Rmagine: 3D Range Sensor Simulation in Polygonal Maps via Ray Tracing for Embedded Hardware on Mobile Robots

Alexander Mock\(^1\), Thomas Wiemann\(^2,3\) and Joachim Hertzberg\(^1,3\)

Abstract—Sensor simulation has emerged as a promising and powerful technique to find solutions to many real-world robotic tasks like localization and pose tracking. However, commonly used simulators have high hardware requirements and are therefore used mostly on high-end computers. In this paper, we present an approach to simulate range sensors directly on embedded hardware of mobile robots that use triangle meshes as environment map. This library, called Rmagine, allows a robot to simulate sensor data for arbitrary range sensors directly on board via ray tracing. Since robots typically only have limited computational resources, Rmagine aims at being flexible and lightweight, while scaling well even to large environment maps. It runs on several platforms like Laptops or embedded computing boards like NVIDIA Jetson by putting an unified API over the specific proprietry libraries provided by the hardware manufacturers. This work is designed to support the future development of robotic applications depending on simulation of range data that could previously not be computed in reasonable time on mobile systems.

I. INTRODUCTION

Triangle meshes are a standard data structure to represent 3D environments in many applications in computer graphics or robotics. Today, it is possible to automatically generate triangle mesh maps even online on embedded systems [1] that accurately capture the real world even in large scale environments [2]. This development allows to use such triangle meshes as an alternative map format to commonly used 2D grid maps or 3D voxel grids. Recently, algorithms for mobile robot navigation were adapted to this format, where the triangle mesh was interpreted as 2D manifold to plan paths over the mesh surface to a given goal [3]. Advances in the gaming industry now also make real time ray tracing feasible in such maps by using optimized algorithms and dedicated hardware. With the power of current ray tracing techniques, it is possible to simulate sensor data even for complex range sensors like LiDARs in real time. Any application that runs on a robot and at some point needs to answer the question “What would your sensor data look like if you were at that location?” could benefit from such 3D range sensor simulations. For example, Monte Carlo localization (MCL) algorithms [6].

In this paper, we introduce our library called Rmagine\(^1\) that enables developers to simulate range sensors on the robot itself even in large-scale triangle meshes. In contrast to commonly used simulators, Rmagine was designed to work as a standalone library without the requirement of a graphical user interface or rendering environment. Rmagine is not intended to simulate complex physics, but to provide efficient ray tracing support on mobile device for future localization and tracking algorithms that use polygonal environment representations. It focuses on a lightweight implementation that allows fast and parallel ray computations on different computational devices as CPUs and GPUs. When hardware-acceleration is available by RTX GPUs, Rmagine is able to simulate thousands of scans from standard sensors per second. This allows an application to be computed either on a server architecture or even on an embedded device on a robot itself. The latter is crucial for algorithms that shall not rely on

---

\(^1\)Knowledge Based Systems Group, Institute of Computer Science, Osnabrück University, Berghoffstraße 11, 49090 Osnabrück, Germany amock@uos.de, joachim.hertzberg@uos.de

\(^2\)Autonomous Robotics Group, Institute of Computer Science, Osnabrück University, Berghoffstraße 11, 49090 Osnabrück, Germany twiemann@uos.de

\(^3\)DFKI Niedersachsen, Plan-based Robot Control, Berghoffstraße 11, 49090 Osnabrück, Germany thomas.wiemann@dfki.de, joachim.hertzberg@dfki.de

The DFKI Niedersachsen (DFKI NI) is sponsored by the Ministry of Science and Culture of Lower Saxony and the VolkswagenStiftung

Rmagine is released under BSD 3-Clause license and available at https://github.com/uos/rmagine.
connectivity to a remote simulation infrastructure. Besides
the hardware abstraction layer, Rmagine provides a number
of built-in sensor models such as RGB-D cameras and LiDARs
and an software interface that allows to implement any scan
pattern as required by the application at hand. Furthermore,
we provide a ROS interface and Gazebo plugins\(^2\) to showcase
elements for this library in robotic contexts. Fig. 1 shows
the results of using Rmagine to simulate different range
sensors in a 3D map rendered in RViz. In our experiments,
we demonstrate the benefits and limits of Rmagine and
comparing it to the state-of-the art simulators Isaac Sim [7]
and Gazebo [8].

