Development of a line-following test environment for a surface adaptive wall-climbing robot

Conor Kirby[*1] and Haruhiko Eto[*2]

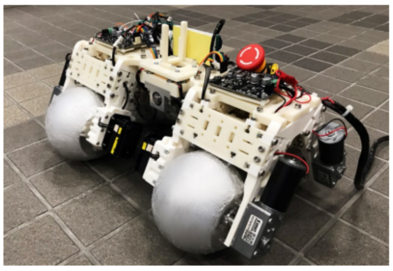
Abstract – This report describes the development of a Unity based simulation environment for a surface adaptive wall-climbing robot. The environment provides a series of experiments for the purpose of designing and testing line following controller schemes. Information about the development and use of the simulation is provided alongside tips for using Unity effectively. Simulation results are provided, presenting evidence for the effectiveness of a linear controller that compensates for the local normal at each wheel.

1 Introduction

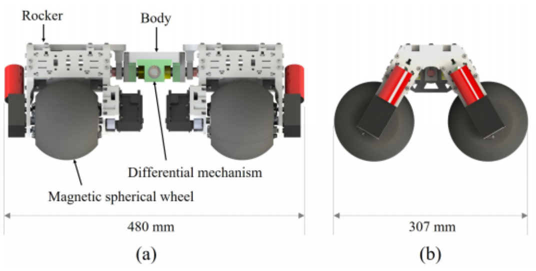
Following the work on a shape adaptive magnetic adhesion mechanism for a wheeled robot [10], the following work describes the development of a simulation test environment for autonomous control of a wheeled robot on a randomly curved terrain. The simulated wheeled robot uses a rocker arm suspension connecting two sides, and four wheels, each with indepently controlled magnetic mechanisms. Each wheel has two attached motors; one for the rotation of the physical wheel, and the other for the magnetic mechanism inside of the wheel.

Figures 1 and 2 show the physical design and dimensions of the robot. The mechanism allows the robot to adapt to an uneven surface on a ship, enabling the robot to be used on welding tasks. The magnet inside of each wheel follows the normal of the surface it is on, and the motor attached to the shaft of the magnet mechanism can be used to drive the magnet to a different angle, for example, when transitioning from the ground to a vertical wall.

The purpose of the simulation environment is to provide a system through which line following and speed control algorithms can be more easily tested and compared. Test environments can be adapted in software and data can be collected and analysed very easily at the same time. The following sections describe: the creation of the physical model of the robot in Blender and Unity; how the simulation is prepared and run to obtain useful information; data collection and analysis and finally a presentation of future work required to improve the simulation.



1 Prototype of the wall-climbing robot with the proposed magnetic adhesion mechanism.



2 Design of the proposed mobile robot: (a) front view, (b) side view.

2 Physical Model

2.1 Unity Software

Unity is a game engine used in a variety of fields to create 3D and 2D real-time experiences [9]. The built-in Physics engine and the ease with which models can be created and manipulated in C# made this software ideal for creating a general test environment for the robot. C# scripts can be written and attached to GameObjects in the Unity environment, and the script will be run every frame. The ease to which scripts can be attached to objects allows for

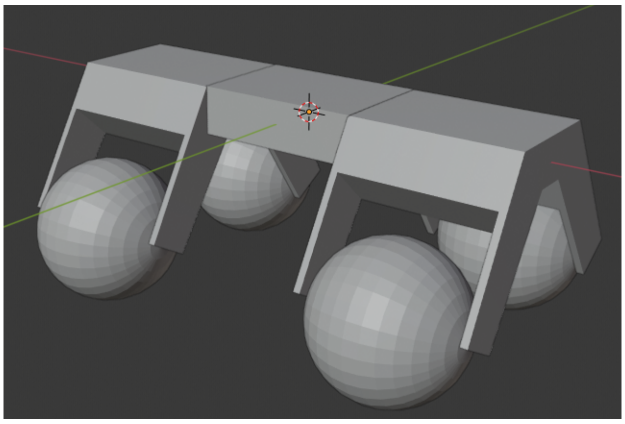
[*1]Massachusetts Institute of Technology
[*2]Sumitomo Heavy Industries

easy experimentation. For example, one could create several robots of different sizes and shapes, and the same C# script can be attached to each robot to compare their performance in a certain task.

