Insert here your thesis’ task.
Bachelor’s thesis

Raytracing on GPU with CUDA

Jan Dundr

Supervisor: Ing., Mgr. Zdeněk Konfršt, PhD.

12th May 2015
Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on 12th May 2015
Czech Technical University in Prague
Faculty of Information Technology
© 2015 Jan Dundr. All rights reserved.

This thesis is school work as defined by Copyright Act of the Czech Republic. It has been submitted at Czech Technical University in Prague, Faculty of Information Technology. The thesis is protected by the Copyright Act and its usage without author’s permission is prohibited (with exceptions defined by the Copyright Act).

Citation of this thesis

Abstrakt

Zkoumané implementace metod vykreslování scény pomocí sledování paprsků běží na paralelní architektuře GPU. Zanalyzoval jsem možné akcelerační struktury a otestoval implementaci různými scénami. Porovnal jsem CPU a GPU implementace a přehodnotil výběr programu MATLAB.

Klíčová slova

GPU, CUDA, sledování paprsku, počítačová grafika, paralelizace, MATLAB

Abstract

Examined raytracing algorithms run on highly parallel GPU architecture. I implemented one acceleration structure after analyzing different options. I experimented on a few scenes with different complexity and compared CPU and GPU implementations. I reevaluated the choice of MATLAB.

Keywords

GPU, CUDA, raytracing, computer graphics, parallelization, MATLAB
Contents

Citation of this thesis ........................................... vi

1 Introduction .................................................... 3
  1.1 The goal .................................................. 3
  1.2 Image rendering ......................................... 3
    1.2.1 Raytracing .......................................... 4
  1.3 Tools .................................................... 6
    1.3.1 GPU programming ................................... 6
    1.3.2 MATLAB versus C++ ............................... 6
    1.3.3 MATLAB language .................................. 6
    1.3.4 MATLAB code performance ....................... 8

2 Rendering acceleration methods ............................. 11
  2.1 Ray and triangle intersection ......................... 11
    2.1.1 Analytic approach ................................ 11
    2.1.2 Triple scalar product approach .................. 12
  2.2 Scene hierarchy structures ............................. 16
    2.2.1 Uniform grid .................................... 17
    2.2.2 Octree (quadtree in 2D) .......................... 20
    2.2.3 k-d tree .......................................... 24
    2.2.4 Bounding Volume Hierarchy (BVH) ............... 25
    2.2.5 Heuristics for k-d tree and BVH splitting .... 27
      2.2.5.1 Surface Area Heuristic (SAH) .......... 27

3 Implementation ................................................ 31
  3.1 Basic raytracing algorithm ............................. 31
    3.1.1 Per-pixel triangle iteration ..................... 31
      3.1.1.1 Primary ray shooting ...................... 32
      3.1.1.2 Ray and triangle intersection .......... 34
      3.1.1.3 Surface interaction .................... 35
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Diagram of the described Whitted raytracing algorithm</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Logarithmic plot showing earth-shatteringly better performance of broken down version</td>
<td>10</td>
</tr>
<tr>
<td>2.1</td>
<td>Diagram of triple scalar product for bottom side bc of triangle abc and ray pq</td>
<td>14</td>
</tr>
<tr>
<td>2.2</td>
<td>Floating point operations in basic and optimized scalar triple product ray and triangle intersection algorithms</td>
<td>16</td>
</tr>
<tr>
<td>2.3</td>
<td>Diagram of uniform grid acceleration structure usage</td>
<td>18</td>
</tr>
<tr>
<td>2.4</td>
<td>Diagram of quadtree acceleration structure usage</td>
<td>21</td>
</tr>
<tr>
<td>2.5</td>
<td>Diagram of k-d tree acceleration structure usage</td>
<td>24</td>
</tr>
<tr>
<td>2.6</td>
<td>Diagram of BVH acceleration structure usage</td>
<td>26</td>
</tr>
<tr>
<td>3.1</td>
<td>Image illustrating camera vectors relationships</td>
<td>33</td>
</tr>
<tr>
<td>3.2</td>
<td>Ray shooting a bounding box intersecting three coordinate planes in a sequence</td>
<td>39</td>
</tr>
<tr>
<td>3.3</td>
<td>Number of triangles and rendering time relationship for naive approach and BVH</td>
<td>41</td>
</tr>
<tr>
<td>3.4</td>
<td>Antialiasing sample pattern</td>
<td>44</td>
</tr>
<tr>
<td>3.5</td>
<td>Antialiasing</td>
<td>45</td>
</tr>
<tr>
<td>3.6</td>
<td>Dragon rendered with a simple shadow algorithm</td>
<td>45</td>
</tr>
<tr>
<td>3.7</td>
<td>Shadow jitter algorithm</td>
<td>46</td>
</tr>
<tr>
<td>3.8</td>
<td>Left points are completely random, right set was formed using Poisson sampling</td>
<td>47</td>
</tr>
<tr>
<td>3.9</td>
<td>Dragon’s smooth shadow raytraced using Poisson sampling</td>
<td>47</td>
</tr>
<tr>
<td>3.10</td>
<td>Dragon’s smooth shadow raytraced using Poisson sampling with random rotations</td>
<td>48</td>
</tr>
<tr>
<td>3.11</td>
<td>Ambient occlusion principle</td>
<td>49</td>
</tr>
<tr>
<td>3.12</td>
<td>Raytraced dragon’s ambient occlusion only</td>
<td>50</td>
</tr>
<tr>
<td>3.13</td>
<td>Raytraced dragon with smooth shadows and ambient occlusion</td>
<td>51</td>
</tr>
</tbody>
</table>
## List of Algorithms

<table>
<thead>
<tr>
<th></th>
<th>Algorithm Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>MATLAB code for dot and cross product with MATLAB functions</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>MATLAB code for dot and cross product broken down to simple math operations</td>
<td>9</td>
</tr>
<tr>
<td>3</td>
<td>MATLAB code of the basic ray–triangle intersection algorithm</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>MATLAB code for basic triangle–ray intersection using triple scalar product</td>
<td>15</td>
</tr>
<tr>
<td>5</td>
<td>Pseudocode of uniform grid acceleration structure construction</td>
<td>19</td>
</tr>
<tr>
<td>6</td>
<td>Pseudocode of uniform grid acceleration structure traversal</td>
<td>19</td>
</tr>
<tr>
<td>7</td>
<td>Pseudocode of quadtree acceleration structure construction</td>
<td>22</td>
</tr>
<tr>
<td>8</td>
<td>Pseudocode of quadtree acceleration structure traversal</td>
<td>22</td>
</tr>
<tr>
<td>9</td>
<td>SAH pseudocode for computing one node’s split action</td>
<td>29</td>
</tr>
</tbody>
</table>
1.1 The goal

Analyzing possible acceleration structure implementations and writing a simple raytracer in MATLAB using GPU acceleration are the goals of this thesis.

1.2 Image rendering

There are currently two main 3d image synthesis algorithms:

- **Realtime rasterization**
  
  This is a very fast way of image rendering. It typically renders a new frame 30 or more times a second, so each frame takes about 33 ms or less to render. GPU is a special hardware with fast rasterization implementation using parallel architecture (hundreds of tasks done at once). The calculations need to be very fast and simple. There is no actual light simulation, it is just a fast approximation. Artist usually manually sets most of the scene parameters to achieve a pleasing look resembling reality.

- **Raytracing**
  
  On the other hand, raytracing simulates real light. It does it by shooting rays colliding with scene. This system approximates behavior of real photons. Raytracing is used in applications which do not require rapid realtime image rendering. The image looks much more realistic and believable this way and optical phenomena occurring as a byproduct in real life are generated without specific artist’s intervention. The only thing needed is a robust lighting model.
However, that does not mean the results of realtime rasterization always look artificial or fabricated. Modern rendering engines together with skilled artists can produce really stunning visuals. But they will never look as good as raytraced images and setting up them will take much longer time for an artist. Carefully fabricated visual effect of rasterization come to life more or less as a byproduct when using raytracing. It is a compromise between rendering speed and artist’s work efficiency.

The obvious downside to raytracing is that it requires large amount of computations. Rendering one image even with decent optimization on CPU can take hours, days or even more. Raytracing uses huge amounts of rays and a large amount of quite simple collision stuff needs to be computed. That sounds like a perfect task for GPU thanks to its highly parallel architecture.

GPU stands for Graphic Processor Unit. It is a special chip with highly parallel architecture typically on a separate board connected to fast PCI port or on the same chip as CPU. CPU is comprised of a few complex and fast cores (2 to 8 in home PCs and tens in servers), but GPU has a marginally larger amount of simpler and slower cores (hundreds or thousands, depends on hardware class and manufacturer).

Unsurprisingly, GPU’s sole purpose was realtime 3D rasterization acceleration. Functionality of small coprocessors was implemented in hardware and they were not programmable. All they were able to do was simple graphics algorithms needed in large amounts.

The programmability of GPU’s was added around 2000 with DirectX8 in form of vertex and pixel shaders. Shaders can be used to manipulate a large amounts of vertex or pixel data. They were still very functionally limited, but that gradually changed over time. High Level Shading Language (HLSL) introduces C–like syntax instead of assembly language, programs could be much longer and complex and use more and more registers or textures. Many types of shaders emerged.

Coprocessors got more complex and much more capable. Compute shaders and entirely non–graphic APIs CUDA and OpenCl (more on these later) emerged. Today, researches use GPU’s highly parallel architecture for a wide varienty of simulations or even machine learning. A lot of these problems need a large number of small calculations. I will revert back to using GPU for graphical purposes, but in a different way than is usual.

1.2.1 Raytracing

Raytracing simulates behavior of light using rays. The basic algorithm used today is by Turner Whitted[1]. It backtraces rays of light that hit pixel on screen to scene. If I simulated rays emitted from light source hitting the scene, there would be a very small number of them hitting camera/computer screen
1.2. Image rendering

The primary (view) ray is shot from a camera through a screen pixel. Shadow ray is shot towards the light from the intersection position if primary ray intersects anything. Pixel is in shadow if there is any intersection of the shadow ray and scene. This illustration is from Wikipedia[2].

and the computation would be very ineffective. This backward algorithm produces the same results and is much, much more efficient.

Ray and surface interaction is evaluated and the backwards philosophy continues. The ray is traced back to the light source. If it hits another surface along the way, the pixel is in shadow. Additional ray can be reflected from the surface, analyzed the same way as a primary ray and added to the resulting color multiplied by the reflective coefficient.

