Saint Vincent College

DEPARTMENT OF PHYSICS

JOHN PIERRE MENEGHINI

COMPUTATIONAL MODELING OF X-RAY ATTENUATION AND SIMULATION OF MEDICAL IMAGING

BACHELOR OF SCIENCE

Saint Vincent College

Draft: *April 5, 2024*

SAINT VINCENT COLLEGE

DEPARTMENT OF PHYSICS

COMPUTATIONAL MODELING OF X-RAY ATTENUATION AND SIMULATION OF MEDICAL IMAGING

JOHN PIERRE MENEGHINI

Thesis submitted in partial fulfillment of the requirements for the degree of *Bachelor of Science*

Adviser: Fr. Michael Antonacci O.S.B
Associate Professor

BACHELOR OF SCIENCE

Saint Vincent College
Draft: *April 5, 2024*

Abstract

Regardless of the language in which the dissertation is written, usually there are at least two abstracts: one abstract in the same language as the main text, and another abstract in some other language.

The abstracts' order varies with the school. If your school has specific regulations concerning the abstracts' order, the NOVAthesis LaTeX (novathesis) (LaTeX) template will respect them. Otherwise, the default rule in the novathesis template is to have in first place the abstract in *the same language as main text*, and then the abstract in *the other language*. For example, if the dissertation is written in Portuguese, the abstracts' order will be first Portuguese and then English, followed by the main text in Portuguese. If the dissertation is written in English, the abstracts' order will be first English and then Portuguese, followed by the main text in English. However, this order can be customized by adding one of the following to the file `5_packages.tex`.

```
\ntsetup{abstractorder={<LANG_1>,...,<LANG_N>}}
\ntsetup{abstractorder={<MAIN_LANG>={<LANG_1>,...,<LANG_N>}}}
```

For example, for a main document written in German with abstracts written in German, English and Italian (by this order) use:

```
\ntsetup{abstractorder={de={de,en,it}}}
```

Concerning its contents, the abstracts should not exceed one page and may answer the following questions (it is essential to adapt to the usual practices of your scientific area):

1. What is the problem?
2. Why is this problem interesting/challenging?
3. What is the proposed approach/solution/contribution?
4. What results (implications/consequences) from the solution?

Keywords: One keyword, Another keyword, Yet another keyword, One keyword more, The last keyword

Contents

List of Figures

List of Tables

List of Listings

Acronyms

novathesis	NOVAthesis LaTeX *(p. i)*

1 X-ray Tracing

Ray tracing is a computer graphics technique that involves tracing the path of rays as they pass through a virtual scene. Due to its ability to create highly-realistic images, it has been used extensively in animations, video games, scientific computing, and by designers who need a physically accurate design of a product. While not the only rendering technique or the fastest in computer graphics, its ability to consistently produce physically realistic renders has made it an indispensable tool in many industries. By accurately simulating the behavior of light, ray tracing can capture intricate details of how light interacts with objects, resulting in realistic shadows, reflections, refractions, and global illumination effects. This level of visual fidelity allows for the creation of visually stunning and immersive experiences that were previously challenging to achieve. As hardware capabilities continue to advance and real-time ray tracing becomes more accessible, its applications are expanding, and its impact on various fields is growing, pushing the boundaries of what is possible in computer graphics and visual simulation [2].

1.0.1 The Brute Force Algorithm

The rays in a scene are described mathematically as the parametric representation of a line. In three dimensions, the point at the end of the line $\vec{r}$ at parameter t is given by the following equation:

$$\vec{r}(t) = \vec{r}_0 + \hat{d}t, \tag{1.1}$$

where $\vec{r}_0$ points from the origin to the start of the line and $\hat{d}$ is a unit vector parallel to the direction of the line. The described ray is shown in Figure 1.1.

