BlenderSim Continuity Document
2013.06.01 to 2013.08.17

David Lettier

1 Trials & Tribulations

1.1 Physics Engine

Over the course of three months, a Blender based simulation engine was developed for HRTeam. Initially, *BlenderSim* was wholly physics based but soon proved to be problematic on three fronts: time, scale, and intricacy.

Initial problems arose when the treads of the *SRV-1* were recreated in the simulation. Even after numerous hours adjusting physics parameters and rigid-body configurations, the treads would consistently behave in erratic fashions. See *Figure 1*.

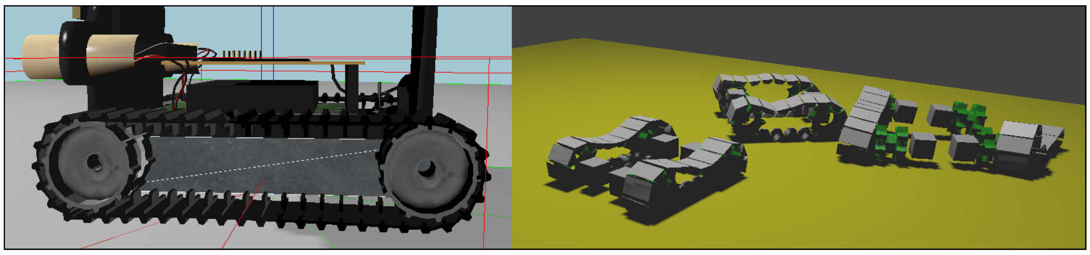

Figure 1: Here you see the to-scale treads modeled after the physical *SRV-1* robot on the left and the physics-based rigid-body tracks in motion on the right.

Scale was problematic as the Blender/Bullet physics engine has difficulty with collisions of objects outside of its intended range of .05 to 10 meters. Non-moving objects smaller than .05 (5cm) Blender/Bullet units in any given dimension erratically bounce when they should be still as no Newtonian force is being applied. For instance, the wheels on the 3D *SRV-1* model of which are 2.11cm x 2.45cm x 2.52cm.

To rectify these issues, the physics engine was largely abandoned–in terms of providing the motion model–and was only kept to keep the robots from running through each other and the arena. In its place, a constant linear and angular velocity motion model was developed of which only moves the 3D robot as if it was a single point body. See *Figure 2* and *Figure 3*.

To increase the fidelity of *BlenderSim*, a probabilistic motion model was discussed. The probabilistic motion model would be generated by data mining previous physical experiments where robot locations were logged over time. Once in place, the probabilistic motion model would be used in a genetic algorithm fitness function. Here each generational offspring of a simulated physics-based robot's movements would be compared with that of a of simulated probabilistic-based robot's movements given some time, initial orientation, and some location both robots must reach.

1.2 Asyncore versus ServerSocket

Originally *Asyncore* was used in both *robot_*_client.py* and *robot_*_server.py* to handle the TCP/IP link that feeds waypoints from the client to the server of which waits for a client connection in the simulator. However, *Asyncore* proved problematic as it was scaled to four robots. It was difficult to control the *asyncore.loop* from within *robot_*_server.py* and ultimately the Blender Game Engine loop. The *asyncore.loop* was ran in its own thread (in *robot_*_server.py*) of which caused the port allocated to stay open even when the Blender Game Engine was exited (but not Blender itself). This required the simulator operator to completely exit and restart Blender in order to run another simulation as the port allocated (and subsequently the *asyncore.loop* thread) was linked to the Blender process. There were also multithreading race conditions affecting the simulation where the robots (controlled by *robot_*_controller.py*) wouldn't receive the correct order of waypoints from their respective servers.