This is still artificially perfect behavior of light; the surfaces in reality do not bounce off rays this precisely and light sources are not perfect points. We can simulate this by Monte Carlo approach[3] using multiple randomized rays. Instead of calculating intersection for one ray only, we can generate new rays, randomize their direction or origin and take the result as an average.

This technique approximates the random aspect of light travel in a lot of usual light phenomena. It is used in tracing rays back to light sources of certain size, shadow penumbras[^1] motion blur, depth of field, BRDF[^2] and much more. Application details of exact techniques will be discussed in implementation section.

[^1]: portion of shadow where the surface is only partially shadowed
[^2]: four-variable function for modelling the way light reflects on a surface
1. Introduction

1.3 Tools

1.3.1 GPU programming

There are a few ways to run algorithms on a GPU. The most frequent one is writing a shader program and using it in some graphics API, usually OpenGL or DirectX. The realtime raytracer could be written this way using the new compute shaders, but OpenGL are DirectX are quite complicated and verbose APIs. They are not very suitable for rapid prototyping like this. I would need to have a lot of code finished already to be able to test my algorithms quickly enough. These APIs are designed with complex high–performance rendering engines in mind.

That leaves us with two other options: CUDA or OpenCL. OpenCL is a framework for writing programs over a lot of platforms including GPU. OpenCL is a wide standard; once the code is written it is compatible with most of GPUs and CPUs. CUDA is an architecture compatible with NVidia cards only. It is much closer to the hardware and able to run very fast. CUDA is much more compatible with various applications and is very frequently used in industry. That is why I chose CUDA. I can write simple applications in MATLAB, use its CUDA GPU acceleration and run applications on GPU this way.

1.3.2 MATLAB versus C++

MATLAB CPU code performance largely depends on type of the algorithm, coding style, amount of vectorization etc., but it is mostly considered slower than the same algorithm written in C/C++. The JIT compiler does some optimizations behind the scenes, but not all most used MATLAB functions are directly implemented as C/C++ code. C/C++ allows for much finer code optimizations. MATLAB tends to allocate most stuff dynamically, often copies data instead of changing a pointer etc. leaving user without any simple way to change this.

1.3.3 MATLAB language

MATLAB language syntax is quite C–like, there are just a few differences and a lot of functions for vector/matrix math and access. Its meaning is quite clear and it is readable, so I will present algorithms here directly in MATLAB code. Here are some key differences between C and MATLAB syntax:

- https://www.opengl.org/
- https://www.khronos.org/opencl/
Blocks of code (conditions, function, cycles...) are separated similar to Pascal code by end instead of {}:

```matlab
if (condition) {
    code
} else {
    code
}
if (condition)
    code
else
    code
end
```

Arrays/matrices are addressed by round parenthesis and coordinates delimited by commas.

```
A[y][x] -> A(y,x)
```

Array elements are addressed starting from 1, not 0.

```
A[0] to A[N-1] -> A(1) to A(N)
```

We can replace any coordinate with colon. MATLAB then returns slice of source array/matrix with appropriate coordinates. For example: let there be matrix `A((1,2),(3,4))`. Printing `A(1,:)` shows vector (1,3) (all elements with `x=1` and any `y`).

```
A(y,x,:)
```

For cycle is written using colons.

```
for (int i=0;i<n;i++) -> for i=1:n
```

We can change the step size.

```
for (int i=0;i<n;i+=3) -> for i=1:3:n
```

Execution time between tic and toc is measured and printed.

```
tic code toc
```

There is no need to define datatype when declaring variable, but there are all classic datatypes available: int, uint, float, double, strings and more... Functions can be nested and inside function can access variables of parent function.
1. Introduction

function main()
    n=5;
    function a()
        k=n+2;
    end
end

1.3.4 MATLAB code performance

There were quite a few surprises waiting for me when I began writing my
algorithm in MATLAB code. The speed difference between coding techniques
was very apparent once I started looping through every pixel on the image (there
are already half a million pixels in 800x600 image so it is very time–consuming
code). My first approach was using MATLAB’s OOP functionality, it was very
slow. Even function calling had a large overhead. Not even functions for dot
product and cross product frequently seen in all of graphic programming were
usable, one of them in loop body was enough to slow down execution time
several times. I had to resort to simple mathematical expressions, I wrote
the cycle body in one big chunk without calling any functions whatsoever.

This simple demo MATLAB program demonstrates the difference. Code
snippet 1 generates Nx3 random vectors in $\mathbb{R}^3$ and uses MATLAB helper
functions to calculate dot and cross products (completely random operation
just for the sake of benchmarking). Another snippet 2 does exactly the same
thing. I did not use any dot and cross MATLAB functions this time. I broke
the products up into simple math operations (additions and multiplications).
This snippet is slightly less readable but much faster, as we can see on logar-
ithmic plot 1.2. The measurements are listed in Appendix A.3.

I will still use dot() and cross() functions in this text for the sake of
readability. The actual MATLAB code has the expanded numerical version.

Another influential MATLAB performance hack is pre–allocating memory.
Adding element to array one by one can noticeably hurt performance. Preal-
locating it with zeros function speeds up the algorithm a lot; whole array
does not have to be copied every iteration. Maximum number of possible array
elements has to be known when calling zeros function, MATLAB reverts back
to copying array on every iteration otherwise. I approximated array’s upper
bound a few times in my code. Some arrays are not guaranteed to not cross
this boundary when using extremely odd values like very small MINDIFF. An-
other possibility is allocating additional blocks of memory as needed in case
we have no maximum array size approximation. This approach is used in
script loading obj files when allocating every data array.
Algorithm 1 MATLAB code for dot and cross product with MATLAB functions

```matlab
function dotcross(N)
    data = rand(N,3,3);
    tic
    for i=1:N
        dotV = dot(data(i,1,:),data(i,2,:));
        crossV = cross(data(i,2,:),data(i,3,:));
    end
    toc
end
```

Algorithm 2 MATLAB code for dot and cross product broken down to simple math operations.

```matlab
function dotcross_opt(N)
    data = rand(N,3,3);
    tic
    for i=1:N
        dotV = data(i,1,1)*data(i,2,1) + data(i,1,2)*data(i,2,2) + data(i,1,3)*data(i,2,3);
        crossVX = data(i,2,2)*data(i,3,3) - data(i,2,3)*data(i,3,2);
        crossVY = data(i,2,3)*data(i,3,1) - data(i,2,1)*data(i,3,3);
        crossVZ = data(i,2,1)*data(i,3,2) - data(i,2,2)*data(i,3,1);
    end
    toc
end
```
Figure 1.2: Logarithmic plot showing earth-shatteringly better performance of broken down version

This chart shows a relationship between execution speed of code 2 and 1. We can clearly see the version with dot and cross product broken down into simple additions and multiplications is a few hundred times faster on average. This is a big difference influencing the execution time a lot.
Chapter 2

Rendering acceleration methods

The most computationally intensive part of raytracing is evaluating the ray and triangle intersections. The scenes can have millions of polygons and naively computing collision for every ray and triangle pair is very ineffective. The time complexity of this approach using ray and triangle intersections is $O(tr)$, where $t$ is number of triangles and $r$ number of rays. There are two main ways to speed this up: using faster ray and triangle intersection algorithm and reducing number of calculated ray and triangle intersections.

2.1 Ray and triangle intersection

Let’s suppose we have a triangle given by three positions in 3d space according to its three vertices $(v_1, v_2, v_3)$ and a ray given by origin in space $r$ and direction vector $d$, together $(r, d)$. We need to calculate the collision between ray in the positive direction from origin and triangle surface (plus its border and vertices). Then we need to know the nearest point of intersection (if it exists). This is the basic operation our raytracing program will do very often so we need to do it as fast as possible.

2.1.1 Analytic approach

Approach to this problem using analytical geometry is straightforward: we can calculate intersection of triangle plane with our ray and then find out if that collision lies within our triangle. Triangle given by points $(v_1, v_2, v_3)$ has its plane-defining vectors $p_1 = v_2 - v_1$, $p_2 = v_3 - v_1$, starting point $v_1$ and normal $n = \text{cross}(p_1, p_2)$. We can find out if the ray (line) intersects triangle plane by comparing plane’s normal and ray’s direction vector:

$$n \cdot \text{dir} \neq 0$$
The line and triangle are parallel if plane’s normal and ray’s direction vectors are orthogonal, which means their dot product is zero. Otherwise, the intersection exists. Now we need to calculate that point.

Any line between point on our plane \( v \) (any linear combination of \( p_1 \) and \( p_2 \)) and \( v_1 \) is by definition orthogonal:

\[
(v - v_1) \cdot n = 0
\]

We can look at point \( v \) as point on the line \( r + dc \):

\[
(r + dc - v_1) \cdot n = 0
\]
\[
(dc) \cdot n + (r - v_1) \cdot n = 0
\]

\[
c = \frac{(v_1 - r) \cdot n}{n \cdot d}
\]

\( c \in \mathbb{R} \) is a parameter and \( n \cdot d \) is not zero, we already classified that case as non-intersecting. We can now easily compute the intersection point as:

\[
tp = r + dc
\]

We can limit this line to our ray by allowing only positive \( c \).

We need to decide if the point is inside the triangle now. Fast algorithm for doing this is checking if the point is on the right side of every triangle line. We can do this by calculating clockness of new triangles created from triangle line and our point against triangle normal:

\[
N = (v_2 - v_1) \times (tp - v_1)
\]
\[
dot = N \cdot n
\]

The right clockness means the new normal \( N \) has the same direction as our original normal \( n \) which means dot product is less than zero. We do this same procedure for other two triangle lines \( (v_2, v_3) \) and \( (v_3, v_1) \) too. If triangle point \( tp \) passes all three clockness/sideness tests, ray intersects our triangle in point \( pt \). The whole algorithm is at 3.