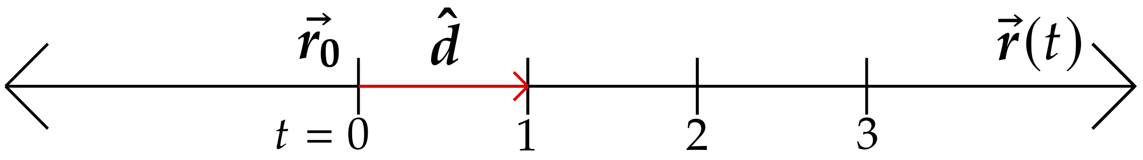

Figure 1.1: A diagram of a parametric ray.

Unlike in real life where light rays end at a camera, we instead trace light rays from the camera to locations in the scene. While one could trace light rays from a light source (forward ray tracing), many of these rays would end up missing the camera entirely, not affecting the image. More efficiently, the light rays are instead traced from the camera to the light source (backward ray tracing), drastically cutting back on the computational load while producing an identical image for most practical cases [2]. So, in the case of backward ray tracing, the method used in this research, rays are emitted from the camera; therefore, $\vec{r}_0$ represents the location of the camera in the scene.

To define the area of the scene visible from the point of view of the camera, one can create a rectangular surface in which all rays pass through, called a viewport. Increasing the size of the viewport increases the amount of the scene that is visible in the final image, and vice versa. This viewport is broken up into a grid, the resolution of which determines the resolution of the final image. Rays are then sent through each point in this grid and out into the scene.

So far, we have a camera, light rays, and a way to define what parts of the scene are visible; all three of which are visualized in Figure 1.2. It is important to note that, by convention, the z-axis in the scene points opposite the viewport.

