CS500 Post-mortem
Ray tracing using CUDA

Jacobo Larsson
Digipen Institute of Technology, Bilbao

1 Introduction

Ray tracing is a rendering technique that consists of casting rays towards a scene, retrieving graphical information by colliding with the geometry, and finally using the gathered data to create an image.

When ray tracing is used to generate an image, the camera position acts as the origin for the rays and create a square in front of it to represent the viewport. After that, two variables *u* and *v*, are created with the purpose of computing pixel positions within the viewport square. We then cast a ray from the camera to each of the computed pixel positions, and intersect the geometry of the scene, to retrieve the corresponding color for that pixel. [1]

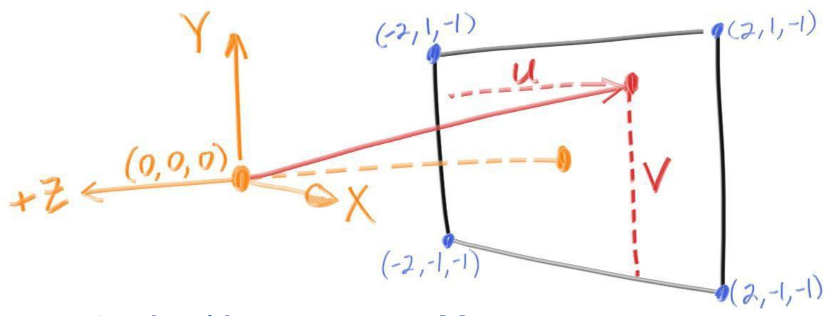


Figure 1 Casted rays´ direction computation [1]

To generate an image with a good resolution, millions of ray castings and geometry intersection queries are needed, which takes a lot of time using a single-threaded, sequential approach. This document presents an optimization to this process using GPU parallel programming techniques, by means of the CUDA API. The proposed approach is to perform all per-pixel operations for all the pixels in the screen in a parallel manner using GPU threads.

This project enables the creation and rendering of different geometrical shapes as well as meshes. These objects can be rendered using different material properties, following the physically based rendering model. [3]

Supported object shapes:

- Spheres, given radius

- Boxes, given length, width and height vectors
- Convex polygons, given a list of vertices
- Meshes, given an .obj file

Supported material types

- Lambertian
- Metal
- Dielectric

A Kd-tree acceleration structure is used for optimizing the ray-triangle queries for the meshes.
The implementation of the features is divided into five milestones, presented in the following sections.

2 Geometry intersections

For the raytracer to render the different objects in the scene, intersection queries must be performed for their geometry.

2.1 Ray-Sphere

Sphere equation: $(x - C_x)^2 + (y - C_y)^2 + (z - C_z)^2 = r^2$

C = centre of the sphere
r = radius

Ray equation: $\rho(t) = P + t\,\vec{v}$

To find the intersection we introduce the ray equation into the sphere one:

$$(P_x + t\,\overrightarrow{v_x} - C_x)^2 + (P_y + t\,\overrightarrow{v_y} - C_y)^2 + (P_z + t\,\overrightarrow{v_z} - C_z)^2 - r^2 = 0$$

It is then solved for t, resulting in:

$$t = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-2(\vec{v} \cdot \overrightarrow{CP}) \pm \sqrt{\left(2(\vec{v} \cdot \overrightarrow{CP})\right)^2 - 4(\vec{v} \cdot \vec{v})\left((\overrightarrow{CP} \cdot \overrightarrow{CP}) - r^2\right)}}{2(\vec{v} \cdot \vec{v})}$$

$$discriminant = \left(2(\vec{v} \cdot \overrightarrow{CP})\right)^2 - 4(\vec{v} \cdot \vec{v})\left((\overrightarrow{CP} \cdot \overrightarrow{CP}) - r^2\right)$$

Pseudo code for the intersection:

```
If  $discriminant = 0$:
    $t = \frac{-2(\vec{v} \cdot \overrightarrow{CP})}{2(\vec{v} \cdot \vec{v})} = \frac{-(\vec{v} \cdot \overrightarrow{CP})}{(\vec{v} \cdot \vec{v})}$

Else if  $discriminant < 0$:
    $No\ intersection$

Else :
    $t_1 = \frac{-2(\vec{v} \cdot \overrightarrow{CP}) - \sqrt{discriminant}}{2(\vec{v} \cdot \vec{v})}$
    $t_2 = \frac{-2(\vec{v} \cdot \overrightarrow{CP}) + \sqrt{discriminant}}{2(\vec{v} \cdot \vec{v})}$

    If  $t_1 \geq 0 : and\ t_2 \geq 0$:
            $\mathbf{2\ intersections}$
            $t_i = \min(t_1, t_2)$

    If  $t_1 \geq 0 : and\ t_2 < 0$:
            $\mathbf{1\ intersection}$
            $t_i = t_1$

    If  $t_1 < 0 : and\ t_2 \geq 0$:
            $\mathbf{1\ intersection}$
            $t_i = t_2$

    If  $t_1 < 0 : and\ t_2 < 0$:
            $\mathbf{No\ intersection}$
```

2.2 Ray-box

Plane equation: $(P - C) \cdot \vec{n} = 0$

C = point in the plane
n = normal of the plane

Introduction of the ray equation into the plane one:

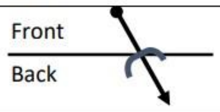
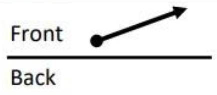
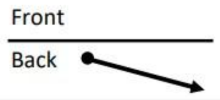
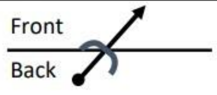
$$(P + t\,\vec{v} - C) \cdot \vec{n} = 0$$

Solve for t:

$$t_i = -\frac{(P - C) \cdot \vec{n}}{\vec{v} \cdot \vec{n}} = -\frac{\overrightarrow{CP} \cdot \vec{n}}{\vec{v} \cdot \vec{n}}$$

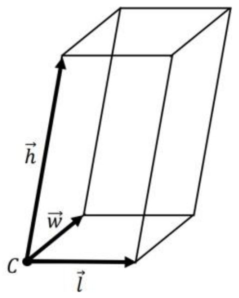
A plane can be used to divide the space in two parts called half-spaces. Pseudo code for the ray-half space:

```
If $\vec{v} \cdot \vec{n} < 0$: (ray towards the back of the plane)
  If $(P - C) \cdot \vec{n} > 0$: (ray starts in front)
       $t_i = -\frac{\overrightarrow{CP} \cdot \vec{n}}{\vec{v} \cdot \vec{n}}$
       $I = [t_i, \infty)$
  Else:              (ray starts behind)
       $I = [0, \infty)$

Else if $\vec{v} \cdot \vec{n} > 0$: (ray towards the front of the plane)
  If $(P - C) \cdot \vec{n} < 0$: (ray starts behind)
       $t_i = -\frac{\overrightarrow{CP} \cdot \vec{n}}{\vec{v} \cdot \vec{n}}$
       $I = [0, t_i]$

  Else:              (ray starts in front)
       No intersection
       $I = \emptyset$
```