2.1.2 Triple scalar product approach

Triple scalar product is product of three 3D vectors and results in a scalar.

\[
A \cdot (B \times C)
\]
2.1. Ray and triangle intersection

Algorithm 3 MATLAB code of the basic ray–triangle intersection algorithm

```matlab
n = cross(v2-v1,v3-v1);
if (dot(n,d)==0)
    return 0;
end

c = dot(v1-r,n)/dot(n,d);
if (c<0)
    return 0;
end

% for every (vv1,vv2) from (v1,v2),(v2,v3),(v3,v1)
trianglePoint = r + dc;
newN = cross(vv2-vv1,tp-vv1);
if (dot(N,n)>0)
    return 0;
end
```

We can use this special product to compute our triangle and ray intersection much faster, as it is described in [4] in 5.3.4 Intersecting line with a triangle. Let $A$ be our ray direction vector, $B$ vector from ray origin to one vertex of a triangle and $C$ vector to another vertex.

$$d \cdot ((v1-r) \times (v2-r))$$

Diagram 2.1 should help with understanding the intersection mechanism. $B$ and $C$ create plane spanning from one triangle side to ray origin. $B \times C$ then gives vector orthogonal to this new plane, directed away from triangle and ray origin. Angle between $B \times C$ and our ray is then $\geq 90^\circ$ when our ray shoots the triangle and $< 90^\circ$ when it does not. We can differentiate between these two states by checking if triple scalar product is non–negative (it can be equal to zero if our ray shoots the triangle line). We can compute three triple scalar products this way. The ray shoots the triangle if they are all not negative.

The choice of side ordering and/or which side of $< 90^\circ$ means intersecting with triangle depends on which coordinate system we use: if it is left–handed or right–handed. We are using left–handed coordinate system and our triangles are defined in anti–clockwise order. Our algorithm has culling turned on.

Backface culling can be turned off by changing the angle between $B \times C$ and our ray check. The dot product sign would need to be the same for every side. Doing this can speed up computation in case of a lot of paper–thin double–sided polygons. There would otherwise need to be two triangles, one for every side.

4 shows the algorithm. We can get barycentric coordinates of intersection point by adjusting triple scalar products so their sum is one. If all the products
2. Rendering acceleration methods

Figure 2.1: Diagram of triple scalar product for bottom side \( bc \) of triangle \( abc \) and ray \( pq \)

Or ray shoots from position \( p \) in direction to another position \( q \). We are determining if the ray shoots the triangle. Let there be another plane defined by two vectors \( B \) and \( C \) (with triangle \( pbc \)) and its normal (orthogonal) vector \( B \times C \). Our ray shoots the triangle if the angle between the ray and plane normal is less than or equal 90°. This occasion can be easily computed by using dot product.

are 0, the program would divide by zero when calculating denominator. But there is only one way our ray could intersect all triangle’s lines at once: the triangle would not exist, it would be a point. We can delete this check if we trust our triangle data. The barycentric coordinates according to vertices \( (v1, v2, v3) \) are now stored in \( (u, v, w) \).

This is already a fast way of ray and triangle collision calculation but we can make it even faster. Triple scalar product is invariant to rotation of its vectors, which is a good property we will use in the next step:

\[
A \cdot (B \times C) = B \cdot (C \times A) = C \cdot (A \times B)
\]
Algorithm 4 MATLAB code for basic triangle–ray intersection using triple scalar product

\[
pq = q-p; \\
pa = a-p; \\
pb = b-p; \\
pc = c-p; \\
u = \text{dot}(pq, \text{cross}(pc,pb)); \\
\text{if} \ (u<0) \\
\quad \text{return} \ 0; \\
\text{end} \\
v = \text{dot}(pq, \text{cross}(pa,pc)); \\
\text{if} \ (v<0) \\
\quad \text{return} \ 0; \\
\text{end} \\
w = \text{dot}(pq, \text{cross}(pb,pa)); \\
\text{if} \ (w<0) \\
\quad \text{return} \ 0; \\
\text{end} \\
\text{if} \ (u==0 \ && \ v==0 \ && \ w==0) \\
\quad \text{return} \ 0; \\
\text{end} \\
denom = 1/(u+v+w); \\
u = u \times \text{denom}; \\
v = v \times \text{denom}; \\
w = w \times \text{denom}; \\
\]

We can reuse cross product \((pq,pc)\) for \(u\) and \(v\) calculation this way.

\[
m = \text{cross}(pq,pc); \\
u = \text{dot}(pb,m); \quad % = \text{product}(pb,pq,pc) = \text{product}(pq,pc,pb) \\
v = -\text{dot}(pa,m); \quad % = \text{dot}(m,pa) = \text{product}(pq,pc,pa) \\
w = \text{dot}(pq, \text{cross}(pb,pa)); \\
\]

It is possible to rephrase this into mathematically equivalent expressions:

\[
m = \text{cross}(pq,p); \\
u = \text{dot}(pq, \text{cross}(c,b)) + \text{dot}(m,c-b); \\
v = \text{dot}(pq, \text{cross}(a,c)) + \text{dot}(m,a-c); \\
w = \text{dot}(pq, \text{cross}(b,a)) + \text{dot}(m,b-a); \\
\]

\[
m = \text{cross}(pq,p); \\
s = \text{dot}(m,c-b); \\
t = \text{dot}(m,a-c); \\
u = \text{dot}(pq, \text{cross}(c,b)) + s; \\
v = \text{dot}(pq, \text{cross}(a,c)) + t; \\
w = \text{dot}(pq, \text{cross}(b,a)) - s - t; \\
\]


![Rendering acceleration methods](image)

### Table 2.2: Floating point operations in basic and optimized scalar triple product ray and triangle intersection algorithms

<table>
<thead>
<tr>
<th>Operation</th>
<th>Basic</th>
<th>Optimized</th>
</tr>
</thead>
<tbody>
<tr>
<td>add/sub</td>
<td>57</td>
<td>20</td>
</tr>
<tr>
<td>mul</td>
<td>45</td>
<td>24</td>
</tr>
<tr>
<td>div</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>comp</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

Three cross products used in $u, v, w$ calculation do not depend on any ray position or direction. Raytracer typically shoots a lot of different rays into the scene. That does not change from ray to ray. We can calculate these cross products before and store them along triangles. The same thing can be done with $c-b$ and $a-c$. There are only 5 dot products and one cross product for every ray and triangle combination now. The rest of the algorithm from [4] does not change.

Let’s get back to the basic algorithm [3] and compare its number of floating point operations with our new, optimized one using this table [2.2]. These sums assume no extra optimizations or cross product acceleration over dot product and such. The difference is obvious anyway, one less comparison will not change much. The optimized one is much faster. The only downside is we need to store 3 cross products and two differences with each triangle now (15 floats). They can be sacrificed and calculated during intersection checking if memory is an issue.

### 2.2 Scene hierarchy structures

We have a very fast algorithm for computing ray and triangle intersection now. But there is another problem waiting for us. The amount of needed ray and triangle intersection computations per pixel can ve very large. The raytraced scene can have millions of triangles and program would need to compute intersection with every smallest triangle in the whole scene, even the ones in completely different parts.

We can save a lot of processing time here by skipping a big amount of triangles in one step. We would like to compute the intersection only for a small amount of triangles around the ray we shot into the scene. There are a number of acceleration structures that can help us with this task. They attempt to sort triangles into small groups according to their locality in the scene, they intersect ray with these groups first and then intersect individual triangles inside those intersected groups, ignoring the ones outside.

I will demonstrate the structures here in 2D. It should be less confusing.
and more intuitive to grasp them this way. Conversion from 2D to 3D (or any other–D) structures is trivial.

It is good to know the bounding box of every triangle, so our algorithm can easily determine the bounds of triangle’s location. That is simple, we just need to look at the smallest and largest coordinate of our triangle and store the bounding box as these two positions (minimum and maximum). It is 2 positions, 4 floats in our 2D case or 2 positions and 6 floats in 3D.

Measurement results from [5] and [6] were used for performance comparisons in the following section.

2.2.1 Uniform grid

The simplest kind of these acceleration structures is uniform grid. The idea is simple here: we introduce an uniform grid of dimensions $W \times H$ over the whole scene. Every cell of the grid has assigned pointer to every triangle inside or partially inside.

The algorithm for every ray shot into the scene is as follows: every cell of the grid is checked for intersection with the ray. If the ray passes through it or touches it, the ray is checked for collision with every triangle of the cell. The method then returns the closest intersection point to the origin of ray (visible to the camera). We can see diagram of the uniform grid 2.3. Blue cells are intersecting our ray, their green triangles are checked for intersection and the red line is our ray. Pseudocode of uniform grid construction is 5 and pseudocode of uniform grid traversal 6.

The upper bound cost of construction stage is $O(TC^D)$ ray–triangle intersection calculations. C is average number of cells along an axis, T is number of triangles and D is the dimension of universe. We need to perform a ray–triangle collision for every cell and triangle pair.

The upper bound cost of traversal stage of one ray is $O(TC^D)$ ray–triangle intersection calculations as well. That would happen if the ray hit every cell with every triangle in it (that means all triangles are large and spanning across all the cells). Ideal average cost for uniform scene would be $\frac{T}{C^D}$.

We can choose what exactly to do when some triangles overlap cell boundaries. We can slice one triangle into 2, 3 or 4 (up to 8 in 3D) if it is on the border or into even more of them if it spans across multiple cells. That speeds up calculation but we need more memory to store more triangles (especially in case of very big ones). Another approach is not to do anything with triangles and have some of them assigned to multiple cells. We might need to calculate more intersections this way but the memory footprint would be zero.

This simple acceleration structure has a few obvious problems. We need to
2. Rendering acceleration methods

Figure 2.3: Diagram of uniform grid acceleration structure usage

This diagram illustrates cell traversal when using an uniform grid. There are many triangles referenced in grid’s cells. Red line is our ray shot through the uniform grid. Intersected cells are colored blue. Triangles referenced in these cells are green. They are checked for an intersection. It is possible some triangles are referenced by more cells and checked for collision multiple times.
2.2. Scene hierarchy structures

Algorithm 5 Pseudocode of uniform grid acceleration structure construction

\begin{algorithm}
\textbf{for all} WxH cells \textit{c} \textbf{do}
\begin{algorithmic}
\State \textit{c} $\leftarrow$ newCell
\end{algorithmic}
\textbf{end for}
\textbf{for all} WxH cells \textit{c} \textbf{do}
\begin{algorithmic}
\State \textbf{for all} triangle \textit{t} \textbf{do}
\begin{algorithmic}
\State \textbf{if} collision\textit{(c,t)} \textbf{then}
\State \textit{c} $\leftarrow$ \textit{t}
\State \textbf{end if}
\end{algorithmic}
\textbf{end for}
\end{algorithmic}
\textbf{end for}
\textbf{return} every cell
\end{algorithm}

Algorithm 6 Pseudocode of uniform grid acceleration structure traversal

\begin{algorithm}
\textit{nearestColl} $\leftarrow$ \textit{inf}
\textbf{for all} WxH cells \textit{c} \textbf{do}
\begin{algorithmic}
\State \textbf{if} collision\textit{(c,t)} \textbf{then}
\begin{algorithmic}
\State \textbf{for all} triangle \textit{t} assigned to \textit{c} \textbf{do}
\begin{algorithmic}
\State \textbf{if} collision\textit{(t,ray)} \textbf{then}
\State \textbf{if} \textit{coll} $<$ \textit{nearestColl} \textbf{then}
\State \textit{nearestColl} $\leftarrow$ \textit{coll}
\State \textbf{end if}
\State \textbf{end if}
\end{algorithmic}
\textbf{end for}
\textbf{end if}
\end{algorithmic}
\textbf{end for}
\textbf{return} \textit{nearestColl}
\end{algorithm}

Go through every cell for every computed ray even though most of the cells are missed completely. This is especially apparent in the 3D case where the number of cells is cubic instead of square in this 2D universe. While the 2D scene of 10 $\times$ 10 units would have 100 cells, the 3D equivalent of 10 $\times$ 10 $\times$ 10 units would have much larger amount 1000 cells and the iteration through cells would be one order slower.