II. RELATED WORK

In computer graphics (CG), rendering is the process of
generating the image of a virtual camera in a virtual scene.
The research field of classical photorealistic rendering mainly
aims to render images that appear as realistic as possible to
a human viewer. Contrary, simulations aim to be as physically
correct as possible compared to real sensor data. Physical
correctness and realistic rendering however are often not so far
apart. The research field Physically Based Rendering (PBR),
seeks to improve light-physics in order to let the renderings
appear more realistic to human viewers [9]. Researchers
of PBR early searched for a rendering strategy alternating
classical rasterization that models the transport of light
more accurately in terms of physics. The most promising
approach is ray tracing, which simulates light rays in a scene,
taking into account reflections, transmissions, and emissions
at the surface. Due to the long computation time of the light
traversals, for many years this technique has not been used in
real-time applications. In recent years algorithms accelerating
scene traversals with BVH-trees on CPU and GPU improved
significantly, especially when implemented in hardware on
a GPU [10]. Such realtime capable and physically correct
ray tracing triggered novel developments in photorealistic
rendering and sensor simulation. In 2018, NVIDIA released
the Isaac SDK [7] that includes the simulator Isaac Sim to
simulate robots in 3D scenes. Isaac Sim utilizes NVIDIA's
hardware accelerated ray tracing to improve runtime and
accuracy of physical simulations including sensor simulations.
To simulate physics, it uses NVIDIA's PhysX API. However,
the software can only be used on computers with built-in
NVIDIA RTX GPUs. NVIDIA advertises Isaac Sim for
having such a high level of render realism that it could even
be used for reinforcement learning using simulated data [11].
An established alternative to Isaac Sim is Gazebo, that has
been successfully used in robotic applications [12]. Gazebo
introduced the support of PBR with version 11, but the used
OGRE rendering engine does not support ray tracing. To
compensate for that, with Ignition-Gazebo they integrated the
NVIDIA OptiX [13] ray tracing engine in addition to OGRE.
This software is currently in an experimental state. Besides
Isaac Sim and Gazebo there are many more approaches out

\(^2\)Rmagine Gazebo plugins is a software package released under
BSD 3-Clause license and available at https://github.com/uos/
rmagine_gazebo_plugins.

Applications running on robots such as localization [4],
[5], [15], change detection [6], or online inspection planning [16]
require very frequent spawns and removals of sensors at
certain locations in the map. Using Isaac Sim or Gazebo
for these problems, however, leads to large computational
overheads. In addition, Isaac Sim and Gazebo must first
be installed on a mobile robot, which may not be possible
depending on the limitations of its computing units. From the
point of view of simple algorithmic ray tracing operations,
these simulators come with a significant overhead, as they
also provide complex physical simulations and complex
user interfaces, that are not required when running within
a standalone algorithmic library. Furthermore, these software
packages do not provide a suitable API for standalone
applications. Hence, both Isaac Sim and Gazebo are not
intended for use on the robots directly [8], [17].

Rmagine was especially developed to bridge this gap
between external simulation environments and the need to
perform fast sensor simulation on the robots for different
tasks. Since the available computational devices on robots
differ significantly in hardware architecture, ranging from
small embedded SoCs to standard laptops with or without
dedicated GPUs, Rmagine provides a high level API that
abstracts from the underlying hardware to provide ray tracing
support on many devices and architectures.

