Swarm Robotics Simulation Using Unity

Binh-Son Le[†], Vy-Long Dang[‡], and Trong-Tu Bui[†]

Faculty of Electronics and Telecommunications, University of Science, VNU-HCM, Vietnam

E-mail: [†]{lbson,bttu}@fetel.hcmus.edu.vn, [‡]vylong93khtn@gmail.com

Abstract Unity is well-known as one of the top game engines for game development. However, with the ability to switching between 2D and 3D environments, using state of the art physics engine (NVIDIA PhysX), flexible programming model, and a good debug interface, it also shows a great potential to simulate swarm-robotic systems. In this paper, we will present how Unity can be used to model a wide range of sensors and communication channels, which are vital structures of a swarm-robotic system. Then, we also demonstrate its simulation ability by implementing and verifying a localization algorithm of a swarm of 2D robots.

Keywords Swarm, Robotics, Simulation, Unity

1. Introduction

In recent years, the study of how collective behaviors between relatively simple robots can accomplish distributed tasks, such as foraging, collective transport of objects or mapping, and pattern formation, in a reliable and scalable way, has been studied extensively [5, 7]. As inspired from decentralized behaviors of natural swarms of insects, the term ”Swarm Robotics” is usually used to denote these systems.

In swarm robotics, each robot should be able, but not required, to communicate and sense each other in order to reach a common goal. For example, if each robot can measure the distance from its location to its neighbors and pass these information to others, they can build a network coordination (localization) then self-assembly into a particular shape [8]. However, validating swarm-robotic algorithms in real world environments is not a trivial task, as researchers have to build tens to hundreds of mobile robots which would pose a problem on the cost, time and area required for each experiment. Even if these requirements can be met by using small and low-cost robots such as kilobot [5], it is still better to simulate these algorithms before implementing them on real systems, as researchers can control how realistic an experiment can be and separate hardware failures with software problems. For instance, sensor noises can be temporary excluded from the simulation to test the correctness of the algorithm, then, different noise models can be used to verify its robustness.

For simulating swarm-robotic systems, a simulator should support both 2D and 3D environments, as it will be much faster to simulate the communication between thousands of dots in a 2D environment before observing how a few tens of them interact in a realistic 3D scene. Obviously, physical realism is also an important factor in simulating collective behaviors of the system. Furthermore, since each type of mobile robot can use a different method for sensing and communicating with their surrounding, the software architecture should be highly flexible and customizable to adapt to specific situations. Finally, the simulator should support a debug interface and a way to control the data of each robot.

Nevertheless, to the best of our knowledge, none of the existing commercial and open-source robot simulators can meet all of the requirements above. 3D commercial tools such as Webots [6] and V-REP [4] are integrated with various well-known robot models and support many programming languages and platforms. However, their distance sensors are not flexible, as they function both as a transmitter and receiver at the same time, which is not the case for real sensors such as infrared, sonar or laser. Though, the open source project, player/stage [1], together with project gazebo [3] provide both 2D and 3D environments, their physics engine, Open Dynamics Engine (ODE), is not as precise as other libraries such as NVIDIA PhysX engine. In addition, their sensor models and visual effects also suffer similar limitations as Webots and V-REP. On the other hand, to provide realistic 3D environments, an open source simulator named MORSE, built on top of the game engine mode of Blender (a 3D digital content creation tool), has also been proposed. It only supports Python, uses Bullet engine and follows component based approach to model sensors, actuators, and robots as other simulators. However, since Blender is more of an artist tool than a programming tool, its game engine is not as good as other tools such as Unity or Unreal Engine. Nevertheless, utilizing a game engine for robot designs and simulating is a fascinating idea, since a game engine is highly customizable in order to create and model nearly every real world situations, from complicated aerial vehicles to diving machines.

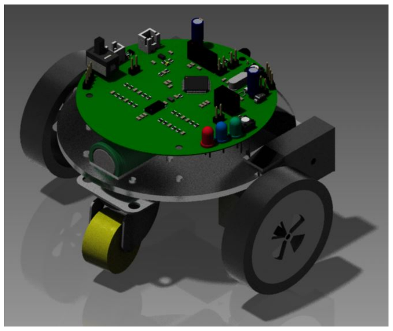
Among the top game engines, Unity shows a great potential to be used in swarm-robotic simulations. Its free version is flexible enough to re-create existing robot platforms with different types of actuators and movements, as proven in [10]. Additionally, it also supports three different programming languages: C#, Unity Script (JavaScript), and Boo (Python), with a wide range of visual effects in 2D and 3D environments and uses NVIDIA PhysX as its physics engine. In this paper, we will continue to show that Unity is capable of creating a wide range of sensor models and communication channels, which are vital structures of a swarm-robotic system. Then, the trilateration algorithm [9] used in the localization process of a swarm of robots is also implemented on simple 2D robots and simulated.