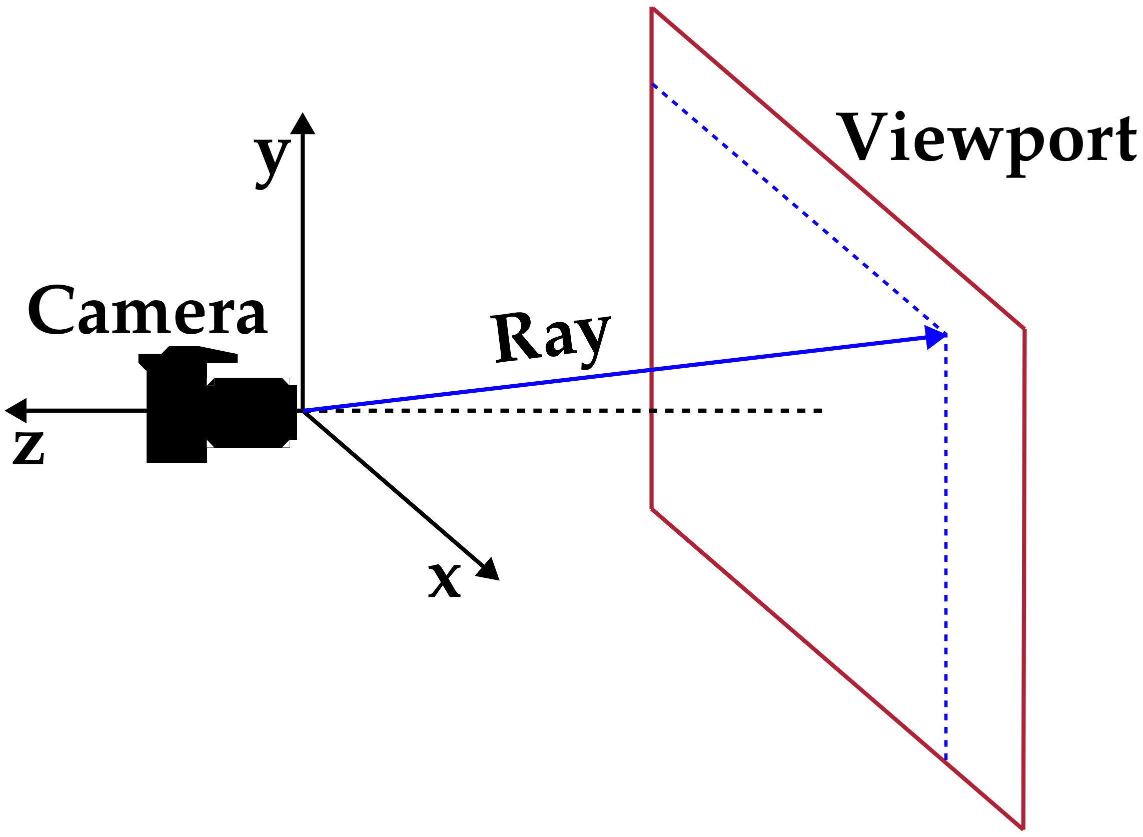

Figure 1.2: A basic ray-tracing setup with a camera, ray, and viewport.

To set the location of the viewport in the scene, we must first define three new vectors:

$\overrightarrow{\boldsymbol{HO}}$, $\overrightarrow{\boldsymbol{VE}}$, and $\overrightarrow{\boldsymbol{FL}}$, which are vectors pointing from the left most to right most point (-x to x direction), the bottom most to top most point (-y to y direction), and from the center of the viewport to $\vec{r}_0$, respectively. A vector pointing from $\vec{r}_0$, to the bottom-left corner $\overrightarrow{\boldsymbol{LHC}}$ can be calculated using the following equation:

$$\overrightarrow{\boldsymbol{LHC}} = \vec{r}_0 - \frac{1}{2}\overrightarrow{\boldsymbol{HO}} - \frac{1}{2}\overrightarrow{\boldsymbol{VE}} - \overrightarrow{\boldsymbol{FL}}, \tag{1.2}$$

which was obtained geometrically from Figure 1.3 [3].

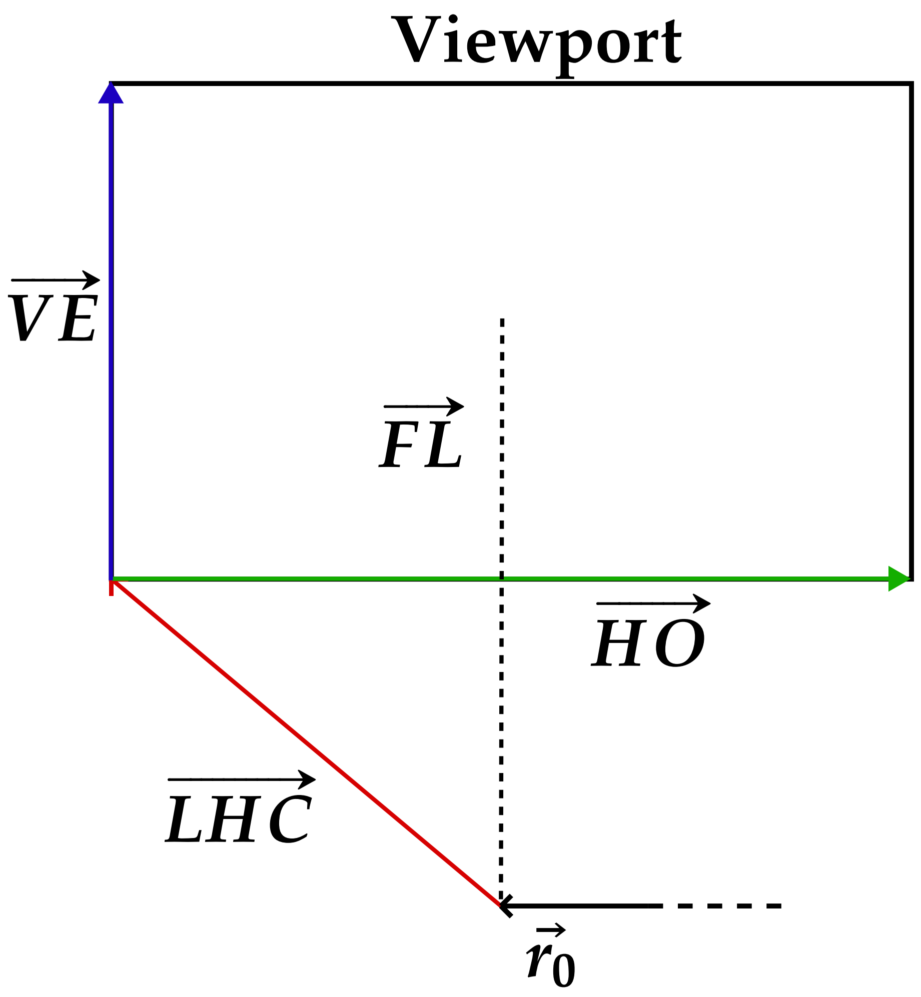
Viewport
$\overrightarrow{\boldsymbol{VE}}$
$\overrightarrow{\boldsymbol{FL}}$
$\overrightarrow{\boldsymbol{HO}}$
$\overrightarrow{\boldsymbol{LHC}}$
$\vec{r}_0$