III. RMAGINE

Rmagine is written in C++ and implements functions for
simulating range sensors, accelerated with BVH trees, which
are known to scale logarithmically in terms of render time
with increasing map size [18]. In addition, ray casting allows
to model sensors more flexible and precise compared to
traditional rasterization [19]. Currently, Rmagine uses Intel
Embree [20] and NVIDIA OptiX [13] as backends to support
robots with CPUs and NVIDIA-GPUs equally. Embree was
improved for Intel CPUs, but can also be executed on other
architectures. To actually simulate data of a sensor, a map,
a sensor model, and its pose relative to robot’s base need
to be provided. Each of these elements can be placed and

Authorized licensed use limited to the terms of the applicable license agreement with IEEE. Restrictions apply.
modified either in the CPU’s RAM or in the GPU’s VRAM. Consequently, custom implementations can update the map the sensor models and placements continuously either via CPU procedure calls or CUDA kernels during runtime. The API of Rmagine is visualized in Fig. 2. The following examples can be tested with our Rmagine Gazebo plugins.

A. Map

In Rmagine, a map consists of a scene graph. One scene holds many geometries or instances. A geometry is, for example, a triangle mesh. Each instance instantiates another subscene which again holds many geometries or instances. Many instances can refer to a single subscene. Instances of the same subscene may differ by their pose and scale in the scene. It is possible to spawn many instances of the same subscene without actually duplicating their memory. Rmagine implements one scene graph for each backend. The backends provide similar interfaces differ internally in storing the scene data. An Embree scene holds its elements in RAM while the OptiX scene stores the geometry data as CUDA buffers. The only interface difference is, that the Optix scene can hold either instances or geometries while the Embree scene can handle instances and geometries in the same scene.

These scenes can either be constructed programmatically or loaded from file formats for triangle meshes or scene graphs in general. In order to provide a unified interface that supports a wide range of file formats, we use the Open Assets Importer Library (Assimp) [21] to load meshes and scene graphs into our software. The Assimp scene graph is then internally converted into Embree or OptiX scenes. The respective acceleration structures are then constructed over the Rmagine scene graph.

Once a map is constructed, it also can be modified during runtime. Modifications are, for example, adding, removing, rotating, moving, scaling instances or geometries. Geometry modifications, such as moving single vertices, are also possible during runtime. OptiX scene elements can also be changed using CUDA kernels, which allows a developer to animate scenes without any overheads in copy operations. After committing the modifications of the map, Rmagine updates the acceleration structures. Mostly, they do not require complete rebuilds, and instead, only those tree depths affected by the committed modifications are partially updated. All these functionalities make the map dynamic, which allows to detect environmental changes and transfer them into the map.

B. Sensor-Models

Robots may be equipped with several range sensors with different mathematical models. Rmagine implements the sensor models Spherical, Pinhole, Cylindrical and the custom models 01Dn and OnDn. The SphericalModel consists of three values theta, phi and range, which represent polar angles θ, φ and the range r, respectively. Since sensors that can be modeled with a spherical model typically have constant horizontal and vertical sampling increments, the values for θ and φ can be discretized accordingly. For example, a Velodyne VLP-16 LiDAR sensor has a 30° field of view with 16 scan lines with a fixed resolution of 2°. Thus, φ ∈ [−15.0°, +15.0°] occupies 16 possible values with an increment of 2.0°. Horizontally, the VLP-16 has a 360° field of view with an adjustable resolution from 0.1° to 0.4°. Setting the horizontal resolution to 0.4°, one scan line would provide 900 single measurements. Thus, θ ∈ [−360°, 360°) takes 900 possible values with an increment of 0.4°. The measurements of a VLP-16 are valid up to a 100 m distance: r ∈ [0, 100]. Depth cameras can be modeled by a PinholeModel with the parameters image size (w, h), focal lengths (fx, fy) and principal point (cx, cy). In the last two models provided in Rmagine, 01Dn and OnDn, the origins and directions of the rays can be fully customized, allowing developers to implement sensors following arbitrary scan patterns. 01Dn has one origin and N directions while OnDn has N origins and N directions where the i-th ray is constructed from the i-th origin and the i-th direction. Tab. I shows common sensor models that can be simulated with Rmagine. Each simulated sensor can be attached to the virtual body of a robot by defining the transformation of the sensor to the robot’s base Tsb. Fig. 1 visualizes simulations for each of Rmagine’s sensor models.

