Using the Unity Game Engine to Develop SARGE: A Case Study

Jeff Craighead, Jennifer Burke, and Robin Murphy

Abstract— **This paper discusses the implementation of the Search and Rescue Game Environment (SARGE) using the Unity game engine. The paper will explain the key benefits of using Unity versus other popular platforms and how the various components of the Unity API and environment editor are used to create SARGE. This discussion is divided into sections covering the various robots, sensors, environments, and user interfaces in SARGE.**

I. Introduction

With robot simulation becoming popular again it is important for the robotics research community to be able to focus on the important and interesting part of their work, such as building real robots and algorithms, not building robot simulators. It it the results of the simulations that we are after. This leads to the question addressed in this paper, "What simulation engine is easiest to use to accomplish a given simulation task?" In the past many researchers have chosen from the available commercial or open source game engines such as Unreal [1], [2], [3], while others have used purpose built robot simulation engines such as Stage, Gazebo [4], [5], [6], [7], and Webots [8], [9], [10], and others still have used math packages such as Matlab [11], [12], [13], [14]. Each type of engine has plusses and minuses and choosing the right package depends heavily on the accuracy and desired output of the simulation and tends to be a very project specific choice. The Search and Rescue Game Environment (SARGE), a distributed, multi-player, robot operator training game and robot simulator [15], [16], [17], was initially based on the Unreal2 engine as an extension of USARSim, however due to complications with Unreal2, SARGE development was moved to the Unity engine [18].

Jeff Craighead, Jennifer Burke, and Robin Murphy are with the Institute for Safety, Security and Rescue Technology at the University of South Florida. 4202 E. Fowler Avenue, Tampa, FL, USA. {craighea,jlburke4,murphy}@cse.usf.edu. This work was sponsored, in part, by the US Army Research Laboratory under Cooperative Agreement W911NF-06-2-0041. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the ARL or the US Government. The US Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright notation hereon. All aerial photography provided by GoogleEarth.

The goal of the SARGE project is to create a video game that requires players to master the skills necessary to operate several types of robots that are typically used in search and rescue and law enforcement. This includes the iRobot Packbot, the Inuktun VGTV, and a rotary-wing UAV. Additionally online play requires players to work in teams to conduct a search of a disaster area. The game will track a player's performance over multiple play sessions and provide them with in-game feedback on their skill improvement. The Unity engine has made development of SARGE simple and has allowed the developers to concentrate on content development instead of worrying about integrating various open source physics and rendering components or fighting with buggy environment editors. This paper will provide an outline of the benefits of using Unity versus other simulation engines, then provide a detailed description of how SARGE is implemented using the engine.

II. Related Work

There are a large number of robot simulators available, as mentioned in Section I, including USARSim, Webots, and Player/Stage robot simulators as well as Matlab. This section will focus on the key features of these popular simulators and discuss why we decided to create another from the ground up.

USARSim [19] made its debut in 2003, making it one of the first high fidelity, open source, 3D robot simulators available. The benefits were numerous to USARSim, it was cheap (it was based on the best game engine at the time, Epic's Unreal2 engine) and USARSim was created as a mod which meant that the cost for developers was the same as users (the price of the game UnrealTournament). The Karma physics engine was high fidelity for the time and allowed developers to focus on art and code development instead of integrating a custom physics engine with a rendering system. However, Unreal2's world editor UnrealEd is buggy (the editor will often crash, causing the developer to loose work) and cumbersome to use. Additionally the documentation for Unreal2's API is seriously lacking. There is no official source for documentation of the Unreal API for modders, only a few wiki-like documents

that are maintained by the community with no input from Unreal's developers. This makes development of new code unnecessarily difficult. At the time Unreal was the best choice and the benefits outweighed the faults and in fact the benefits are the same as we are claiming for Unity today. The problem with continuing to use Unreal2 is that now there are better choices available, including Unreal3 which still suffers from a lack of documentation.

Webots [8], currently late in the development cycle of version 5, combines the high fidelity Open Dynamics Engine (ODE) with a VRML-like rendering engine and provides a C/C++ and Java API. Robots are developed by adding nodes to a hierarchy, some of which define the shape of the robot, while others define sensor and actuator locations. Environments are constructed in a similar manner. While the ODE provides Webots with a high fidelity physics simulation, the VRML rendering system is not sufficient to reproduce real-world locations and objects with sufficient detail for high fidelity operator training.

