A Customizable, Multi-Host Simulation and Visualization Framework for Robot Applications

Tim Braun, Jens Wettach, Karsten Berns
Robotics Research Lab
University of Kaiserslautern
Kaiserslautern, Germany
Email: {braun, wettach, berns}@informatik.uni-kl.de

***Abstract*—A highly flexible framework for visualization and sensor simulation in three-dimensional environments is presented. By allowing the insertion of freely programmable elements for online scene modification, a programmer can customize the framework to fulfill the exact simulation or visualization needs of an application of interest. Furthermore, the framework provides simple external interfaces so that multiple clients can be attached to it with ease. The frameworks' capabilities are demonstrated with two complex robotic applications that require both a high quality simulation of cameras and lasers scanners and an intuitive 3D visualization.**

I. Introduction

Due to the high complexity of modern robotic systems, almost any research conducted in the area of robotics can benefit from a *simulation* of the system behavior before experiments on a real platform take place. Aside from reduced development time, a simulation allows to validate safety properties and test new algorithms more objectively with increased control over sensor/actor noise and more repeatable conditions. On the other hand, the control and understanding of an operating robotic system can often be substantially improved by offering the user a good *visualization* of the current robot situation.

Both of these aspects can (and commonly are) be approached using 3D-models of the involved objects. For example, in order to simulate a camera mounted on a mobile robot, a three-dimensional scene needs to be modeled and an image of it must be rendered from the camera's current viewpoint. For visualization, the same approach can be applied: Modeling a scene after the actual area where the robot is deployed and displaying it to the user from a virtual camera he or she controls. However, in both cases it is crucial to use a framework that is powerful enough to support all simulation and visualization requirements that arise for the application at hand. For example, the visualization of a robot arm might require to allow the parametrization of all joint angles and the highlighting of internal collisions or invalid configurations, while the visualization of a mobile robot might need to display navigational points at varying positions.

Commercially available toolkits typically provide several pre-made robots and a variety of standard sensors like cameras, bumpers and laser scanners. With these, an user can simulate or visualize a scene as long as the desired functionality is provided by the toolkit. However, these toolkits are inadequate for highly specific requirements not foreseen by the developers. For example, if a specialized simulation for outdoor robotic research needs to control the visual transparency of foliage to test image processing algorithms in thick vegetation, this effect cannot be realized by standard out-of-the-box software.

The **SimVis3D** framework presented in this paper aims at providing a general approach to such complex and highly specific adaptation requirements. It is a modular framework usable both for the simulation of optical sensor-systems like cameras, laser scanners or PSDs and the visualization of spatial information such as 3D-environments, maps or topological graphs. Besides providing basic functionality to construct and parametrize three-dimensional scenarios, SimVis3D offers strong extensibility by providing a mechanism to easily include new, manually coded components. Since its main field of application are complex real time robotic systems which normally contain more than one controlling computer, an additional feature of SimVis3D is the ability to support situations where its input is generated by multiple client computers.

In the next section, a short overview of related robot simulation toolkits is given. Section III describes the SimVis3D framework architecture itself, while section IV elaborates how the framework performed in two real applications. Based on these results, a conclusion is given.

II. Related Work

There already exists a variety of toolkits for realistic robot simulation. In *SimRobot*[1], arbitrary robots can be defined based on an XML description using predefined generic bodies, sensors and actuators. Available sensor types are cameras, laser scanners and bumpers. Robot dynamics are simulated via ODE[1]. *Gazebo*[2] is a 3D multi-robot simulator that contains predefined models of real robots as Pioneer2DX and SegwayRMP. Provided sensors are sonar, range scanner, GPS, inertial system and cameras. Robot and sensors are created as plug-ins and the simulation environment is described via XML. *Webots*[3] is a commercial simulation tool containing several models of real robots as Aibo, Khepera and Koala. It provides