Uniform grids have cells distributed uniformly across the whole scene (hence the name). This can have good speed implications for scene with uniform polygon density across it, for example some dense bushes.

That is unfortunately practically never the case; scenes nearly always tend to have big density discrepancies in them. Huge parts of them are completely empty and without triangles (air, areas with one thin wall, insides of big objects (we are keeping records of the surface representation only)) while others have more objects in one place (forests) or are modelled in a much
2. Rendering acceleration methods

more detailed way (faces, hair).

That means our algorithm can push a big amount of triangles in one cell and very small number of them in others. This can very easily result in a 99% of triangles in one cell only for a very large scenes and virtually no acceleration at all or even slower speeds if there are a lot of rays shot trough that cell. We could try to fight this by increasing the number of cells. That would also mean more ray and cell collision checks, possibly more than resulting ray and triangle ones (the amount of cells rises cubically in 3D).

That does not mean uniform grids are completely useless acceleration structure. They are easy and fast to construct and their performance is good for small uniform scenes. They are a good choice when there is not enough or nearly no time for structure construction or possibly a simpler method for deepest levels of more advanced acceleration structures described next. Following smarter acceleration structures try to counter the weaknesses of uniform grids.

2.2.2 Octree (quadtree in 2D)

We can replace our uniform grid spanning over the scene with quadtree. Quadtree is a hierarchical structure which builds itself recursively above our scene by dividing its leafs. It respects the density of the scene and divides only when it is still advantageous. That means the areas of high triangle density are populated by more octree cells.

The construction algorithm runs like this: We assign some threshold value MINTRI of minimum triangles in a leaf needed for its further division. We assign some threshold value MINSIZE as the minimum size of cell to still divide. The root node of tree is constructed above all triangles and all triangles are saved into temporary array. We go through every triangle in current node. If it is inside or touches the node, we save the pointer to the triangle to the temporary array. If the number of triangles in the node is equal or more than MINTRI or the cell size is smaller than minimal MINSIZE, we divide the node in half in both axes forming four new cells this way. We give every new cell our temporary array and proceed the same way as with this node. If the number of triangles in this node is less then MINTRI and the cell size is more or same as MINSIZE, we do not divide current node into four new cells and save our temporary array into this node.

That was much more complicated than in case of uniform grid already. Now the traversing algorithm: We start in a root node and with empty stack. If current node does not have any child nodes, the ray is checked for intersection with every triangle of current node and the closest intersection to the ray origin (if any) is remembered. When the current node has child nodes, every child node is checked for intersection with our ray. If the ray intersects or
2.2. Scene hierarchy structures

Figure 2.4: Diagram of quadtree acceleration structure usage

This is similar diagram to the previous one. Each intersected node’s children are checked for intersection. If there are no children, the node is colored blue and all its referenced triangles are checked for intersection and colored green.

If a triangle touches this child node, the child node is pushed onto stack. After all child nodes have been dealt with, the stack is checked. If it is empty, the traversal has ended. If there is a node, we pop it out and process the same way as our root node.

We can see diagram of the quadtree. This quadtree was generated with maximum 3 triangles per node. Pseudocode of quadtree construction is \[7\] and pseudocode of quadtree traversal \[8\].

I explained the acceleration structure on its 2D quadtree version for simplicity and good illustration: the structure and algorithm is practically the same in 3D universe. It is called octree, the nodes are not 2D patches but 3D cubes and one leaf is not divided into four patches, it is divided into eight cubes. It simply halves the nodes along all three axes. The rest of the algorithm stays
2. Rendering acceleration methods

**Algorithm 7** Pseudocode of quadtree acceleration structure construction

```
node ← rootNode  // creates root node
CONSTRUCTNODE(node,t)

function CONSTRUCTNODE(node n, triangles t)
    for all triangle t do
        if collision(node, t then
            temp ← t
        end if
    end for
    if node.triangles.size >= MINTRI then
        if node.size >= MINSIZE then
            node.childs ← node.divide  // divides node into children
            for all child nodes cn do
                CONSTRUCTNODE(cn, temp)
            end for
        else
            node.triangles ← t
        end if
    end if
end function
```

**Algorithm 8** Pseudocode of quadtree acceleration structure traversal

```
earestColl ← inf
return nearestColl

function SHOOTRAY(ray, node)
    if n.children then
        for all child cn do
            if collision(ray, cn) then
                SHOOTRAY(ray, cn)
            end if
        end for
    else
        for all triangle t from node.triangles do
            coll ← collision(t, ray)
            if coll < nearestColl then
                nearestColl ← coll
            end if
        end for
    end if
end function
```
2.2. Scene hierarchy structures

the same.

Time complexity derivation is a little tricky and not that useful. The worst case scenario for tree construction is probably around $O(nodeDivisions + leafNodes \times T) = O(T \times constant + \frac{T^2 \times SIZE}{MINSIZE}) = O(T^2)$ where $N$ is number of divided nodes and $T$ number of triangles. Number of divided nodes and number of leaf nodes are both linearly dependent on $T$ and every leaf node can have at worst $T$ triangles. $SIZE$ is size of the root node. The average complexity should be around $T$ (constant small number of triangles per leaf) but is highly dependent on density distribution of the scene which can grow the amount of nodes considerably.

The worst time complexity of one ray traversal is limited by minimum node size as well. It is $O(\frac{T^2 \times SIZE}{MINSIZE}) = O(T^2)$ when dealing with all triangles spanning across all nodes divided as small as possible. The ideal average complexity is around $\log(T)$.

We can see major improvements over uniform grids here. The construction stage is more complicated involving a recursive algorithm, but the structure behaves much better. More nodes are created in dense areas and empty space is not divided further. The traversal stage recursively ignores cells without any geometry.

This algorithm still has certain weaknesses. Every division is spatially constant, every child node is divided in the same way. This is not a problem for gracefully detailed scenes, scenes with gradually appearing detail in them. If there is a cell with a lot of detail in one very small part and nearly empty space in another, the quadtree divides itself several times even if it could cut off the very dense portion right away. Imagine a person staying in an empty field on ideally bad position. Quadtree algorithm would divide the field many times before it got to the person. Better solution would be cutting off the person right away.

Another potential problem with this algorithm could be the overhead. We always cut the leaf in 3D version into 8 cubes and proceed to compute the next step for every cube. This means larger memory footprint and potentially lot of useless empty nodes generated. The scene might be less dense in one direction then another and it might not be ideal to always cut along all the dimensions equally.

This acceleration structure already behaves very well and is quite frequently used. There is no need for extra heuristic and the division step is very simple. Following acceleration structures try to attack these weaknesses once again.
2. Rendering acceleration methods

Figure 2.5: Diagram of k–d tree acceleration structure usage

This diagram is very similar to the last one. Each node’s two children are checked for an intersection.

2.2.3 k–d tree

This tree structure is very similar to the octree. The algorithm for acceleration structure construction and traversal are practically same as with octree. The only difference is node division. There is only one cut when dividing a node (not one cut per axis as with octree) and it does not need to be in the exact middle of a node. The axis which is cut and the place of the cut is chosen for every node individually based on some heuristic. Fast, but not best simple heuristic could be choosing the longest axis and placing the cut so that the number of triangles on both sides of the cut is equal. Better, but computationally intensive heuristic will be discussed later.

There is no point in describing the algorithm in detail again. It is identical with octree with small difference in the way the nodes are divided. We can see the k–d tree diagram 2.5.
k–d tree is effectively a binary tree version of octree, so the same order time complexity limits should apply when using constant time heuristic. The time complexity greatly depends on the type of heuristic used. Finding the true optimum is NP hard problem.

k–d tree effectively reduces octree’s node division cost with cheaper one. This proves as a good strategy as k–d tree defeats octrees by a large amount in most cases. k–d tree is able to divide a scene much better than octree and the division is faster (if very slow heuristic is not used). The number of pointless nodes in octree is lower than number of extra divisions of k–d tree.

Interesting approach to k–d trees are k–d trees with ropes[7] (pointers to neighbouring). They seem to be fast but memory intensive. It is also possible to split a node into more than two parts. That can introduce extra overhead.

2.2.4 Bounding Volume Hierarchy (BVH)

Bounding Volume Hierarchy is next slight upgrade to our tree–based method. Algorithm body stays the same once again, only the division method is different. There is still single cut determined by some heuristic but children are different. They are not cut–off part of their parent node, they are a new bounding box of their geometry. The ray intersection is calculated only against the new, small bounding box. We can see BVH diagram[2.6]. The first image shows bounding boxes after the root division because the hierarchy is not very clear in the second image.

The order of time complexity stays the same, it is a binary tree again.

This potentially cuts off more empty space around the child node decreasing depth of the tree. That is true and the construction stage is shorter this way. However, checking for ray and bounding box intersection is slower than checking intersection of ray and two planes in k–d tree case: we need to test for the whole bounding box which is six planes in AABB case. k–d tree and BVH are in practice about the same speed, the performance differs with scene type. BVH can be constructed in less time because the nodes get smaller faster.

Child nodes in k–d tree are completely disjoint and there is no place between them. That is not the case with BVH. Their children’s geometry and its bounding boxes can overlap or not touch at all. The algorithm tests all six sides of bounding boxes anyway. Care must be taken about the amount of overlap. Some amount of overlap can save triangle memory but it could severely deepen the tree and/or slow down the traversal. If there are some distinctly bigger triangles in the scene, the child nodes containing them could not be smaller. There needs to be some threshold of overlapping amount.
BVH tree's nodes have children nodes defined as two bounding boxes. These two bounding boxes can overlap or be disjoint. This makes this diagram not very clear. The first illustration show the division of root node only.
2.2. Scene hierarchy structures

BVH is a very popular acceleration structure. Its performance is about the same as k-d trees (it highly depends on the way the scene is organized) and the construction phase is shorter. It’s often used with animated scenes where tree is modified every frame.