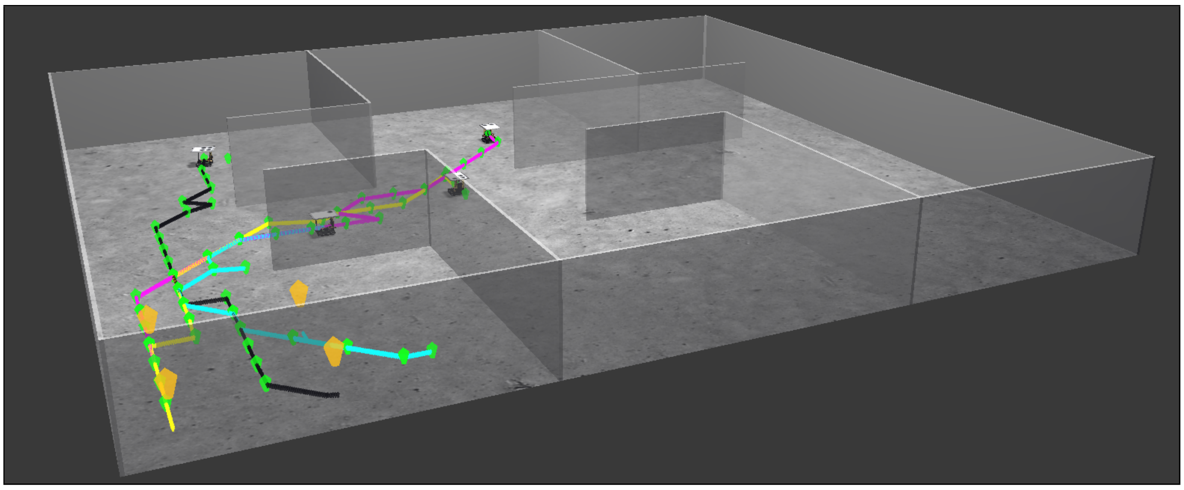

Figure 2: Here you see the constant velocity motion model at work in *BlenderSim* with four robots traversing their respective paths of which consist of way-points scrapped from a previously recorded physical lab experiment.

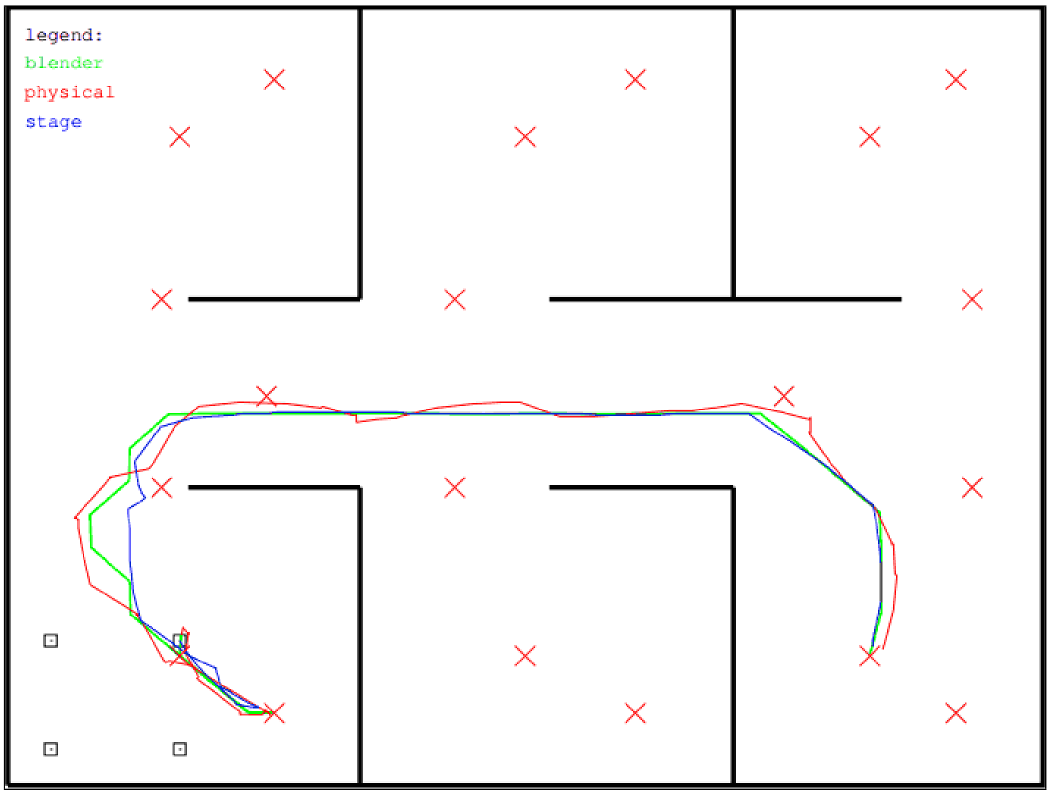

Figure 3: Here you see the path plots–as plotted by Dr. Elizabeth Sklar–of the *BlenderSim* robot, the physical robot, and the *Stage* robot plotted in comparison.

To remedy these issues, *robot_*_server.py* was rewritten with *ServerSocket*. *ServerSocket* allows for greater control over its serve loop and threading capabilities. And as an added side bonus, it also reduced the code base making it easier to follow.

Note though that *Asyncore* is still utilized in *robot_*_client.py* as it scaled well and didn't exhibit any errors while running the simulation client-side.