[1]http://www.ode.org

```
<part file="world.iv" name="WORLD" insert_at="ROOT" pose_offset="0 0 0 0 0 0"/>
<part file="robot.iv" name="ROBOT" insert_at="WORLD" pose_offset="0 0 0 0 0 0"/>
<element type="pose" name="R_POSE" insert_at="ROBOT" x="1" y="2" z="3" roll="0" pitch="0" yaw="-90"/>
<sensor type="camera" name="ROBOT_CAM" insert_at="ROBOT" pose_offset="3 0 5 0 0 0"/>
```

Fig. 1. Example XML based scene description

a virtual time to speed up simulation processes. Physics are implemented by ODE and sensor classes comprehend light, touch, force and distance sensors, scanners, GPS and cameras. *USARSim*[4] is a simulation tool for **u**rban **s**earch **a**nd **r**escue robots based on the *Unreal Tournament* game engine. Virtual environments can be modelled via the *Unreal Editor* and dynamics is based on the *Karma*[2] engine. Sensor types are sonar, laser scanner and forward looking infrared (FLIR). *EyeSim*[5] is a simulator for *EyeBot* robots (a development platform for vehicle, walking and flying robots). Sensor classes contain virtual cameras and position sensitive devices which can provide realistic, noisy data.

Although these tools provide excellent support for standard mobile robot setups, highly specialized scenarios having special scene manipulation requirements are not adequately supported due to the toolkits' limited customizability and extensibility. Also, most do only support mobile robot settings; other robot types like stationary manipulators are not available.

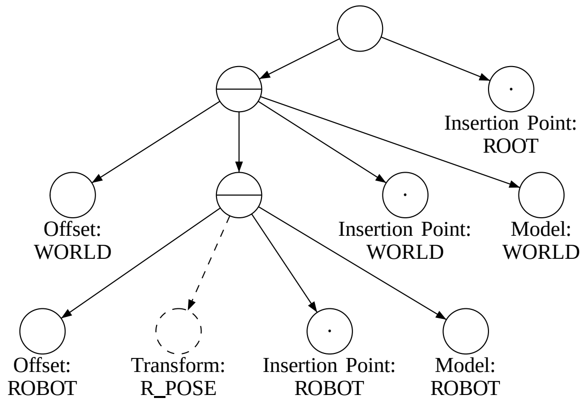

Fig. 2. Scene graph after parsing command file from Fig.1

III. Framework Architecture

SimVis3D was designed to fulfill four goals:

- Allow users to easily create custom setups by combining basic building blocks to meaningful scenarios.
- Allow users to parametrize exactly the scene aspects they want to control.
- Provide strong support for people wanting to add new components which alter previously static aspects.
- Use very simple external interfaces to facilitate data transport to and from remote processes.

To achieve the desired level of customizability, it was decided to use only open source libraries and eventually make SimVis3D available as a GPLed open source project. The framework is built on top of the widely used 3D rendering library *Coin3D*[3], which is API-compatible to Open Inventor. Both rely on OpenGL for the actual rendering process and use a graph data structure called **scene graph** to store and render graphics.

In the next section, the main ideas behind the scene graph data structure will be shortly revisited in order to prepare the presentation of how a scene is composed from basic blocks (Section III-C) and how specific scenario aspects can be made parameterizable (Section III-D).

A. Scene Graph

A three-dimensional scene in Coin3D is created from nodes of the scene graph (see Fig.2). Information that defines actual 3D shapes, attributes, cameras and light sources is stored in *leaf nodes*, which are children of a hierarchy of *group nodes*. The group nodes serve to provide a logical structure by grouping nodes together that for example make up the same object.

To render an image, the graph is recursively traversed starting with the scene graph root (group) node. When a group node is traversed, all children of it are traversed in a fixed order. The order is indicated in pictures by the child nodes relative position: The left-most node comes first, the right-most node last. Each traversed graph node can manipulate the current OpenGL state by changing parameters like the active model transformation matrix or transmit geometrical primitives to the graphics hardware. Geometry is then rendered using the current OpenGL state; its appearance therefore depends on all nodes that have been traversed earlier. To limit the scope of a state change, there exists a special variant of group nodes called *separator nodes*. Separator nodes behave like ordinary group nodes but also save the OpenGL state at the beginning of their traversal and restore this state upon its end. They are generally indicated by a horizontal line inside the node.