Gazebo [4] Gazebo is a 3D environment simulator which is part of the Player/Stage project. Gazebo also uses ODE as a physics engine. Gazebo uses the open source Ogre rendering engine, which is used in a variety of commercial projects and can provide a high quality visuals. However, Gazebo robots and environments are created by hand coding XML files, which limits the possibilities for visually and physically complex scenes and robots to more VRML-like structures.

Matlab [11] is extremely popular for robot simulation, particularly because the available Matlab toolboxes allow users to do rapid prototyping of robot control systems. However Matlab does not provide any type of physical environment simulation system, so each user must create their own. This has produces many 2D and 3D graph based visualizations for robot simulations because plotting graphs is simple in Matlab. Matlab does have a VR toolbox which is a VRML rendering environment, so while this is sufficient for evaluation of low level controllers and some high level algorithms, the lack of high quality visualization makes Matlab insufficient for operator training purposes.

The robot simulators discussed in this section are all popular in current robotics research, yet all suffer from some deficiency that makes them insufficient for operator training work. Unreal2 was cheap and high fidelity for its time, however there are several better choices today. Unreal2's rendering engine and environment editor is still superior to the other simulators discussed, however the Karma physics engine that Unreal2 uses is no match for ODE and is probably on par with the average Matlab simulation. None of these simulators are developer friendly, they either require hand coding of environment layouts, are lacking necessary documentation, or in the case of Matlab are a well documented blank slate with no physics or rendering capabilities. For this reason other engines were investigated and the Unity engine seems to be the best solution for a high fidelity, developer friendly, cost effective simulation environment.

III. The Unity Engine

The Unity game engine is developed by Unity Technologies in Denmark. Unity integrates a custom rendering engine with the nVidia PhysX physics engine and Mono, the open source implementation of Microsoft's .NET libraries. The benefits of using Unity are many when compared to the engines discussed in Section II. This section provides a breakdown of what we consider to be the key features of Unity that make it an excellent robot simulation engine.

Documentation. The Unity engine comes with complete documentation with examples for its entire API. This is the biggest benefit of Unity and leads to increased productivity when compared to other engines such as Unreal or Source which only provide partial documentation for non-paying customers (mod developers).

Developer Community. There is an active on-line developer community which can often provide assistance for new users. The Unity Technologies developers also are very willing to add features to the engine at a users request, which will never happen if using a big-name engine such a Unreal. In fact several of the features existing in the Unity API are a result of requests from the SARGE developers.

Drag-n-Drop. Unity's editor is by far the easiest to use when compared to Unreal, Source, or Torque. Content is listed in a tree and is added to an environment in a drag-n-drop manner. Objects in the environment are listed in a separate tree, each of which can be assigned multiple scripts written in C#, a Javascript-like language, or Boo as well as physics and rendering properties. Script developers have access to the complete Mono API. Scripts can give objects interactive behaviors, create user interfaces, or simply manage information. Figure 1 shows a screenshot of the Unity Editor being used to develop a user interface for the iRobot Packbot.

Physics & Rendering. By using physics properties, objects can be given mass, drag, springiness, bounciness, and collision detection as well as be assembled using a variety of joints. The physics properties are simulated by nVidia's PhysX engine, which is used in many AAA

commercial games. The rendering properties include shader and texture assignment which affect the appearance of visible objects. Unity's custom rendering engine uses a simplified shader language which is compiled into DirectX 9 or OpenGL 2.0 shaders depending on the target platform.

Multiplatform Distribution. The Unity engine's editor runs on OSX, however applications created using Unity can be compiled for OSX, Windows, or as a Web-Player (which runs in a web browser via a plugin, similar to Adobe Flash). There are no restrictions on distribution of applications created with Unity and because applications created with Unity are not mods of existing games the end user does not need to own a copy of anything. Complete binaries can simply be distributed as the developer wishes.

