
            closestHit ← dist;
    return closestHit;
else
    d ← (node.split − ray.origin) ∗ ray.rdir;
    if d ≤ tnear then
        return recTraverseA(node.backSon(), tnear, far, hit);
    else
        if d ≥ tfar then
            return recTraverseA(node.backSon(), tnear, tfar, hit);
        else
            thit ← recTraverseA(node.frontSon(), tnear, d, hit);
            if thit ≤ d then
                return thit;
            return recTraverseA(node.backSon(), d, tfar, hit);
```

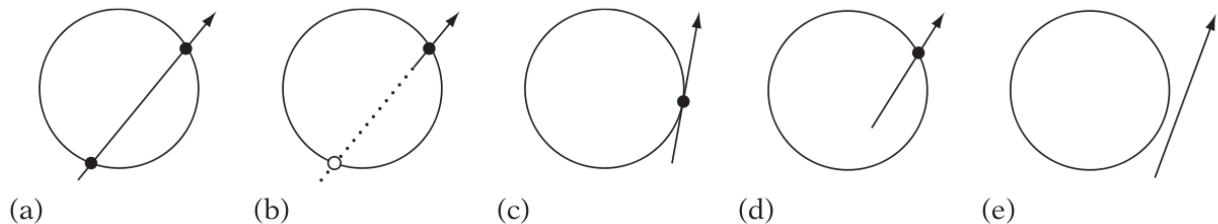


Figura 1.6: Casos de intersecção raio-esfera

são amplamente usados em sistemas de detecção de colisão (Ericson 2004).

1.4.1 Raio-esfera

Ericson define a intersecção raio-esfera como visto no algoritmo 6, baseado nos cinco casos possíveis de intersecção entre um raio e uma esfera, descritos na figura 1.6. Em (a) Raio intersecta a esfera (duas vezes) com $t > 0$; (b) Falsa intersecção com $t < 0$; (c) Raio intersecta a esfera na tangente; (d) Inicia na esfera; (e) Sem intersecção. A equação para um ponto na superfície da esfera é $(X - C) \cdot (X - C) = r^2$, onde C é o centro da esfera e X um ponto na superfície.

Algoritmo 6: Intersecção entre raio e esfera descrita por Ericson (Ericson 2004)

```
// Intersects ray r = p + td, |d| = 1, with sphere s and, if intersecting,
// returns t value of intersection and intersection point q
int IntersectRaySphere(Point p, Vector d, Sphere s, float &t, Point &q) {
    Vector m = p - s.c;
    float b = Dot(m, d);
    float c = Dot(m, m) - s.r * s.r;

    // Exit if r's origin outside s (c > 0)
    //  and r pointing away from s (b > 0)
    if (c > 0.0f && b > 0.0f) return 0;

    float discr = b*b - c;
    // A negative discriminant corresponds to ray missing sphere

    if (discr < 0.0f) return 0;

    // Ray now found to intersect sphere,
    //  compute smallest t value of intersection
    t = -b - Sqrt(discr);

    // If t is negative, ray started inside sphere so clamp t to zero
    if (t < 0.0f) t = 0.0f;
    q = p + t * d;
    return 1;
}
```

1.4.2 Raio-triângulo

Um ponto qualquer em um triângulo pode ser dado por coordenadas baricêntricas. Assim:

$$P = P_1 + a(P_2 - P_1) \tag{1.6}$$

$$P = a_1 P_1 + a_2 P_2 + a_3 P_3 \tag{1.7}$$

$$P - P_1 = a_2(P_2 - P_1) + a_3(P_3 - P_1) \tag{1.8}$$

Na igualdade 1.6, observa-se a definição de um ponto da reta que inicia em P_1 e segue até P_2. Na fórmula 1.7, mostra-se a definição de um ponto em um triângulo como coordenadas baricêntricas. A equação 1.8 pode ser solvida em R^3, porém é mais eficiente solvê-la em R^2, utilizando o plano que contém o triângulo:

$$b = P_3 - P_1 \quad (1.9)$$

$$c = P_2 - P_1 \quad (1.10)$$

$$a_2 = (b_x.h_y – b_y.h_x)/(b_x.c_y – b_y.c_x) \quad (1.11)$$

$$a_3 = (h_x.c_y – h_y.c_x)/(b_x.c_y – b_y.c_x) \quad (1.12)$$

Algoritmo 7: rayTriIntersect

input : ray, tri
output: if the ray hit the triangle
$a \leftarrow ray.origin - tri.vertex[0]$;
$b \leftarrow tri.point[2] - tri.point[0]$;
$c \leftarrow tri.point[1] - tri.point[1]$;
$distance \leftarrow -dot(a, tri.normal)/dot(ray.direction, tri.normal)$;
if $(distance < epsilon) \vee (distance > maxDistance)$ **then**
  return *MISS*;
$axis \leftarrow$ `dominantAxis(`$tri.normal$`)` ;
$u \leftarrow (axis + 1) \mod 3$;
$v \leftarrow (axis + 2) \mod 3$;
$p_u = ray.origin_u + distance * ray.direction_u$;
$p_v = ray.origin_v + distance * ray.direction_v$;
$a_2 = (b_u * P_v - b_v * P_u)/(b_u * c_v - b_v * c_u)$;
if $a_2 < 0$ **then**
  return *MISS*;
$a_3 = (c_v * p_u - c_u * p_v)/(b_u * c_v - b_v * c_u)$;
if $a_3 < 0$ **then**
  return *MISS*;
if $a_2 + a_3 > 1$ **then**
  return *MISS*;
return *HIT*;

Utilizando-se dessas equações podemos definir o algoritmo 7. O qual foi otimizado por Wald (Wald 2004), guardando alguns valores numa *cache* por triângulo. Tal cache guardaria os seguintes valores:

$$n_u = N_u/N_{axis} \quad (1.13)$$
$$n_v = N_v/N_{axis} \quad (1.14)$$
$$n_d = dot(N,P_0) \quad (1.15)$$
$$bn_v = -b_v/(b_u * c_v - b_v * c_u) \quad (1.16)$$
$$bn_u = b_u/(b_u * c_v - b_v * c_u) \quad (1.17)$$
$$cn_u = c_v/(b_u * c_v - b_v * c_u) \quad (1.18)$$
$$cn_v = -c_u/(b_u * c_v - b_v * c_u) \quad (1.19)$$

Onde N é a normal do triângulo e P_n é o vértice n do triângulo. O resultado é o algoritmo 8.

Algoritmo 8: cachedRayTriIntersect

input : ray, tri
output: if the ray hit the triangle
$axis \leftarrow$ `dominantAxis`$(tri.normal)$;
$u \leftarrow (axis+1) \mod 3$;
$v \leftarrow (axis+2) \mod 3$;
$d \leftarrow 1.0/(ray.direction_{axis} + n_u * ray.direction_u + nv * ray.direction_v$;
$distance \leftarrow d * (n_d - ray.origin_{axis} - n_u * ray.origin_{axis} - n_v * ray.origin_{axis})$;
if $!(0 < t < maxDistance)$ **then**
  return *MISS*;
$p_u = ray.origin_u + t * ray.direction_u - a_u$;
$p_v = ray.origin_v + t * ray.direction_v - a_u$;
$a_2 = P_v * bn_u + P_u * bnv$;
if $a_2 < 0$ **then**
  return *MISS*;
$a_3 = Pu * cn_u + P_v * cnv$;
if $a_3 < 0$ **then**
  return *MISS*;
if $(a_2 + a_3) > 1$ **then**
  return *MISS*;
return *HIT*;

1.4.3 Raio-caixa

A colisão entre raio e caixa pelo teste de linha e caixa, o qual demonstrou um maior grau de estabilidade numérica em relação a raio caixa foi utilizada na implementação do raytracer. O algoritmo 9, que implementa o cálculo é proveniente de Ericson (Ericson 2004), adaptado por Tiago Nobrega para integrar-se as bibliotecas C3DE.

Algoritmo 9: Intersecção raio-caixa

