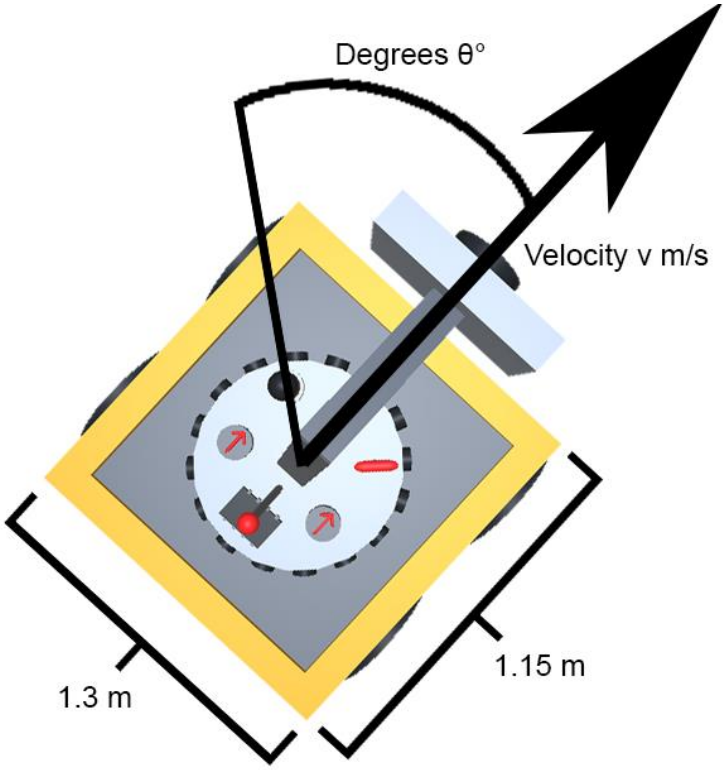
Vehicle Physics



Abstract

The Unity game engine that the AVL is built on top of has fantastic physics simulation capabilities that can meet a variety of needs. This includes gravity, wind resistance, Rigidbody collisions, wheel rotation dynamics, and raycasting.

However, ultra-realistic physics can dramatically complicate control problems for autonomous vehicles. For this reason, much of this physics simulating functionality has been disabled or its effects otherwise removed from the entities that operate in the assignment scenarios. This was done to simplify the problems for new students, as well as ensure a level of complete control over the environment and the Ego vehicle that one would not normally have.

As a part of this, the “PhysicsBody” class has been created to handle the physics calculations of the Ego vehicle. This class replaces the functionality of the traditional Rigidbody class that is typically used in a Unity project, and offers a level of guaranteed control regarding the state of the Ego vehicle that the Rigidbody class does not. The trade-off is that the physics system modeled by the PhysicsBody is much more rudimentary and inaccurate than the Rigidbody class. This is considered an acceptable price, however in the future Rigidbody support may be added for assignment scenarios of higher difficulty.

Collisions

At present time, there is no support for collision detection or realistic collision behavior from the Ego vehicle. When the Ego vehicle “collides” with an obstacle, it will move directly through the obstacle as if it wasn’t there. This was determined to be an acceptable loss of accuracy, as student-created control systems should not be collided with any objects in the first place. The goals of the current assignment scenarios is to avoid collisions completely, not try to rectify a collision once it has occurred.

Control systems designed to increase the safety of passengers during an unavoidable collision is a fascinating research topic, and support for such assignment scenarios may be added in the future, however for the time being such scenarios are considered beyond the scope of the current coursework involving the AVL.

Position Updating

At the start of every frame, the physics body updates the position of the Ego vehicle. It does so according to the following pseudo-code:

```
UpdatePosition(current_position, forward_direction, frame_velocity):
	offset = forward_direction * frame_velocity

	new_position = current_position + offset


	return new_position
```

The current_position and forward_direction variables are vectors, and frame_velocity is a scalar. The current_position variable contains the current position of the Ego vehicle, and the forward_direction variable contains the unit vector which points in the direction that the Ego vehicle is facing. A unit vector has a magnitude of 1, so when it is multiplied by a scalar number, then the result vector will have a magnitude of equal to the original scalar.

When the forward_direction is multiplied by the frame_velocity, the resulting vector gives the amount of offset necessary from the current position to a new position in the direction that the Ego vehicle is facing. The frame_velocity is not the actual velocity, as the actual velocity is given in meters-per-second, and there are 8 frames every second. Thus the frame_velocity is equal to the actual velocity divided by 8.

The new position is then calculated to be the current position plus the newly calculated offset.

There is no momentum in the PhysicsBody. Thus the Ego vehicle will move towards whatever direction it is facing, regardless of what the direction it faced in the previous frame.

Velocity Updating

The AddSpeed(float) function of the PhysicsBody increases/decreases the velocity by the given amount that is passed as an argument. If the new velocity has a larger magnitude than the max speed, then the new velocity is truncated to be the max speed.

Once during every frame, after the position of the Ego vehicle has been updated, drag is applied to the velocity. This drag is calculated as a simple multiplication of the velocity with a number that is less than 1. If no new speed is added using the AddSpeed(float) function, then the Ego vehicle will eventually slow to a stop after a few seconds. This is why the accelerator device continually maintains a desired speed – it keeps adding a speed amount to the PhysicsBody to ensure that student-created tasks within the RTOS do not need to babysit the velocity.

Rotation Updating

The Rotate(float) function applies a rotation to the Ego vehicle based on the value passed as an argument. This rotation amount is measured in degrees, and if a value outside the bounds of [-360, 360] is given then Unity will automatically wrap the rotation around. Thus -360, 360, and 0 will all produce the exact same final orientation of the Ego vehicle.

As previously mentioned, there is no momentum in the PhysicsBody. Thus when then the Ego vehicle rotates to a new orientation, the position update in the next frame will immediately move in the new forward-facing direction without any sliding. There is also no rotational momentum nor is there rotational drag.