Low Cost. The Unity engine has a relatively low cost for a complete game engine (although more expensive than the free open source engines). The Indie version of the engine is US $199 while the Pro version, which is required to publish for Windows is US $750 for an Academic license or US $1499 otherwise. This pricing is comparable to Torque, yet Unity's Editor is in our opinion much easier to use. While this is more expensive than modding an existing game, which usually only requires the investment in a copy of a game, it provides far more developmental freedom. If one were to use the Unreal or Source engines in a non-mod project, the cost for engine license and complete documentation would run well over US $300,000.

The remainder of this paper will discuss how Unity has been used to implement various pieces of SARGE, including the robots, sensors, environments, and user interface.

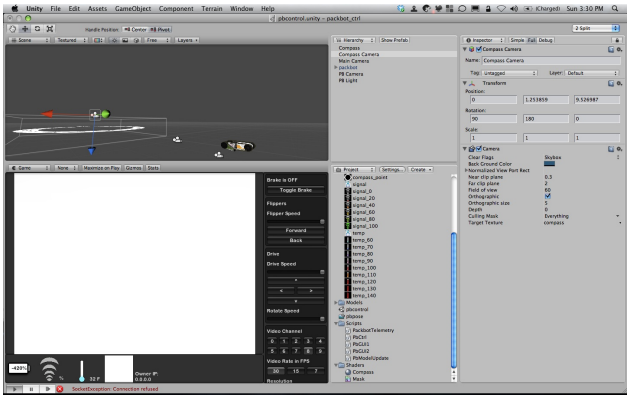

Fig. 1. A screenshot of an interface for a Packbot being developed in Unity.

IV. Robots

This section will describe the implementation of four SARGE robots using the Unity game engine. The robots available in the current version of SARGE include the iRobot Packbot Scout, the iRobot ATRV-Jr., the Inuktun VGTV Extreme, the USF SeaRAI, and the iSensys IP3. The Packbot and VGTV are both tracked ground vehicles and are implemented similarly using a custom tracked vehicle class, due to space limitations this paper will focus on the Packbot.

Each of the robots is modeled in Cheetah3D and consist of a hierarchy of objects. The root object of each hierarchy is the robot's largest body piece. Movable parts such as the Packbot's flippers or the IP3's rotor are parented to the root. Unity imports the native Cheetah3D files and maintains the object hierarchy which allows the movable components to be controlled individually through one or more scripts.

A. iRobot Packbot

Tracked vehicles are difficult to simulate. Typically tracks are simulated using many wheels[1] or using a sliding surface. SARGE uses a sliding surface to simulate the the tracks of tracked vehicles. At each contact point a force is applied in the opposite direction of the worlds gravity to keep the vehicle from falling through the surface it is resting on. To move the vehicle a force is applied at each contact point in the direction the track would move while the texture of the track is shifted to give the illusion that the track is turning. This is the typical approach used in video games which include tracked vehicles such as tanks. By applying a force at the contact points for each track, the vehicle responds similarly to the real Packbot. Drag forces are applied automatically by the physics engine to each track depending on the surface properties of the object the track is in contact with. Figure 2 shows a screenshot of the Packbot Scout model in SARGE. The flippers have the full 360° range of motion that the real Packbot flippers have and apply a force along the contact surface. The simulated Packbot currently has a single sensor, a standard video camera.

B. iRobot ATRV-Jr

The ATRV-Jr. model in SARGE makes use of Unity's built in wheel collider system. The body and wheels were modeled in Cheetah3D with the wheels parented to the body. Unity's built in wheel colliders are assigned to each wheel object in the hierarchy. The wheel colliders

[1]USARSim uses this method

take the wheel radius and friction properties as parameters. A control script is attached to the robot's body object which listens for commands from the user, the script then adjusts the power applied to each wheel. The sensors attached to the vehicle include the laser scanner, IMU, GPS, compass. Figure 3 shows the ATRV-Jr. as it appears in SARGE.

C. USF SeaRAI

The USF SeaRAI is an unmanned surface vehicle designed for inspection of port and littoral environments. This vehicle is shown in Figure 4. It consists of a main beam attached to two pontoons with two motors in the aft. The vehicle is steered using the motors. The simulated vehicle floats on objects with an attached buoyancy script which applies a buoyant force at the contact points based on the estimated volume of the submerged portion of the vehicle. This allows the simulated vehicle to run aground if it is steered too close to shore. The motors operate by applying a force at the point on the vehicle where the motor is attached. A control script attached to the boat translates user commands into motor forces.