	Ray towards back	Ray towards front
Ray starts in front		
Ray starts in the back		

Now its possible to extrapolate this information to perform the ray-box intersection, since a box can be defined as the intersection of 6 half spaces.

$$\begin{cases} C = corner\ of\ the\ box \\ \vec{w} = width\ vector \\ \vec{h} = height\ vector \\ \vec{l} = length\ vector \end{cases}$$



The 6 planes:

	Point	Normal
Front (α_1)	$C_1 = C$	$\vec{n_1} = \vec{l} \times \vec{h}$
Back (α_2)	$C_2 = C + \vec{w}$	$\vec{n_2} = -\vec{n_1} = \vec{h} \times \vec{l}$
Left (α_3)	$C_3 = C$	$\vec{n_3} = \vec{h} \times \vec{w}$
Right (α_4)	$C_4 = C + \vec{l}$	$\vec{n_4} = -\vec{n_3} = \vec{w} \times \vec{h}$
Bottom (α_5)	$C_5 = C$	$\vec{n_5} = \vec{w} \times \vec{l}$
Top (α_6)	$C_6 = C + \vec{h}$	$\vec{n_6} = -\vec{n_5} = \vec{l} \times \vec{w}$

The final pseudo code for the ray-box intersection would be the following:

```
J = [0,∞)
i = 1
while i ≤ 6 and $J \neq \emptyset$:
        $I = R \cap H(\alpha_i)$
        $J = J \cap I$
        $i = i + 1$
```

Intersection of intervals:

$$[a,b] \cap [c,d] = [\max(a,c), \min(b,d)]$$
$$when\ I = [a,b]\ and\ b < a \rightarrow I = \emptyset$$

Interpretation of the resultant interval $J = [t_{min}, t_{max}]$ when $J \neq \emptyset$:

```
If  $t_{min} = 0$: (ray starts inside de box)
        $P_i = P + t_{max}\,\vec{v}$

Else :
        $P_i = P + t_{min}\,\vec{v}$
```

2.3 Ray- polygon

Triangle:

Data for a triangle:

$$\text{Triangle: }\begin{cases} V_1 \\ V_2 \\ V_3 \end{cases}$$

Or:

$$\text{Triangle: }\begin{cases} C = V_1 \\ \vec{a} = V_2 - V_1 \\ \vec{b} = V_3 - V_1 \end{cases}$$

Where:

$$P = C + \alpha\,\vec{a} + \beta\,\vec{b} \quad with \begin{cases} \alpha \geq 0 \\ \beta \geq 0 \\ \alpha + \beta \leq 1 \end{cases} \; if\ inside\ the\ triangle$$

This triangle representation format is named affine coordinates.

$$P = O + \lambda_1\overrightarrow{v_1} + \lambda_2\overrightarrow{v_2} + \cdots + \lambda_n\overrightarrow{v_n}$$
$$\lambda_i \rightarrow affine\ coordinate$$

Ray-Triangle:

Equation of triangle:

$$P = C + \alpha\,\vec{a} + \beta\,\vec{b}$$
$$\alpha\,\vec{a} + \beta\,\vec{b} = P - C$$

Both sides dotted with $\vec{a}$

$$(\alpha\,\vec{a} + \beta\,\vec{b}) \cdot \vec{a} = (P - C) \cdot \vec{a}$$
$$\alpha\,(\vec{a} \cdot \vec{a}) + \beta\,(\vec{b} \cdot \vec{a}) = (P - C) \cdot \vec{a}$$

Both sides dotted with $\vec{b}$

$$(\alpha\,\vec{a} + \beta\,\vec{b}) \cdot \vec{b} = (P - C) \cdot \vec{b}$$
$$\alpha\,(\vec{a} \cdot \vec{b}) + \beta\,(\vec{b} \cdot \vec{b}) = (P - C) \cdot \vec{b}$$

$$\begin{pmatrix} (\vec{a} \cdot \vec{a}) & (\vec{b} \cdot \vec{a}) \\ (\vec{a} \cdot \vec{b}) & (\vec{b} \cdot \vec{b}) \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \end{pmatrix} = \begin{pmatrix} (P - C) \cdot \vec{a} \\ (P - C) \cdot \vec{b} \end{pmatrix}$$

$$M\begin{pmatrix}\alpha\\\beta\end{pmatrix} = \begin{pmatrix}(P-C)\cdot\vec{a}\\(P-C)\cdot\vec{b}\end{pmatrix}$$

$$\begin{pmatrix}\alpha\\\beta\end{pmatrix} = M^{-1}\begin{pmatrix}(P-C)\cdot\vec{a}\\(P-C)\cdot\vec{b}\end{pmatrix}$$

$$M^{-1} = \frac{1}{\det M}adjoint(M)$$

$$adjoint(A) = cofactor(A)^T$$

$$cofactor\begin{pmatrix}a & b\\c & d\end{pmatrix} = \begin{pmatrix}d & -c\\-b & a\end{pmatrix}$$

$$M^{-1} = \frac{1}{(\vec{a}\cdot\vec{a})(\vec{b}\cdot\vec{b}) - (\vec{b}\cdot\vec{a})(\vec{a}\cdot\vec{b})}\begin{pmatrix}(\vec{b}\cdot\vec{b}) & -(\vec{b}\cdot\vec{a})\\-(\vec{a}\cdot\vec{b}) & (\vec{a}\cdot\vec{a})\end{pmatrix}$$

$$\begin{pmatrix}\alpha\\\beta\end{pmatrix} = \frac{1}{(\vec{a}\cdot\vec{a})(\vec{b}\cdot\vec{b}) - (\vec{b}\cdot\vec{a})(\vec{a}\cdot\vec{b})}\begin{pmatrix}(\vec{b}\cdot\vec{b}) & -(\vec{b}\cdot\vec{a})\\-(\vec{a}\cdot\vec{b}) & (\vec{a}\cdot\vec{a})\end{pmatrix}\begin{pmatrix}(P-C)\cdot\vec{a}\\(P-C)\cdot\vec{b}\end{pmatrix}$$

Pseudocode:

```
Create plane with triangle data: $\vec{n} = \vec{a} \times \vec{b}$
Intersect plane with ray
If no intersection with plane:
    $No\ intersection$
Else:
    $P_i = P + t_i\vec{v}$
    $\begin{pmatrix}\alpha\\\beta\end{pmatrix} = M^{-1}\begin{pmatrix}(P_i-C)\cdot\vec{a}\\(P_i-C)\cdot\vec{b}\end{pmatrix}$
    If $\alpha \geq 0$ and $\beta \geq 0$ and $\alpha + \beta \leq 1$:
        $Intersection\ at\ P_i$
    Else:
        $No\ intersection$
```

To compute the ray-polygon, the polygon is divided into triangles by creating a triangle fan and perform ray-triangle in each of them. Note that this type of triangulation only works for convex polygons.