C. Simulation

Given the Rmagine map and the sensors models, simulated data can be generated for arbitrary poses in the map. First, a map is passed to the sensor simulator object as well as the particular transformation from sensor to base Tsb. This information together with the defined sensor model is used to simulate the intersections of the respective scan pattern with the 3D model. Besides the Cartesian coordinates, additional

| Attr. | Type | Stride | Attr. | Type | Stride |
|-------|------|--------|-------|------|--------|
| Hits  | uint8 | 1      | Primal | uint32 | 1      |
| Ranges| float | 1      | Geomals| uint32 | 1      |
| Points| float | 3      | InstIds| uint32 | 1      |
| Normals| float | 3      |       |        |        |

Fig. 3: Rmagine simulation of a LiDAR in a 3D map consisting of several instances. Here, the intersection attributes Points, Normals and InstIds (colors) are queried.

TABLE I: Examples of well known sensors that can be modeled and simulated by Rmagine.

| Model | Sensors |
|-------|---------|
| Spherical | Velodyne VLP-16, VLP-32, Ouster OS-0, OS-1, OS-2 |
| Pinhole | MS Kinect, Asus Xtion, Intel RealSense D400 series |
| 01Dn | Livox Mal-40, Avia |
attributes of the intersection point can be requested. The hit attribute can be either 1 or 0, depending on whether the ray intersects the map or not. The range attribute represents the distance from the ray’s origin to the first intersection with the map along the ray’s direction. Querying the normal attribute enables the calculation of the normal at the intersected face. The ID of instance, geometry, or triangle can be retrieved by ID of instance, geometry, or triangle can be retrieved by the InstId, GeomId, or PrimId attribute respectively. A list of all available intersection attributes is summarized in Tab. II. The implementation automatically optimizes the OptiX and Embree code depending on which combination of intersection attributes were queried. Thus, we assure the ray casting operation is performed only once per requested bundle of intersection attributes. The resulting attribute buffers are located in the sensors reference frame. Fig. 3 visualizes the simulation results querying the attributes Points, Normals and InstIds. Depending on the device that computes the intersections, the results are stored either into buffer in CPU or GPU memory. This allows other applications to flexibly post-process the simulated data without any overhead for transferring data between computing devices. This way, Rmagine distributes the workloads flexibly to computing devices of the robot. For example, Rmagine provides functions to apply several noise models to the simulated measurements. Rmagine selects the actual implementation of the function according to the memory location of the simulated measurements. If the data is located in a CUDA buffer on the GPU, the random numbers are generated directly on the device by using cuRAND, CUDAs random number generation library. Otherwise, the functionalities of the C++ standard random number library are used. Examples of noise models implemented in Rmagine are shown in Fig. 4.

D. Example Modeling

In this section, we present an example for modeling a LiDAR and a depth camera in Rmagine with GPU and CPU-based ray tracing. For this, we need to load the map data both to CPU and GPU. The depth camera is modeled using suitable parameters for a pinhole model, the LiDARs is modeled as an instance of a spherical model. The actual simulator classes are named by the sensor-model type X as prefix and the computing backend Y as suffix: XSimulatorY. Accordingly, we need a SphereSimulatorOptix and a PinholeSimulatorEmbree to compute a LiDAR via OptiX and a depth camera via Embree. For sensor data simulation, we define a set of possible poses that is loaded both to CPU and GPU memory. After each simulation request, the scans simulated on the GPU are then downloaded into CPU memory. The workflow of this example is sketched in Fig. 5. As robots often execute several applications running in parallel, it might be desirable to support some mechanism for load balancing. For that, Rmagine allows to switch the usage of computing devices during runtime. Switching devices in this example can be realized by adding a SphereSimulatorEmbree and PinholeSimulatorOptix that can be called respectively.