D. iSensys IP3

The iSensys IP3 is an R/C helicopter based unmanned aerial vehicle. The simulated vehicle works by applying a motor force at the rotor head and at the tail rotor. The rotor head tilts with user input which adjusts the direction the lift force is applied. The force at the tail rotor is adjusted in the control script to maintain the heading of the vehicle, which simulates the effect of a heading hold gyro. The vehicle is equipped with two video cameras, one fixed nose camera for the pilot and one on the pan/tilt module for the payload operator. The IP3 is shown in Figure 5, note that the cameras are not shown in the model at this time.

Fig. 3. A screen capture of the simulated iRobot ATRV-Jr. in SARGE.

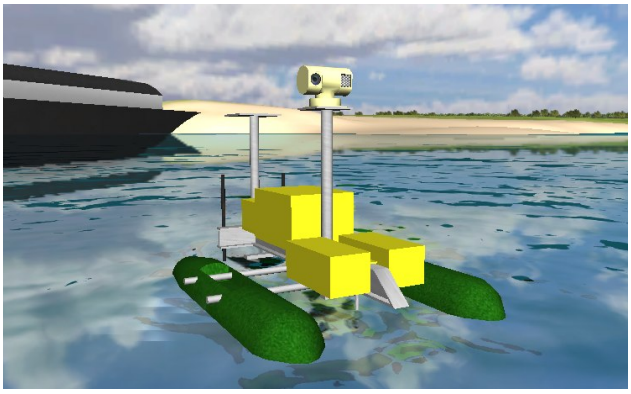

Fig. 4. A screen capture of University of South Florida's SeaRAI unmanned surface vehicle in SARGE.

Fig. 2. A screen capture of the simulated iRobot Packbot Scout in SARGE.

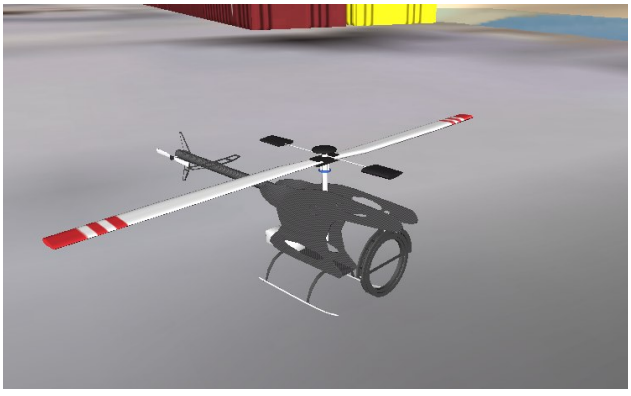

Fig. 5. A screen capture of the iSensys IP3 unmanned aerial vehicle in SARGE.

V. SENSORS

This section focuses on the implementation of various sensors in SARGE. The sensors discussed in this section include a planar laser range scanner, 3D camera, compass, global positioning (GPS), odometry, and inertial measurement (IMU) sensors. SARGE also includes a standard video camera sensor, however this functionality is built into Unity and due to space limitations is not discussed in this section. All sensors are empty Unity game objects with a script attached that performs the necessary operations to produces a reading. Range sensors in SARGE are implemented using the physics system's raycast functions, which return the distance to an object given an origin, direction and distance. The GPS and compass data are calculated as an offset from a reference point in the Unity environment. The odometry data is calculated for each wheel by accumulating the rotation of each wheel every frame. Finally the inertial measurements are calculated based on the translation and rotation of the vehicle each frame. Figure 6 shows the area without any sensor debug information displayed as seen in the Unity editor.

A. Planar Laser Scanner

The simulated planar laser is a straightforward use of Unity's Raycast function. The script takes as parameters the maximum distance for the raycast, the scan angle in degrees, the number of rays to cast, and the number of scans per second. The script uses a timer to kick off a loop that casts the specified number of rays within the specified angle. The script attempts to reproduce the behavior of the SICK LMS-200, as such the scan starts on the left side of the device and plus or minus up to 4mm of random error is introduced into each distance measurement. Figure 7 shows the laser scanner's debug display in the Unity editor. The start of the scan is colored green and fades to red as the scan completes. The blue rays indicate a hit detected by the Raycast. For the rays that do not indicate a hit the distance is set to the maximum distance parameter, which is consistent with the data returned by the LMS-200.