The rest of this paper is organized as follows. Section II goes in details how Unity can be used as a robotics simulator. In section III, the localization algorithm is implemented and verified on a swarm of simple 2D robots. Finally, section IV draws the conclusion.

2. Swarm Robotics Simulation Using Unity

2.1. Programming in Unity

Similar to other robotic simulators, each object in Unity can be attached with various kinds of scripts (written in C#, Unity script or Boo), which govern the behaviors of that object in the simulated scene. After a component is created, it can be stored as a prefab to be reused later on. For example, a motor model can be attached with a script indicating its maximum velocity, its mass, what parts of it can rotate, and some random parameters to replicate a real motor. Then, a motor prefab can be attached many times into a robot model to form another more complicated prefab. Additionally, in the robot model, other scripts can access the script in a motor model to control its parameters and invoke its functions. By using scripting, researchers can first test an algorithm in a 2D environment with simple 2D robots, then test it again in a 3D scene with a more realistic robot model.


(a)

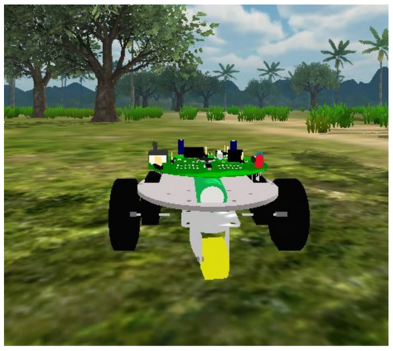

(b)

Figure 1: (a) The model of a T-Bot in Autodesk Inventor (b) The robot model rendered in the free version of Unity.

2.2. Robot Design

A robot model can be imported directly into Unity from digital content creation (DCC) tools such as Blender or Maya. On the other hand, designers who model their prototype robot on CAD softwares, such as Autodesk Inventor or Solidworks, can import the model into DCC tools first, then, export it to Unity. For example, Fig. 1 shows the model of our prototype robot (T-Bot) in Inventor and how it was rendered on the free version of Unity. Since all the constrains from CAD softwares are lost during the importing process, designers will need to specify rigidbody components, colliders, joints and actuators [10] again in Unity. It should be noted that Unity also supports "wheel colliders" to simulate a wheel vehicle movements. This method is less accurate but also requires less computation power than using mesh colliders. Aside from wheel robots, flying objects, such as quadrotors, planes and helicopters, have also been created by other game designers and can be accessed through the asset store.

2.3. Sensors Modeling

As mentioned before, most robotic simulators use a technique known as ray casting to implement distance sensors. This method draws a straight line from the transmitter to the first obstacle in its

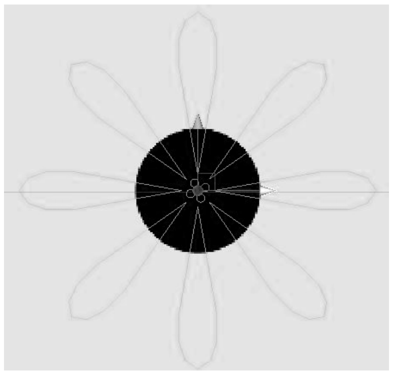


Figure 2: The model of an infrared ring on a simple 2D robot.

measured range, then it returns the distance between the two objects. As a result, infrared, sonar or laser sensors modeled by this way are only different in the area they can cover. In swarm robotics, each robot should be able to measure the distance from itself to its neighbors. However, the ray casting technique in most simulators does not provide much knowledge about the object it hits, so it requires some workaround ways to know if the obstacle is another robot or not. In Unity, this technique is also available and returns much more meaningful information, but sensors can also be modeled using trigger zones, which can specify which types of objects entering a zone can cause a predefined event. Hence, we can create a receiver that only reacts when it enters an appropriate transmitter zone. For example, Fig. 2 describes an infrared ring, with 8 transmitters forming 8 trigger zones and 4 circles in the center playing as receivers, attached to a simple 2D robot. Thus, each robot will know exactly when it enters another robot sensor range. Furthermore, by combining both of these techniques, we can model complex behaviors of various kinds of distance sensors which is not possible in other simulators.