It is possible to introduce more than two bounding boxes per node. The bounding boxes do not even have to be AABBs. They can be spheres, convex hulls etc. instead. There is also Bounding Interval Hierarchy [8] which combines k-d tree and BVH. The nodes are divided by two planes in the same axis. The children’s bounding boxes are then stuck to the parent’s borders, only their inner border can move freely.

2.2.5 Heuristics for k-d tree and BVH splitting

Tree quality, construction time and traversal speed of k-d trees and BVH depend highly on the heuristic used. This heuristic is usually some middle ground between tree construction and traversal time. Quickly split trees will be a bit slower and vice versa.

There are simple one-step heuristics with quite good results.

- **Middle split**: the axis in which the node is largest is chosen for splitting and is cut right through the middle. This heuristic is very simple, does not need to look at any geometry and generates well predictable node size. It is not content-aware and getting to a few triangles can take a lot of splits.

- **Mean split**: the node is split at the mean of geometry in selected maximum-sized axis. The mean is calculated from triangle’s vertices or their centers.

- **Median split**: mean split’s weakness are outliers. If there is a bunch of triangles in one place and one or two far away, the mean algorithm chooses split in the empty space between, placing a few triangles on one side and too many on the other. Median split fights this by splitting the node in the middle over the maximum sized axis at the median of triangles. It is split in the way that approximately half the triangles are on one side. Median of vertices or triangle center coordinates is still very cheap to compute: $O(n)$ for $n$ points.

2.2.5.1 Surface Area Heuristic (SAH)

Median split might sound like an optimal heuristic, it splits the nodes into two parts with the same amount of triangles after all. It is not the case, unfortunately. Tree constructed this way has one big disadvantage: every
node has a large amount of triangles in and ray has to compute its intersection with some triangles every node it visits. It is more optimal to leave some big nodes with no or a few triangles from time to time. The ray passing through this node does not have to compute that much intersections.

Finding the optimal solution for tree configuration like this is not a simple problem. This is the reason we have SAH. It attempts to approximate the ideal tree splitting by considering the volume and number of primitives (triangles in our case) in split parts. The cost of splitting a node is:

\[
\text{cost}_{\text{split}} = \text{cost}_{\text{traverse}} + \text{cost}_{\text{intersect}} \times \frac{\text{vol}_{l} \times \text{prims}_{l} + \text{vol}_{r} \times \text{prims}_{r}}{\text{vol}}
\]

vol is volume and prims is number of primitives. The cost of not splitting the node is simply:

\[
\text{cost}_{\text{notsplit}} = \text{cost}_{\text{intersect}} \times \text{prims}
\]

Every node evaluates its not splitting cost. The algorithm iterates over all three axes and all splitting positions between primitives. The new splitting cost is computed every iteration. The action with smallest cost is then performed: the node is either split at minimum–cost position or not split at all. Pseudocode for this algorithm is [9].

This heuristic’s calculation ignores the possibility of node splitting after current split. This would exponentially increase the computation time and this simplistic heuristic was proven to work quite well [9]. The resulting tree’s splitting performance is believed to be around two times better than that of median split trees. This algorithm is quite slow compared to previous methods.

The good sortless implementation uses bucket sorting. Primitives are not sorted once in the beginning. Every node iterates through all its primitives, adds them into fixed amount of “buckets” per axis and splits between these buckets. Resulting algorithm is faster, does not need any memory for linked lists and produces approximate results.
Algorithm 9 SAH pseudocode for computing one node’s split action

\[
\begin{align*}
\text{costNotSplit} & \leftarrow (\text{costIntersect} \times \text{t.size}) \\
\minCost & \leftarrow \text{costNotSplit} \\
\minAction & \leftarrow \text{notSplitAction} \\
\text{for all axis a} & \text{ do} \\
& \quad \text{ for all split position p between triangles on axis a do} \\
& \quad \quad t \leftarrow \text{t.split}(p) \\
& \quad \quad \text{temp} \leftarrow (\text{t.left.volume} \times \text{t.left.size} + \text{t.right.volume} \times \text{t.right.size}) \\
& \quad \quad \text{splitCost} \leftarrow (\text{costTraverse} + \text{costIntersect} \times (\text{temp} / \text{t.volume})) \\
& \quad \quad \text{if splitCost} < \minCost \text{ then} \\
& \quad \quad \quad \minCost \leftarrow \text{splitCost} \\
& \quad \quad \quad \minAction \leftarrow \text{splitAction}(p) \\
& \quad \quad \text{end if} \\
& \quad \text{end for} \\
& \text{end for} \\
\minAction \text{.do()} \quad \triangleright \text{ do the best action}
\end{align*}
\]
3.1 Basic raytracing algorithm

3.1.1 Per–pixel triangle iteration

I implemented the algorithm in CPU MATLAB code with simple GPU porting in mind. I wanted to write CPU code first and then simply port it to GPU by exchanging the for cycle for calculating every pixel by GPU function. The code does not use MATLAB dot and cross helper functions.

My testing program has very simple scene settings. Image size, camera attributes and directional light attributes are set globally at the beginning of the main m script file. The number of scene attributes is too large to input meaningfully using arguments. They are used only for a limited number of frequently changed settings.

My program’s input is Wavefront object file *.obj. It is simple universally accepted geometry definition format. It is easily readable as asimple ASCII text. I import this file format by using Wavefront OBJ toolbox MATLAB import script\[8\] Vertex positions, vertex normals and faces ale loaded into MATLAB array optimized for raytracing speed.

String to float conversion script was the slowest part of obj loading script. I replaced it with Fast String to Double Conversion\[9\] mex file. This is Windows binary file executed by MATLAB. It exploits fast C++ string handling capabilities and is much faster than MATLAB version. Obj file loading is about 30% faster now.

Primary ray is shot from camera position trough every resulting image

\[8\]http://www.mathworks.com/matlabcentral/fileexchange/27982-wavefront-obj-toolbox
\[9\]http://www.mathworks.com/matlabcentral/fileexchange/28893-fast-string-to-double-conversion
3. Implementation

pixel. The ray is tested against every triangle in the scene and closest position to the camera is found. The pixel is lit and surface and light interaction is computed. The resulting image is shown.

Each stage of the algorithm is described in detail.

3.1.1.1 Primary ray shooting

Camera position and orientation target (point in 3D space seen by camera in the center of the screen) are set using two vectors pos and tg, the scalar field of view fov is in degrees vertically and asp is aspect ratio of the image width/height.

There is a primary ray shot for every pixel on the resulting image. Ray’s origin is camera position pos and the direction is towards the camera target tg deviated by screen position in bounds defined by fov as we can see on the image 3.1. The formula for primary ray calculation follows:

\[
\text{rayDir} = \text{dir_{tg}} + \text{fov}_{y} \times \text{up} \times y + \text{fov}_{x} \times \text{right} \times x \\
\text{dir_{tg}} = \text{normalize}(\text{tg} - \text{pos}) \\
\text{fov}_{y} = \frac{\text{fov} \times \text{DEGTORAD}}{2} \\
\text{fov}_{x} = \frac{\text{fov} \times \text{asp} \times \text{DEGTORAD}}{2} \\
\text{right} = [0, 0, 1] \times \text{dir} \\
\text{up} = \text{dir} \times \text{right} \\
x, y \in < -1, 1 >
\]

\( \text{dir_{tg}} \) is normalized direction vector from camera position pos to camera target tg. \( \text{fov}_{y} \) is the screen space vertical component of the ray deviation from direction vector in degrees is converted to radians by constant \text{DEGTORAD} (which is \( \frac{\pi}{180} \)). The same thing happens to horizontal component \( \text{fov}_{x} \) but it is converted from vertical to horizontal field of view by multiplying it with aspect ratio \( \text{asp} = \frac{\text{width}}{\text{height}} \).

Right vector points to the right in the camera space. If one looks through the camera, it is pointing to the right, hence the name. The right vector is orthogonal to the camera direction and general up direction vector and it is calculated by taking a cross product of these two. I defined general up direction as vector \((0, 0, 1)\). I defined Z as up direction of my coordinate system this way. Defining the general up direction like this limits us to the upright camera orientation. The roll could be computed by simple modification: the general
3.1. Basic raytracing algorithm

**Figure 3.1**: Image illustrating camera vectors relationships

Camera is at position \texttt{pos}, vector from \texttt{pos} to \texttt{tg} is camera’s direction vector \texttt{dir}. \texttt{up} and \texttt{right} vectors are calculated from camera’s direction vector \texttt{dir} and general up direction using previous formula. These two vectors are scaled using field of view information. Adding these two vectors to the direction vector results in a new direction vector corresponding to pixel position on screen.
up direction would have to be rotated by roll along camera direction vector axis.

The real, camera–aligned up vector $\mathbf{up}$ (up direction in camera space, $y$ coordinate in screen space) is then calculated simply as a cross product of direction $\mathbf{dir}$ and right camera vector $\mathbf{right}$. $x$ and $y$ are normalized screen space pixel coordinates from $-1$ to $1$. They are from $-1$ because we need to deviate primary ray’s direction from camera direction to both sides of up and right vector directions (down and left too). That also means we need to divide our $\text{fov}$ by two, it would be twice as big than needed otherwise. We need to take care about $\mathbf{dir} = (0,0,1)$ case, cross product would be zero.

There is primary ray shot trough every pixel on the screen this way. Changing outer $\text{for}$ of pixel iteration to $\text{parfor}$ easily enables algorithm to make better use of all processor cores.

### 3.1.1.2 Ray and triangle intersection

I decided to prioritize speed over memory usage in my initial implementation. I would change the algorithm to presave less data and compute more while running if the memory consumption would be issue when raytracing standard scenes.

The ray and triangle intersection algorithm used is the fast triple scalar approach described in [4]. I implemented the speed optimized version with three cross products and two triangle sides vectors ($\mathbf{v3-v2}$ and $\mathbf{v1-v3}$) presaved with vertices data along position and normal.