B. Range Camera

Like the planar laser scanner the simulated range camera uses of Unity's Raycast function. By default this produces a 64x64 array of range data, like the Canesta range sensor used in the CRASAR lab. The script allows the user to configure a maximum distance, horizontal resolution, vertical resolution, and field of view. For each frame the script simply calls the Raycast function XRes*YRes number of times (the default is 64x64 or 4096 raycasts). This makes this sensor relatively processor intensive to run, however as long as a single instance of SARGE is not running more than two or three of these sensors simultaneously there is no noticeable performance degradation. As CPU speed continues to increase this will become less of a problem. Figures 8 and 9 show the debug display and the output of the range camera. The output images are colored similarly to the output of Canesta's debug application where red indicates a near 0 distance and violet is the maximum distance detectable by the sensor. Unlike the real time-of-flight range sensors which have a problem with range banding[2] the simulated range sensor does not yet implement this type of error. Figure 8 shows the range camera with a resolution of 64x64 pixels. Figure 9 shows the range camera with a resolution of 256x256 pixels. The difference can clearly be seen in the output images (right), additionally the low resolution debug output (left) displays a fringing at the edges indicating a lower resolution. The higher resolution range sensor uses 65536 raycasts every frame. Running at this resolution produces a noticeable slowdown in the simulation since this is equivalent of running 16 low resolution sensors.

C. Compass

The compass script is very simple. Unity game objects, like objects in pretty much all 3D applications have an orientation in addition to a position. When designing an environment a reference object is placed with its Z axis facing the desired North direction. The compass script then uses the offset of its Z axis[3] from the reference direction to determine the compass heading. Figure 10 shows a reference point with the Z axis oriented to an arbitrary North along with two robots, one facing North ($0°$), the other facing East ($90°$).

D. Global Positioning

The simulated GPS, like the compass, uses an offset from a reference point to convert from an arbitrary point in SARGE's world space to a real-world pair of coordinates. If the environment is modeled after a real-world location, the reference point is placed in a known location and assigned the corresponding real-world latitude and longitude using a script that simply holds these variables. Each world unit in Unity and SARGE corresponds to 1m in the real-world which makes the offset calculation trivial. The distance between the robot and the reference point with respect to the X and Z axes

[2]Objects that are farther than the maximum detectable range still appear in the image, but with invalid distance readings, causing the ranges to appear as a series of bands.

[3]The Z axis of the object it is attached to.

of the reference point. These distances are divided by the number of meters per degree for the given latitude and longitude of the reference point. Equation 1 shows the formula used to calculate the number of meters per degree of latitude and Equation 2 shows the formula used to calculate the number of meters per degree of longitude. This is necessary because these distances change depending on the given latitude. The accuracy of formulas were obtained from [20] and have been verified experimentally in through their use in [16].

$$\begin{aligned} latlen = {} & 111132.92 \\ & + (-559.82 * Mathf.Cos(2 * lat)) \\ & + (1.175 * Mathf.Cos(4 * lat)) \\ & + (-0.0023 * Mathf.Cos(6 * lat)) \end{aligned} \quad (1)$$

$$\begin{aligned} lonlen = {} & (111412.84 * Mathf.Cos(lat)) \\ & + (-93.5 * Mathf.Cos(3 * lat)) \\ & + (0.118 * Mathf.Cos(5 * lat)) \end{aligned} \quad (2)$$

E. Odometry

The simulated odometry sensor operates similarly to its real-world counterpart. Wheeled robots in SARGE use Unity's built in wheel system. This wheel system provides the current rotations per minute (RPM) of each wheel in the system. For every frame of the simulation the RPM of a wheel is read and converted to rotations per second and then multiplied by the fraction of a second since the last update and the circumference of the wheel. This gives the linear distance the wheel would travel if there is no slippage, just like an encoder based odometer on a real robot.

F. Inertial Measurement

The simulated Inertial Measurement Unit (IMU) calculates inertial changes each frame by comparing the position of the game object with its previous position and orientation. This allows the IMU to provide both linear and angular velocities and accelerations. At this point the fidelity of this simulated sensor has not been validated empirically, it is likely that the simulated sensor is less noisy than its real-world counterpart.