```
bool intersects(const LineSegment& disp, const AABB& b) {
    Point3D p0 = disp.origin();
    Point3D p1 = disp.corner();
    Point3D c = b.center(); // Box center-point
    Vector3D e = b.corner() - c; // Box halflength extents
    Point3D m = p0 + ((p1 - p0) * 0.5); // Segment midpoint
    Vector3D d = p1 - m; // Segment halflength vector
    //can't make diff with two points
    Vector3D tmp(c.x(), c.y(), c.z());
    m -= tmp;
    // Try world coordinate axes as separating axes
    Float adx = Mathf::abs(d.x());
    if (Mathf::abs(m.x()) > e.x() + adx) return false;
    Float ady = Mathf::abs(d.y());
    if (Mathf::abs(m.y()) > e.y() + ady) return false;
    Float adz = Mathf::abs(d.z());
    if (Mathf::abs(m.z()) > e.z() + adz) return false;

    //counter arithmetic errors
    adx += RAY_EPSILON; ady += RAY_EPSILON; adz += RAY_EPSILON;

    // Try cross products of segment direction vector with coordinate
        axes
    if (Mathf::abs(m.y() * d.z() - m.z() * d.y()) > e.y() * adz + e.z
       () * ady) return false;
    if (Mathf::abs(m.z() * d.x() - m.x() * d.z()) > e.x() * adz + e.z
       () * adx) return false;
    if (Mathf::abs(m.x() * d.y() - m.y() * d.x()) > e.x() * ady + e.y
       () * adx) return false;