B. Insertion points

SimVis3D extensively uses the hierarchical structure of the scene graph to express dependencies between components. Since coordinate transformations accumulate during the rendering traversal, scene components placed as a child of another component automatically inherit any transformations that influence the parent. Thus, a

[2]http://wiki.beyondunreal.com/wiki/Karma

[3]http://www.coin3d.org

sensor connected to a mobile robot is automatically moved together with it if the sensor node is inserted as a child of the nodes that make up the robot.

When inserting new nodes into an existing scene graph, it is therefore especially important to control the exact place of insertion. For this, SimVis3D uses named placeholder leaf nodes called **insertion points**, which can be located in a scene graph by their name but have no influence on rendering. Adding such insertion points in a scene allows to mark semantically meaningful locations, for example the attachment point of a camera sensor or places where other parts could be attached to a robot. To prime scene construction, SimVis3D initially provides a group node with an insertion point called ROOT.

C. Scene Construction

The SimVis3D framework allows users to specify the construction of a scene with a XML based description file (Fig.1 shows an example). Each line contains either a **part**, **element** or **sensor** command.

The **part** command simply adds another scene graph stored in a separate file (using a standard format like Open Inventor or VRML) to the current scene. Parts are the basic building blocks that constitute the scene and generally make up individual objects like robots, chairs or the environment. New parts can be added to the existing scene only at insertion points, but it is possible to supply a static offset which is interpreted relative to the coordinate frame active at the specified insertion point. Each part command also creates a new insertion point in this local frame with the parts name, leading to the possibility to create nested parts without manually adding insertion points. More exactly, a part command inserts a separator node containing the static offset transformation, the new insertion point and the actual part right *before* the specified parent insertion point. Fig.2 shows a schematic scene graph after the example scene description has been processed. The dashed node is added by the `pose` element discussed in section III-D.

With this mechanism, it is very easy to create a hierarchical scene by combining several parts. By using nested insertion points, the user can specify relations between objects, i.e. parts that are attached to another (for example, a scanner attached to a robot sitting upon a table). In this case, the pose offset and all subsequent transformations of a nested part are automatically interpreted in the coordinate frame of the parent part, agreeing with intuition.

D. Scene Parametrization

The **element** command is the most versatile instruction to SimVis3D and lies at the core of the frameworks flexibility and extensibility. It causes the instantiation of a class object identified by the element name using a factory pattern. The class constructor receives an insertion point and the whole XML command that led to its construction. It is then free to extract any needed extra configuration data from the command and modify the scene graph in any way it sees fit (but generally adding nodes at the position indicated by the insertion point). It is important to stress that all modifications are encapsulated inside the element; the SimVis3D framework is unaware of the elements' internal workings. The only interface between them are named **parameters** exported by the element, which are floating-point scalar values that can be modified by the framework. Parameter modifications are propagated through the element to effect changes in the scene graph. The exact nature of the changes thereby depends again solely on the instantiated element. Coming back to the example in Fig.1, the third line adds an element to the scene that allows to modify the pose of the robot relative to the environment - a natural requirement for mobile robots. Internally, the `pose` element created here could for example read its initial position from the XML command, export the six parameters (x, y, z, roll, pitch, yaw) to the framework and propagate them to a new coordinate transformation node inserted right before the specified insertion point.

This architecture allows users to extend SimVis3D with almost any effect or interaction capability he or she desires by implementing a custom element class and adding it to the element factory. The encapsulation guarantees that no changes to other components of SimVis3D are required. It is easy to envision elements that change the shape or color of parts, alter scene lighting, add geometrical data like point clouds visualizing 3D scanner data etc. By customizing elements, the SimVis3D framework can be tuned to the special modeling requirements for the problem at hand with ease and efficiency.