Futhermore because Unity is used for a variety of fields, and is not exclusively used for a very specific purpose like many simulation software packages, it is possible to find support to solve problems through other users or user-created assets that can be imported into the Unity environment. However, Unity is not used for creating customised models of different objects, as there are only a few set shapes available to the user, such as cubes, planes and spheres. For creating a model to match the physical design of the robot, a separate software is needed.

2.2 Blender Model

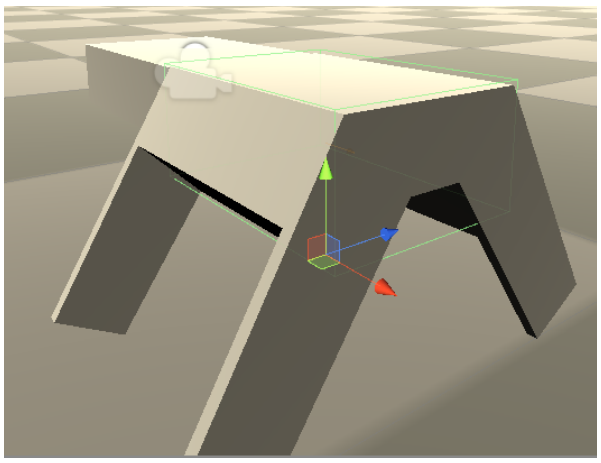
The physical parts of the robot model were modelled in Blender and imported as a FBX file. Blender is a powerful, free 3D modelling software, allowing one to create and adapt meshes that define some shape. The robot model is simplified into 7 parts - the 4 hollow wheels and 3 robot sections - and a mesh is created for each part to match the real robot dimensions as close as possible, including wheel inner and outer radius, and robot width and length. The model can be seen in Figure 3.


3 Robot model made in the Blender software

2.3 Unity Physics Engine

Once the FBX file is imported into Unity, the parts can be attached to each other by first adding rigid bodies and mass to each part and then adding hinge joints to match the real physical robot. Every rigid body in the simulation, at every frame, will have gravity and friction acting upon them. Gravity acts upon the centre of mass of each rigid body of the Robot body, while angular drag can be added to the wheels to slow down the angular rotation of the wheels[8]. In this simulation, gravity has also been defined separately in code, as a force acting at the centre of mass of each object, so that it can be turned on and off and magnetic forces can be analysed on their own. When using the self-implemented gravity, the Unity based gravity, seen as a useGravity boolean on the rigid body object, must be turned off for each object to ensure correct results.

A Collider element must be added to each part of the model to ensure that the robot interacts with the world, as the collider handles physical collisions in Unity. Using a Mesh Collider[4] is most accurate as it matches the exact physical shape of the robot parts, but due to some difficulties with the FBX importation method, box colliders for the robot body and spherical colliders for the wheels is recommended to ensure the colliders do not interact with each other and cause the robot to collide with itself. Figure 4 depicts the green box collider used by the physics engine to represent collisions to this object. A full mesh collider would cover the whole robot body, but in this case would cause self-collisions, so the simplified box collider works better.