2 Running BlenderSim

As of this writing, *BlenderSim* is located on *GitHub* at *https://github.com/lettier/blendersim/*.

2.1 Preparing the Log Files

- As of this writing, *BlenderSim* reads scrubbed log files of which contain only waypoint and campose lines from a master historical log file from some previously run physical simulation.
- The scrubbed log files are located in *./waypoint_logs*. The log files must be named *1.log*, *2.log*, *3.log*, and *4.log*. *1.log* corresponds to *robot_1*, *2.log* corresponds to *robot_2*, etc. in the simulator.
- These scrubbed log files are read by *log_reader.py* of which grabs all the waypoints (x,y) from the first set of (x,y) points on any given waypoint line in the log file.
- These points are then in turn fed to *robot_*_client.py* of which then feeds the waypoints to *robot_*_server.py* via a TCP/IP connection.
- The log files that currently reside in *./waypoint_logs* were created from *PHYS_CS_Auction_A_1_3.log*. *1.log* refers to blackfin-17, *2.log* refers to blackfin-18, *3.log* refers to blackfin-16, and 4.log refers to blackfin-15. The reason being the robots' respective standard starting points from *PHYS_CS_Auction_A_1_3.log*. See *Figure 4*.

To generate *1.log*, *2.log*, *3.log*, and *4.log* *cd* to *./scripts/* and run:

```
$ python waypoint_log_populator.py <historical_physical_experiment_log_file>
```

For example:

```
$ python waypoint_log_populator.py ../archive/PHYS_CS_Auction_A_1_3.log
```

Expected output:

```
[WAYPOINT_LOG_POPULATOR.PY] Looking for robots in master log file: ../archive/PHYS_CS_Auction_A_1_3.log
[WAYPOINT_LOG_POPULATOR.PY] Writing out waypoint log file: ../waypoint_logs/1.log
[WAYPOINT_LOG_POPULATOR.PY] Writing out waypoint log file: ../waypoint_logs/2.log
[WAYPOINT_LOG_POPULATOR.PY] Writing out waypoint log file: ../waypoint_logs/3.log
[WAYPOINT_LOG_POPULATOR.PY] Writing out waypoint log file: ../waypoint_logs/4.log
```

The resulting *1.log*, *2.log*, *3.log*, and *4.log* will be located in *../waypoint_logs/*.

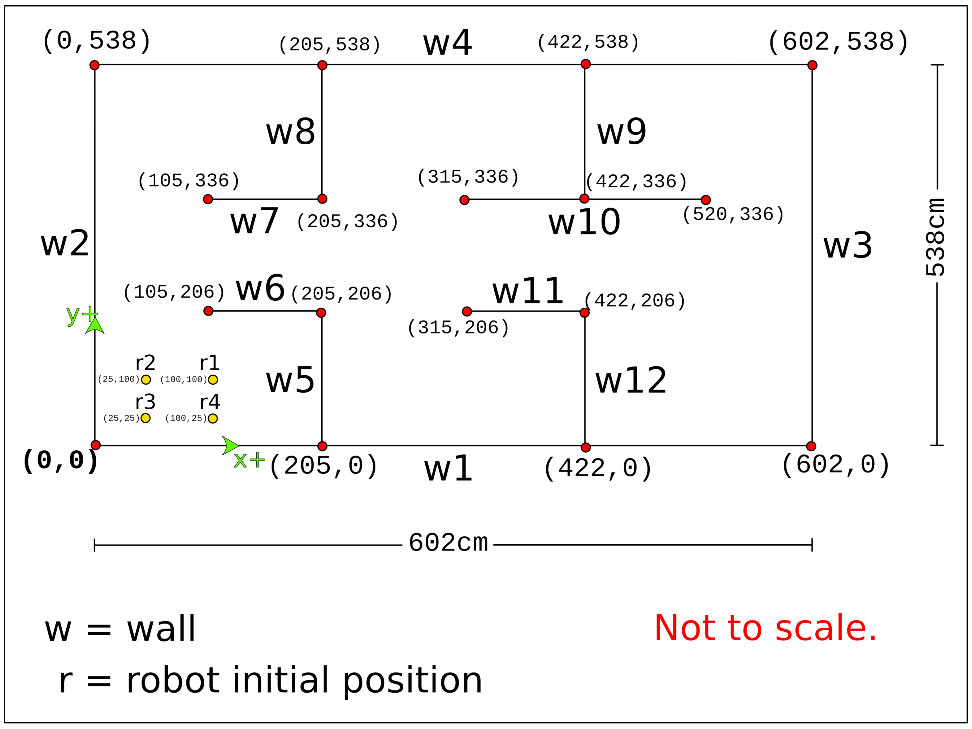

Figure 4: Here you see the arena map (not to scale) with wall coordinates and robot starting locations.