E. Sensors

The **sensor** command of SimVis3D is required for the inclusion of simulated sensors in the scene, for example a camera attached to a robot. Employing the same mechanism as the instantiation of elements, the sensor command effects the creation of a sensor object of the requested type at a given insertion point. Line 4 in Fig.1 creates a camera attached to the robot with a static offset relative to the robots local frame center. Differing from elements, sensors do not offer parameters to modify their behavior online. Instead, they offer sensor data to the user - plain images in the case of camera sensors, distance data in the case of laser scanners or PSDs. However, parametrized sensors can be emulated by inserting elements right before the sensor.

Although currently only cameras and laser scanners are supported by SimVis3D, the encapsulation and insertion mechanism of sensors is closely modeled after the one used for elements, so the inclusion of custom sensor types can be implemented with effort comparable to a new element class. The encapsulation principle is especially important here, since for example the inclusion of a new camera requires modifications of the scene graph not only at the insertion point, but also at its root. However, since the actual modifications are hidden for the framework anyway, this implementation detail does not bear any architectural consequences and is thus not further discussed.

F. Additional Framework Services

For increased ease of use, the SimVis3D framework allows to embed XML commands directly into Open Inventor files. When a part is created using this file, the embedded commands are retrieved and executed. It is thus possible to package a single file describing a complete robot with all degrees of freedoms, sensors etc. using element and sensor commands inside the file.

Furthermore, the framework optimizes the scene graph after all commands have been executed by replacing all scene graph nodes having identical content with references to one of these nodes. Identical nodes can easily occur in the scene if for example two identical part commands are given (two tables need to be placed inside a room). The optimization ensures that no performance penalty is incurred, regardless how a scenario is constructed.

G. Accessing SimVis3D from multiple hosts

Aside from easy customizability and extensibility, SimVis3D explicitly supports that parts of the simulation and visualization input data come from different processes or even different physical machines. The actual job of transporting data between computers is not in the scope of the framework and needs to be performed by standard interprocess communication (IPC) using shared memory, named pipes etc. or libraries like ACE. However, SimVis3D was designed to support a variety of these techniques by offering a very simple interface to the outside world.

This interface consists of *four data arrays* located at given memory locations and stored compactly. The arrays contain

- structs of element descriptors
- floating-point parameter values
- strings forming a ’scene change request log’
- strings forming a ’scene change log’

The element descriptors are structures containing all information required to identify a specific element present in the current scene and locate their parameter values in the second array. For each element, they hold the elements’ name and insertion point, the number and names of its parameters and most importantly, the starting index of the parameter values in the parameter values interface array. This second array is a simple vector of floating point scalars containing, as the name suggests, the actual values of all element parameters stored consecutively. With access to these two interfaces, any process can analyze an existing scene and update the parameters of any element that it requires to access. For example, a process that calculates a robot position based on various sensors could locate the `R_POSE` element of the running example by scanning the element descriptors, extract the index of its parameter values and write the calculated robot pose into the value array at this location. The reason to split the element descriptors and the actual values into two arrays is efficiency: After looking up the exact array index of a parameter value once, the client process can from this point on directly manipulate the float values, reducing data transfer to the SimVis3D host process enormously. Fig.3 shows the interfaces that the example used in the previous section would produce.