4 Box Mesh used for robot body collision detection

2.4 Motors

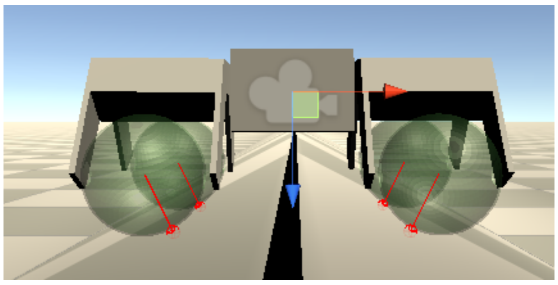
The robot model is able to move by attaching motors to each wheel hinge joint and specifying a maximum torque and target velocity for the motor. The max torque and target velocity must be set before the hinge joint useMotor boolean can be set to true, which will then attach the motor to the hinge joint of the game object[2]. The maximum torque is specified using the motors from the actual robot, the Dynamixel XM540-w270-r[12], with a stall torque of 10.6 Nm at a 12V input voltage. The target velocity for each motor is then varied at each frame to complete the line following task and to control the overall robot velocity. The system will then attempt to reach the target velocity without exceeding the maximum specified torque. Relevant script: LineFollowingPIDV2.cs attached to object 'Middle'

2.5 Magnetic Adhesion & Surface Normal

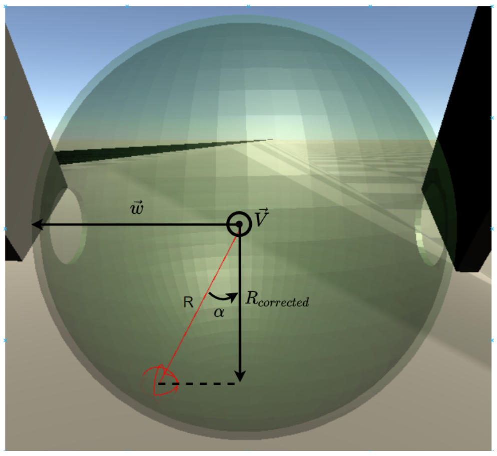
Robot magnetic adhesion is simulated using ray casting Physics in Unity. From the centre of each wheel, a spherical ray is fired directly downwards, and the contact point between that sphere and the terrain surface is detected [5]. The ray cast will only detect game objects in a certain 'Layer', with the terrain added to the 'Terrain' layer so that the robot ray cast will detect it. A force can then be added to the robot wheel rigid body, with a magnitude equal to the average magnetic force from the wheels in the real robot, and a vector direction from the wheel centre to the ray cast collision point. The collision point position in 3D space is then transformed into spherical coordinates to obtain the theta and phi angles that represent the vector from wheel centre to the contact point on the wheel. These angles are then used in the line following and velocity control task.

Figures 5 and 6 depict the ray casting physics computed at every frame. The red lines from the centre of the wheel to the surface follows the normal of the surface. The contact point can be defined in terms of two angles, α and β. The angle α, as shown in figure 6, is defined by projecting the contact point vector on to a plane defined by the rotation axis $\vec{w}$ of the wheel as its normal. The angle between the original vector and the projected vector is α, which can be used to calculate $R_{corrected} = R/cos(\alpha)$. $R_{corrected}$ is used to calculate the velocity of the wheel more accurately when given the angular velocity of the wheel. This process will be referred to as normal compensation. Similarly, β can be calculated by projecting the contact point vector onto a plane defined by the Velocity vector $\vec{V}$ of the wheel as its normal. This angle defines the slope of the hill in front of the robot, allowing for speed compensation on an uphill or downhill slope.