The standard indexed mesh representation keeps two arrays in memory. One of them has data for every vertex of the mesh (position, normal, texture coordinates...) and other holds triples of vertex IDs to connect to create a mesh. This is memory friendly approach because the most memory–heavy vertex data stuff is saved only once even when there are more triangles with this vertex. The downside of this approach is one extra array lookup: we need to look up ID of the triangle vertices and then lookup every vertex data in another array. GPU’s random global memory operations are known to be slow. I decided to store triangles in one array only, directly listing every vertex with its data and hopefully speeding up the algorithm this way.

Data for $n$ triangles are saved in a few large arrays:

- $\hat{\text{triPos}}(n,3,3)$

  Three–dimensional array describing position of triangle’s vertices in 3D space.
3.1. Basic raytracing algorithm

- \( v_3.v2(n,3) \) and \( v_1.v3(n,3) \)
  Two triangle vectors (subtractions of two border vertices) saved for fast lookup when intersecting the ray.

- \( \text{triNorm}(n,3,3) \)
  Normal of every triangle’s vertex imported from obj file (can differ from actual triangle’s geometry normal for artist’s purposes). This normal vector is always normalized and only its direction is interesting to us. It’s possible to save each normal’s two coordinates with the sign of the third and calculate the third one it at runtime if memory usage is a concern. Imagine we only saved \( x, y \) and sign of \( z \) \( z_{\text{sign}}: \)

\[
\sqrt{x^2 + y^2 + z^2} = 1 \\
z = z_{\text{sign}} \times \sqrt{-x^2 - y^2 + 1}
\]

- \( \text{triCross}(n,3,3) \)
  Three triangle’s cross products saved for fast lookup when intersecting the ray. Can be left out and computed at runtime if memory usage is a concern.

- \( \text{faceNorm}(n,3) \)
  Actual geometry–based triangle normal used for triangle culling. Can be left out and computed at runtime or the culling can be ignored completely.

Each ray is tested for intersection with every triangle in the scene. The triangle is facing away from the ray and ray can never intersect it if the dot product between ray direction and triangle normal is \( > 0 \). This “backface culling” rejects about half the intersection tests early. Ray either does not intersect the scene and pixel is colored black or the intersection closest to the camera is saved.

3.1.1.3 Surface interaction

The first implemented lighting model is simple. The resulting pixel value is defined as \( \text{ambient} + \text{diffuse} \) components of Phong shading model[10]. The normalized value \( p \) of resulting pixel in range \((0,1)\) is calculated using this formula:

\[
p = a + (1 - a) \text{max}(0, (n \cdot d))
\]

\( a \) is manually defined ambient minimal value of pixels unaffected by direct light. \( n \) is a surface normal and \( d \) is a light direction vector. The brightness
3. Implementation

of a pixel on a surface is determined by surface’s angle towards the light (or towards general light direction in case of directional light) lightDir. The dot product is a great tool for this.

We need to know the surface normal \( n \) for this calculation. It is computed by interpolating normals of intersected triangle’s vertices using weights \( u, v, w \) computed while intersecting the triangle:

\[
\mathbf{n} = \text{normalize}(u \mathbf{n}_1 + v \mathbf{n}_2 + w \mathbf{n}_3)
\]

\( n_1, n_2, n_3 \) are three border triangle’s normals. The resulting normal has to be further normalized because interpolating three normalized vectors does not guarantee normalized result (it is actually never normalized except border cases). We need to save the closest intersection’s triangle ID and \( u, v, w \) weights so we can use it for lighting. Another solution is calculating lighting model right after any intersection of primary rays. That can have very decremental effect on performance, especially with more expensive surface interaction like shooting additional number of shadow rays.

3.1.1.4 Performance

Every performance in this thesis was measured on my home computer. Every new set of tests was running on a rebooted PC with fresh Windows 8.1 installation. Every procedure was measured four times. The first measurement was rejected, because the data might not be cached properly yet and MATLAB generated CUDA code is compiled on the first measurement. The median value of following three measurements was treated as a result to eliminate any random spikes. CPU side multithreaded algorithms were launched with 8 local worker threads. PC configuration was:

CPU: Intel Xeon Processor E3-1240v3
(8M Cache, 3.40 GHz, 4 cores, 8 threads)
GPU: NVidia GeForce GTX 760
(1152 CUDA cores, 2 GB, EVGA, 1072 - 1137 Mhz)
OS: MS Windows 8.1

All these parts put together already function as a simple raytracer. I loaded simple obj file with 8 triangles and set up camera and directional light meaningfully. The resulting 640x480 image is [B.1]

This image takes 5.28 seconds to render on my machine. The problems arise when rendering a little more complex scene: here is hemisphere on a plane consisting of 362 triangles [B.2]

This took 142.09 seconds to render. Even modern games are able to render millions of triangles in realtime, typical raytracer works with much more.
This is unacceptable. The same implementation in C++ could be optimized much better and result in much better performance but the linear relationship between triangle count and render time would stay the same.

Let’s try to run this algorithm on GPU. I used `arrayfun` running an assigned function for every input array element. `arrayfun` calculates all array elements parallelly on GPU using CUDA. Input and output arrays have to be converted to GPUArray.

I assigned the code calculating every pixel’s color to `arrayfun` and converted both triangle definition and output arrays to GPUArray. Some things need to be changed when using `arrayfun`. Indexing is possible only for global GPUArrays and is not allowed for any local variables. I rewrote the code to address local variables like `pqX`, `pqY`, `pqZ` instead of indexed `pq(1)`, `pq(2)` and `pq(3)`.

This implementation was much faster. Pyramid scene was rendered in 0.96 seconds and the sphere in 1.75 seconds. Let’s see how it copes with 3842 triangles: [3.3]

It takes 7.88 seconds. Rendering time still (unsurprisingly) increases linearly with triangle count. We need our acceleration structure to fight that.

### 3.1.2 BVH to the rescue

I decided to use Bounding Volume Hierarchy. The BVH tree is built in memory after the mesh is loaded and before rendering. The hierarchy is written in single threaded MATLAB code; BVH acceleration using GPU is possible, but quite complex topic beyond this thesis, which focuses on raytracing itself.

I used the median heuristic for node splitting decisions. It is not as good as SAH but MATLAB is quite slow. It would take a really long time to compute SAH for large scenes; median heuristic is a good compromise. On the other hand, some splitting decisions done by median heuristic might be dumb. The triangles themselves are not cut when overlapping node boundaries and one triangle can be referenced by more smaller nodes. Two child nodes overlap by an adjustable node size ratio.

Simplest structure for BVH implementation would be structures with pointer references between them. There is a slight problem with that: it is not possible to send pointer referenced structures to GPU. One can send only continuous blocks of data. I implemented Bounding Volume Hierarchy using arrays and array indexing. I created these arrays holding information about \( n \) BVH nodes:
3. Implementation

- **treeId(n,2)**
  
  Two-dimensional array describing relationship between nodes. Every row \( i \) represents one tree node with index \( i \). Each node has the right and left child’s ids. Every non-leaf node has two children. If it is a leaf node and it has no children, the first id is 0 and second is index into the treeData. MATLAB indexes arrays from 1 so the 0 is unique.

- **treeData(m)**
  
  One-dimensional array referenced by leaf nodes in treeId. Each entry index starts by number of following ids. List of triangle ids to array used for triangle intersection array follows. The new unused position is used as a new entry index. The size of this array \( m \) is unknown. It is bigger than number of nodes + number of triangles. Border triangles are referenced twice because of our BVH implementation.

- **treeBox(n,2,3)**
  
  Array describing each node’s bounding box used for BVH node intersection checking. Each bounding box is described by two 3D positions: minimum and maximum position in each coordinate.

3.1.2.1 Bounding box intersection

We need to intersect a lot of nodes of BVH tree as fast as possible when traversing. The simplest way to intersect the boxes is, similarly to the naive triangle intersection algorithm, checking ray’s intersection with 6 planes surrounding the box. Our bounding boxes are aligned to the world’s axes so checking ray’s intersection with them is fast and simple. Consider calculating the x coordinate of intersection of our ray \((r, d)\) and plane \(x = x_P\):

\[
r.x + d.x \cdot p = x_P \\
p = \frac{x_P - r.x}{d.x}
\]

\(p\) is our ray’s parameter describing intersection’s position on plane \(x = x_P\). We can do this for all 6 planes and their adequate coordinates and gain 6 parameters. We must take care of dividing by direction vector’s zero. The lazy solution would be to declare the intersection every time.

Our ray shoots the bounding box if it enters our bounding box in every coordinate before leaving our bounding box in any coordinate. We can see this on illustration 3.2. The red ray enters Z plane fist, Y plane second end enters the box with last X plane. This means the maximum parameter of all first (entering) intersections in every coordinate is smaller than minimum
3.1. Basic raytracing algorithm

We are testing red colored ray for intersection with the box. The ray intersects the ground Z plane first (its parameter is the smallest), Y plane second and X plane last. It is inside the box after the last intersection.

Figure 3.2: Ray shooting a bounding box intersecting three coordinate planes in a sequence

parameter of all second (leaving) intersections in every coordinate (this is a bit of a tongue twister which can be botched up pretty easily).

\[
p_{\text{min}} = \max(\min(p_{X1}, p_{X2}), \min(p_{Y1}, p_{Y2}), \min(p_{Z1}, p_{Z2}))
\]

\[
p_{\text{max}} = \min(\max(p_{X1}, p_{X2}), \max(p_{Y1}, p_{Y2}), \max(p_{Z1}, p_{Z2}))
\]

Our ray intersects the bounding box if \(p_{\text{min}} < p_{\text{max}}\). The intersection is going against ray direction (behind the camera) when \(p_{\text{min}} < 0\) and \(p_{\text{max}} < 0\). This simple check potentially improves performance nearly twice if our camera is somewhere in the middle of a scene.

3.1.2.2 GPU traversal implementation

There is no recursion support for MATLAB generated GPU code. I simulated stack by an expanding array of postponed node indices from \texttt{treeId}. The CPU implementation was simple and the acceleration improved rendering
3. Implementation

Times a lot. The torus knot scene that previously took 115 seconds was now rendered after 5 seconds. That is great, let’s port it to the GPU!

There were some problems with implementing stack–based traversal in GPU generated code in MATLAB. MATLAB GPU code can index only in global GPU Arrays (like our triangle or BVH tree node data) and not in any local, temporary variables in one thread. I need an unique stack array for every calculated pixel on the image. There is no feasible way to achieve this using `arrayfun` code in MATLAB. Array indexing is a crucial feature needed for stack operation.