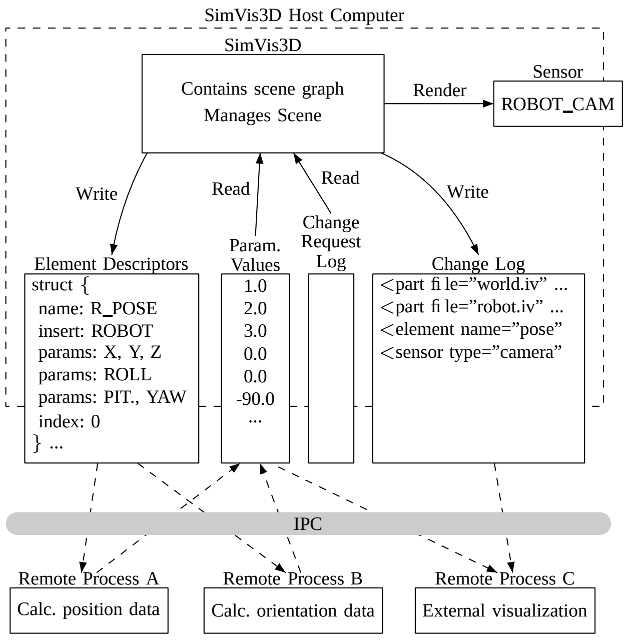

Fig. 3. Remote access interfaces provided by SimVis3D. Two use cases are depicted: a) Remote processes A and B manipulate part of the robot pose by setting the parameters exported by the R_POSE element. b) An external visualization process C copies the scene setup by repeating the commands in the change log and copying all parameter values. The external visualization could then for example render the scene on screen from a remote viewpoint without any performance penalty for the SimVis3D host computer.

While the first two arrays allow the manipulation of existing elements, the last two interfaces permit external processes to add new components or track these structural changes. In order to add a new scene component, a client can write XML commands similar to those in Fig.1 to the scene change request log. When SimVis3D notes a new command, the framework executes it and confirms the execution by adding it to the scene change log. This mechanism allows clients to request the addition of new elements or parts during runtime. It also permits the construction of an identical scene by simply repeating all commands stored in the change log on a local scene graph. Thereby, an external client could render the same scene as is used for simulation from a different viewpoint for visualization purposes, or even exclusively render a scene that is only set up but not rendered on a robotic platform. With this approach, SimVis3D supports both simulation *and* visualization tasks (and even mixtures of both) seamlessly within the same basic framework. Process C in Fig.3 shows an example of such a use, and Fig.6 has been created using this setup.

IV. Applications

The SimVis3D framework has been used to simulate sensors and visualize complex 3D scenes for two research projects involving a mobile indoor robot named MARVIN and an outdoor robot termed RAVON[4]. Although both projects focus on different topics and therefore have differing simulation and visualization needs, the framework was easily adapted to their requirements by implementing custom elements as described in III-D.

For both projects, SimVis3D has been embedded into already existing robotic control systems based on MCA2[6]. The core functionality has been wrapped into two MCA2-modules (see [6] for details), with one handling all scene graph related tasks and another providing frame grabber and scanner interfaces for the virtual sensor data. Both control systems are split across two PCs and control data needed to be passed to SimVis3D from multiple hosts using the interfaces described in section III-G. The IPC component was implemented based on the 'Blackboard' mechanism available in MCA2, which essentially provides blocks of shared memory that are transparently passed to different hosts via TCP/IP if needed. For this, it proved especially beneficial that SimVis3D exhibits simple, compact array interfaces and no complicated data exchange protocols were used.

A. *MARVIN Scenario*

MARVIN is an autonomous mobile robot for exploration and mapping of indoor environments. Its obstacle avoidance and navigation capabilities primarily depend on two planar laser range scanners at its front and rear with horizontal measurement planes 10 cm above the floor. Furthermore, a third scanner is mounted on a frontal tilt unit 1 m above ground to take 3D scans of environmental features (walls, doors, etc). Thus in order to test algorithms like line and plane extraction from distance data, the simulation for the MARVIN project mainly required a realistic 3D scanner simulation. This has been realized by adding a distance sensor class to the SimVis3D framework that provides a settable number of distance values calculated from a given sensor center to nearest objects in the scene. The calculation had been based initially on ray tracing operations which are directly supported by *Coin3D*. Due to the bad runtime performance of this method (about 2s for one scan with 361 values), a more sophisticated method has been developed that evaluates the depth buffer of a standard camera sensor which is virtually attached to the scanner. With this, all three scanners on MARVIN can be simulated at a rate of 15 Hz providing distance data in a half-circle measurement plane with $0.5°$ resolution (about 360 values per scanner) which is fast enough to test obstacle avoidance strategies in real time.