Relevant script: RayCastingTest.cs attached to each wheel



5 Ray cast point normal to robot velocity direction



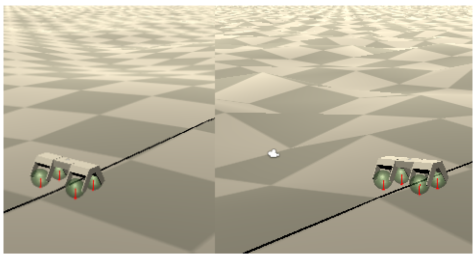
6 Close Up of Geometry of Ray Cast

3 Simulation Preparation

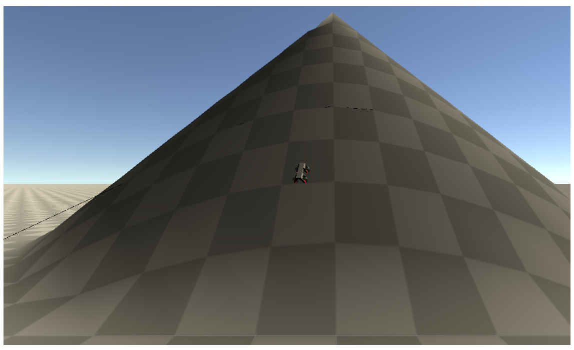
3.1 Terrain Generation

The terrain game object in Unity allows for a variety of ground surfaces to be generated, enabling multiple tests of the robot line following algorithm. The X and Z axes of the terrain are specified to give the surface a width and length, and a maximum Y axis height value is also specified. The width and length values must be powers of 2. A list of floats in the range [0,1] is then generated to give the height at each [X,Z] coordinate, and each value of that list is a percentage of the maximum height value[6]. For example, if the maximum height value is given as 10, and the height value specified at an [X,Z] coordinate is 0.5, then the height value will be 5 at that point. A number of height generators can then be created to enable a variety of tests to be conducted. These include:

- A randomly bumpy surface using a Perlin noise filter with a specified noise scale (Figure 7)
- A cylindrical wall that resembles a steel pipe (Figure 8)
- A simple linear slope in x and z (Figure 9)



7 Terrain surface with a Perlin Noise height map (Left) with 100 noise scale (Right) with 1000 noise scale

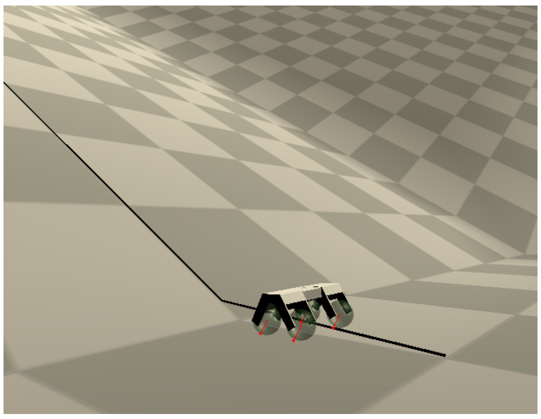


8 Terrain surface with a cyclindrical magnetic wall

Relevant script: TerrainGenerator.cs attached to Terrain

3.2 Line Renderer

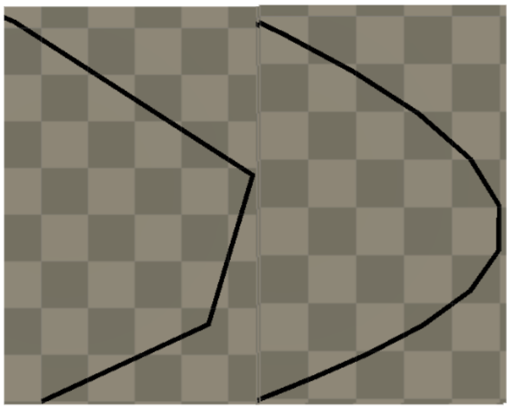
Once the surface is generated, a line for line following purposes must also be generated. One way of doing



9 Terrain surface with a 30 degree slope in X and 30 degree slope in Z

this is by using the terrain 'Paint' method to spray a black line on the terrain surface. However, this is done manually and lacks precision, so instead a Line Renderer object can be created. The Line Renderer object takes an array of two or more points in 3D space, with a straight line generated between each point [3]. Once these lines are specified, a smoothing algorithm is then used to create a smooth interpolated line passing through the points. Figure 10 shows how the smoothing algorithm changes the line, allowing for either sharp or smooth corners to be created easily. From each point on the line renderer, a ray cast is then used to find what height the point should be at so that the line follows the bumps of the terrain surface.