2.2 Starting the Game Engine

- To run the simulation, either enter via the command line [**simulator_1**]**$ blender way_point_comparison.blend** or double click *way_point_comparison.blend*.
- Orient the 3D view as you wish and press the *p* key. The simulator is now running and awaiting the client connections.
- Controls for *BlenderSim* are:
  - Non-keypad *1*, *2*, *3*, *4*, *5*, and *6* correspond to the overhead cameras in the arena.
  - Non-keypad *7*, *8*, *9*, and *0* correspond to the onboard first person view cameras of the *SRV-1*s.
  - *a*, *b*, *c*, *d*, and *e* correspond to the preconfigured task point configurations found in *./task_points_configurations/*.
  - *Esc* exits the simulation and preps it for another run without having to close Blender itself.

2.3 Running the Client

- *cd* over to *./scripts/* and run from the command line [**scripts**]**$ python robot_clients.py**.

2.4 Resulting Simpose Logs

- As the simulator runs, resulting *simposes* for each robot will be recorded to *./robot_logs/1.log*, *./robot_logs/2.log*, *./robot_logs/3.log*, and *./robot_logs/4.log*.
- Each file is *CSV* formatted to be opened in a spreadsheet application.

3 How BlenderSim Works

3.1 Servers

- The servers are expressed, in the simulation, as the amber colored *gems* located just above the 3D modeled *SRV-1*s. See *Figure 5*.
- *robot_*_server.py* contains all of the logic governing over the *BlenderSim* servers corresponding to the four robots in the simulation.
- Server logic utilizes the *ServerSocket* Python module.
- Each of the four servers run in separate threads parented to the Blender process.
- *robot_1_server.py* listens on *port 5001*, *robot_2_server.py* listens on *port 5002*, *robot_3_server.py* listens on *port 5003*, and *robot_4_server.py* listens on *port 5004*.
- As a connection is made to a client, the server performs a short handshake and begins requesting waypoints from the client.
- The waypoints are put into a thread safe queue for the robot_*_controller.py to read from at its leisure.
- Once the client notifies the server that there are no more waypoints, server puts *done* in the waypoint queue, sends *done* to the client, shuts down the port, and its thread is terminated.
- If the Blender Game Engine is terminated before the servers receive all of their waypoints, the servers shut down their ports and their threads are terminated. This allows the simulator to be started again fresh (no [*Errno 98*] *Address already in use.* errors) without having to close Blender itself in order to run another simulation.

3.2 Clients

- *robot_***1-4***_client.py* connect to ports *5001-5004* respectively.
- Utilizing *log_reader.py*, the clients read in their respective waypoints from *./waypoint_logs/***1-4***.log*.
- Once connected to their respective servers, the clients perform a short handshake and then begin trasmitting waypoints as each of their respective servers request them.
- Once the clients run out of waypoints to transmit, they signal the servers that there are *nomore* waypoints.
- Once they receive *done* from their respective server, they close the connection and terminate.
- Out of convenience, *robot_clients.py* runs *robot_***1-4***_client.py* in four separate asynchronous sub-processes utilizing the *Subproccess* Python module.

3.3 Robot Controllers

- The robot controllers are expressed, in the simulation, as the metal box bases of the *SRV-1* 3D models. See *Figure 5*.
- *robot_***1-4** are controlled via the logic contained in robot_*_controller.py.
- No robot will move while their respective waypoint queues are empty.
- Once their waypoint queues begin being populated, by their respective servers, the robots will traverse a linear path to each waypoint first by turning to face the waypoint at a constant velocity and then by moving forward towards the waypoint at a constant linear velocity. Velocities were calculated from a historical physical lab experiment log.
- Once the robots detect *done* in their waypoint queue, they will cease movement. This indicates that they have traversed the complete *A** previously-calculated-path as read from the historical physical lab experiment master log.

3.4 Task Points Manager

- The task points manager is expressed, in the simulation, as the giant torus located above the 3D modeled arena. Once the Blender Game Engine is started, it hides itself. See *Figure 5*.
- The logic is contained in *task_points_manager.py*.
- At the start of the simulation, the task points manger reads the task points configurations from the *./task_points_configurations/* directory. The individual task points configurations must be named *a.conf*, *b.conf*, *c.conf*, *d.conf*, and *e.conf*.
- Once it detects keys **a**-**e** are pressed, it will place the task points in the 3D modeled arena as small red toruses. See *Figure 6*.