    // No separating axis found; segment must be overlapping AABB
    return true;
}
```

2 Implementação

Utilizando-se do sistema de renderização atual que utiliza as bibliotecas *Cyclops 3D Environment* (C3DE), principalmente as C3DE-Core e C3DE-AL. Mudanças as bibliotecas foram feitas de modo a tentar trazer o desempenho do raytracing para algo próximo de tempo real. Nesse capítulo descreve-se as implementações e otimizações feitas. As bibliotecas C3DE estão organizadas segundo a figura 2.1. As bibliotecas C3DE-Core e C3DE-AL, proveem, respectivamente, as partes básicas para computação gráfica (geometria, estruturas matemáticas) e renderização por meio de OpenGL das estruturas geométricas.

Um diagrama simplificado do protótipo de raytracing está ilustrado na figura 2.2. Para as implementações SSE e OpenCL dos algoritmos de instersecção a parte mais modificada foi a classe RtMath. Contudo, para uma implementação completa, provavelmente a classe CLSceneFactory criará uma associação direta com a classe Raytracer ou um RaytracerManager que intermediará o que rodará em CPU e o que rodará em GPU. Assim, Esse diagrama, com a implementação completa das intersecções em GPU sofrerá várias modificações, principalmente acréscimos a esse modelo atual.

2.1 Implementações com vetorização

Na arquitetura X86 existem as instruções que operam sobre registradores multimídia, os quais até a versão 4 do *Streaming SIMD Extensions* (SSE) são 4 registradores de ponto flutuante (16 bytes) equivalentes a um ponto ou vetor no R^4. Tais implementações utilizam registradores específicos da CPU, os quais dependem de alta utilização para prover o maior desempenho. Dessa forma, *branches* condicionais ou incondicionais no código depredam significativamente o desempenho, tornando proibitivo o uso de SSE com grandes quantidades de condições e recursões.

Tentou-se também implementar a intersecção de esfera com raio com SSE, porém dado a quantidade significativa de condicionais no código original e poucas operações que atuem sobre

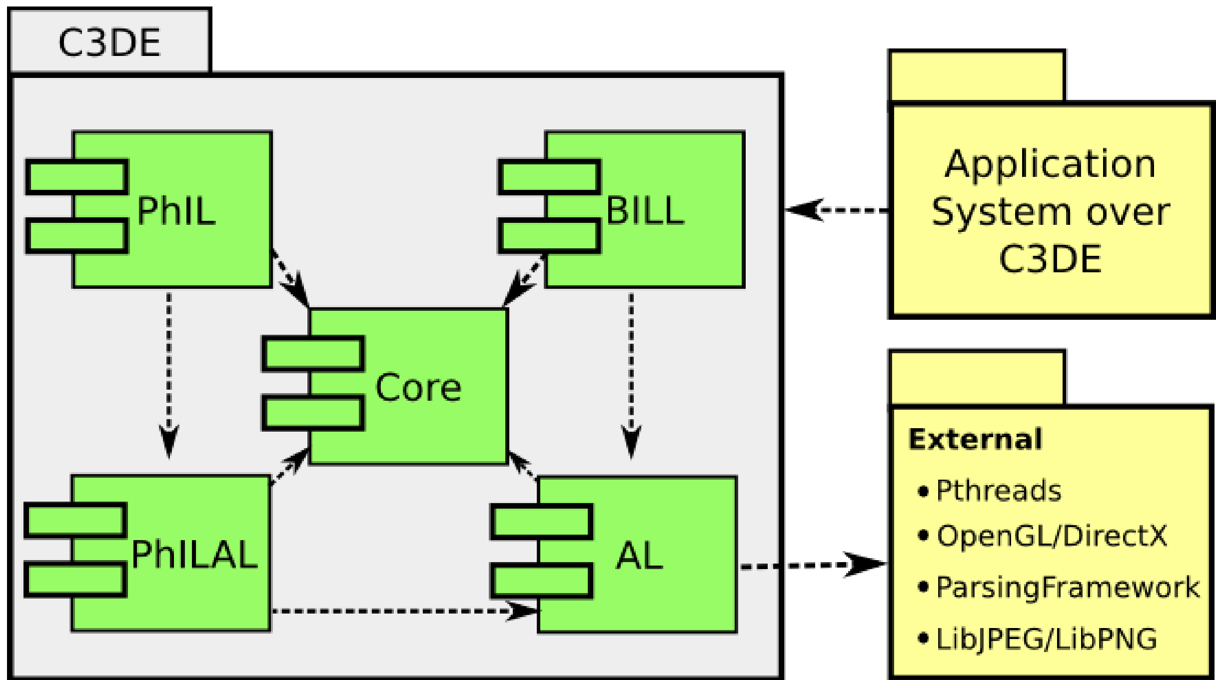


Figura 2.1: Organização das bibliotecas C3DE, em diagrama de componentes.

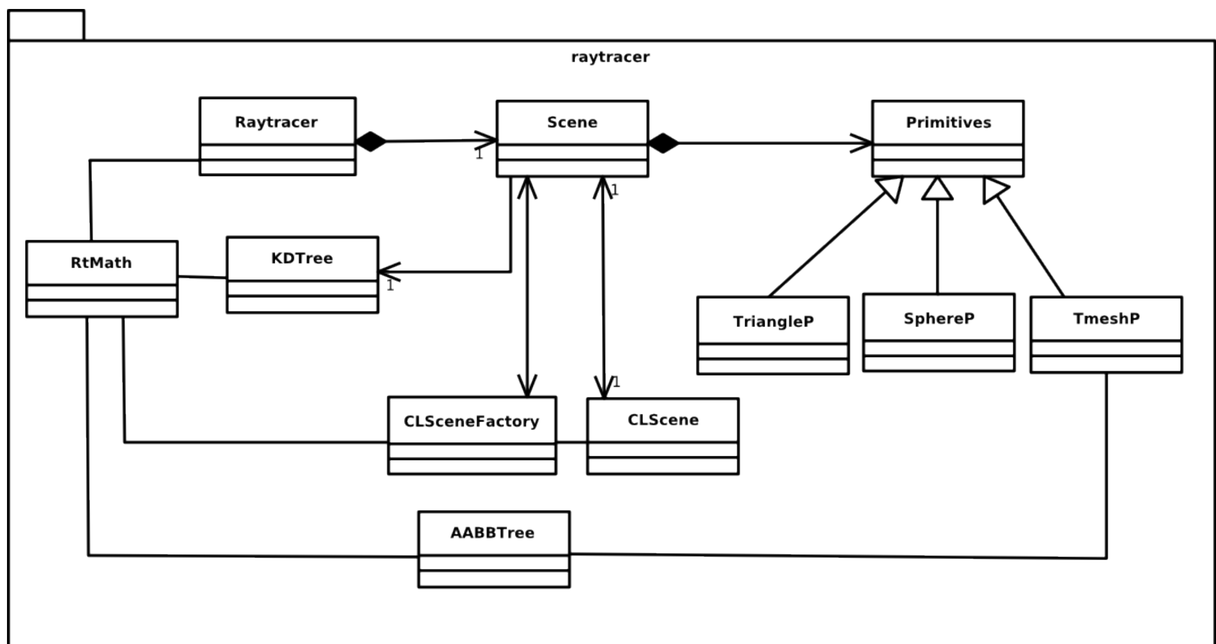


Figura 2.2: Diagrama representativo da situação atual do raytracer.

vetores, o desempenho do código SSE acabou por sem inferior a versão original. A intersecção com caixa e raio foi implementada também em SSE, seguindo a implementação original feita por Ericson (Ericson 2004).

As instruções de vetorização funcionam melhor com memória alinhada ao tamanho dos registradores. Assim, um raytracing que utilize pacotes de raios implementados alinhados a 16 bytes pode-se beneficiar dessas instruções. Os registradores podem ser acessados por bibliotecas padrões C, que implementam algumas instruções que atuam sobre as variáveis de tipo *__m128* (registrador SSE). Entre as mais comuns estão operadores de soma, divisão, multiplicação, produto escalar, as quais atuam sobre vetores de elementos. Vários exemplos de utilização das instruções SSE podem ser observadas nos anexos, principalmente na classe Vec4 implementada para utilizar os registradores SSE com uma interface C++.