Figure 1.3: A diagram showing the geometric representations of $\overrightarrow{\boldsymbol{HO}}$, $\overrightarrow{\boldsymbol{VE}}$, $\overrightarrow{\boldsymbol{FL}}$, and $\overrightarrow{\boldsymbol{LHC}}$ with respect to the viewport.

With $\overrightarrow{\boldsymbol{LHC}}$ pointing to the left-hand-corner of the viewport in the scene, $\overrightarrow{\boldsymbol{HO}}$ and $\overrightarrow{\boldsymbol{VE}}$ can be scaled to define a ray pointing from the camera to any point on the viewport. In particular, the viewport's horizontal and vertical coordinates, u_i and v_j, are defined as such,

$$u_i = i/(\text{image width} - 1); \tag{1.3}$$

$$v_j = j/(\text{image height} - 1), \tag{1.4}$$

where the image width and height are the resolution of the rendered image, i is an integer ranging from [0, image width - 1], and j is an integer ranging from [0, image height - 1]. Note that both u_i and v_j range from [0, 1].

Therefore, using these newly defined viewport coordinates, a ray taking the form of Equation 1.1, pointing from the camera to the pixel coordinates (i, j) is given by the following equation:

$$\vec{r}_{i,j}(t) = \vec{r}_0 + t\left(\overrightarrow{LHC} + u_i\overrightarrow{HO} + v_j\overrightarrow{VE}\right), \tag{1.5}$$

which is represented geometrically in Figure 1.4 [3].