In Fig.4 the visualization result of the 3D scanner simulation is shown. The simulation uses the same base scene graph as the visualization as input for calculating the intersection of the laser beams with the scene objects (floor, walls, tables). The only difference to the visualization is that in the simulation scene the laser beams and data points are not visible as they might disturb the ray tracing operation.

[4]http://agrosy.informatik.uni-kl.de/en/robot_gallery/

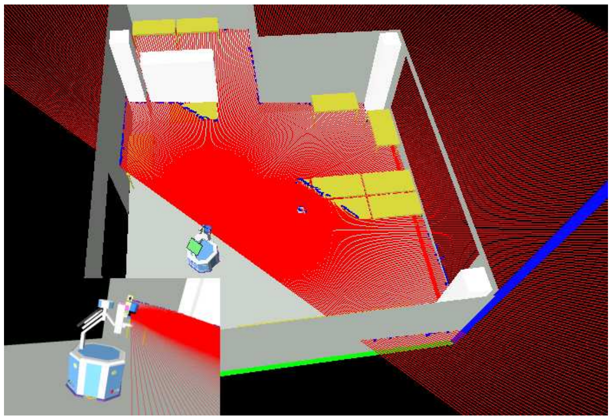

Fig. 4. Visualization of laser beams and simulated distance measurements for tiltable 3D scanner mounted on mobile robot MARVIN in a typical indoor scene. In the lower left corner a detailed view of the robot and the tilted scanner is given.

Consequently this setup results in three different programs with four scene graphs that might transparently be spread on three computers if a suitable network layer is provided, as for example in the *MCA* framework that is used in this project:

- simulation node with scene graph composed of robot and environment (lab) as basis for scanner data calculation
- visualization node with same scene graph as simulation, enhanced with plotted laser beams and intersection points
- GUI node for visualizing the simulation and visualization scene, reflecting these two scene graphs in two *SoQt* viewer plugins (for ease of illustration only the visualization scene is shown in Fig.4)

The only precondition for this strategy is that simulation and visualization node get the same basic scene description file as input. The GUI node takes all information for the scene graph setup from data structures provided by the other nodes via the network link.

For easy evaluation of the simulation results a new element has been added to the SimVis3D toolkit that allows to plot an arbitrary 3D point cloud with definable number, color and size of points. To create a realistic data set the points are affected by Gaussian noise so they do not exactly fit the intersection between ray and object. This is for example the cause that some points on the wall diagonally opposite to the robot are hidden and do not appear in the visualization. This procedure allows feeding the data filtering and feature extraction algorithms with realistic distance measurements and facilitates the offline test of navigation and mapping strategies before downloading them on the real robot.

B. RAVON Scenario

The RAVON outdoor project focuses on autonomous obstacle avoidance and navigation and the respective algorithms rely heavily on visual stereo reconstruction and visual odometry calculation. Furthermore, color and lighting invariance algorithms are used for preprocessing.

To create an useful simulation for this research project, SimVis3D was required to deliver simulated camera images with highly realistic textures and allow the variation of lighting color and direction. In order to meet these requirements, an outdoor scene similar to the standard testing area of RAVON has been modeled with detailed obstacles and high resolution textures. Furthermore, elements allowing the on-line adjustment of foliage transparency and lighting color settings were created.

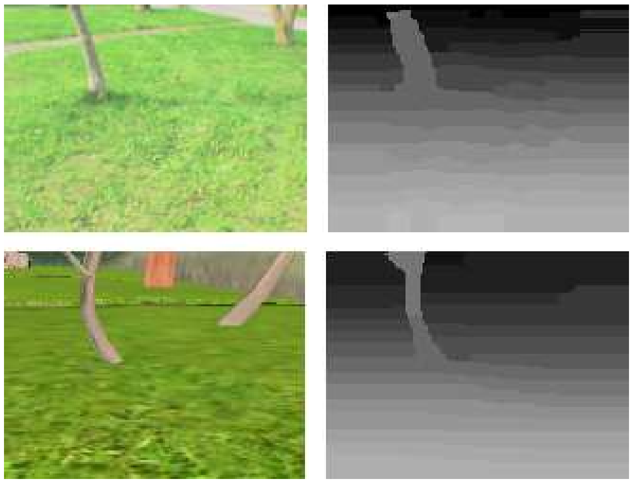

Fig. 5. Real (upper row) and simulated (lower row) camera images and disparity maps.