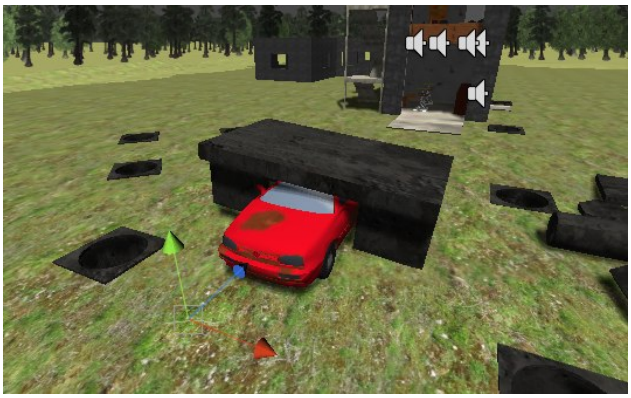

Fig. 6. The SARGE scene used to demonstrate the laser and range camera.

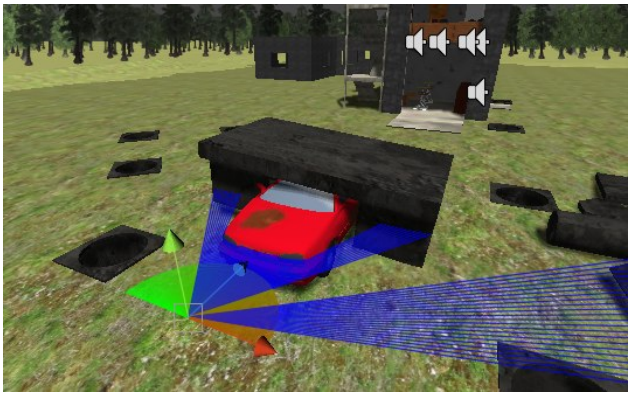

Fig. 7. The debug display for the planar laser range scanner.

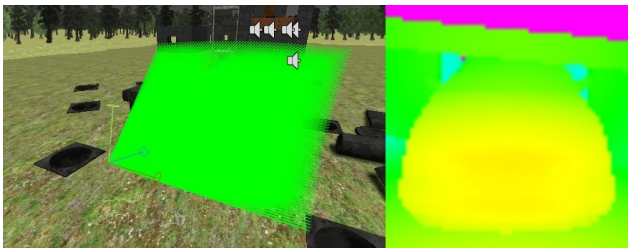

Fig. 8. The debug display and output of the range camera (64x64).

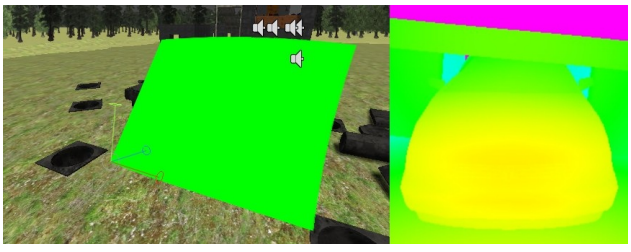

Fig. 9. The debug display and output of the range camera (256x256).

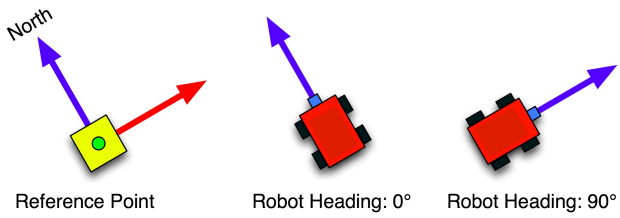

Fig. 10. The compass heading is calculated using an offset from a stationary reference point in the simulated world.

VI. Environments

Creating environments in Unity is only as complex as the environment that will be created. To create a simple checkerboard with sides (similar to the Webots "My First World" tutorial) the developer can model the environment in an external application and then import it into Unity and assign a MeshCollider or several Box-Colliders to the object's floor and sides. Or several Box primitives could be placed in space and sized as desired. To create a more complicated environment such as those in SARGE it is necessary to model the individual pieces in an external editor and import and place them in Unity. Previous work [16] explains in detail how SARGE environments are created. The short explanation is that SARGE environments are created using reference photos for modeling real-world buildings. Then the buildings and foliage are placed in space using an aerial photo of the environment being reproduced as a template. This allows SARGE environment objects to be placed with an accuracy of less than 1m relative to the other objects in the environment. Figure 11 shows the University of South Florida's robot testbed partially completed. The terrain is a combination of Unity's built in terrain system and a plane with holes in it which is necessary to allows the robots to enter the underground portion of the environment. At this time Unity's terrain system doesn't support holes that pass through the terrain, so the terrain directly under the testbed is set to a very low height and covered with the plane with holes.

