Assembly accelerated Path Tracing algorithm

Final report

Team W3:
Jakub Leśniak
Gracjan Jeżewski
Radosław Rzeczkowski

Supervisor: Ph.D., Eng. Piotr Czekalski
17 December 2021

Table of Contents

Final report 1

Table of Contents 2

1. Algorithm description 3

1.1. Light transport theory 3

1.2. Intersection functions 3

1.3. Sphere intersection 4

1.4. Monte-Carlo simulation 4

2. Program description 5

2.1. Architecture 5

2.2. Rendering 5

2.3. Library 5

3. User Interface 6

3.1. Image rendering 6

3.2. User controls 7

4. Execution measurements 8

4.1. Measurements table 8

4.2. Data interpretation 9

5. Testing and Debugging 10

5.1. Debugging 10

5.2. Testing 10

6. Conclusions 11

1. Algorithm description

1.1. Light transport theory

Path Tracing is a light transport (ray tracing) algorithm following the rules of Monte Carlo simulation. The general approach, is to integrate illuminance arriving at a single point on the surface of an object, in the scene. What's important, is that integration of illuminance starts in the camera (point of view), and follows an inverse path of light ray, from the observer, to the light source. Such way of integration allows us to discard rays that wouldn't be able to reach the observer under any case.

1.2. Intersection functions

To find aforementioned points, custom intersection functions are required, allowing us to find a common solution for light ray and geometry. In case, where two intersection points are found (i.e ray intersecting sphere in the middle) intersection function returns the smallest of the two time values which refers to the closest hit point possible in the scene for a ray with specific origin and direction.

To determine the closest possible intersection in the scene, we are iterating through objects in the scene, and comparing results of each ray-geometry intersection to find the smallest possible time value.

Position of intersection can be calculated using ray function which is a linear function of time:

$$f(t) = Origin + (t \times Direction)$$

1.3. Sphere intersection

To make the task approachable, we have decided to focus on one custom intersection function, which is a ray-sphere intersection function.

Sphere equation, with center $C = (a, b, c)$ and radius r:

$$(x - a)^2 + (y - b)^2 + (z - c)^2 = r^2$$

Above equation can be reinterpreted and rewritten using a dot product of two vectors , where P is a point on the sphere:

$$(P - C) \cdot (P - C) = r^2$$

If we consider, that point P can be represented by a linear function of time, we obtain:

$$((Origin + t \times dimension) - C) \cdot ((Origin + t \times dimension) - C) = r^2$$

After expansion of the above we obtain a quadratic equation in a form of

$$at^2 + bt + c = 0 \text{ where:}$$

$$a = Direction \cdot Direction$$

$$b = 2 \times (Direction \cdot (Origin - C))$$

$$c = (Origin - C) \cdot (Origin - C) - r^2$$

To find the intersection means to solve the above quadratic equation, roots obtained (if any) represent the time of a collision between a ray and a sphere in a 3-dimensional space.

1.4. Monte-Carlo simulation

Averaging results of repetitions of the algorithm for one point, with ray direction randomisation, provides improved rendering quality (noise reduction) and physical accuracy of the image.

2. Program description

2.1. Architecture

We have implemented a very basic version of the algorithm in higher level language (C#), and accelerated sphere intersection function using Assembly procedures. Such implementation allows us to compare execution times of hybrid and monolingual version.

2.2. Rendering

Main rendering method contained within *Renderer* class performs light integration for every pixel of the viewport (imaginary viewable space). Along with resolution data and number of samples, a boolean flag is passed, which defines whether assembly or C# native method should be used for every intersection test.

To use assembly procedure, an Asm*Proxy* object is used. It contains a method, using *unsafe* external C# function call with DLLImport attribute. *Sphere* object contains a method with native high level implementation.

Assembly source code resembles the native C# implementation line by line. There are only a few small differences in the code, which are a result of optimisation of assembly procedure (optimisation of delta calculation).

2.3. Library

Project solution has a strictly defined order of object creation. Firstly, we are creating a dynamic link library from ASM source code in working directory of the project. Afterwards, C# source code is compiled and ready to be executed. Program executable can dynamically import the library in case, where user specifies that Assembly accelerated version of the algorithm should be performed.

3. User Interface

3.1. Image rendering

At the beginning of program execution, user is greeted with minimalistic and straightforward interface. It consists of two large square placeholders with different background colour, slider, two checkboxes with labels and a button marked with “Start” label.