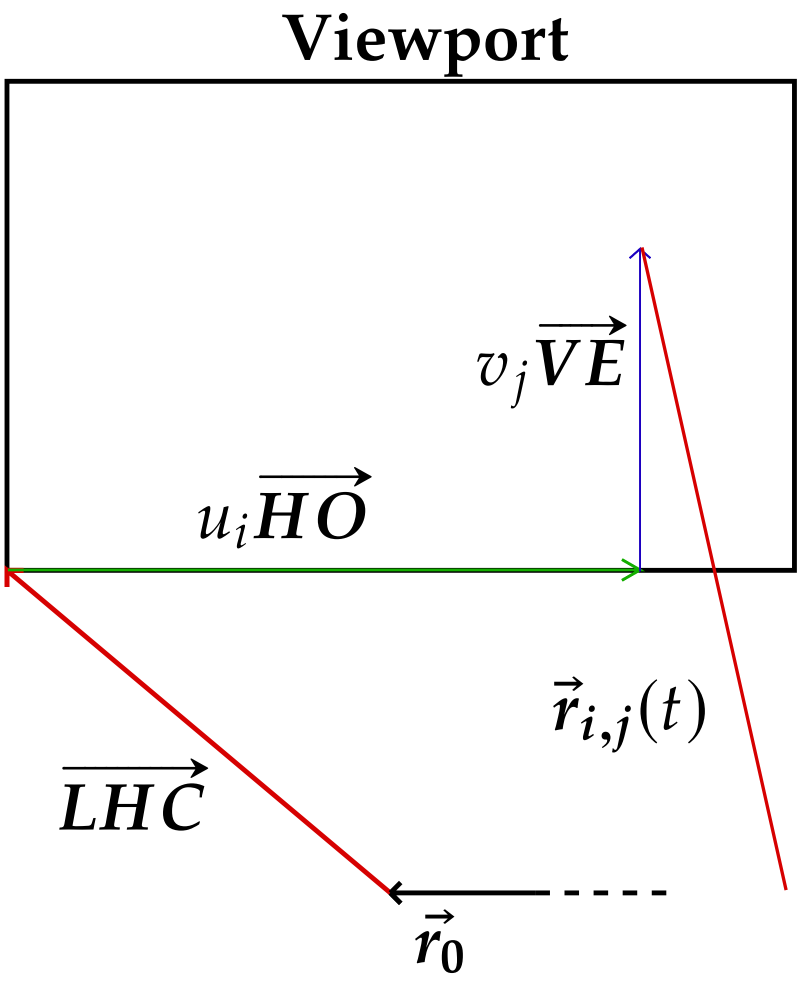

Figure 1.4: A diagram showing the geometric representation of Equation 1.5.

1.0.2 Ray-Triangle Intersections

With the end goal to be able ray-trace x-rays through an arbitrary user-defined mesh, we first need to choose which shape to compose said mesh with. Due to its efficient ray-intersection algorithm and its ease of parallelization on Graphics Processing Units (GPUs), these constituents, referred to as primitives, are typically chosen to be triangles. This following section will develop the ray-triangle intersection algorithm used by the x-ray tracer. In particular, the algorithm used was invented by Tomas Möller and Ben Trumbor, and is subsequentially refered to as the Möller-Trumbore ray-triangle intersection algorithm [1].

Barycentric Coordinates

To motivate the unique coordinate system used by the algorithm, let us first review the concept of center of mass for 3 massive point particles. If 3 objects each have their own location in space $\vec{r}_i$, each with mass m_i, the center of mass $\vec{R}$ is given by:

$$\vec{R} = \frac{m_1\vec{r}_1 + m_2\vec{r}_2 + m_3\vec{r}_3}{m_1 + m_2 + m_3}. \tag{1.6}$$

Notice a few properties of $\vec{R}$ that are intuitively clear:

- $\vec{R}$ will always lie in the plane containing the point particles.
- $\vec{R}$ will always lie in the triangle T with vertices $\{\vec{r}_i\}$.

This means that by changing $\{m_i\}$, $\vec{R}$ will span all set of points in T. This is the motivation behind barycentric coordinates, the coordinate system employed by Möller and Trumbore. In particular, to represent a vector $\vec{v}$ pointing to the surface of a triangle $T \in \mathbb{R}^3$ with vertices $\vec{v}_0$, $\vec{v}_1$, and $\vec{v}_2$, barycentric coordinates can be utilized. These coordinates $\{\alpha, \beta, \gamma\} \in \mathbb{R}$ act as the masses of the vertices of T. Thus, any point $\vec{v}$ along the plane P of T can be represented by

$$\vec{v} = \alpha\vec{v}_0 + \beta\vec{v}_1 + \gamma\vec{v}_2. \tag{1.7}$$

For $\vec{v}$ to be located on or within T, the following requirements on the barycentric coordinates must be satisified:

- $\alpha + \beta + \gamma = 1$ (1.8)
- $\alpha, \beta, \gamma \geq 0$ (1.9)

With regard to the center of mass interpretation, Equation 1.8 removes the computation of the denominator in Equation 1.6, while Equation 1.9 eliminates the possibility of negative mass. For a rigorous proof of the requirements, see (find something).

Möller-Trumbore Intersection Algorithm

The algorithm consists of 2 main steps:

1. Check if ray intersects with P;
2. If so, check if intersection lies outside of T.

1. To check if the ray described by Equation 1.1 intersects with P, one can define two vectors that lie along 2 edges of T:

$$\vec{e}_1 = \vec{v}_1 - \vec{v}_0; \tag{1.10}$$

$$\vec{e}_2 = \vec{v}_2 - \vec{v}_0. \tag{1.11}$$

If $\vec{p} = \hat{d} \times \vec{e}_2$, then let $C = \vec{e}_1 \cdot \vec{p}$. The line $\vec{r}$ does not intersect the plane P iff $C = 0$, which happens iff $\hat{d} \parallel P$. Thus, if $C = 0$, then $\vec{r}$ does not intersect P; otherwise, $\vec{r}$ will intersect P.