2.2 Processamento paralelo

De modo a produzir resultados mais eficientes, é possível efetuar-se todos os cálculos de renderização de um raytracer em paralelo, uma vez que as cores são calculadas por pixel e sem relação direta com os valores vizinhos. Para aproveitar o processamento intrinsecamente paralelo do raytracing, pode-se implementar versões em CPU com múltiplas threads e SSE, assim como versões em GPU dos algoritmos de travessia e intersecção. Para implementar a parte em CPU, utilizou-se o *Threadpool*, um *conceito* da biblioteca Boost, o qual implementa um gerenciador de threads. Os algoritmos em GPU, rodam em OpenCL e são versões similares as originais escritas em C++ (porém adaptadas para uma interface C das estruturas envolvidas).

2.3 Resultados

Os algoritmos citados nas seções anteriores foram implementados e testados em vários casos de estudo. Sobretudo, as diferenças de desempenho em cada diferente versão deles são descritas de modo a mostrar a vantagem da técnica sobre outras. Nos testes demonstrados, foi utilizado a configuração vista na tabela .

Itens	PC 1	PC 2
Processador	Core 2 Duo 2.13 Ghz	Phenom X4 2.3 Ghz
GPU	Geforce 250 GTS	Geforce 9600 GT
Memória	4GB DDR2 800mhz	4 GB DDR2 800mhz

Ilustrado na figura 2.3, está o desempenho para realizar-se a travessia sem BVH (KDTree) e com BVH. O número de primitivas (objetos) variam nesses testes, demonstrado no eixo das abcissas. Observa-se uma tendência quadrática no algoritmo *naive*, enquanto com a KDTree (BVH) os testes são bem mais lineares.

Na figura 2.4, demonstra-se os tempos de execução para cenas com números diferentes de primitivas. Observa-se que o melhor desempenho geral para o PC 1, nessa cena, são 3 threads, enquanto para o PC 2 são quatro. Observando-se que o PC 1 tem um processador de dois núcleos e que o PC 2 um de quatro essas constatações fazem sentido. Também, observa-se uma certa linearidade no ganho de desempenho, o qual torna-se mais linear para cenas mais complexas.

A parte em OpenCL, vista na figura 2.5, e comparada com as duas implementações em CPU, foi aproximadamente 3000% mais rápido que as implementações em CPU para a configuração do PC 1. No gráfico a escala é logarítmica (base 10). Note-se que esse tipo de comparação depende da GPU e CPU associadas, assim como número de threads no qual o código roda. No caso do gráfico, os códigos em CPU consideram uma implementação *single core*, enquanto em GPU utiliza todos os 128 *CUDA cores* da placa 250 GTS. O código SSE para triângulo-raio manual foi 32% mais rápido em relação ao código original otimizado pelo GCC (com vetorização também). Já o caixa-raio, foi 50% mais rápido. Claramente, para esses algoritmos, as implementações em SSE manuais foram significativamente mais rápidas, mas quando comparadas com a implementação paralelizada em uma GPU mediana, seus ganhos parecem menos significativos.

2.4 Discussão conclusiva

Dado os resultados obtidos por meio das implementações e o objetivo final de construir-se um raytracer processado em GPUs e CPUs, observa-se que esse estudo dirigido contribuiu, principalmente para o amadurecimento da parte que diz respeito a lapidação da implementação em CPU. Tal implementação é vital, uma vez que serve como base e comparação para futuras

