Scanning Simulation Speed Improvement in Robotic Nuclear Decommissioning System

Wonmook Jeong*, Hyoseok Lim, Sungmoon Joo, Ikjune Kim and Jonghawn Lee
Korea Atomic Energy Research Institute, Daedeok-daero 989-111, Yuseong-gu, Daejeon, Korea
**Corresponding author: jwonmook@kaeri.re.kr*

1. Introduction

There are two ways to obtain point cloud data: using a physical scanner or using a scan model in a virtual environment. This paper presents an efficient method for a scanning simulation method using C++ or python. Simulations generate synthetic point cloud data used to train deep learning models for classifying reactor parts in robotic nuclear decommissioning system. The purpose of this study is to select an optimal algorithm for scanning simulation.

This paper is organized as follows. Section 2 introduces how our scan simulator works. Section 2.2 shows a comparison of the two methods of converting meshes to point clouds. Section 3 describes multiprocessing and Section 4 describes numerical experiment environment. Finally, Section 5 concludes the paper.

2. Simulator development

2.1 Simulator Overview

Simulators are built by applying a raycasting mechanism by default. As shown in **Fig. 1**, one of the intersections of the line segment connecting the two points(red point – scanner position, green point – gazing point) and the mesh (STL file) is converted into point cloud data by extracting the first intersection.

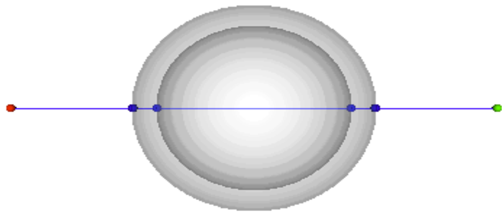

Fig. 1 Intersection point line to mesh

2.2 Implementation Method (C++, Python)

To achieve optimal performance, we have implemented the simulator in various methods and compared the performance in terms of simulation speed.

2.2.1 C++ (VCGlib)

The Visualization and Computer Graphics Library (VCG for short) is an open source portable C++ template library for manipulating, processing, and displaying using OpenGL of triangular and tetrahedral meshes [1].

A simulator written in C++ of the same concept was implemented using VCGlib. A simulator was created using the Find Intersection function, which is one of the representative functions of VCGlib.

2.2.2 Python (Pycaster)

We have created a user-friendly simulator using Python. The simulator is built using a library called 'Pycaster' which contains the same functionality as VCGlib to find intersections.

2.2.3 Python with binding (pybind11)

Pybind11 is a lightweight header-only library that exposes C++ types in Python and vice versa, primarily for generating Python bindings of existing C++ code [2].

Pybind11 is used for C++ scripts to take advantage of Python's user-friendly environment and C++'s fastest computational speed. Based on the runtime analysis result (**Fig. 4**), the intersection function, which takes the largest portion in runtime, was converted and applied through pybind11.

3. Multiprocessing

The problem that always accompanies Python-based scripts is that they are slower than C++, in general. The same problem was observed in the simulator implementation described above, and we applied multiprocessing to improve speed.

3.1 Effects of Multiprocessing

Python supports multiprocessing (since Python version 2.6.X), which can significantly shorten the runtime required for iterative tasks.

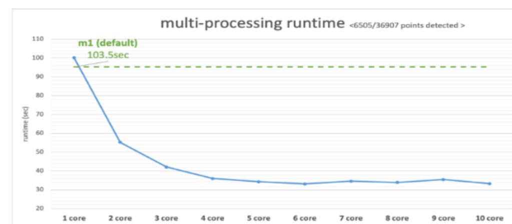

Fig. 2 Runtime comparison to the number of CPUs using (dotted line: not using multiprocessing)

We calculated the core efficiency (**Eq.** 1). The runtime reduction rate is defined as how much of the runtime has decreased compared to the previous step. Core growth is defined as how many cores have increased compared to the previous step. As the number of cores increases, the efficiency decreases along the trend line in **Fig. 3**, which means that the cost effectiveness decreases.

$$Core\ efficient\ rate = \frac{Runtime\ decrease\ rate}{Core\ increase\ rate} \quad \text{Eq. 1}$$

Fig. 3 Core efficiency (calculated by Eq. 1)