2. Solving Equation 1.8 for α and substituting into Equation 1.7 results in

$$\vec{v} = (1 - \beta - \gamma)\vec{v}_0 + \beta\vec{v}_1 + \gamma\vec{v}_2. \tag{1.12}$$

After distributing and rewriting Equation 1.12 in terms of $\vec{e}_1$ and $\vec{e}_2$,

$$\vec{v} = \vec{v}_0 + \beta\vec{e}_1 + \gamma\vec{e}_2. \tag{1.13}$$

Setting $\vec{r} = \vec{v}$ to find the barycentric coordinates for the ray-triangle intersection and solving for $\vec{r} - \vec{v}_0$ yields

$$\vec{r} - \vec{v}_0 = -\hat{d} + \beta\vec{e}_1 + \gamma\vec{e}_2. \tag{1.14}$$

One can then show that taking $(\vec{r} - \vec{v}_0) \cdot \vec{p}$ gives the following equation for β:

$$\beta = \frac{(\vec{r} - \vec{v}_0) \cdot \vec{p}}{C}, \tag{1.15}$$

and similarly, taking $\hat{d} \cdot [(\vec{r} - \vec{v}_0) \times \vec{e}_1]$ results in

$$\gamma = \frac{\hat{d} \cdot [(\vec{r} - \vec{v}_0) \times \vec{e}_1]}{C}. \tag{1.16}$$

To then determine t at which the ray-triangle intersection occurs, one can take $\vec{e}_2 \cdot [(\vec{r} - \vec{v}_0) \times \vec{e}_1]$, yielding

$$t = \frac{\vec{e}_2 \cdot [(\vec{r} - \vec{v}_0) \times \vec{e}_1]}{C}. \tag{1.17}$$

```cpp
 bool triangle::hit(const ray &r, float t_min, float t_max, hit_record &rec) const {
   float kEpsilon = 1e-9;

   vec3 e1 = v1 - v0;
   vec3 e2 = v2 - v0;
   vec3 pvec = cross(r.direction(), e2);
   float C = dot(e1, pvec);

   // ray is parallel to triangle
   if (std::abs(C) < kEpsilon) return false;

   float inv_det = 1.0 / C;

   vec3 tvec = r.origin() - v0;
   float u = dot(tvec, pvec) * C;

   // hit point is outside of triangle
   if (u < 0.0 || u > 1.0) return false;

   vec3 qvec = cross(tvec, e1);
   float v = dot(r.direction(), qvec) * C;

   // hit point is outside of triangle
   if (v < 0.0 || u + v > 1.0) return false;

   float t = dot(e2, qvec) * C;

   // hit point is outside of ray
   if (t < t_min || t > t_max) return false;

   // record intersection point
   rec.t.push_back(t);
   rec.p.push_back(r.at(t));

   return true;
}
```

Listing 1: C++ implementation of the Möller-Trumbore ray-triangle intersection algorithm.

Bibliography

[1] T. Möller and B. Trumbore. “Fast, minimum storage ray/triangle intersection”. In: *ACM SIGGRAPH 2005 Courses*. 2005, 7–es (cit. on p. 5).

[2] J. Peddie. *Ray Tracing: A Tool for All*. en. Cham: Springer International Publishing, 2019. ISBN: 978-3-030-17489-7 978-3-030-17490-3. DOI: `10.1007/978-3-030-17490-3`. URL: `http://link.springer.com/10.1007/978-3-030-17490-3` (visited on 2023-05-12) (cit. on pp. 1, 2).

[3] P. Shirley. *Ray Tracing in One Weekend*. 2020-12. URL: `https://raytracing.github.io/books/RayTracingInOneWeekend.html` (cit. on pp. 3, 4).