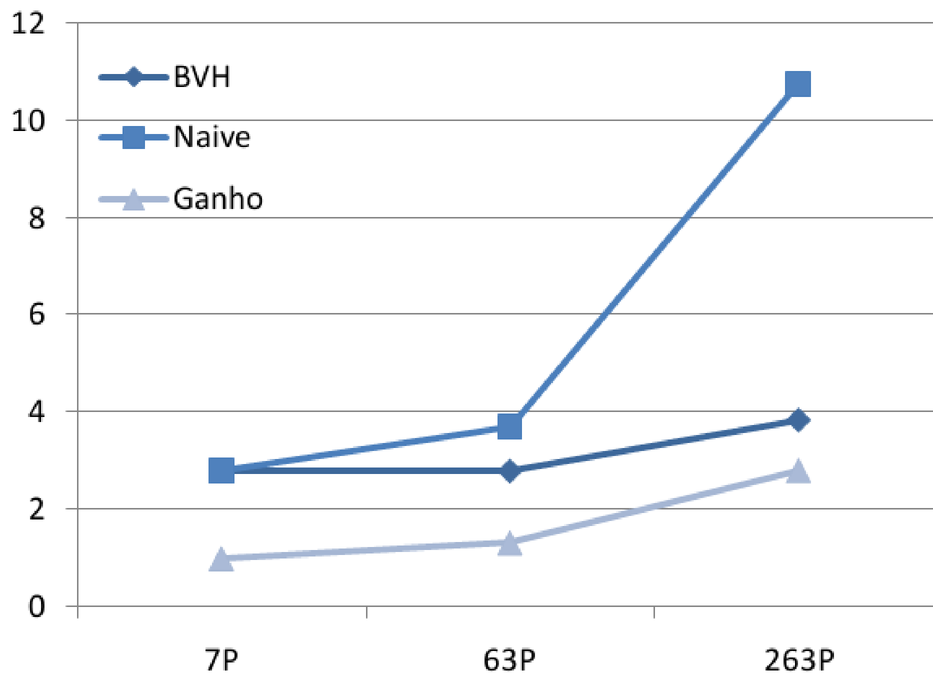
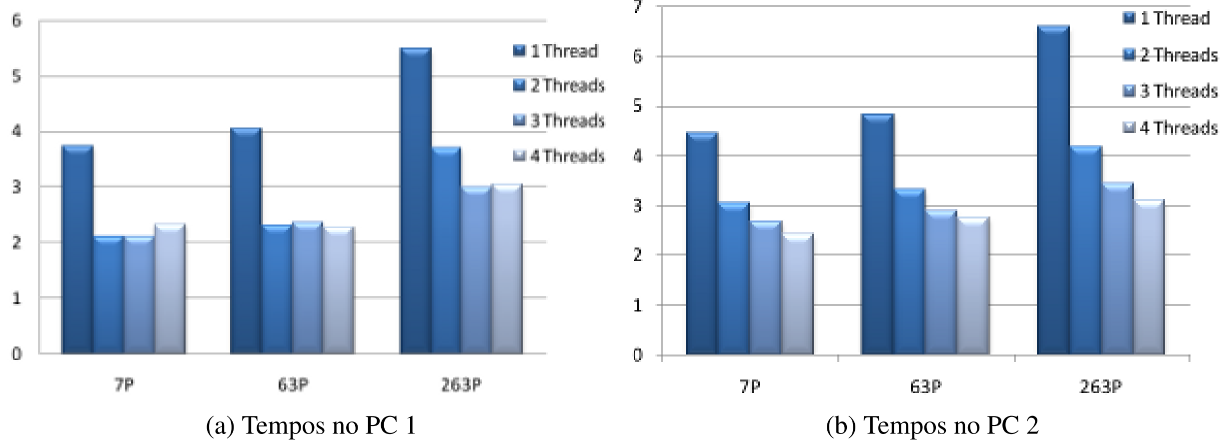


Figura 2.3: Comparação entre os algoritmos *naive* e BVH para travessia, em segundos. O ganho de desempenho da BVH em relação a primeira é mostrado pela terceira função.



(a) Tempos no PC 1

(b) Tempos no PC 2

Figura 2.4: Tempos, em segundos, de execução para diferentes quantidades de Threads do raytracer implementado.

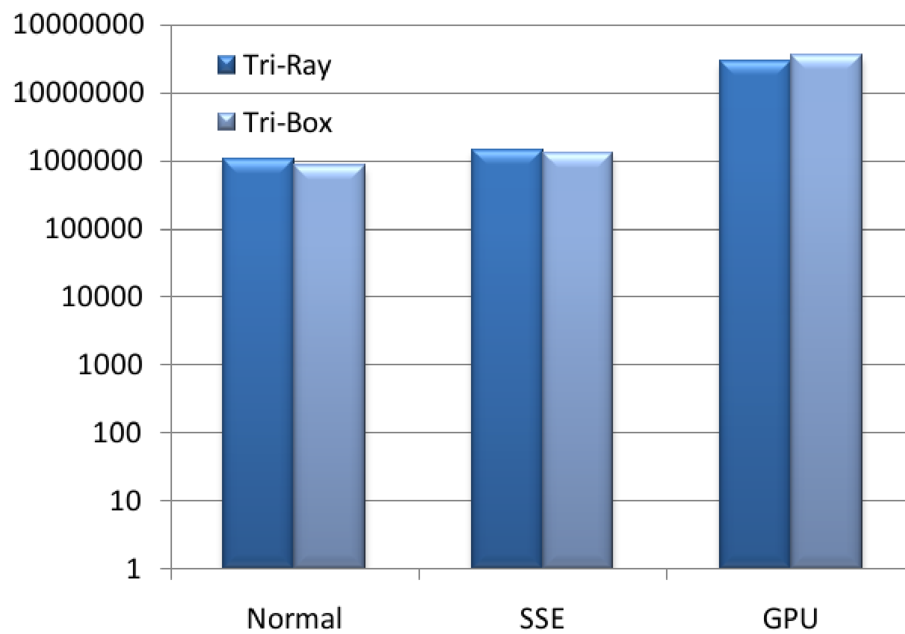


Figura 2.5: Comparação de execuções por segundo das versões, normais, SSE e GPU dos referidos algoritmos de intersecção. O teste foi realizado no PC 1.

implementações dos mesmos algoritmos em GPU. As técnicas empregadas ainda não possibilitaram um raytracing em tempo real, o qual deve ser possível com a implementação completa das estruturas de aceleração em GPU e com a otimização dos algoritmos em CPU com instruções de vetorização, conforme foi demonstrado. Com a utilização tanto de GPU como CPU, fica claro que será possível reduzir o custo computacional do raytracing para a CPU, e possivelmente trazer técnicas consideradas como não interativas para um grau de interatividade de mais de 10 quadros por segundo. Apesar disso, programar-se para o paradigma de GPU ainda requer um grande esforço e mesmo códigos mais simples como os das intersecções são difíceis de depurar.

Referências Bibliográficas

Bikker 2010 BIKKER, Jacco. *Arauna Real-time Raytracing*. 2010. Disponível em: <http://igad.nhtv.nl/ bikker>.

Ericson 2004 ERICSON, Christer. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN 1558607323.

Günther et al. 2007 GÜNTHER, Johannes; POPOV, Stefan; SEIDEL, Hans-Peter; SLUSALLEK, Philipp. Realtime ray tracing on GPU with BVH-based packet traversal. In: *Proceedings of the IEEE/Eurographics Symposium on Interactive Ray Tracing 2007*. [S.l.: s.n.], 2007. p. 113–118.

Havran 2000 HAVRAN, Vlastimil. *Heuristic Ray Shooting Algorithms*. Tese (Ph.D. Thesis) — Czech Technical University in Prague, November 2000. Disponível em: <http://www.cgg.cvut.cz/ havran/phdthesis.html>.

Lauterbach et al. 2009 LAUTERBACH, C.; GARLAND, M.; SENGUPTA, S.; LUEBKE, D.; MANOCHA, D. Fast bvh construction on gpus. *Computer Graphics Forum*, Blackwell Publishing, v. 28, n. 2, p. 375–384, 2009. ISSN 0167-7055. Disponível em: <http://dx.doi.org/10.1111/j.1467-8659.2009.01377.x>.

Rost 2006 ROST, Randi J. *OpenGL(R) Shading Language (2nd Edition)*. [S.l.]: Addison-Wesley Professional, 2006. Paperback. ISBN 0321334892.

Silva et al. 2009 *A multi-layered development framework for medical imaging applications*. 1–4 p. Disponível em: <http://dx.doi.org/10.1109/CBMS.2009.5255379>.

Stone 2009 STONE, John. *An Introduction to OpenCL*. Dec 2009.

Wald 2004 WALD, Ingo. *Realtime Ray Tracing and Interactive Global Illumination*. Tese (Doutorado) — Computer Graphics Group, Saarland University, 2004.

3 Anexos

Nesse capítulo estão alguns códigos básicos utilizados na implementação.

3.1 Implementação de Vec4 e Float4