3 First milestone implementation

The ray tracer in the first milestone is almost the simplest ray tracing rendering that can be done: a single ray is casted for each pixel in the image, and it retrieves the color of the closest object it intersected. The surface normal of the intersection point is also computed, and the possibility of rendering its vector values as RGB color is provided.

Functionalities developed:

- Added support for sphere objects.
- Scene is determined by a .txt scene file.
- The resolution of the image is configurable through command arguments.
- The generated image can be exported as a .bmp file.

Implementing these features was straightforward, and time was also spent on defining how the renderer's structure should be (which, looking in retrospective, was a good idea).

4 Global illumination

The idea is to make casted rays bounce off objects they collide with, so that the retrieved color is a combination of all the surfaces' color it collided with. This model mimics the way photons work in real life: when a photon reaches the surface of an object, it is absorbed by a molecule. Some electrons in this molecule get stimulated and go into higher energy orbitals, and then quickly move back to their original orbitals, emitting new electrons in the process.

In the real world, light is emitted from light sources, gets affected by the environment, and finally reaches our eyes for us to see. Since the objective is to mimic real physics to render our scene, it would make sense to cast rays from light sources in the scene, make them bounce off objects and render the rays that reach the camera. The problem with this approach is that most rays would be irrelevant, since only a very small number of rays would collide with the camera, and we would also need to make the camera have a volume to collide with.

A better approach consists of, instead of casting rays from the light source and render the rays that collide with the camera, we cast the rays from the camera and render the rays that collide with the light source.

The direction of the new, bounced ray depends on the surface normal at the collision position, and the material of the collided object.

5 Lambertian material

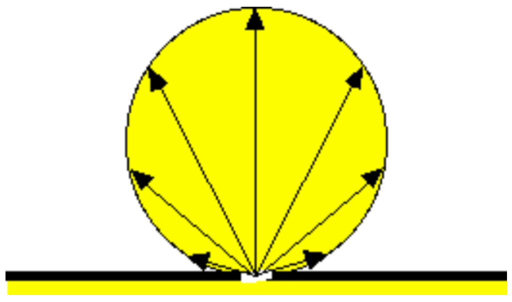


Figure 2 Reflection vectors using Lambertian distribution [4]

Lambertian materials' ray bounces follow the Lambert's cosine law, that says that luminous intensity is directly proportional to the cosine of the angle between the normal of the surface and the incident ray. To mimic this behaviour, rays bounced from objects have their new direction be similar to the surface's normal, as if a higher number of rays in a certain general direction imitates luminous intensity.

Tracing one ray per pixel only determines one possible path that ray may have followed, so to make the final render more accurate, multiple pixel samples may be averaged by tracing multiple rays per pixel.

As for the Lambertian material implementation for this project, the taken approach is to generate a unit sphere tangent to the ray's point of intersection with the surface, taking a random point from the surface of this sphere, and defining the bounced ray's origin as the point of intersection (we move it a tiny bit in the surface normal's direction in order to avoid colliding with the same object, due to floating point error while computing the ray's time of intersection) and the normalized vector from the intersection to the random sphere point as the direction.

6 Second milestone implementation

Functionalities developed:

- Added support for box objects.
- Added support for Lambertian and light materials.
- Global illumination.
- Adjustable number of ray bounces and samples per pixel through a config file.

A problem occurred when working on this milestone, that was that when computing a new bounced ray, it would intersect with the object it originated from. This was because when computing the time of intersection of the ray to get the intersection point, a small error was introduced due to a floating point error. The solution for this was to offset the new ray's origin by a small scalar times the surface normal.

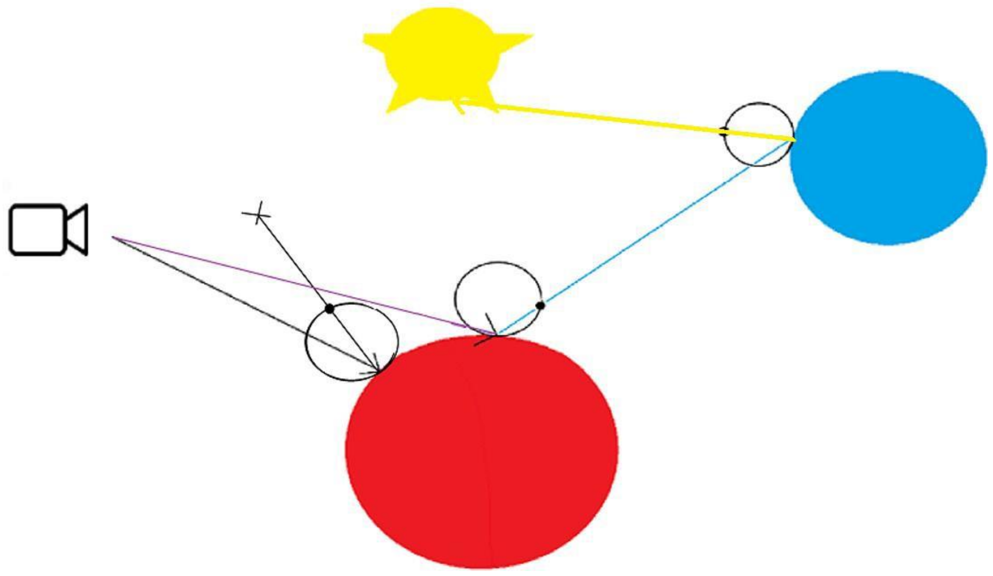


Figure 3 Milestone 2´s global illumination model

6 Metallic material

When an object has a metallic material, rays bounce off them following the reflection vector:

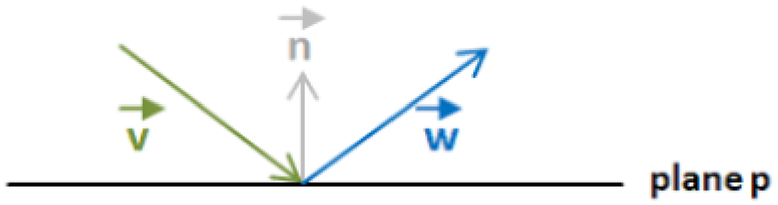


Figure 4 Mirror reflection. V is the incident vector, N the normal and W the reflected vector[5]

Where the incident ray's angle respect to the normal is the same as the bounced' s one.