Acceleration and position sensors can also be implemented easily by reading the velocity, position, or rotation of the object. For pressure sensors, we can multiply the mass of the object with its velocity after a collision event occurs. For visual sensors, we can use the camera component in Unity to capture images or stream videos from the simulated environment or directly from the real world using webcam. In addition, the plugin feature in the pro-version also supports developers to call OpenCV library through wrapper methods. Moreover, users can interact with the simulated robot in real-time through a wide range of input devices, such as Leap Motion or Kinect. Thus, this will enables a wide range of interesting applications and experiments.

2.4. Communication Channel

Similar to distance sensors, ray casting, trigger zone and scripts can be used to define a communication zone from one robot to others. For example, a sphere shape can replicate an RF zone in which any robots with an RF receiver can listen to the transmitter. In addition, using trigger zone and ray casting, the distance from the transmitter to the receiver and obstacles between them can be used to control noise parameters in the script attached in the receiver. This feature is similar to that of commercial tools like Webot, but more flexible, as users can freely customize the channel to suit their requirements.

3. Localization Algorithm Simulation

3.1. The Simulated 2D Robot

The target of this paper is to verify the localization algorithm on Unity. Therefore, a simple 2D robot similar to the one in Fig. 2 is used. However, the led ring is replaced by a circle, which is a trigger zone that will notify the robot if another robot enters its range and start measuring the distance from the two robots. For the communication channel, we replicate an ideal RF channel with following characteristics:

- Each robot has a unique global ID, which can be used as its address in the communication channel.
- A robot can send a message to all other robots belonging to its RF range.
- Only one robot is able to transmit data at a time.
- If a robot cannot transmit data, it waits a random time before trying to access the channel again.

3.2. The Localization Algorithm

One of the most basic features of swarm robotics is network coordinate building (localization), as only after that, algorithms such as pattern formation, mapping, or collective movements can continue to be integrated into the swarm. In this paper, we use the trilateration algorithm, proposed in [9], which has been used extensively in wireless sensor networks. This algorithm has two phases. First, each robot builds their own local coordinate system by using information about the distance of surrounding robots. Fig. 3 depicts the simulation results on Unity after phase 1 is complete. It should be noted that, since this process is random, each time the simulation is run, a different result is obtained.

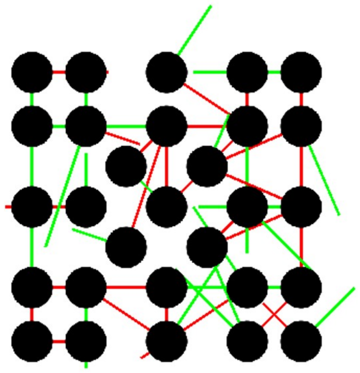
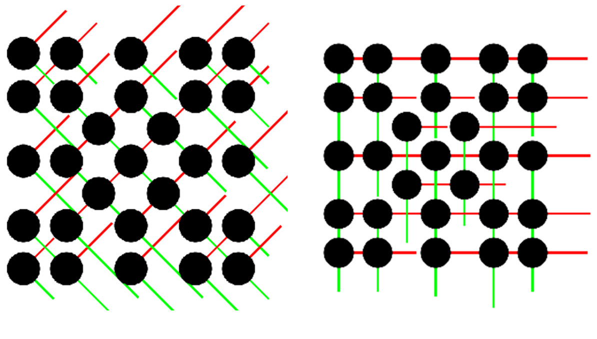


Figure 3: The simulation results after phase 1, with two straight lines representing X and Y axis.



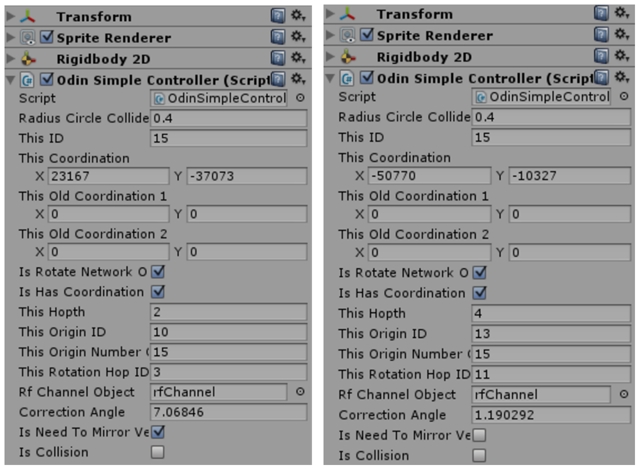
(a) (b)

Figure 4: The simulation results after phase 2 is completed (a) The first run (b) The second run.