To test the framework performance, images taken with the real robots stereo camera system have been compared with images generated with virtual camera sensors in simulation. Fig.5 exemplarily shows a pair of such real and simulated images along with disparity maps calculated from them (the images depict the typical performance of the stereo algorithm). In both cases, the actual stereo matching algorithm used in the RAVON project was used without any adjustments. The results demonstrate that the simulation clearly suffices for stereo reconstruction. Similar experiments performed for the visual odometry computation also led to realistic results, although a texture with higher contrast had to be used in the simulation.

Asides from simulation, the framework has also been used for visualization of mapping and route planning. For this, elements that overlay markers for map nodes and node connections at parametrized locations have been implemented (Fig.6). By using two separate instances of SimVis3D for simulation and visualization, both could be used simultaneously but contained different elements - therefore, the overlaid map does not appear in the simulated camera images, although the other environment and the robot pose are equal in both Fig.5 and Fig.6.

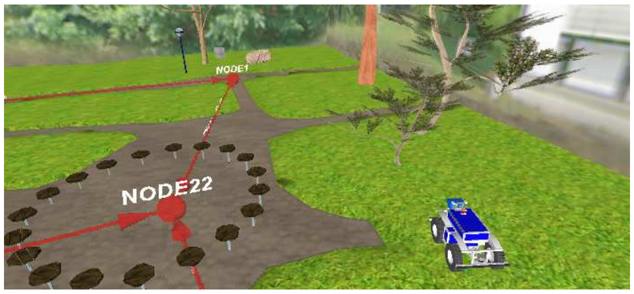

Fig. 6. Visualization of RAVON scene with overlaid navigation graph.

V. Conclusion

A framework for simulation and visualization of three-dimensional scenes has been presented that is geared towards robotic applications with highly specific scene parametrization needs. By supporting manually coded extensions, a programmer can easily customize all control possibilities that an application of interest requires. The use in two complex mobile robot scenarios has provided examples for the frameworks extensibility and has shown that realistic sensor data can be produced. Although currently not as feature-rich as commercially available toolkits, the capability for extensions and the multi-host friendly architecture make SimVis3D a useful addition to available simulation/visualization frameworks that adds several new aspects.

VI. Future Work

Future work includes the application of SimVis3D to stationary robots (arms, robotic heads) to explore their special requirements and the integration of a dynamics simulation based on the element inclusion methodology. Preliminary experiments with the *newton*[5] library are very promising. As already mentioned, the publication of SimVis3D as an open-source project is also one of the next steps.

References

[1] T. Laue, K. Spiess, and T. Refer, "A general physical robot simulator and its application in robocup," in *Proc. of RoboCup Symposium.*, Bremen, Germany, 2005.

[2] N. Koening and A. Howard, "Gazebo-3d multiple robot simulator with dynamics," http://playerstage.sourceforge.net/gazebo/gazebo.html, 2006.

[3] O. Michel, "Webots: Professional mobile robot simulation," *International Journal Of Advanced Robotic Systems*, vol. 1, no. 1, pp. 39–42, 2004.

[4] J. Wang, M. Lewis, and J. Gennari, "A game engine based simulation of the nist urban search & rescue arenas," in *In Proceedings of the 2003 Winter Simulation Conference*, 2003.

[5] A. Koestler and T. Braeunl, "Mobile robot simulation with realistic error models," in *ICARA*, December 2004.

[6] K.-U. Scholl, J. Albiez, B. Gassmann, and J. Zöllner, "Mca - modular controller architecture," in *Robotik 2002, VDI-Bericht 1679*, 2002.

[5]http://www.newtondynamics.com/