```
struct Float4 {
    static const unsigned int SIZE = 4;

    union {
        struct {
            float x, y, z, w;
        };

        float cell[SIZE];
    };

    inline Float4(const float xx, const float yy, const float zz, const float ww) : x(xx), y(
        yy), z(zz), w(ww) {}
    inline Float4() : x(0), y(0), z(0), w(0) {}

    std::string toString() const {
        std::stringstream sstream;
        sstream << "[" << x << " " << y << " " << z << " " << w << "]";

        return sstream.str();
    }

} _MM_ALIGN16;

// some defines first
union ieee754_QNAN {
   const float f;
   struct {
      const unsigned int mantissa:23, exp:8, sign:1;
   };

   ieee754_QNAN() : f(0.0), mantissa(0x7FFFFF), exp(0xFF), sign(0x0) {}
};

_MM_ALIGN16 const ieee754_QNAN absMask;
static const __m128 abs4Mask = _mm_load1_ps(&absMask.f);
```

```
struct Vec4 {
    __m128 xmm;

    Vec4() {
        xmm = _mm_setzero_ps();
    }

    Vec4(__m128 v) : xmm(v) {}

    Vec4(float v) {
        xmm = _mm_set1_ps(v);
    }

    Vec4(float x, float y, float z, float w) {
        xmm = _mm_set_ps(w, z, y, x);
    }

    Vec4(const float *v) {
        xmm = _mm_load_ps(v);
    }

    Vec4(const c3deCore::math::Vector3D& v) {
        xmm = _mm_set_ps(0, v.z(), v.y(), v.x());
    }

    Vec4(const c3deCore::math::Point3D& v) {
        xmm = _mm_set_ps(0, v.z(), v.y(), v.x());
    }

    //!a, b, c and d are indexes from this vector to compose the shuffled vec
    template<int a, int b, int c, int d>
    Vec4 shuffle() const {
        return Vec4(_mm_shuffle_ps(xmm, xmm, _MM_SHUFFLE(d,c,b,a)));
    }

    //!a and b are indexes of the v vec while d and c are indexes of this vector
    template<int a, int b, int c, int d>
    Vec4 shuffle(const Vec4 &v) const {
        return Vec4(_mm_shuffle_ps(xmm, v.xmm, _MM_SHUFFLE(d,c,b,a)));
    }

    // calculate absolute value
    Vec4 abs() {
        return _mm_and_ps(abs4Mask, xmm);
    }

#ifdef __SSE3__
    Vec4 hadd(const Vec4 &v) const {
        return Vec4(_mm_hadd_ps(xmm, v.xmm));
    }
#endif