`arrayfun` transforms limited MATLAB code to CUDA kernel. CUDA C code has no problems with local array indexing and C kernels can be loaded using MATLAB. I rewrote the traversal code from MATLAB code to C and loaded it into MATLAB as a kernel.

Loading CUDA C kernel in MATLAB is not very complicated. Arrays are converted to GPU Arrays the same way as while using `arrayfun`. There needs to be manually specified block size: this is 96 threads for my machine and highly depends on architecture. The kernel is launched by calling `feval` with all global arrays as parameters and the output data are loaded back into CPU using `gather`. MATLAB accepts kernels in ptx format which is compiled using NVIDIA CUDA Compiler Driver NVCC\(^\text{[http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/\]}\) from the source *.cu file in CUDA C language:

```
nvcc -ptx kernelname.cu
```

Writing C CUDA code is a bit trickier than MATLAB code. We need to declare every variable before assigning to it and care about data types. Every MATLAB array is now pointer to linear C array, but there is a catch: array dimensions are organized in exactly opposite order than in C. Array with A, B, C dimension lengths in MATLAB is now array(A,B,C) then becomes array[C][B][A] in C or equivalently array[C*B*A] in its linear version. We need to write array[a + b*A + c*A*B] when addressing array(a,b,c).

The array addressing code gained some length this way and became a little less readable. I tried to replace this by defining a macro, but the code was less intuitive and I stayed with this explicit addressing scheme. The other difference from MATLAB is addressing from 0 instead from 1. I needed to keep this property for differencing between leaf and non–leaf node in treeId. I added some –1s to array addressing to make the code even less readable.

3.1.2.3 Performance

GPU performance improvement was similar to CPU implementation. Torus

\(\text{http://docs.nvidia.com/cuda/cuda-compiler-driver-nvcc/}\)}
3.1. Basic raytracing algorithm

We can see the BVH approach yields logarithmic execution time relative to the number of triangles rendered. The naive (iterating through every triangle for every pixel) approach is perfectly linear and takes about 10 seconds for 100,000 triangles. That is why BVH approach is much more usable for practical raytracing scenario.

Figure 3.3: Number of triangles and rendering time relationship for naive approach and BVH

The knot scene which took 0.18 seconds previously now renders in only 0.041 seconds. Rewriting into CUDA itself boosted the performance significantly: 7.88 seconds in generated CUDA code became 0.18 seconds with native CUDA.

The important thing to note about our acceleration structure is its ability to find ray/triangle intersection in logarithmic time. Chart 3.3 shows relationship between number of triangles in a raytraced mesh and rendering time. Naive approach iterating through all triangles for every pixel renders scene in a perfectly linear time. Mesh with 100 thousand triangles is rendered in about 10 seconds. My BVH implementation is practically always faster than naive iterative approach and rendering time grows in logarithmic fashion. This means it is much better suited for scenes with large number of triangles.

Both algorithms were run using GPU on a dragon scene of 100k triangles. The dragon model is from The Stanford 3D Scanning Repository[1]. I deleted random triangles to achieve lower triangle count. I did not want to reload new scene for every new test, that would take a very long time.

Deleting triangles decreases surface area shootable by our rays. This can speed up our algorithm and skew the measurement results for BVH. I increased triangle size to keep surface area approximately the same along all the tests. Let $a$ be diameter of a flat 2D shape, its surface $S$ is approximately:

$$S = a^2$$

Deleting $n$ of $N$ triangles decreases surface area approximately $\frac{N}{n} = r$ times. We need to increase it back $r$ times again using new triangle diameter $b$:

$$Sr = ra^2 = b^2$$

$$b = \sqrt{ra^2} = \sqrt{r}a$$

I increased triangle diameter $\sqrt{r}$ times and gained approximately same surface area for every mesh variant and appropriate BVH results. This unfortunately introduced some noise to the results and results using very low number of triangles can be a bit skewed. The logarithmic relationship is still obvious.

Performance measurements of basic raytracing and its numerous CPU and GPU implementations are in appendix B: A.1. All measured scenes were set up exactly the same with 848x480 render target, measured rendering times are in seconds. “Naive” algorithms iterate all triangles for every pixel. CPU naive rendering times for torus and dragon scenes are linear estimates. We can clearly see the linear and logarithmic relationship here again. BVH is a bit slower for an extremely small number of 10 triangles than basic iterative algorithm.

### 3.2 Advanced raytracing

My raytracing algorithm is very fast now. It is possible to shoot secondary rays and increase picture fidelity without waiting for a long time. I implemented a few advanced features expanding my simple raytracing algorithm.

#### 3.2.1 Supersampling

Every our rendered image till now has visibly jagged borders of objects. This aliasing happens because the intersection tests are discrete towards resulting image pixels. There either is one, clearly defined intersection or no intersectional at all. This property can result in signal aliasing in contrast areas especially visible in moving images due to its “shimmering”. This can be partially solved by supersampling. I can shoot a few primary rays insted of one
and average the result. This is similar to rendering larger image and resizing it down. The aliasing should be much less apparent this way.

I implemented 2x2 rotated grid supersampling. I shot and evaluated 4 rays for every pixel and colored it using the average. The rays are not shot from four corners inside a pixel but from a bit rotated square as seen on 3.4 to fight further vertical or horizontal aliasing. Results can be seen on 3.5. The raytracing now runs 4 times slower.

Aliasing is not visible in every resulting image pixel. Supersampling could be computed as an additional post process only in areas with high contrast. I chose not to do this. It complicates rendering pipeline a bit and I this supersampling helps me a lot with two following raytracing expansions. More rays per pixel could be used for even better looking image. It would also be possible to intersect a ray oriented infinite cylinder instead of a ray and compute average results.

3.2.2 Shadows

Shadow rays are shot for every primary ray if it intersects the scene. They are directed toward light position (or light direction if there is no such thing as light position). If the ray intersects any geometry before it travels to light position, the pixel is in shadow and ambient color is assigned. The pixel is lit and processed with our previous diffuse lighting model otherwise. 3.6 shows the results for our dragon scene and directional light.

3.2.2.1 Smooth shadows

This technique renders pin sharp shadows. Real shadows are almost never perfectly sharp unless the caster is very close to the shadow surface. It is because real light sources have some shape and size. They are not a single, sizeless point and they shine light from more than one position. I can simulate this behavior in my directional light by jittering shadow ray direction like it was not hitting a single point of light source but the sun, window, lightbulb or any other lightsource of a shape and size instead. This is illustrated by 3.7.

It is not a good idea to jitter the shadow ray direction by a regular grid, the aliasing caused by their regularity would be very apparent even for a large number of rays. Irregular jittering is much better. Jittering ray directions randomly is better. Poisson disc sampling gives the best results. It places all points approximately same distance from each other and achieves uniform look even for a small number of them. Random sampling gains this property only for a very large number of points. Random and Poisson disc sampling is illustrated at 3.8.
3. **Implementation**

Figure 3.4: Antialiasing sample pattern

This image illustrates black circle rendering with our supersampling enabled. Intersection of the pixel and the circle would be checked in the center of every pixel without any supersampling. We are checking the circle intersection on three pixel positions in every pixel. These positions are defined by rotated grid point positions to fight horizontal and vertical aliasing. The pixel is colored by background and circle in ratio defined by number of inside points intersecting the circle.
3.2. Advanced raytracing

Figure 3.5: Antialiasing

This enlarged small portion of the image illustrates our supersampling algorithm in action. We can clearly see difference in pixel blending around sharp contrasty edges of the teeth.

Figure 3.6: Dragon rendered with a simple shadow algorithm

This dragon mesh was rendered using a simple shadow algorithm with one shadow ray shot for every primary ray intersection.
3. Implementation

Every primary ray intersection now shoots multiple rays to simulate size of the light source. There are multiple rays shot from the intersection point to the sun to achieve smooth shadow penumbra. The resulting shadowed ratio are averaged shadow ray intersection results.

Computing a Poisson disc distribution for every pixel would be very time consuming. I precomputed single Possion disc point set and used it when calculating every pixel. I implemented this smooth shadow filtering using precomputed Poisson disc of 16 points. The resulting image is 3.9.

Positions from Poisson disc are added to the vector from intersection point to the light. They are transformed to live inside plane orthogonal to the intersection and light vector to wiggle it around. This is achieved by computing up and right vectors like we did for camera.

There are obvious sampling patterns visible because Poisson pattern is the same for every pixel. Computing new point distribution of every pixel would solve this issue, but rotating it is usually enough. We can rotate position $(x, y)$ around origin to new position $(x_2, y_2)$ by angle $\alpha$ using this formula:

\[
x_2 = x\cos(\alpha) - y\sin(\alpha) \\
y_2 = y\cos(\alpha) + x\sin(\alpha)
\]

It would be possible to compute sinus and cosinus for unique random
3.2. Advanced raytracing

Figure 3.8: Left points are completely random, right set was formed using Poisson sampling

The Poisson sampling defines a minimum distance between two points and results in a much better covered space. We can use only 16 rays per pixel and achieve a pleasing look. There would need to be a much larger number of required random rays to achieve the same smooth look. Random rays tend to form small groups. The images are from blog [11].

Figure 3.9: Dragon’s smooth shadow raytraced using Poisson sampling

There was the same random ray Poisson distribution used for every pixel and we can clearly see artefacts caused by this regularity.
3. Implementation

Figure 3.10: Dragon’s smooth shadow raytraced using Poisson sampling with random rotations

Random ray distribution was modified by rotating it using different angle for every pixel. Resulting shadow looks much smoother now.

angles every pixel. Goniometric functions are very slow, precomputing a few rotations and randomly swapping them is much better solution. Used rotation is chosen from 32 precomputed ones based on a random value computed from screen coordinates \((x, y)\) and supersampling iteration value \(i\):

\[
index = (7583x + 8269y + 9257i) \mod 32
\]

7583, 8269 and 9257 are random prime numbers hopefully larger than normal image dimensions. Using large primes for rotation choice hides any possible repeating patterns. Resulting shadows look much better.\[3.10\]

There is one simple optimization available: We do not need to know where is the shadow intersection, just that there is some. We can end searching BVH tree once the first intersection is found. Performance advantages of this fix will be discussed in advanced raytracing performance evaluation.

3.2.3 Ambient occlusion (Environment lighting):

Current raytracer implementation treats everything not hit by a direct light in a very simple way: it is an ambient color. Real light does not work this way; it bounces around for some time. That is why there is just a little light inside bushes or in small crevices on objects.
3.2. Advanced raytracing

Figure 3.11: Ambient occlusion principle

There are rays shot from every primary intersection point. Resulting scene intersection is converted to ambient occlusion. This approximates average pixel accessibility to environment light.