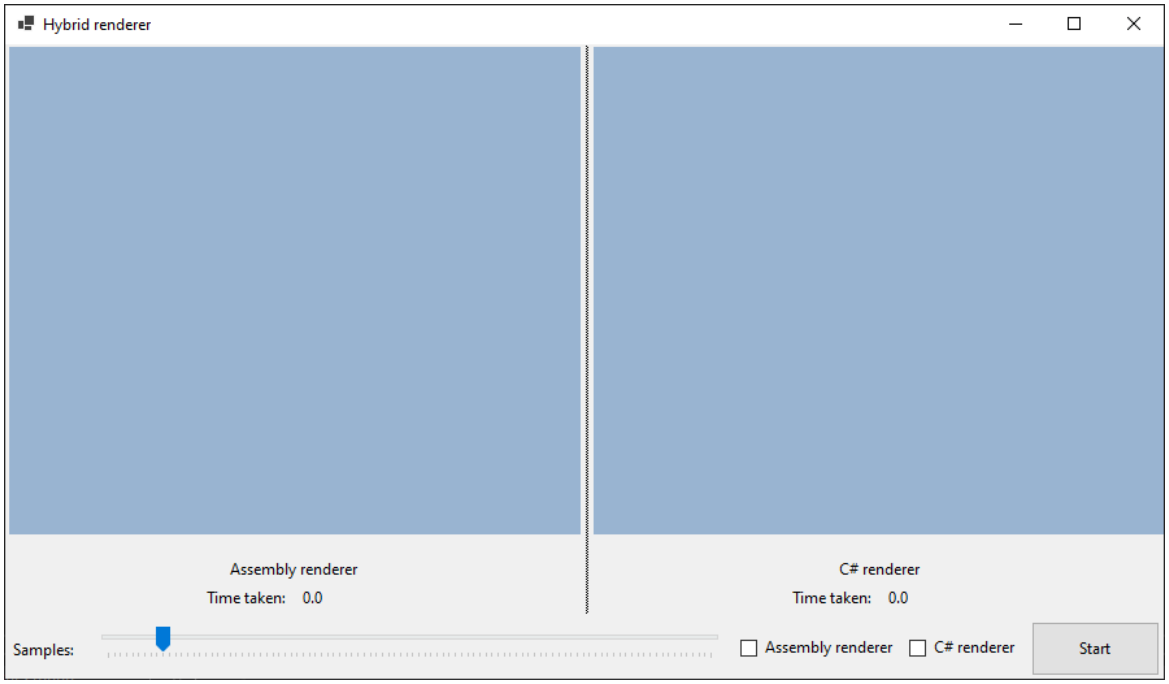


Fig. 1 First execution of the program.

Below each image placeholder, an appropriate text label is presented with time measurement below.

Program requires at least one checked checkbox to start. It is possible to start both versions of the implementation at the same time, program performs rendering in a specific order:

- C# Native implementation,
- Assembly accelerated implementation,

Such serial order of execution allows us to sensibly measure time.

3.2.User controls

User is able to specify modes of execution, and most importantly, number of samples for the algorithm. Slider values are contained within a range of $[1,100]$. Such range provides reasonable time execution, and noticeable noise reduction for upper and lower limits of range.

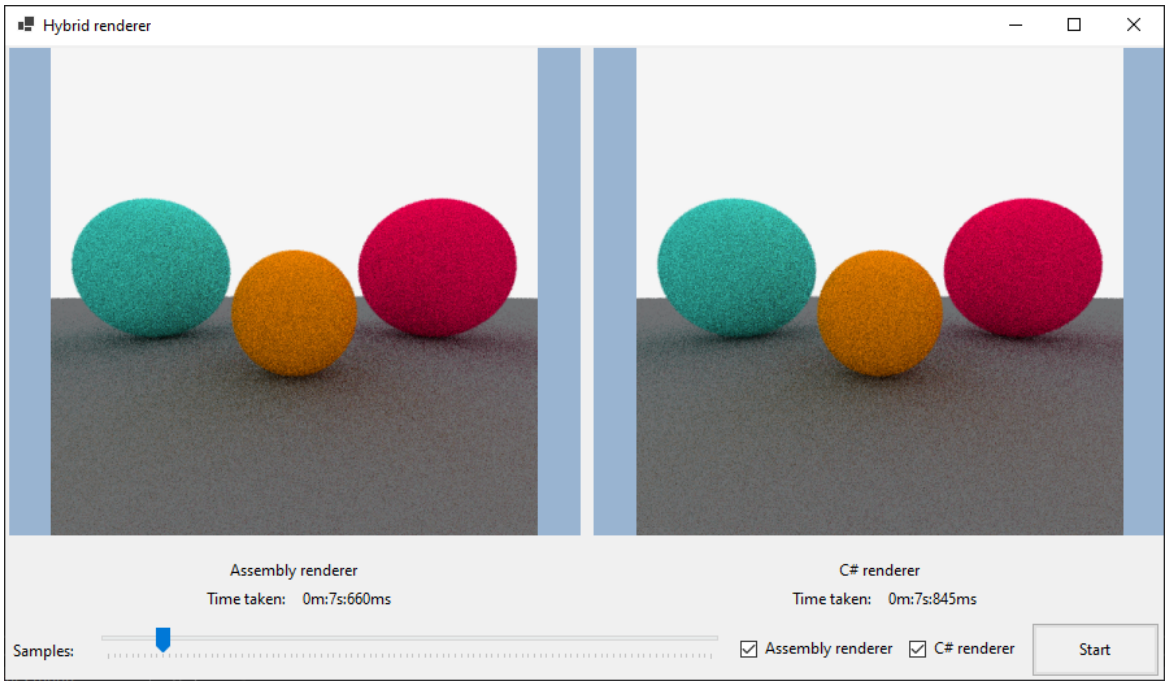


Fig. 2 User interface after execution of both implementations.