3.2 Amdahl's Law

As the number of cores used increases, the runtime can decrease, as shown in **Fig. 2**. Runtime reduction follows Amdahl's law [3].

$$speed\ boost = \frac{1}{\frac{1}{p} + a\left(1 - \frac{1}{p}\right)} \quad \text{Eq. 2}$$

where a is Amdahl's ratio, and p is number of processor

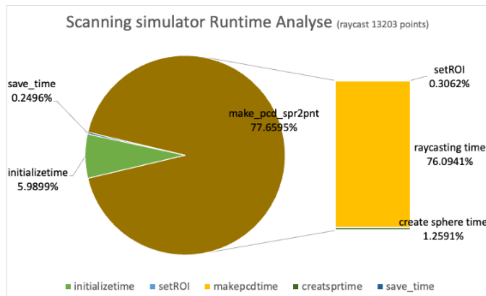

Fig. 4 Scanning simulator partial runtime analysis

About 76% of simulators are capable of multiprocessing as shown in **Fig. 4**. According to Amdahl's Law(**Eq.** 2), the 8-core runtime performance improvement is calculated to be approximately 3x. The calculation results are in good agreement with **Fig. 4**.

4. Experiment Environment

Numerical experiments were performed in a VMware environment using 8GB RAM, M1 (8 core CPU), Ubuntu 18.04 LTS.

We used a mesh model of a reactor part as a scanning target(**Fig. 5).** The three implementations of the ray-casting feature were tested.

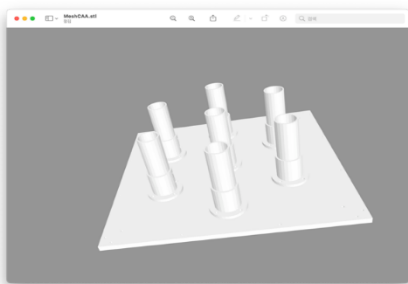

Fig. 5 Reactor part model

5. Result

Fig. 6 shows an example of scan simulation. As shown in **Fig.7**, pure C++ implementation performed the best among the implementations, and the Pybind11 implementation showed similar performance. In the case of Pycaster, since it is a code composed of pure Python, the performance improvement by multiprocessing was the greatest. The results are summarized in Table 1.

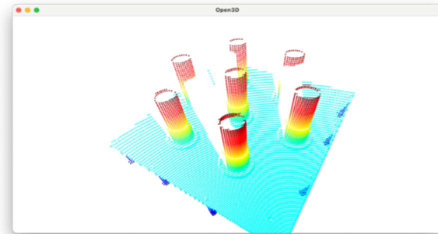

Fig. 6 Scanning result point cloud data (.ply)

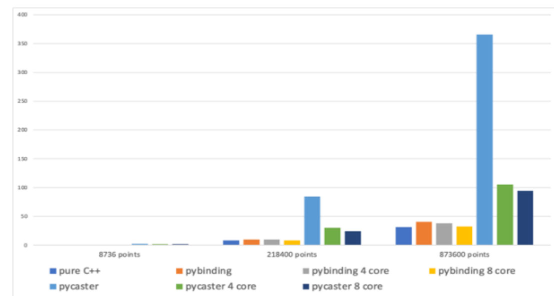

Fig. 7 Runtime comparison

Table 1. Summary of runtime result (sec)

	pure C++	pybinding			pycaster		
multiprocessing	defualt	defualt	4core	8core	defualt	4core	8core
8736 points	0.3600	0.4291	0.3567	0.3575	2.2080	1.6980	1.5980
218400 points	8.0400	9.7700	9.7010	8.3070	84.6342	29.9790	24.1872
873600 points	31.2700	40.5500	37.6100	32.4900	365.2913	105.4451	94.4333

ACKNOWLEDGEMENT

This work was supported by the nuclear research and development program through the National Research Foundation of Korea, funded by the Ministry of Science and ICT, Republic of Korea (Grant Code: 2017M2A8A5015146).

REFERENCES

[1] http://vcg.isti.cnr.it/vcglib/
[2] https://pybind11.readthedocs.io/en/stable/
[3] G. M. Amdahl. Validity of the Single-Processor Approach to Achieving Large Scale Computing Capabilities. In AFIPS Conference Proceedings, pages 483–485, 1967.