This can be simulated by approximating quantity of light receiveable by primary ray intersection point. This can be done by analyzing geometry around that point, more specifically distance between the point and all other geometry inside hemisphere oriented by surface normal as seen on [3.11].

Other geometry shadows pixel more when the distance to the point is smaller. Occlusion $o$ of one pixel $x$ by a single point $p$ is computed using this formula:

$$o = \frac{\max(0, 1 - \frac{\text{dist}(x, p)}{\text{aoDist}})}{\text{PoissonNum}}$$

$\text{aoDist}$ is a constant defining maximum distance where occlusion still occurs. $\text{PoissonNum}$ is a number of Poisson disc distribution points, 16 in our case.

Ray jittering is the same as the one used in smooth shadows. This ray jittering is a very usual occurrence in raytracing and the same algorithm can have a lot of different uses: motion blur, depth of field, any kind of blurring, BRDF, anisotropic materials... The hemisphere can be easily sampled using
3. Implementation

Figure 3.12: Raytraced dragon’s ambient occlusion only

This shows ambient occlusion pass calculated by pixel accessibility described before only.

2D posson disc coordinates by projecting them along surface normal. Resulting image with 16 ambient occlusion Poisson disc rays transformed by 32 random rotations with maximum ambient occlusion distance of 1 is 3.12.

Ambient occlusion is just an approximation of real environmental lighting. Advanced raytracers compute a few bounces of light. A whole room can be lit by a light coming trough one small window this way. Our ambient occlusion just shadows creases ignoring any entering light.

3.2.4 Putting it all together

Our lighting model is a little more complicated now. A single pixel’s value $v$ is computed using this formula:

$$ v = (a + d(1 - a))(1 - o) $$
$$ d = \max(0, \dot(l, n))s $$

$v$ is resulting value of a one supersampling iteration: four of these values are averaged into final image color. $a$ is value of ambient lighting, $d$ is direct
3.2. Advanced raytracing

Figure 3.13: Raytraced dragon with smooth shadows and ambient occlusion

This final image uses all three described advanced raytracing techniques: it is supersampled using $2 \times 2$ rotated grid sampling, it uses smooth shadows with 16 Poisson distributed random vectors and 32 random rotation lookups and ambient occlusion up to one distance unit with the identical random ray sampling.

\[ \text{light and } o \text{ is amount of occlusion of the pixel, which is the sum we computed from sampling the hemisphere. } l \text{ is light direction (direction of directional light or vector between pixel and light position), } n \text{ surface normal of pixel and } s \text{ amount of pixel that is not in shadow.} \]

This results in a much more dynamic image: Supersampling helps a lot with all Poisson sampled areas: it effectively quadruples number of rays.

Rendering with this amount of visual fidelity using realtime rasterization techniques would be a complex task. There would need to be a lot of manual work and hacks. Using raytracing is a very simple, elegant way; all it takes is shooting a few rays and averaging the result.

Performance measurements of advanced raytracing are in appendix B: All three implementations run on GPU with CUDA code and BVH, rendering times are in seconds. We can see very small time footprint of primary rays: it would be possible to shoot more primary rays only for supersampling purposes or evaluating more complex materials. Shadow rays are one third faster to evaluate than occlusion rays even though there is the same count of them because of our early out optimization.
Conclusion

I successfully achieved the goal of my thesis: I wrote a simple raytracer in MATLAB. Bounding Volume Hierarchy acceleration structure was used to achieve scene intersection in logarithmic time relative to number of triangles.

4.1 Using CPU MATLAB code

There were some problems with using MATLAB for this task. CPU MATLAB code is very simple to parallelize, one just needs to change `for` into `parfor` and loop itself slices into multiple CPU threads given the code is well-behaved. However, normally written MATLAB CPU code can be potentially very slow. Any OOP features or even function calls are very slow and calling them every loop iteration has awful performance consequences.

This results in one long piece of code with no helpful functions like dot or cross products. The code becomes very long, hard to read and easily susceptible to errors. Writing anything more complex this way is not very practical. It is not possible to build a time sensitive part of a program from hierarchy of many small short functions, CPU would spend most of its time on MATLAB overhead.

Another big time saver is preallocating growing arrays. Growing array by single element is very slow because MATLAB copies array into new place in memory every iteration. This is issue in every language, not only in MATLAB. I think this should be addressed in much more user-friendly way like `std::vector` in C++. MATLAB already recognizes growing array and should automatically expand it dynamically only once in a while like `std::vector` does. Preallocating array manually is bothersome and programmer should not worry about that in high-level MATLAB environment.

MATLAB CPU code runs much faster once we write cycle bodies in one go and preallocate everything dynamic. However, it is still much slower than
C++ code. New MATLAB versions use JIT optimizations but MATLAB still does some redundant operation and lowlevel optimizations of C++ code are not possible here.

4.2 Using CUDA with MATLAB

CUDA is implemented into new versions of MATLAB very tightly. It is possible to write quite complex algorithms in MATLAB code using `arrayfun` and easily run them on GPU. It however has its limitations: the same large function calling overhead as in normal MATLAB code and no temporary array indexing. `arrayfun` GPU code can only index inside GPUArray and not in arrays/matrices created inside single thread. That is why it is impossible to write stack–based algorithm using `arrayfun`.

I had to resort to writing CUDA code without any MATLAB functionality in C code. I called it from MATLAB with manually defined block size. Algorithms took a little longer to write this way but the speed gains were major.

4.3 Reevaluating MATLAB usage

Writing programs in MATLAB code is quite easy and fast, simple algorithm prototyping or data manipulation can be done very easily. It is even possible to write a simple raytracer, but not a very fast and compact one.

Using stack–based algorithms in `arrayfun` is not possible and native CUDA C code needs to be used. Mesh loading script and BVH construction written in MATLAB code are quite slow and would highly benefit from begin rewritten in C++. Loading mesh with only 100k triangles takes over 40 seconds on my computer while most today raytraced scenes have millions of them. I used only simple BVH construction with median heuristic. Surface Area Heuristic in MATLAB code would be very slow. C++ mex extension could exploit it and gain marginal tree traversal performance. Whole rendering could potentially become even few times faster. Memory consumption could be problem for larger scenes, C++ would allow us to optimize mesh and memory space a lot.

So we finally arrive at this point: mesh loading and BVH are currently slow or limited and would gain a lot by porting to C++. GPU part of the code is already in native CUDA C. The final conclusion of this thesis is this: prototyping something as complex as GPU raytracer in MATLAB without a lot of lowlevel optimization effort is possible. GPU code executed from MATLAB can be very fast, CPU–side tasks written in MATLAB are however quite slow. Writing them in C++ would be very beneficial to their clarity and execution speed.
Bibliography


Bibliography


Performance measurements

<table>
<thead>
<tr>
<th>scene</th>
<th>pyramid</th>
<th>sphere</th>
<th>torus</th>
<th>dragon</th>
</tr>
</thead>
<tbody>
<tr>
<td>triangles</td>
<td>8</td>
<td>362</td>
<td>3842</td>
<td>100032</td>
</tr>
<tr>
<td>mesh loading [s]</td>
<td>0.007260 0.120421</td>
<td>1.246037 42.740288</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BVH building [s]</td>
<td>0.020141 0.078825</td>
<td>0.859565 25.684082</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU naive [s]</td>
<td>5.281012 142.092658</td>
<td>1380.215984 est 35935.915386 est</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU BVH [s]</td>
<td>6.699966 12.780933</td>
<td>77.167670 105.534854</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU MATLAB naive [s]</td>
<td>0.963156 1.750168</td>
<td>7.878562 16.409162</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU CUDA naive [s]</td>
<td>0.005276 0.025146</td>
<td>0.183155 4.353027</td>
<td></td>
<td></td>
</tr>
<tr>
<td>GPU CUDA BVH [s]</td>
<td>0.006571 0.007724</td>
<td>0.041268 0.070222</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure A.1: Basic raytracing performance

<table>
<thead>
<tr>
<th>scene</th>
<th>dragon</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary rays [s]</td>
<td>0.54</td>
</tr>
<tr>
<td>primary + shadow rays [s]</td>
<td>10.39</td>
</tr>
<tr>
<td>primary + shadow + occlusion rays [s]</td>
<td>25.38</td>
</tr>
</tbody>
</table>

Figure A.2: Advanced raytracing performance
### A. Performance Measurements

<table>
<thead>
<tr>
<th>products</th>
<th>functions [s]</th>
<th>numerical [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>10000</td>
<td>0.385140</td>
<td>0.000929</td>
</tr>
<tr>
<td>20000</td>
<td>0.725887</td>
<td>0.001618</td>
</tr>
<tr>
<td>30000</td>
<td>1.129777</td>
<td>0.002477</td>
</tr>
<tr>
<td>40000</td>
<td>1.511559</td>
<td>0.004980</td>
</tr>
<tr>
<td>50000</td>
<td>1.873704</td>
<td>0.003943</td>
</tr>
<tr>
<td>60000</td>
<td>2.136268</td>
<td>0.005493</td>
</tr>
<tr>
<td>70000</td>
<td>2.463898</td>
<td>0.005789</td>
</tr>
<tr>
<td>80000</td>
<td>2.835621</td>
<td>0.006533</td>
</tr>
<tr>
<td>90000</td>
<td>3.192534</td>
<td>0.007630</td>
</tr>
<tr>
<td>100000</td>
<td>3.521308</td>
<td>0.014755</td>
</tr>
</tbody>
</table>

Figure A.3: Function and numerical MATLAB performance comparison

Table enumerating the difference between calculating dot products in MATLAB using functions and calculating them from scratch.
Figure B.1: Pyramid mesh with 10 triangles
B. Test meshes preview

Figure B.2: Sphere mesh with 362 triangles

Figure B.3: A raytraced torus with 3842 triangles
Figure B.4: Dragon mesh
Appendix C

Contents of enclosed CD

- readme.txt .................... the file with CD contents description
- thesis.pdf .......................... text of this thesis
- matlab_src ..................... directory with raytracing programs
  - raytrace_X.m .................... raytracing MATLAB source files
  - buildBVH.m ..................... MATLAB BVH building file
  - read_wobj.m .................... MATLAB mesh loading file
  - img_X.cu ........................ CUDA source files
  - img_X.ptx ........................ compiled CUDA files
- obj ............................. directory with test meshes
- thesis_src ..................... directory with source files of this thesis