But this approach yields only mirror-like reflections, in order to achieve a rougher look, we must alter the reflected ray's direction. In this project, the approach consists of taking a random point within the volume of a sphere positioned at the end of the reflected ray. The radius of this sphere determines the amount of roughness.

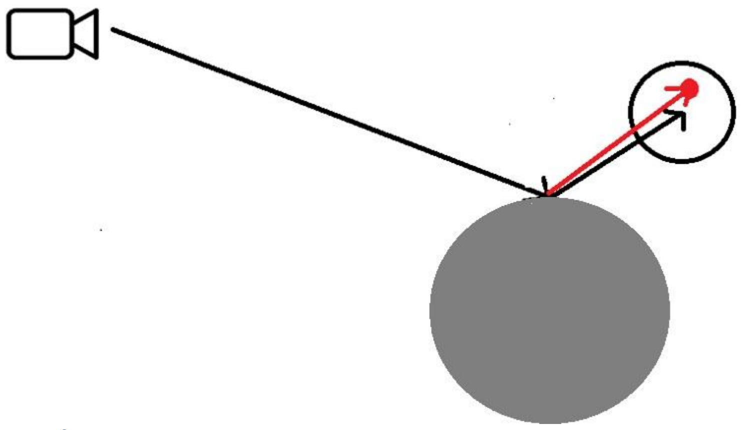


Figure 5 Reflection using the roughness sphere

7 Antialiasing

Antialiasing in the context of graphics consists of blending rendered objects' edges with their background. The approach for this project is to apply a small random offset to the ray's direction when casting rays against the scene, so that when multiple samples are taken for a pixel that is in an object's edge, some of the rays will intersect the object and some not, so the final averaged pixel color would be a blend between the object's edge and it's background. [1]

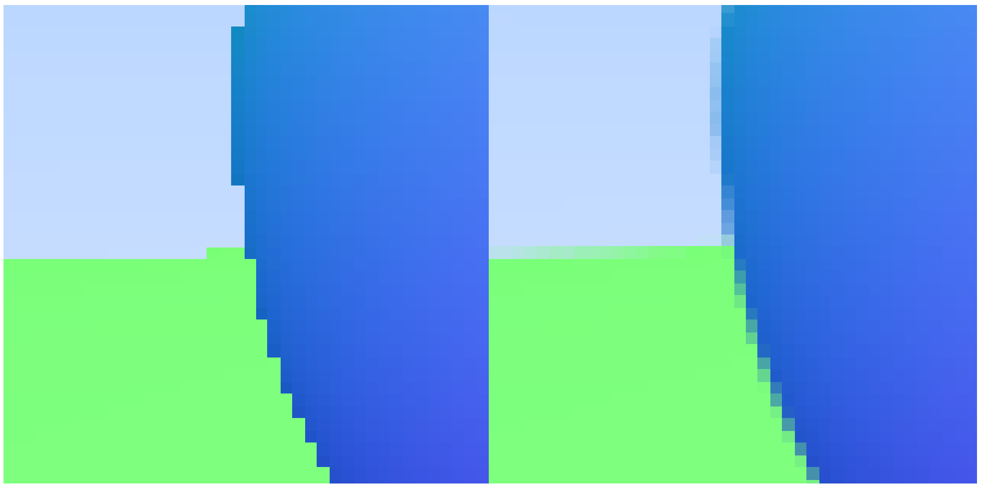


Figure 6 Antialiasing in action [1]

8 Third milestone implementation

Functionalities developed:

- Ray-mesh is performed by iterating through all the mesh's triangles and checking for intersection for each of them.
- Added support for mesh and convex polygon objects.
- Models can be loaded through .obj files.
- Added support for metal materials, with roughness configurable through the scene file's material data.
- Antialiasing.

9 Dielectric materials

Objects with this material both reflect and refract light, meaning that some of the light is reflected, and some is transmitted. Snell's law is used for determining the direction of the transmitted and reflected rays, and Fresnel's Equations gives the fraction of how much of the light reflected or refracted.

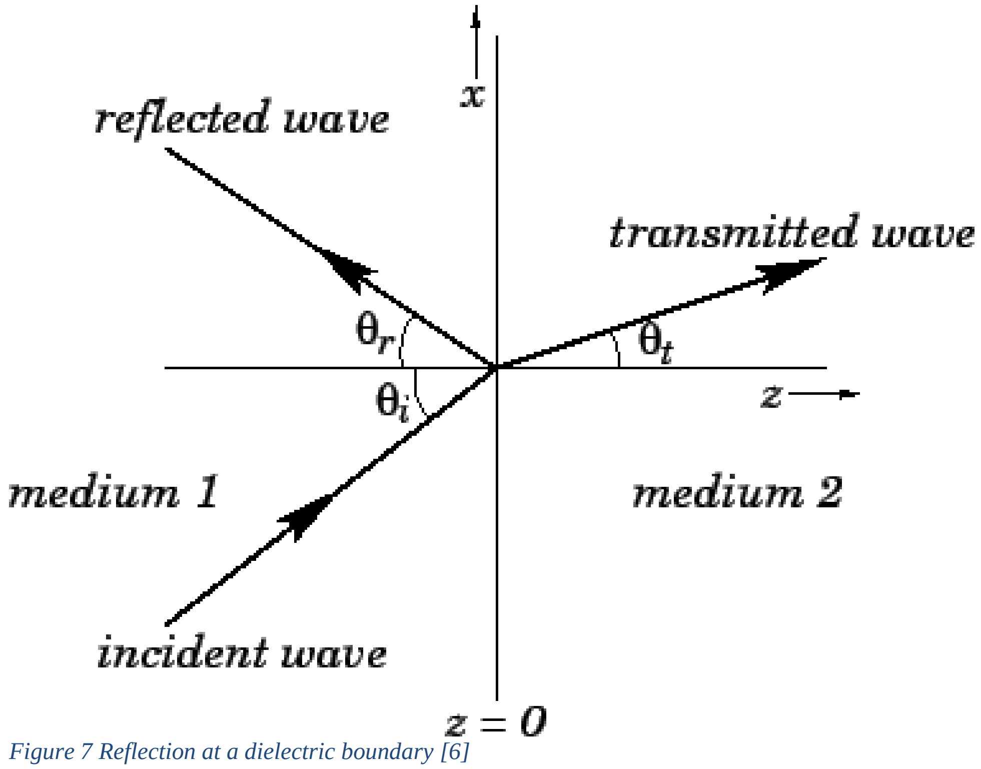


Figure 7 Reflection at a dielectric boundary [6]

Snell's Law:

$$n_i \sin \theta_i = n_t \sin \theta_t$$

Fresnel Equations:

Parallel to the plane of incidence:

$$\frac{E_r^{\parallel}}{E_i^{\parallel}} = \frac{\frac{\mu_i}{\mu_t}\cos\theta_i - \frac{n_i}{n_t}\sqrt{1-\left(\frac{n_i}{n_t}\right)^2(1-(\cos\theta_i)^2)}}{\frac{\mu_i}{\mu_t}\cos\theta_i + \frac{n_i}{n_t}\sqrt{1-\left(\frac{n_i}{n_t}\right)^2(1-(\cos\theta_i)^2)}}$$

Where:

- $E_r^{\parallel}$ = amplitude of the **reflected** light wave's **parallel** electric field
- $E_i^{\parallel}$ = amplitude of the **incident** light wave's **parallel** electric field
- θ_i = angle of incidence
- n_i and n_t = indices of refraction of incident and transmitted materials
- μ_i and μ_t = magnetic permeability of incident and transmitted materials

Perpendicular to the plane of incidence:

$$\frac{E_r^{\perp}}{E_i^{\perp}} = \frac{\frac{n_i}{n_t}\cos\theta_i - \frac{\mu_i}{\mu_t}\sqrt{1-\left(\frac{n_i}{n_t}\right)^2(1-(\cos\theta_i)^2)}}{\frac{n_i}{n_t}\cos\theta_i + \frac{\mu_i}{\mu_t}\sqrt{1-\left(\frac{n_i}{n_t}\right)^2(1-(\cos\theta_i)^2)}}$$

Where:

- $E_r^{\perp}$ = amplitude of the **reflected** light wave's **perpendicular** electric field
- $E_i^{\perp}$ = amplitude of the **incident** light wave's **perpendicular** electric field
- θ_i = angle of incidence
- n_i and n_t = indices of refraction of incident and transmitted materials
- μ_i and μ_t = magnetic permeability of incident and transmitted materials

The reflection coefficient R = $\frac{1}{2}\left\{\left(\frac{E_r^{\perp}}{E_i^{\perp}}\right)^2 + \left(\frac{E_r^{\parallel}}{E_i^{\parallel}}\right)^2\right\}$

By the law of conservation of energy, the transmission coefficient T = 1 - R

If we are to implement this, we would have to create a new bounced ray for the reflected fraction of light and another for the refracted one. But creating multiple rays per collision results in exponential growth, and this is highly undesirable, because it would be a great penalty in performance.

The approach for this project is to use statistical ray tracing, meaning that whenever we intersect a dielectric material, we randomly create a reflected or transmitted ray, depending on the coefficients for both, with the higher coefficient being more likely to be the type of ray to be generated.

```
RayTrace(ray r, int d)
        Check d and return if necessary
        o = FindNearestObject(r)
        R = ReflectionCoeffcient(o,r)
        T = TransmissionCoeffcient(o,r)

        x = Random(0,1)
        If $x < 1 - T$ then
                r ← Reflection(o,r)
                C ← C + RayTrace(r,d-1)
        Else
                r ← Transmission(o,r)
                C ← C + RayTrace(r,d-1)
        Endif

        return C
```

Rays that traverse through the inside of dielectric materials also experience color attenuation, that is computed as follows:

$$I_t = I_0 * A^t$$

Where:
I_t: The energy after traversing distance t
I_0: Initial energy of the light
A: Attenuation coefficient
t: Distance traversed in the medium

10 Kd-tree accelerator

A Kd-tree is a type of binary space partitioning, that can be used in ray tracing to reduce the number of ray queries for triangle meshes.

First, we start with a bounding volume covering all the meshes' geometry, and then we recursively divide the space in two, choosing an axis and a splitting point. Each of these partitions are interior nodes, if they possess other child nodes, or leaf nodes, which contain the triangles that are within the node's volume.

The axis of split is chosen depending on the current recursive depth, and the splitting point is chosen using a cost function:

$$c(A,B) = c_t + p_A \sum_{i=1}^{N_A} c_i a_i + p_B \sum_{i=1}^{N_B} c_i b_i$$

c_t is the traversal cost (user input).

c_i is the intersection cost (user input).

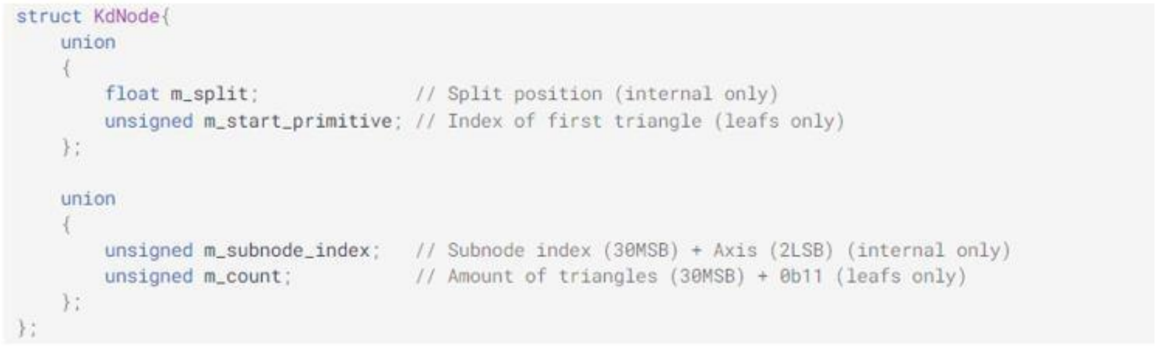
N_A is the amount of objects on the left.

N_B is the amount of objects on the right.

$$p_A = \frac{S_A}{S_{PARENT}} \qquad p_B = \frac{S_B}{S_{PARENT}}$$

To make a ray query against a Kd-tree, we check for the intersection of the ray and the kd- tree's bounding volume, and compute the minimum and maximum values for t. We then compute the time of intersection with the splitting plane and use these three time values to explore the Kd-tree. When we reach a leaf node, all the triangles inside it are checked for intersection.

For the Kd-tree container, instead of using the usual approach of having all the data and two child pointers inside each node, we can make an optimization by having a primitives and a nodes array, with each node having this structure:



```
struct KdNode{
    union
    {
        float m_split;              // Split position (internal only)
        unsigned m_start_primitive; // Index of first triangle (leafs only)
    };

    union
    {
        unsigned m_subnode_index;   // Subnode index (30MSB) + Axis (2LSB) (internal only)
        unsigned m_count;           // Amount of triangles (30MSB) + 0b11 (leafs only)
    };
};
```

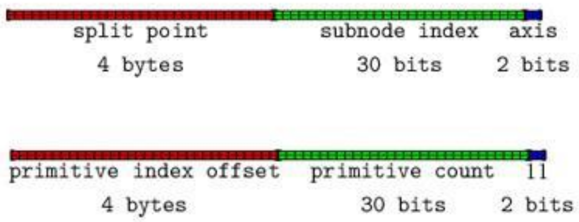


Figure 8 Kd-tree node structure [7]

11 Fourth milestone implementation

Functionalities developed:

- Added support for dielectric materials.
- Acceleration of the process for ray querying for meshes using a Kd-tree.