VII. GUIs

The user interfaces in SARGE are constructed using Unity's GUI API. GUI components are laid out using blocks of code that specify the position and contents of each GUI control. SARGE's GUI relies on a mix of images created in Photoshop and text labels for each of the controls. The GUI is contained in a single script and uses a set of flags to switch between various menus. Figures 12 and 13 show examples of the SARGE GUIs. Figure 12 shows the main menu which consists of a label containing the background image and a box

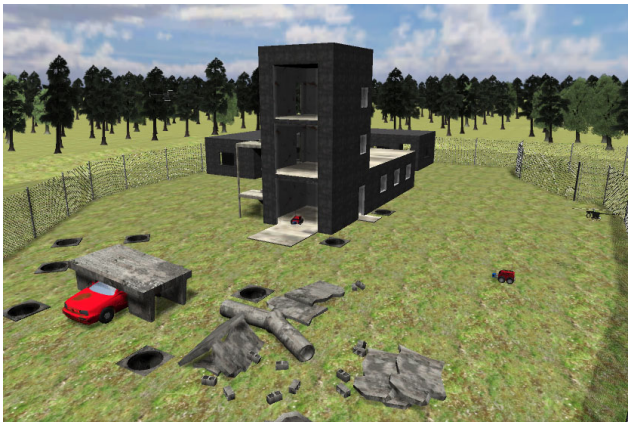

Fig. 11. The USF robotics testbed environment.

filled with the four buttons. The buttons set flags to determine which menu will be displayed during the next frame. Figure 13 shows the graphics options menu which consists of a background image label, a box, a toolbar, buttons, and text labels.

Fig. 12. The SARGE main menu.

VIII. Summary

This paper discussed the benefits of using the Unity game environment versus other popular simulation engines as well as explained how Unity was used to create the Search and Rescue Game Environment (SARGE). The main benefits of Unity are

- Complete API documentation with simple examples.
- Active developer community.
- Drag-n-Drop environment editor.

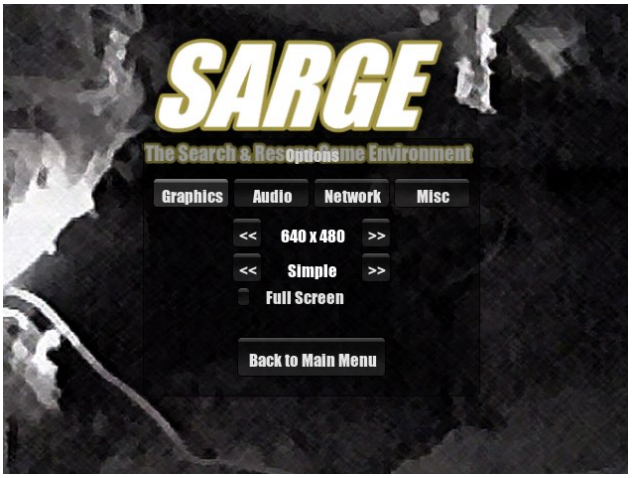

Fig. 13. The SARGE options menu.

- Uses up-to-date physics (nVidia's PhysX) and rendering engines.
- One-click multi-platform (Windows, Mac, Web) distribution.
- Low cost.

These features combined make Unity a high fidelity and easy to use simulation environment. SARGE currently has 5 robots that span air, ground, and sea modalities. Each were modeled in an external 3D modeling application, imported into Unity, and assigned physical properties and control scripts. The robots each have a compliment of sensors ranging from odometry to 3D range cameras. The sensors are simply scripts which make use of various Unity API functions attached to objects in the robot models' hierarchy. SARGE's environments are created using a combination of Unity's terrain system and imported models. The takeaway message should be to give Unity and/or SARGE a try for your robot simulation or visualization needs. Moving the development of SARGE to the Unity engine provided a huge boost in developer productivity because of the benefits listed above.