IV. EXPERIMENTS

This section evaluates Rmagine with respect to our design goals. For this, we installed Rmagine on several test devices to prove the compatibility with different computing architectures and to compare it to Isaac Sim and Gazebo. Furthermore, we present an in-depth runtime analysis of Rmagine in different environments.

A. Compatibility

Rmagine completely inherits the device compatibility of its ray casting backbones Intel Embree and NVIDIA OptiX. Embree at version 3 runs on devices from Intel, AMD or on ARM CPUs. The developers state that it performs approximately 10% better on a Intel CPU when compiled it with the Intel C++ compiler. However, to allow a fair comparison, we compiled Embree with the GNU compiler (version 9.3.0) on an Ubuntu 20.04 operating system. GPU support requires OptiX of version 7.2 or newer. To support this in Rmagine, the OptiX headers need to be downloaded from NVIDIA, while the OptiX library itself is automatically installed through the NVIDIA drivers. OptiX runs on both NVIDIA GTX and NVIDIA RTX graphics cards but not on GPUs of other manufacturers. The OptiX host API can be installed on both Intel and AMD processors. Since version 7.3, OptiX also supports installation on ARM chips. For evaluation, we
TABLE III: Specifications of the devices that were used to evaluate Rmagine, i.e., a desktop PC (PC), a Lenovo Legion (Legion), a Intel NUC (NUC), a Lenovo X1 (X1) and a Jetson Xavier NX (NX).

| ID | CPU Type          | NVIDIA GPU Type    |
|----|-------------------|--------------------|
| PC | Ryzen 7 3800X     | RTX 2070 Super     |
| Legion | Ryzen 7 5800H | RTX 3070 Laptop   |
| NUC | i7 1165G7         | RTX 2060 Max-P     |
| X1  | i7 8750H          | GTX 1050 Ti Max-Q  |
| NX  | Carmel ARM 6C     | Volta 48TC         |

installed Rmagine on an office PC, Lenovo Legion 5 Pro 16ACH6H laptop, an Intel NUC RNUC11PHKg7CAA2, a Lenovo 20MF000XGE laptop and a NVIDIA Jetson Xavier NX developer board. The specifications of the used devices are summarized in Tab. III.

On the Jetson NX, however, we could not install the GPU support since the OptiX library is currently not deployed with the NVIDIA driver for Volta architectures. Nevertheless, compared to the install specifications of Isaac Sim and Gazebo, our software covers a wider range of compatible devices as shown in the left part of Tab. IV. Regarding the required compute capabilities, (NV) stands for NVIDIA and (RT) for ray tracing. A negation means non-NVIDIA or no ray tracing support respectively. In contrast to Rmagine, Isaac Sim necessarily requires a dedicated GPU. However, Gazebo allows to simulate sensor data also on non-NVIDIA GPUs, which Rmagine currently does not support. Isaac Sim and Gazebo both implement interfaces to let developers receive the simulated sensor data on CPU side through buffers placed in RAM. Rmagine additionally implements an interface to directly access the GPU simulations via CUDA buffers to avoid unnecessary downloads when passed to post-processing CUDA kernels afterwards.

B. Performance

In the next experiments the performance of Rmagine was analyzed. We evaluated Rmagine’s runtime in detail using several devices and considering different computational loads and map sizes. In each of the following experiments, we simulated the Velodyne VLP-16 LiDAR sensor, which has 16 scan lines, with 900 range measurements each. Per sensor frame, we thus needed to simulate 14,400 intersections with the map. For the first evaluation we used a synthetic sphere with 1 million faces as map. Inside this sphere, we placed multiple VLP-16 sensors at random poses. The runtime measurements for increasing numbers of simulated poses are presented in Fig. 6. Each experiment was repeated 100 times with different distributions on all devices. The results show that the runtime is linear to the number of simulated sensors.