#ifdef __SSE3__
```

```
    Vec4 hsub(const Vec4 &v) const {
        return Vec4(_mm_hsub_ps(xmm, v.xmm));
    }
#endif

    float sum() const {
        float c;
#ifdef __SSE3__
        Vec4 s = hadd(*this).hadd(*this);
        _mm_store_ss(&c, s.xmm);
#else
        float v[4] __attribute__((aligned(16)));
        (*this) >> v;
        c = v[0] + v[1] + v[2] + v[3];
#endif
        return c;
    }

    float dot(const Vec4 &v) const {
        return (*this * v).sum();
    }

    float sqLength() const {
        return dot(Vec4(*this));
    }

    float length() const {
        return std::sqrt(sqLength());
    }

    void selfNormalize() {
        *this *= 1.0 / length();
    }

    Vec4 normalize() {
        Vec4 r = *this;
        r.selfNormalize();

        return r;
    }

    Vec4 cross(const Vec4& other) {
        Vec4 v[5];
        v[0] = *this;
        v[1] = other;

        v[2] = _mm_shuffle_ps(v[1].xmm, v[0].xmm, _MM_SHUFFLE(3, 0, 2, 2));
        v[3] = _mm_shuffle_ps(v[0].xmm, v[1].xmm, _MM_SHUFFLE(3, 1, 0, 1));

        v[4] = v[2] * v[3];

        v[2] = _mm_shuffle_ps(v[0].xmm, v[1].xmm, _MM_SHUFFLE(3, 0, 2, 2));
        v[3] = _mm_shuffle_ps(v[1].xmm, v[0].xmm, _MM_SHUFFLE(3, 1, 0, 1));
```

```
    v[2] = v[2] * v[3];
    v[2] = v[4] - v[2];

    Float4 out;
    v[2] >> out.cell;
    out.cell[1] *= -1;

    return Vec4(out.cell);
}

Vec4 operator*(const Vec4 &v) const {
    return Vec4(_mm_mul_ps(xmm, v.xmm));
}

Vec4 operator+(const Vec4 &v) const {
    return Vec4(_mm_add_ps(xmm, v.xmm));
}

Vec4 operator-(const Vec4 &v) const {
    return Vec4(_mm_sub_ps(xmm, v.xmm));
}

Vec4 operator/(const Vec4 &v) const {
    return Vec4(_mm_div_ps(xmm, v.xmm));
}

void operator*=(const Vec4 &v) {
    xmm = _mm_mul_ps(xmm, v.xmm);
}

void operator+=(const Vec4 &v) {
    xmm = _mm_add_ps(xmm, v.xmm);
}

void operator-=(const Vec4 &v) {
    xmm = _mm_sub_ps(xmm, v.xmm);
}

void operator/=(const Vec4 &v) {
    xmm = _mm_div_ps(xmm, v.xmm);
}

void operator>>(float *v) const {
    _mm_store_ps(v, xmm);
}

//logic functions
Vec4 operator>(const Vec4& v) {
    return _mm_cmpgt_ps(xmm, v.xmm);
}

Vec4 operator<(const Vec4& v) {
    return _mm_cmplt_ps(xmm, v.xmm);
}
```

```
    int getMovemask() {
        return _mm_movemask_ps(xmm);
    }

    std::string toString() const {
        Float4 r; *this >> r.cell;
        return r.toString();
    }

    static Vec4 toVec4(const c3deCore::math::Vector3D& v) {
        return Vec4(v.x(), v.y(), v.z(), 0.0);
    }

    static Vec4 toVec4(const c3deCore::math::Point3D& v) {
        return Vec4(v.x(), v.y(), v.z(), 0.0);
    }
} _MM_ALIGN16;
```

3.2 Intersecções SSE

Códigos das intersecções reescritas para utilizar registradores SSE.

3.2.1 Raio-caixa

```
bool intersects(const LineSegment& disp, const AABB& b) {
    Vec4 p0(Vec4::toVec4(disp.origin())), p1(Vec4::toVec4(disp.corner()));
    Vec4 c = (Vec4(Vec4::toVec4(b.origin())) + Vec4(Vec4::toVec4(b.corner()))) * 0.5;
    Vec4 e = Vec4(Vec4::toVec4(b.corner())) - c;
    Vec4 m = p0 + ((p1 - p0) * 0.5);
    Vec4 d = p1 - m;

    m -= c;

    Vec4 ad = d.abs();

    //ad = Vec4(1, 2, 3, 4);
    Vec4 am = m.abs();

    if ((am > (ad + e)).getMovemask()) return false;

    // Add in an RAY_EPSILON term to counteract arithmetic errors when segment is
    // (near) parallel to a coordinate axis (see text for detail)
    ad += Vec4(RAY_EPSILON, RAY_EPSILON, RAY_EPSILON, 0);

    Vec4 lOps = m * d.shuffle<1, 2, 0, 3>() -
            m.shuffle<1, 2, 0, 3>() * d;

    Vec4 rOps = e.shuffle<0, 1, 0, 3>() * ad.shuffle<1, 2, 2, 3>() +
            e.shuffle<1, 2, 2, 3>() * ad.shuffle<0, 1, 0, 3>();
```

```
    if ((lOps.abs() > rOps).getMovemask()) {
        return false;
    }

    // No separating axis found; segment must be overlapping AABB
    return true;
}
```

3.2.2 Raio-triângulo

```
int intersect(Math::RayIntersectionInfo& info, const TriAccel& triAccel, const Triangle&
    triangle, const Ray& ray) {

    Vec4 a(triangle.getPoint(0));
    Vec4 n(triangle.getNormal());

    real d = -(Vec4(ray.getOrigin()) - a).dot(n) / Vec4(ray.getDirection()).dot(n);

    if (d < 0.0 || info.distance < d) return Math::MISS;

    unsigned int axis = triAccel.k;
    unsigned int uaxis = Math::AXIS3_MODULE[axis + 1];
    unsigned int vaxis = Math::AXIS3_MODULE[axis + 2];