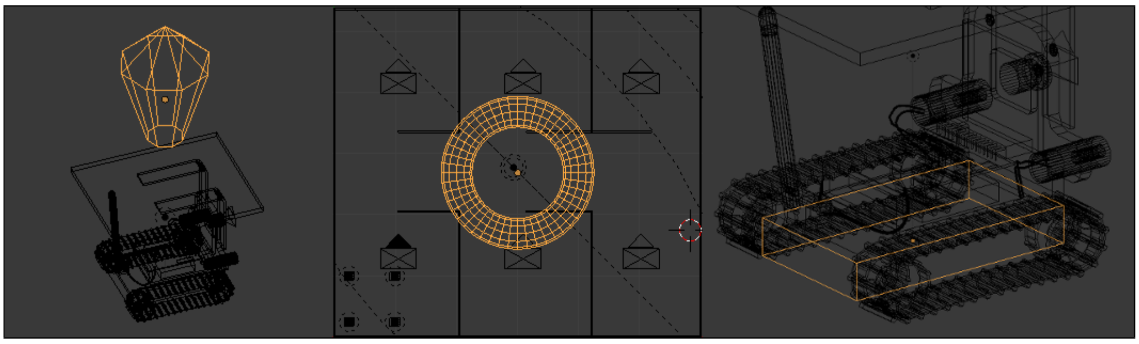

Figure 5: Here you see from left to right, the server *gem*, the task points manager, and the robot controller manifestations in the simulation.

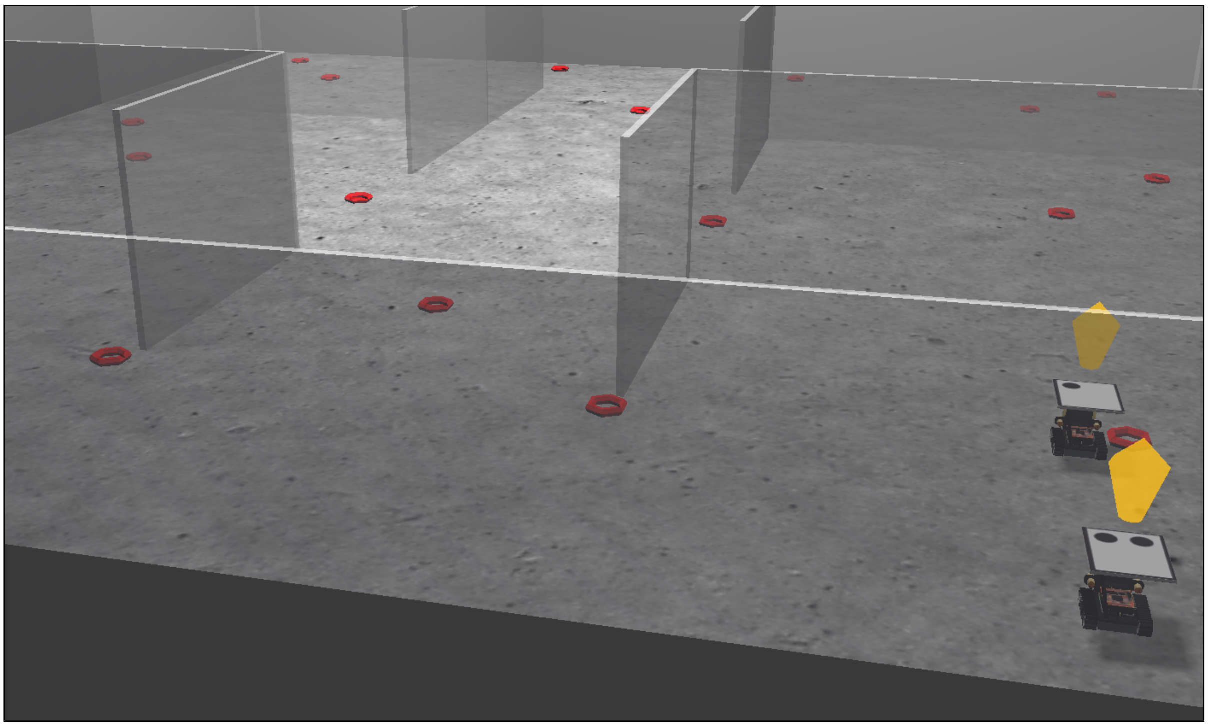

Figure 6: Here you see the red torus task points laid out in the arena.

4 References

[1] Bullet Physics Engine Scale
[2] Blender Python API
[3] Python Socket Programming HOWTO