In phase 2, all coordinate systems are rotated or mirrored to converge into one unique network coordinate. Fig. 4 describes the simulation results after this phase is completed and Fig. 5 shows the details about the coordinate of a robot (with ID is 15). As can be seen, each time the simulation is run, a different network coordinate system, with another origin ID and rotation, is obtained. Fig. 5 is also a great example of how all public properties of a script can be viewed directly in Unity without the need to manually print each of them out in the debug console. Thus, this greatly simplifies the debugging process.

4. Conclusion

In this paper, we have presented and verified how Unity can be used to simulate swarm-robotic system. With the ability to switching between 2D and 3D environments, using state of the art physics engine, flexible programming model, and a good debug interface, Unity has met all of our requirements and is comparable to other robotic simulators such as V-REP, Webot, Gazebo and MORSE. Therefore, in future works, we will continue to implement and simulate the S-DASH algorithm [8] on both the simple 2D robot and



```
Transform
Sprite Renderer
Rigidbody 2D
Odin Simple Controller (Script
Script                     OdinSimpleControl
Radius Circle Collide      0.4
This ID                    15
This Coordination
   X 23167            Y -37073
This Old Coordination 1
   X 0                Y 0
This Old Coordination 2
   X 0                Y 0
Is Rotate Network O        ✓
Is Has Coordination        ✓
This Hopth                 2
This Origin ID             10
This Origin Number (       15
This Rotation Hop ID       3
Rf Channel Object          rfChannel
Correction Angle           7.06846
Is Need To Mirror Ve       ✓
Is Collision               ☐
```

(a)

```
Transform
Sprite Renderer
Rigidbody 2D
Odin Simple Controller (Script
Script                     OdinSimpleControl
Radius Circle Collide      0.4
This ID                    15
This Coordination
   X -50770           Y -10327
This Old Coordination 1
   X 0                Y 0
This Old Coordination 2
   X 0                Y 0
Is Rotate Network O        ✓
Is Has Coordination        ✓
This Hopth                 4
This Origin ID             13
This Origin Number (       15
This Rotation Hop ID       11
Rf Channel Object          rfChannel
Correction Angle           1.190292
Is Need To Mirror Ve       ☐
Is Collision               ☐
```

(b)

Figure 5: The simulation results after phase 2 at a robot (a) The first run (b) The second run.

the prototype 3D robot using Unity.

References

[1] R. T. Vaughan B. P. Gerkey and A. Howard. The player/stage project: Tools for multirobot and distributed sensor systems. In *Proc. Int. Conf. on Advanced Robotics*, pages 317–323, Jul. 2003.

[2] A. Degroote G. Echeverria, N. Lassabe and S. Lemaignan. Modular openrobots simulation engine: Morse. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 46–51, May 2011.

[3] Howard Koenig, N. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 2149–2154, Sep. 2004.

[4] F. Ozaki M. Freese, S. Singh and N. Matsuhira. Virtual robot experimentation platform V-REP: A versatile 3D robot simulator. In *Proc. Int. 2nd Conf. on Simulation Modeling and Programming for Autonomous Robots*, pages 51–62, Nov. 2010.

[5] C. Ahler M. Rubenstein and R. Nagpal. Kilobot: A low cost scalable robot system for collective behaviors. In *Proc. IEEE Int. Conf. on Robotics and Automation*, pages 3293–3298, May. 2012.

[6] O. Michel. Webots: professional mobile robot simulation. *Journal of Advanced Robotics Systems*, 1(1):39–42, 2004.

[7] Yogeswaran Mohan and S. G. Ponnambalam. An extensive review of research in swarm robotics. In *Proc. World Congress on Nature Biologically Inspired Computing*, pages 140–145, Dec. 2009.

[8] M. Rubenstein and Wei-Min Shen. Automatic scalable size selection for the shape of a distributed robotic collective. In *Proc. IEEE/RSJ Int. Conf. on Intelligent Robots and Systems*, pages 508–513, Oct. 2010.

[9] M. Hamdi S. Capkun and J.-P. Hubaux. GPS-free positioning in mobile ad-hoc networks. In *Proc. 34th Annual Hawaii Int. Conf. on System Sciences*, pages 1–15, Jan. 2001.

[10] R. Paris N. Smith J. Blevins W. A. Mattingly, D. Chang and M. Ouyang. Robot design using Unity for computer games and robotic simulations. In *Proc. 17th Int. Conf. on Computer Games*, pages 56–59, Jul. 2012.