Relevant script: LineFollowingPIDV2.cs attached to object 'Main'



10 The same line defined with a smoothness of 1 (left) and 10 (right)

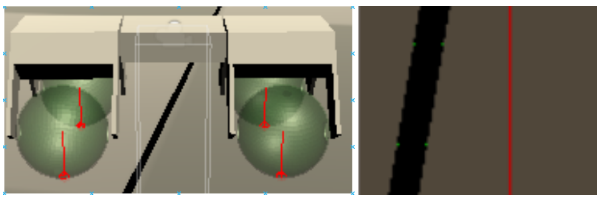
3.3 Machine Vision

In order to simulate a real world line following task, the robot is given a Camera object attached at the robot centre and facing directly downwards. The camera used in this simulation is orthographic, meaning the size of objects does not change with depth, so that the mapping from pixels to simulation world distances can be easily calculated. The camera is given: an aspect ratio a that specifies the ratio of image width to height; an orthographicSize O_{size} value, specifying half the vertical viewing volume of the camera [1], and an Image size I that is a power of 2 to create a square image. The HeightFactor H relating image height to simulation distance is then given by $H = (2O_{size}/I)$, and WidthFactor W relating image width to simulation distance is given by $W = Ha$.

Instead of rendering the camera image to the game display, it is instead rendered to a Unity RenderTexture object, which will update every frame. This is a costly process mathematically, since it involves sending the image from GPU to CPU to read the pixels, before sending it back to have the Render Texture render again. As the image size is increased, the process slows down drastically and will result in noticeable frame rate effects. An alternative is to use AsyncGPUReadback[7] in the call to get image pixels, which will result in some frame latency but will not slow down the GPU-CPU pipeline. This will allow image size to be increased further without drastic frame rate effects, which is desirable as image size effects the resolution of distances that can be measured for line distance error.

The left and right edge of the line is detected at two points, as shown in figure 11, from which the distance from the middle of the image and the line is found, as well as the angle between vertical and the line. In figure 11, on the left, the white lines show the viewing box of the orthographic camera. These values are extracted to be used in the Line Following task presented below.

Relevant script: ImageProcessing.cs attached to object 'CV Camera'



11 How the machine vision image is processed

4 Line Following

4.1 Task Description

The Line Following task presents a simulation of the task required of the robot when accurately following a weld line on a surface. Figure 12 depicts the geometry of the task. To simplify the problem, we can assume that the desired path is a horizontal line and the error vector will then be $e = [d_{err}, \Psi_{err}]^T$. [11]

We then create a linear controller for the robot, commanding a rotation value $\omega = Ke$ where $K = [K_d, K_\Psi]$.

By considering linearized vehicle dynamics and applying the small angle approximation for Ψ_{err}, we get a characteristic equation in the LaPlace domain of:

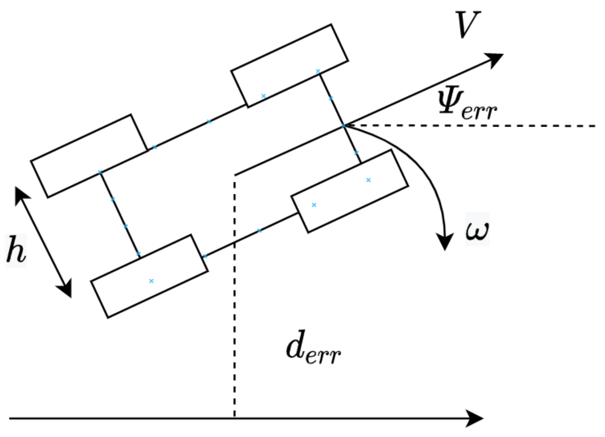
$$s^2 - K_\Psi s - VK_d = 0$$

Which we can interpret in terms of the generic second-order system response, $s^2 + 2\zeta\omega_n s + \omega_n^2 = 0$ where ζ is a damping term and ω_n is the natural frequency.