REFERENCES

[1] "Unrealtournament 2004," http://www.unrealtournament.com/. [Online]. Available: http://www.unrealtournament.com/

[2] S. Carpin, M. Lewis, J. Wang, S. Balakirsky, and C. Scrapper, "Usarsim: a robot simulator for research and education," in *Proceedings of the 2007 International Conference on Robotics and Automation*, April 2007, pp. 1400–1405.

[3] S. Balakirsky, C. Scrapper, S. Carpin, and M. Lewis, "USARSim: Providing a Framework for Multi-robot Performance Evaluation," *Proceedings of the 6th Performance Metrics for Intelligent Systems (PerMIS)*, 2006.

[4] "Player/stage/gazebo," http://sourceforge.net/projects/playerstage. [Online]. Available: http://sourceforge.net/projects/playerstage

[5] R. Rusu, A. Maldonado, M. Beetz, and B. Gerkey, "Extending Player/Stage/Gazebo towards Cognitive Robots Acting in Ubiquitous Sensor-equipped Environments," *IEEE International Conference on Robotics and Automation (ICRA) Workshop for Network Robot System, Rome, Italy, April*, vol. 14, 2007.

[6] P. Karimian, R. Vaughan, and S. Brown, "Sounds Good: Simulation and Evaluation of Audio Communication for Multi-Robot Exploration," *Proceedings of the IEEE International Conference on Intelligent Robots and Systems, October*, 2006.

[7] S. Kim, K. Kim, and T. Kim, "Human-Aided Cleaning Algorithm for Low-Cost Robot Architecture," *LECTURE NOTES IN COMPUTER SCIENCE*, vol. 4552, p. 366, 2007.

[8] "Webots," http://www.cyberbotics.com/products/webots/. [Online]. Available: http://www.cyberbotics.com/products/webots/

[9] L. Hohl, R. Tellez, O. Michel, and A. Ijspeert, "Aibo and webots: Simulation, wireless remote control and controller transfer," *Robotics and Autonomous Systems*, vol. 54, no. 6, p. 472, June 2006.

[10] O. Michel, "Cyberbotics ltd - webotstm: Professional mobile robot simulation," in *International Journal of Advanced Robotic Systems*, vol. 1, no. 1, 2004, pp. 39–42.

[11] "Matlab," http://www.mathworks.com. [Online]. Available: http://www.mathworks.com

[12] D. Ernst, K. Valavanis, R. Garcia, and J. Craighead, "Unmanned Vehicle Controller Design, Evaluation and Implementation: From MATLAB to Printed Circuit Board," *Journal of Intelligent and Robotic Systems*, vol. 49, no. 1, pp. 85–108, 2007.

[13] F. Song, P. E. An, and A. Folleco, "Modeling and simulation of autonomous underwater vehicles: Design and implementation," *IEEE Journal of Oceanic Engineering*, vol. 28, no. 2, pp. 283–296, April 2003.

[14] T. Prestero, "Verification of a six-degree of freedom simulation model for the remus autonomous underwater vehicle," Master's thesis, Massachusetts Institute of Technology and Woods Hole Oceanographic Institution, September 2001.

[15] J. Craighead, "Distributed, game-based, intelligent tutoring systems - the next step in computer based training?" in *Proceedings of the International Symposium on Collaborative Technologies and Systems (CTS 2008)*, May 2008, pp. 247–257.

[16] J. Craighead, R. Gutierrez, J. Burke, and R. Murphy, "Validating the search and rescue game environment as a robot simulator by performing a simulated anomaly detection task," in *Proceedings of the 2008 International Conference on Intelligent Robots and Systems (IROS 2008)*, September 2008.

[17] "Search and rescue game environment," http://sarge.sourceforge.net. [Online]. Available: http://sarge.sourceforge.net

[18] "Unity," http://www.unity3d.com. [Online]. Available: http://www.unity3d.com

[19] "Usarsim," http://usarsim.sourceforge.net/. [Online]. Available: http://usarsim.sourceforge.net/

[20] (2008) Length of a degree of latitude and longitude calculator. [Online]. Available: http://www.csgnetwork.com/degreelenllavcalc.html