12 CUDA

CUDA is a GPGPU programming API, that allows the programmer to exploit the computing power of the GPU while using C++ syntax. It is widely used in fields such as machine learning, physics engines, cryptography, medicine…etc.

CUDA code is divided into host (run on the CPU) and device (run on the GPU), device functions are called kernels. There are three function identifier to let the compiler know how and from where they should run:

- __host : runs on CPU, callable only from host code.
- __device : runs on GPU, callable only from device code.
- __global : runs on GPU, callable from both host and device code.

Global functions are called with the following syntax:

```
add<<<1, 1>>>(N, x, y);
```

Where values between the triple less-than and greater-than determine the number of threads assigned to run that function. The first value is the number of thread blocks, and the second one the thread count per block.

12.1 Thread hierarchy

Threads are grouped into thread blocks, which are at the same time grouped into a single kernel grid. Threads and thread blocks can be identified using built-in variables to get their indices in the block and in the grid, and these indices can be of up to three dimensions.

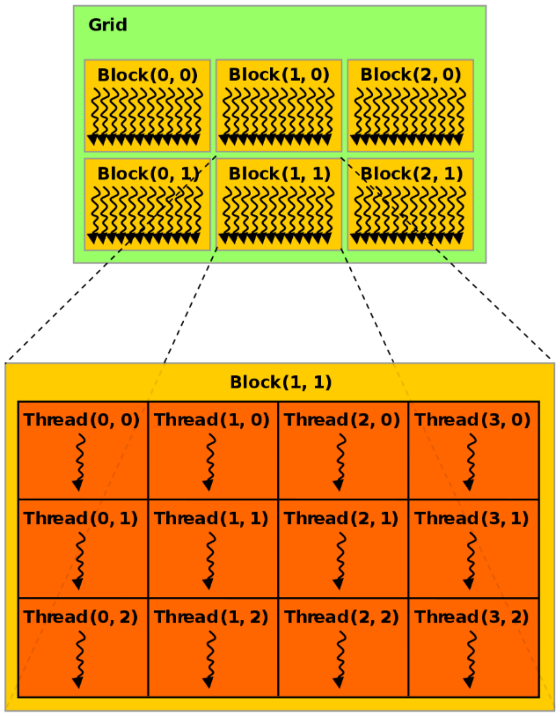


Figure 9 Thread hierarchy on the GPU [8]

12.2 Threads from a hardware perspective

GPUs contain multiple Streaming Multiprocessors (SM) in them, and these are in charge of executing the thread blocks assigned to them. The block count per SM depends on the requested blocks and the number of SMs in a GPU.

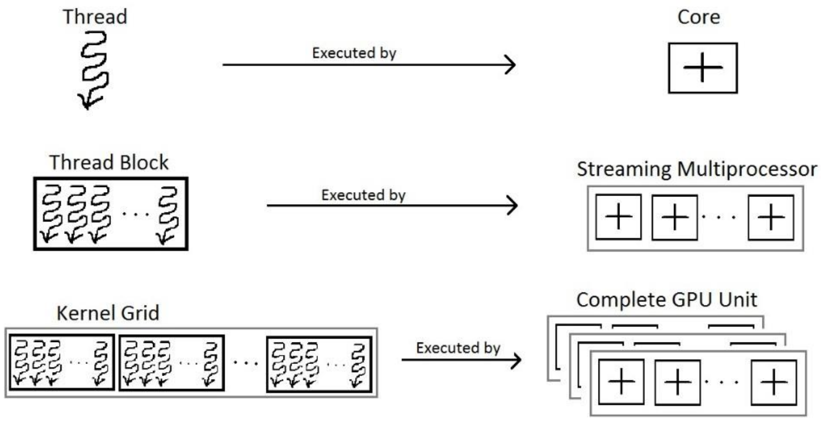


Figure 10 Hardware vs. software threads [9]

12.3 Streaming multiprocessors

SMs are general purpose processors with low clock rate and small cache. They support instruction-level parallelism but not branch prediction. They are composed of:

- Execution cores
    - Execution cores
    - Special Function Units (cos, sin, 1/sqrt, log2…etc)
- Caches and shared memory
- Thousands of registers
- Schedulers for warps (in charge of issuing instructions to warps)

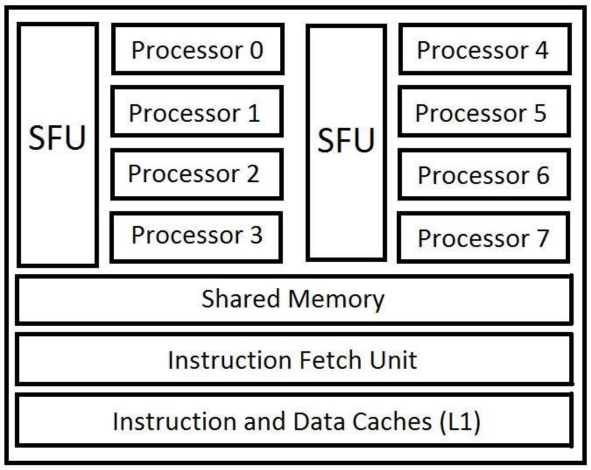


Figure 11 Streaming multiprocessor and its resources [9]

When the SM executes a thread block, all the threads inside the block are executed at the same time. SMs may contain up to eight thread blocks at the same time.

To free the memory of a thread block, all the threads within it must have finished ITS execution, so threads within a block should take about the same time to finish, to avoid inactivity.

Threads inside a thread block are grouped in thread warps of 32 threads in size.

12.4 Thread warps

Thread warps use SIMT (Single Instruction Multiple Thread) model. This model is similar to Flynn's taxonomy SIMD (Single Instruction Multiple Data), but it has the advantage of being able to perform an instruction in arbitrary data (since we are executing the instruction in different threads), instead of needing to have all data contiguous in memory.

12.5 Memory hierarchy

To access memory in kernel functions, we must first allocate it in the GPU RAM (DRAM), but memory allocated in the DRAM must be copied onto RAM for the host to be able to access it, and vice versa.