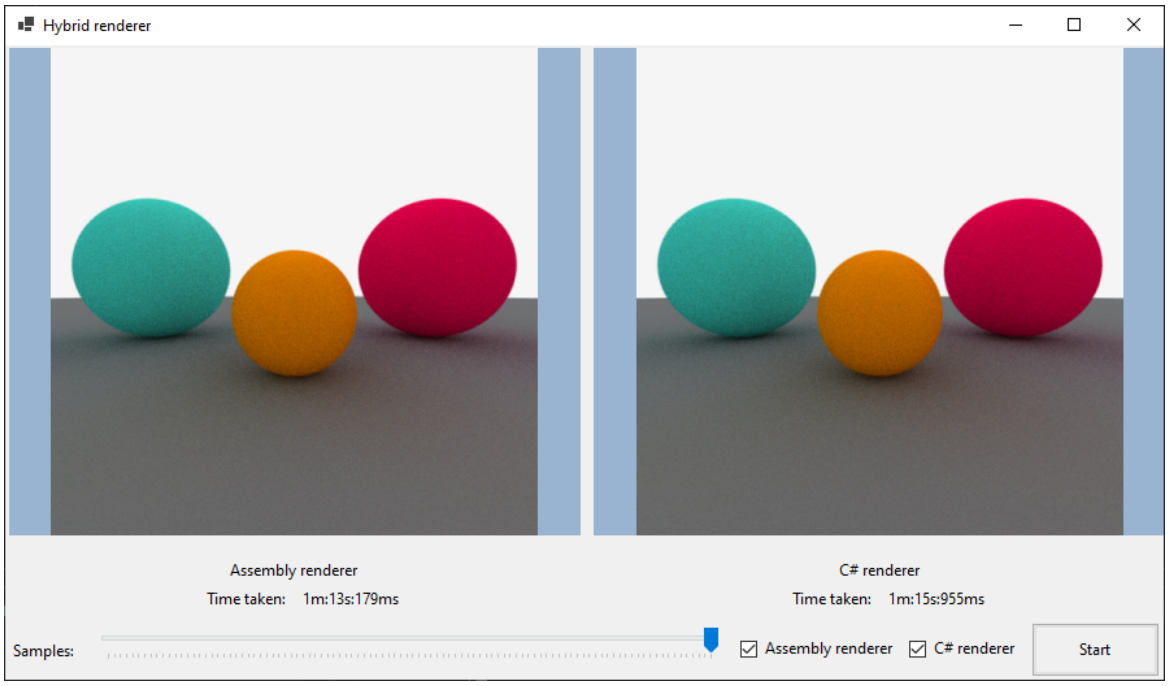


Fig. 3 Noise reduction for sample count equal 100.