The linear controller can then be optimized through a number of experiments that measure the distance error over time when the robot starts with some initial error e_{init}, and then treating the system as a step response, which can be characterised using the following values:

- Rise Time - Time it takes for the error to fall from 90% to 10% of the initial error
- Settling Time - Time it takes for the error to fall within 2% of the initial error
- Overshoot - Percentage overshoot past the desired line
- Steady State Error - The average error after the system has settled

These values characterising the system can then be optimised by varying the ζ and ω_n controller values.

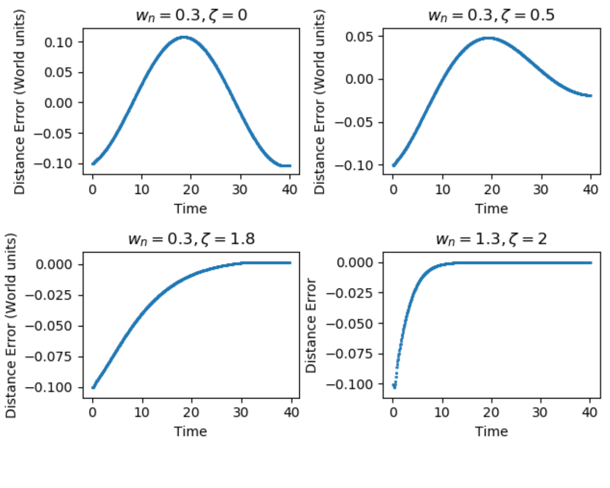


12 Line Following Task Geometry

4.2 Data Collection & Presentation

When a simulation has finished after a set amount of time, an excel sheet is created with the raw distance error, speed error, average rotation rate of the wheels, and the X and Z position of the robot over time. Two other excel sheets are also creating, containing the simulation settings and step response characteristics (rise time, settling time etc) for the distance error and speed error. Figure 13 shows the results from an experiment run on a straight line on a flat surface, where the robot starts with an initial error d_{init}. The robot distance error d_{error} is plotted over time. The simulation can be used to optimise the natural frequency and damping term of the distance controller, by analysing the overshoot and rise time and how these two values change as ζ and ω_n are varied.

Relevant scripts: DataSaver.cs attached to object 'Main'



13 Distance error as damping term is varied on a flat surface

4.3 Data Analysis

The damping and natural frequency terms for the line following linear controller behave as expected, with the damping term required to prevent the robot from overshooting and oscillating around the target line, while the natural frequency term effects how quickly the robot reaches the target line. Due to expected differences between the simulation and the real world physics of the robot line following problem, the simulation should not be used to find the exact values to be used on the real robot. However, by comparing the relative behaviour of the robot when different damping or natural frequency values are used, or if a completely different line following control scheme is used, one can obtain some intuition for the expected behaviour of the real world robot when autonomous control is added