There are four types of memory allocation in with CUI

- Pageable memory
- Pinned memory
- Mapped memory
- Unified memory
- GPU memory

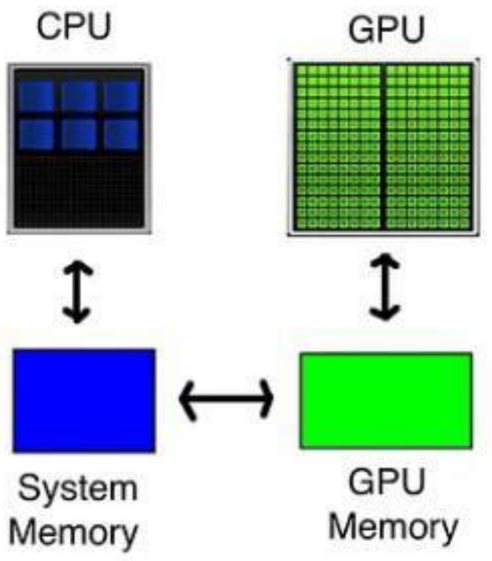


Figure 12 View without "Unified memory" [10]

12.5.1 Pageable and pinned memory

The memory allocated by the host is by default pageable. To use this data in the GPU we must make a transfer, but CUDA implicitly copies this memory into a temporary pinned memory buffer, and then transfers it to the device. We can avoid this by directly allocating pinned memory.

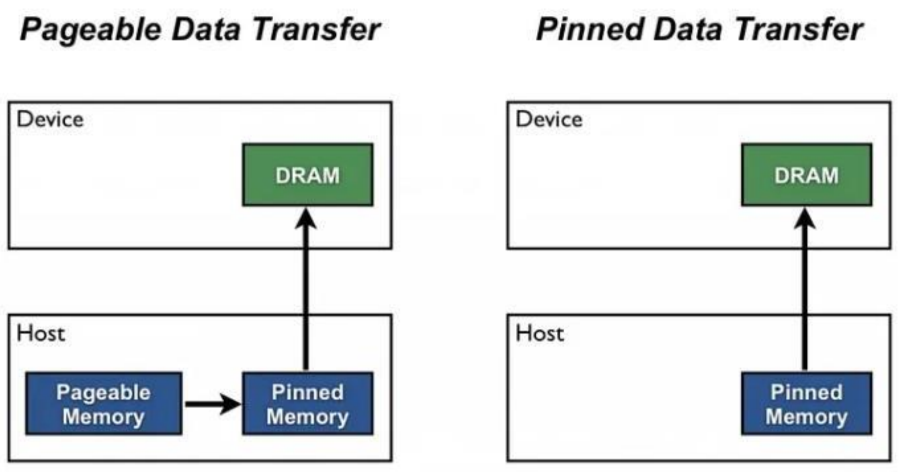


Figure 13 Pageable Data transefr vs. Pinned data transfer (source: https://devblogs.nvidia.com/how-optimize-data-transfers-cuda-cc/)

12.5.2 Mapped and unified memory

- Mapped memory is pinned memory that can be mapped into a device address. Data will not be copied into device memory, but transfers will happen during execution.

- Unified memory is a pool of memory which is accessible by both device and host, but data migration still happens under the hood.

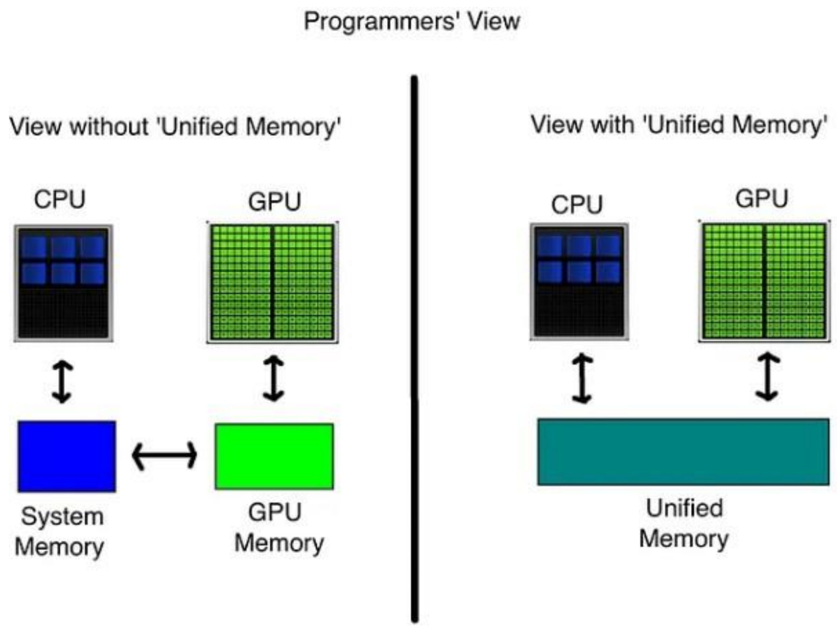


Figure 14 View with or without "Unified memoriy" [10]

12.6 Random numbers

Using a single random number generator for all the threads would be problematic, because since execution happens concurrently, many times the returned values would be the same for multiple threads.

For this reason, we must generate a single random number generator per each thread in a kernel. We can do this by using CUDA's cuRAND library.

13 Final milestone

For the final project, the idea was to remake all the previous milestones, but using CUDA to accelerate the rendering process.

Mapped memory was used for the frame buffer, since after testing different types of memory allocation, this was the one that yielded the best results.

All scene objects are allocated on DRAM, so that accessing them while rendering is faster.

As for the rendering, the first approach was to create a render kernel function that assigns a single thread for each pixel, with thread blocks of size 8x8, and cast a ray for each pixel sample. But a problem occurred when meshes were added to the scene: since the number of polygon queries is very high, thread execution would cause thread timeouts when having more than ~20 samples per pixel.

The first solution to this was calling the render function multiple times, and divide the sample computation between the render calls. The problem with this is that we must wait until all the pixels in the screen are done rendering to execute the next kernel call, and since different pixels take very different amounts of time to complete computation, this would cause a lot of threads to be inactive, so I needed to find a better way.

The final approach is to divide the screen in multiple uniform squares and run the kernel in each of these squares. A thread block is assigned to every pixel in the square, and each of the pixels in the thread block is in charge of computing a single sample.