In the next experiment, we placed 10,000 VLP-16 sensors on static poses while varying the map size. The entire experiment was again computed and repeated 100 times on our test devices. The average runtimes are shown in Fig. 8. Since both Gazebo and Isaac Sim were originally not designed to run on mobile robot hardware, a performance comparison here would not be meaningful. Therefore, we added a Gazebo baseline at Fig. 8a just for orientation. Here, Rmagine outperforms Gazebo in every instantiation. Since both Embree and OptiX internally use BVH-trees to accelerate ray traversals, we expect the simulation to scale well in larger maps. Matching this expectation, the experimental results demonstrate the simulation time rises logarithmically with increasing map sizes.

The first two experiments were done on synthetically generated maps. In actual applications, maps of the real-world environments are more unstructured than the test data in the first experiments. To account for that, we performed additional experiments on several real world maps of sites at Osnabrück University (indoor and outdoor), either manually built or reconstructed from point cloud data. Example renderings of these data sets are shown in Fig. 7. The left part of Tab. V summarizes the size of the captured areas and the number of triangles in the maps. The column “Data Type” specifies if the mesh was synthetically hand modeled or reconstructed with LVR2 [2]. Within each environment, we recorded the number of simulated VLP-16 scans that can be simulated in one second. The results are summarized in Tab. V.

C. Quality

In the last experiment we compared the quality of the LiDAR scans simulated by Rmagine, Isaac Sim, and Gazebo. To compare the simulations to a shared baseline, we also placed a real Velodyne VLP-16 in a scene that looks similar to the virtual scene and recorded its measurements.
TABLE IV: Supported architectures, compute capabilities and memory location of simulation results.

| Target Architecture | Compute Capabilities | Memory Location |
|---------------------|----------------------|-----------------|
| Gazebo              | CPU, GPU (NV)        | CPU, GPU        |
| Isaac Sim           | CPU, GPU (NV-RT)     | CPU, GPU        |
| Rmagine             | CPU, GPU (NV RT)     | CPU, GPU        |

Table IV: Supported architectures, compute capabilities and memory location of simulation results. The results of Rmagine are shown in (b). As baseline we measured the runtime for Gazebo as well (a). For that, we chose the office environment of 7b. The results of Rmagine are shown in (b). As baseline we measured the runtime for Gazebo as well (a).

Fig. 8: Runtimes of simulating 10,000 VLP-16 scans on different devices (Tab. III) while varying the map’s size (x-axis). The results of Rmagine are shown in (b). As baseline we measured the runtime for Gazebo as well (a).

Fig. 9: Quality of LiDAR data simulated in map 7b compared to real data of a Velodyne VLP-16 (a). For that, we chose the office environment of 7b. The results show that the quality of the Rmagine simulations such as Isaac Sim and Gazebo are similar to the real scan, except for the GPU ray sensor of Gazebo (v11.9.1) as presented in Fig. 10. These anomalies only occur at rays hitting the surface at sharp angles. The accuracy of a simulation is closely tied to how accurate the 3D map resembles the real world. Consequently, it’s challenging to differentiate between errors in mapping and those in the simulation itself, which makes it inexpressive to perform quantitative evaluations.

V. CONCLUSION AND FUTURE WORK

We presented a library to simulate range sensor data on different platforms called Rmagine. In contrast to established simulators, Rmagine is developed to directly run on the embedded computing devices of a mobile robot. Rmagine makes the simulation of arbitrary range sensors feasible for future development of robotic algorithms. The evaluation demonstrated that our approach is fast compared to established simulators even without GPU acceleration and that it scales well to large maps. The possibility to keep the simulation results in GPU memory allows to directly use the computed 3D data in GPU-accelerated algorithms. Future work should enhance the library by additional map representations such as 3D occupancy grids, TSDF fields or 2D occupancy grids to be downwards compatible. Furthermore, we aim to support GPUs of other manufacturers from AMD or Intel by implementing an OpenCL interface. In principle, it might even be possible to integrate more application-specific devices like FPGAs. We also aim to integrate materials and textures to allow developers to integrate more application-specific devices like FPGAs. We also aim to integrate materials and textures to allow developers to integrate more application-specific devices like FPGAs.