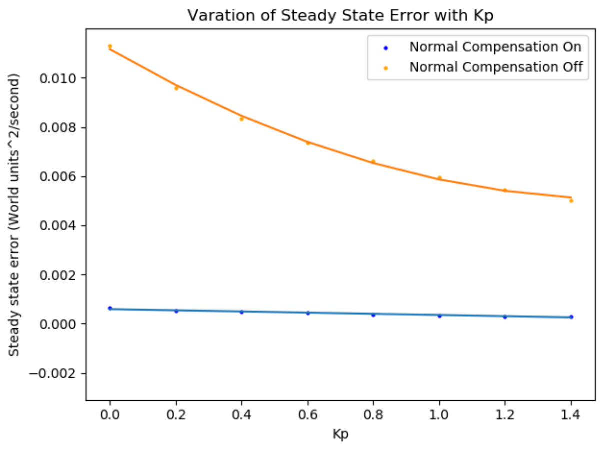
5 Velocity Control

5.1 Task Description

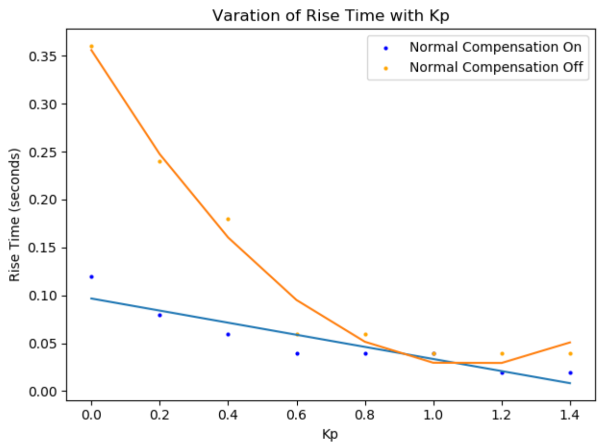
The welding task not only requires the robot to follow a desired line accurately, but it also requires the robot to maintain the constant velocity of 200-300 mm/min needed when welding a joint. If the robot does not accurately control velocity, then a separate PID controller would be needed for the welding arm, adding further complexity to the system. Velocity can be directly measured from the motor, and a PID controller can be used to maintain some target velocity. The task is characterised similarly to the distance controller, with the step input from zero velocity to the target velocity given and the robot velocity error over time measured, allowing the PID controller to be optimised.

5.2 Data Collection

The steady state error and rise time for the step response of robot speed is shown in Figures 14 and 15, with normal compensation (as described in section 2.4) On and Off compared as the Proportional value of a PID value is varied. This experiment is run on a randomly bumpy terrain surface, where the robot is tasked with reaching and maintaining some target speed V_{target}.



14 Steady State Error as Kp is varied, with normal compensation on and off on a bumpy surface



15 Rise Time as Kp is varied, with normal compensation on and off on a bumpy surface

5.3 Data Analysis

This experiment reveals the effectiveness of a method for normal compensation, even when a well-optimised PID speed controller is being used. Since normal compensation is purely based on the geometry of the problem, it does not require any further experimentation to optimise, like the PID controller does. Since the magnets in the wheels of the real robot will always follow the normal of the surface, the α angle used to calculate $R_{corrected}$ can be found with a sensor that tracks the angle of the magnet relative to the original downwards position, such as a linear potentiometer attached to the centre of the wheel.

6 Future Work

6.1 Comparing Simulation with Real Experiments

In order to ensure that the simulation is giving reasonable and useful results, simple experiments should be run with the real robot to compare with the performance in the simulation. For example, the real robot, attached with a camera to observe the ground, could be given a line following task with the same original distance error as a simulation. By comparing the expected results in the simulation with the actual obtained results, the simulation can be further refined.

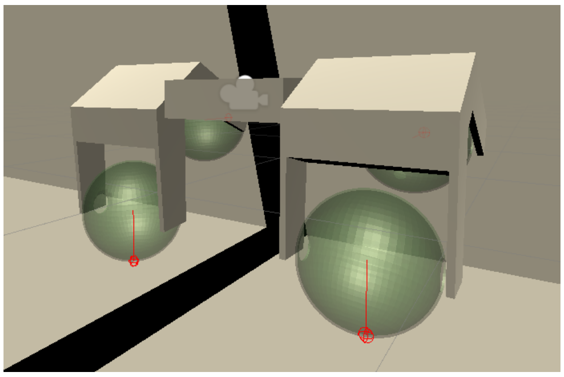
6.2 Line Following on Walls

The simulation can be further extended by analysing the robot behaviour when line following on vertical magnetic walls. So far wall climbing has been implemented by simulating a LIDAR sensor on the robot front, detecting any surfaces in front of the robot that is in the 'Terrain' layer. Once a wall is in contact with a robot wheel, a separate control scheme can be applied so that the robot corrects its position, aligning both front wheels with the wall, and then transitions on to the vertical surface and continues to follow the line. The robot mid-transition is shown in Figure 16. Further work is needed to design a robust control scheme to deal with different wall transitions and to ensure that the line following task is still completed accurately.