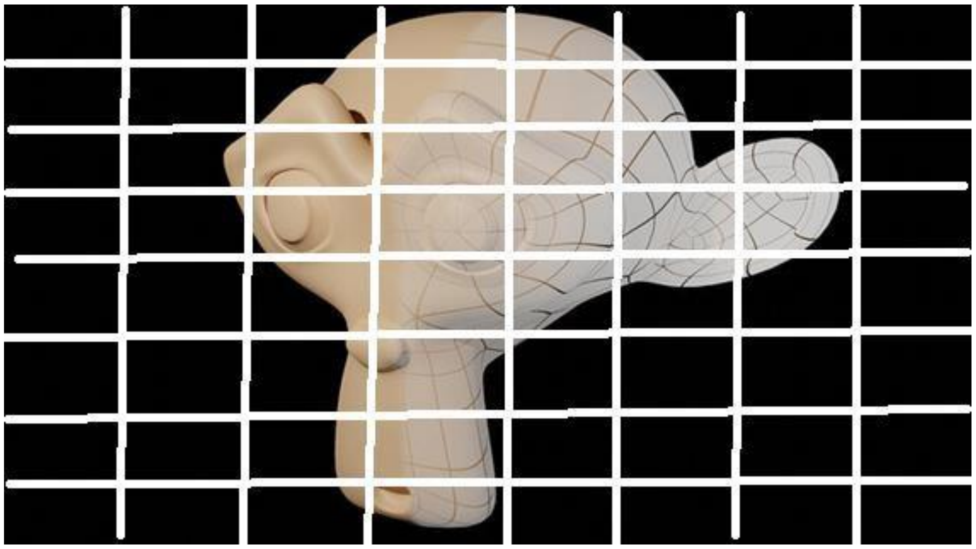


Figure 15 Example of division of the screen into squares of equal size

14 Final project results

Having 128 samples per pixel, 10 maximum bounces per ray, a screen of 1024x1024 in resolution, and running on a Geforce GTX 980 GPU, these are some comparisons of the single-threaded and the CUDA multi-threaded version of the ray tracer renderer:

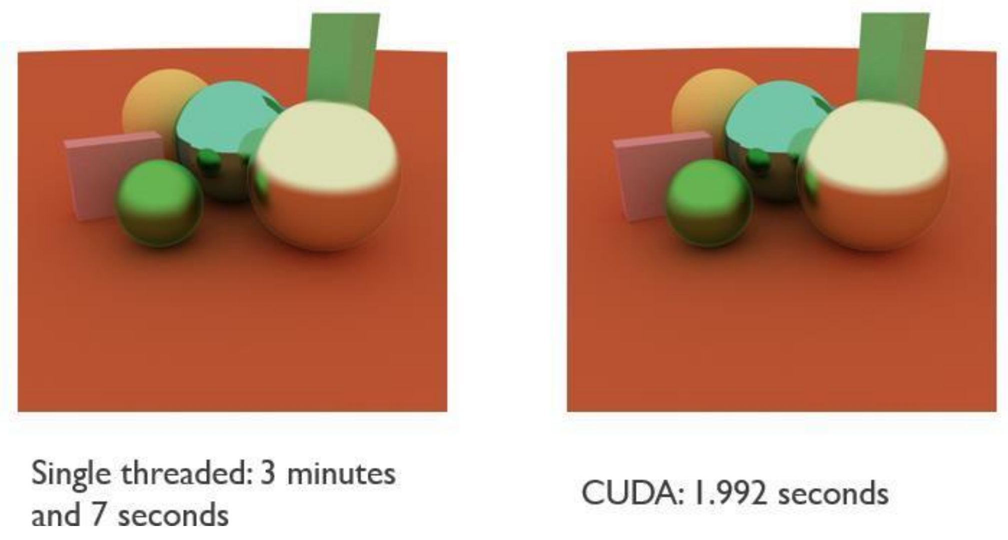


Figure 16 Single-thread vs. multiple thread – Test 1 X93 Speedup

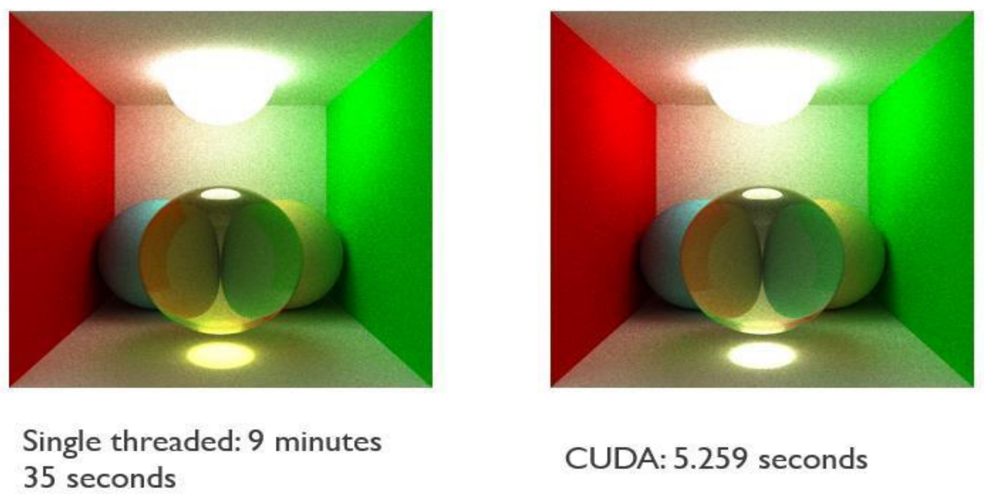


Figure 17 Single-thread vs. multiple thread – Test 2 X109 Speed

15 Conclusions

Using CUDA to make use of the GPU´s multi-threading capabilities for per-pixel operations in raytracing provides an improvement of orders of magnitude compared to a sequential, single-threaded approach, as seen in the performed tests, where the CUDA implementation took from 90 to 110 times less time to render.

16 Future work

The results could potentially improve even more in combination with some of the following actions:

- Improve bandwidth usage by performing parallel memory transfers
- Test different types of memory allocation
- Make the render kernels run in parallel
- Use shared memory to avoid cache misses
- Implement other raytracing techniques

17 References

[1] Shirley, P. (2018). Ray tracing in one weekend. Amazon Digital Services LLC, 1, 4.

[2] Allen, R. (2018). Accelerated Ray Tracing in One Weekend in CUDA, Nvidia Developer Technical Blog

[3] Pharr, M., Jakob, W., & Humphreys, G. (2023). Physically based rendering: From theory to implementation. MIT Press.

[4] Steven L. Jacques, Scott A. Prahl (1998). Lambertian pattern of escaping light. Oregon Graduate Institute

[5] Molkenthin, B. (2023). Vector reflection at a surface. Sunshine.

[6] Fitzpatrick, R. (2014). Classical Electromagnetism.

[7] Beldad, E. Cs350 course webpage notes. Digipen Institute of Technology.

[8] Molnar, B., Tolnai, G., Legrady, D., & Szieberth, M. (2018). Vectorized Monte Carlo for GUARDYAN–a GPU accelerated reactor dynamics code. In This Conference.

[9] Hweu, Wen-mei, Course material Heterogeneous Parallel Programming

[10] Dey, S. (2019). Efficient data input/output (i/o) for finite difference time domain (FDTD) computation on graphics processing unit.