4. Execution measurements

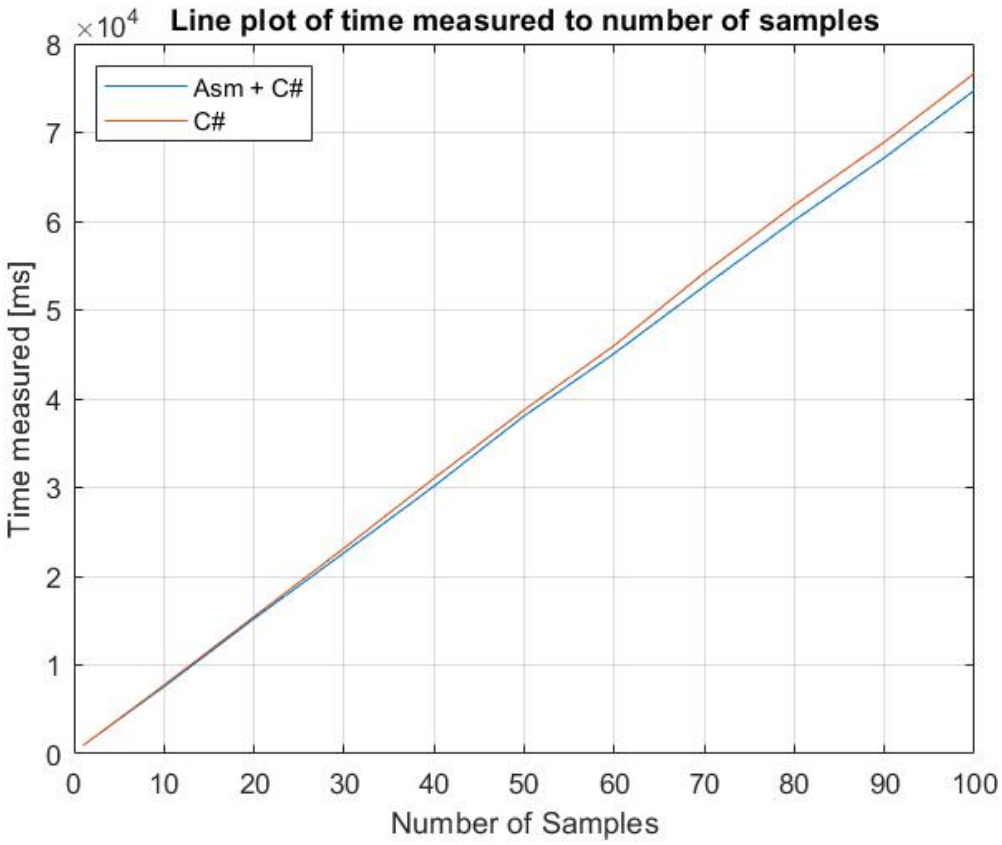
Nature of the algorithm makes it challenging to appropriately compare execution times. Rendering time depends mainly on ray bouncing count, which in turn depends highly on random number generation (to define next bounce direction). The only way to stop the recursion, is to reach the *sky* (outer space, no intersections) or exceed ray bounce limit of 50 bounces.

However, from our testing, it seems that in almost all cases, the execution time of assembly accelerated procedure appears to have an improvement of around 3% compared with native high level implementation.

4.1. Measurements table

Sample count	Hybrid execution time	Monolingual execution time
1	0m:0s:866ms	0m:0s:901ms
10	0m:7s:571ms	0m:7s:727ms
20	0m:15s:218ms	0m:15s:447ms
30	0m:22s:653ms	0m:23s:168ms
40	0m:30s:162ms	0m:31s:43ms
50	0m:38s:62ms	0m:38s:740ms
60	0m:45s:100ms	0m:46s:00ms
70	0m:52s:692ms	0m:54s:187ms
80	1m:0s:101ms	1m:1s:807ms
90	1m:7s:153ms	1m:8s:915ms
100	1m:14s:747ms	1m:16s:669ms

4.2. Data interpretation



As seen above, improvement in rendering times grows linearly with number of samples. Bigger number of samples can accumulate slight time improvement and provide faster execution. Each sample performs identical action (complexity of calculations **does not** grow with sample count) meaning, that we can expect this trend to follow for much larger sample count.

It is worth noting, that hybrid solution performs additional action of loading dynamic-link library. From our testing, it seems that DLL file is loaded once - the first time unsafe proxy method is used.

5. Testing and Debugging

5.1. Debugging

To perform debugging of our assembly (and C#) source, we have used excellent tooling available in Visual Studio IDE. This environment allowed us to observe registers, memory and individual variables in a way, which made the process very straightforward.

To reduce the complexity of the results (large range of values) we have specified a small subset of testing input values for our debugging environment, and observed how algorithm is performed. Running both, the high level (reference) and lower level versions interchangeably, allowed us to match the results and pinpoint the exact location of each calculation error, where results differed.

The most common issues, were:

- Data alignment issues for memory variables,
- Usage of incorrect instructions:
  - Vector instead of scalar instructions and the opposite,
  - Incorrect interpretation of instruction documentation,

5.2.Testing

Apart from synthetic tests for edge cases, our implementation is tested with a very broad range of input variables by design. What is more, because of the visual nature of the implementation, interpretation of results is very straightforward. Visual artefacts are immediately visible and have a huge impact on the resulting image. For an image of 500x500 pixels with just 25 samples, average ray bounce count of 5 and 4 objects in the scene, we are executing our function approximately 125 000 000 times.

6. Conclusions

Taking into account the time it took to implement the procedure in a low-level language, and the resulting improvement in execution time, such an operation turns out to be moderately profitable.

However, the sole act of implementing our procedure using SIMD parallel processing, gave us an invaluable insight that may allow us to create more vectorizable code in a high-level language, ultimately making it easier for the compiler to optimize.

Topic of the project turned out to be a relatively appropriate choice. Visual output of the algorithm made it easier to thoroughly test the implementation together with edge cases, and to judge whether results for a large dataset are consistent with the reference.

Dependence on vector operations enabled the logical and legitimate use of parallel processing, which allowed us to achieve similar levels of performance compared with monolingual solution.