    Vec4 pVec(ray.getOrigin()(uaxis), ray.getOrigin()(vaxis), ray.getOrigin()(uaxis), ray.
        getOrigin()(vaxis));

    pVec += Vec4(ray.getDirection()(uaxis), ray.getDirection()(vaxis), ray.getDirection()(
        uaxis), ray.getDirection()(vaxis)) * d;
    pVec -= Vec4(triangle.getPoint(0)(uaxis), triangle.getPoint(0)(vaxis), triangle.getPoint
        (0)(uaxis), triangle.getPoint(0)(vaxis));

    //multiplies by 0.5 because it will be summed twice later on
    Vec4 bg = pVec * Vec4(triAccel.ac[1], triAccel.ac[0], triAccel.ab[0], triAccel.ab[1]) *
        0.5;

    real beta = bg.shuffle<0, 1, 0, 1>().sum();

    real gamma = bg.shuffle<2, 3, 2, 3>().sum();

    Vec4 v0(beta, gamma, 0.5 + RAY_EPSILON, 0), v1(0, 0, bg.sum(), 0);
    if ((v0 < v1).getMovemask()) return Math::MISS;

    info.distance = d;
    info.u = gamma;
    info.v = beta;
    info.normal = triangle.getNormal();

    //return ray.getDirect().dotProduct(triangle.getNormal()) > 0) ? INPRIM : HIT;
    return Math::HIT;
}
```

3.3 Intersecções OpenCL

Códigos das intersecções reescritas para rodarem em GPU.

3.3.1 Raio-caixa

```
int intersectsRayBox(
        __global float4* rOrig,
        __global float4* rDir,
        __global float4* bOrig,
        __global float4* bCorner) {

    //line
    float4 p0 = *rOrig;
    float4 p1 = (*rDir) * BIG_FLOAT + (*rOrig);
    //box center
    float4 c = ((*bOrig) + (*bCorner)) * 0.5;
    float4 e = (*bCorner) - c;
    //line middle point
    float4 m = p0 + ((p1 - p0) * 0.5);
    float4 d = p1 - m;

    m -= c;
    float4 ad = fabs(d);

    float4 am = fabs(m);

    if (am.x > e.x + ad.x) return MISS;
    if (am.y > e.y + ad.y) return MISS;
    if (am.z > e.z + ad.z) return MISS;

    // Add in an 1.e-6 term to counteract arithmetic errors when segment is
    // (near) parallel to a coordinate axis (see text for detail)
    ad += (float4)(RAY_EPSILON, RAY_EPSILON, RAY_EPSILON, 0);

    // Try cross products of segment direction vector with coordinate axes
    if (fabs(m.y * d.z - m.z * d.y) > e.y * ad.z + e.z * ad.y) return MISS;
    if (fabs(m.z * d.x - m.x * d.z) > e.x * ad.z + e.z * ad.x) return MISS;
    if (fabs(m.x * d.y - m.y * d.x) > e.x * ad.y + e.y * ad.x) return MISS;

    return HIT;
}
```

3.3.2 Raio-triângulo

```
int intersectRayTri(
        float4* result,
        __global float4* rOrig,
        __global float4* rDir,
        __global uint* face,
        __global float4* vertPos,
```

```
    __global float4* vertNormal) {

//can't initialize as array
float4 lvp[3]; lvp[0] = vertPos[face[0]]; lvp[1] = vertPos[face[1]]; lvp[2] = vertPos[face
    [2]];

//can't initialize as array
float4 lvn[3]; lvn[0] = vertNormal[face[0]]; lvn[1] = vertNormal[face[1]]; lvn[2] =
    vertNormal[face[2]];
float4 ab = lvp[1] - lvp[0];//lv[1].position - lv[0].position;
float4 ac = lvp[2] - lvp[0];
float4 normal = lvn[0] + lvn[1] + lvn[2];
normal = normalize(normal);

//real d = -(ray.getOrigin() - triangle.getPoint(0)).dotProduct(normal) / ray.getDirection
    ().dotProduct(normal);
float d = -dot((*rOrig) - lvp[0], normal) / dot(*rDir, normal);

if (d < 0.0 || result[0].x < d) return MISS;

result[1] = normal;

int axis = biggestAxis(&normal);
int uaxis = (axis + 1) % 3;
int vaxis = (axis + 2) % 3;

float aca[VEC3_SIZE] = S_ARRAY3_FLOAT4(ac);
float aba[VEC3_SIZE] = S_ARRAY3_FLOAT4(ab);
float lrDir[VEC3_SIZE] = S_ARRAY3_FLOAT4(*rDir);
float lrOrig[VEC3_SIZE] = S_ARRAY3_FLOAT4(*rOrig);
float a[3] = S_ARRAY3_FLOAT4(lvp[0]);


float pu = lrOrig[uaxis] + d * lrDir[uaxis] - a[uaxis];
float pv = lrOrig[vaxis] + d * lrDir[vaxis] - a[vaxis];

float inv, acnu, acnv, abnu, abnv;
inv = 1.0 / (aca[uaxis] * aba[vaxis] - aca[vaxis] * aba[uaxis]);


// first line equation
acnu = aca[uaxis] * inv;
acnv = -aca[vaxis] * inv;

// second line equation
abnu = aba[vaxis] * inv;
abnv = -aba[uaxis] * inv;

float beta = pv * acnu + pu * acnv;
if (beta < 0.0) return MISS;

float gamma = pu * abnu + pv * abnv;
if (gamma < 0.0 || (gamma + beta) > 1.0) return MISS;
```

```
    //t, u, v, id
    result[0] = (float4)(d, gamma, beta, face[0]);

    return HIT;
}
```