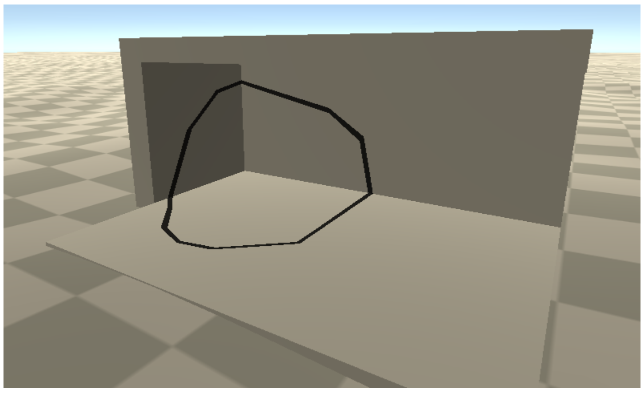
An example of such a test can be seen by looking at the 'Magnetic Wall' object in Unity, as shown in figure 17. This is an object created in Blender and coloured to have black lines on a white surface. Once imported into Unity as an FBX, a small script 'MakeShader' must be attached to the object to ensure that the colours are visible to the robot Image Processing camera. Similar objects to this can be created in Blender, allowing multiple extra tests to be created

6.3 Path Planning

For real life use, in order to create a robot capable of purely autonomous control, a control scheme may



16 Robot during the transition phase from ground to wall



17 Custom object created in and imported from Blender for wall climbing tests

be required when the robot is finishing one weld and having to relocate to start another weld. A path planning algorithm, such as A*, may be used to find the shortest distance to that point while avoiding obstacles detected using LIDAR.

7 Acknowledgements - 謝辞

最後に、適切な指導を賜った指導者の衛藤さんに感謝いたします。また、ロボティクス技術グループのみなさんに感謝いたします。様々な指導を賜った先輩の鈴木さん、サラムさん、明石さん、羽根田さん、山崎さんにも厚く御礼を申し上げ、感謝の意を表します。

References

[1] Unity Documentation. *Camera.orthographicSize.* URL: `https://docs.unity3d.com/ScriptReference/Camera-orthographicSize.html`. accessed: 01.03.2020.

[2] Unity Documentation. *HingeJoint.useMotor.* URL: `https://docs.unity3d.com/ScriptReference/HingeJoint-useMotor.html`. accessed: 01.06.2020.

[3] Unity Documentation. *Line Renderer.* URL: `https://docs.unity3d.com/Manual/class-LineRenderer.html`. accessed: 03.06.2020.

[4] Unity Documentation. *MeshCollider.* URL: `https://docs.unity3d.com/ScriptReference/MeshCollider.html`. accessed: 01.03.2020.

[5] Unity Documentation. *Physics.SphereCast.* URL: `https://docs.unity3d.com/ScriptReference/Physics.SphereCast.html`. accessed: 05.04.2020.

[6] Unity Documentation. *Physics.TerrainData.SetHeights.* URL: `https://docs.unity3d.com/ScriptReference/TerrainData.SetHeights.html`. accessed: 15.05.2020.

[7] Unity Documentation. *Rendering AsyncGPUReadback.* URL: `https://docs.unity3d.com/ScriptReference/Rendering.AsyncGPUReadback.html`. accessed: 03.06.2020.

[8] Unity Documentation. *Rigidbody.angularDrag.* URL: `https://docs.unity3d.com/ScriptReference/Rigidbody-angularDrag.html`. accessed: 30.05.2020.

[9] Unity Documentation. *Unity Real-Time Development Platform.* URL: `https://unity.com`. accessed: 01.03.2020.

[10] E. Haruhiko and H. Asada. "Development of a Wheeled Wall-Climbing Robot with a Shape-Adaptive Magnetic Adhesion Mechanism". In: *2020 International Conference on Robotics and Automation* (2019).

[11] W. Newman. *Systematic Approach to Learning Robot Programming with ROS.* International series of monographs on physics. CRCPress, 2018, pp. 332–333.

[12] Robotis. *XM540-W270.* URL: `http://emanual.robotis.com/docs/en/dxl/x/xm540-w270/`. accessed: 15.03.2020.