AvenueUI - A Comprehensive Visualization, Teleoperation Application and Development Framework for Multiple Mobile Robots

Ethan Gold, CS Dept, Columbia University

Abstract

This thesis describes the user interface of the AVENUE project. The AVENUE project is an autonomous mobile site-modeling robot which creates solid 3-D photometrically registered models in an urban environment. The AvenueUI application provides a user interface into our autonomous robot systems for the purposes of executing experiments and facilitating system development. The AvenueUI also provides its own API for sensor/actuator visualization and control expansion. Data is presented both in a soft real-time 3-D environment as well as raw, allowing the user to interact with the underlying control system at high and low levels simultaneously. Multiple robots, views, and users are all supported in a networked environment.

1 Introduction

The purpose of the AVENUE project is to automate the process of acquiring and building a model of a complex urban environment with a limited amount of a-priori information. We are developing an autonomous mobile site-modeling robot which creates solid 3-D photometrically registered models using a laser range scanner and color cameras. AVENUE is designed to operate in an urban environment such as our campus at Columbia University. Components of the software system include the low-level *Mobility*(tm) control API, the localizer and navigation server, a Voronoi-based path-planner, a range scan view planner, and a scan and photometry integrator.

This thesis describes a 3-D live-data visualization and control application for management of and experimentation with networked mobile robots such as AVENUE. The problems of autonomous and manual control are often intertwined, especially during development of both hardware and software platforms. This system was developed to provide both high and low-level visualization and control of our multiple mobile robot systems in an integrated, portable application which allows continued development and evolution as required by our research. The user-interface described here is designed to be a single point of access for the several lower-level systems comprising the rest of the project.

Our software architecture consists of a roughly three-tier networked control architecture. RWII (Real World Interfaces Inc.), our robot vendor, ships a low-level control API called Mobility(tm) which provides access to individual sensor data and actuator controls. The Mobility services expose their API via the Common Object Request Broker Architecture

(CORBA) [1] , on top of which is layered our higher-level data fusion and control software described in Gueorguiev et al '00[3] and depicted in Figure 3. The user-interface described here is represented by the bottom "Remote Host" rectangles of Figure 3, layered on top of both the high-level fusion/control software and communicating directly to the low-level sensor API as well.

Our mobile robot lab needs an extensive, comprehensive user interface to facilitate development and experimentation with our multiple robots. The mobile robots are equipped with a wide range of sensors, all feeding their data to the LAN via CORBA services [3]. This suite of constantly streaming data types forced us to abandon our original User Interface [3] which was based on RWII's *MOM*(tm) (Mobility Object Management) sensor-management application. We chose in favor of an architecture which provided a single-point of data visualization in which all data can be viewed together in context while allowing the user to also examine each sensor's data in detail. Another key requirement was that we no longer be tied to the MOM source tree as our old system was, and that new functionality be implementable as runtime loadable modules without modifying the main source code of the application. MOM is extremely useful as a low-level testing platform as it breaks each sensor and actuator interface out into separate data and visualization windows, independent from others in the system. The great disadvantage is that there is no centralized environment in which to collect the data together for contextual visualization. Additionally, each interface window must be explicitly requested by the user who must dig through the nested architecture list to find the ones he wants. It can be extremely tedious to get a satisfactory set of windows visible for a specific task, especially if the UI and/or robot are being reset frequently.

Our primary mobile robot is an all-terrain ATRV2 (figure 2) from RWII. Our secondary robots are RWII Pioneer-1s with radio serial links to a Linux host machine which are useful for testing indoors.

1.1 Application Requirements

1. *Live-data feedback for visualization of the robot in relation to its environment:* We want to see data visualized in 3-D in soft real-time as it comes in over the network. Network latency is relatively low, bandwidth reasonably high, and the constant data streams must be rendered onto the 3-D canvas in an integrated context quickly enough to feel live.
2. *Multi-level data and control architecture exposure:* We need to be able to examine data from individual sensors outside of the main visualization environment, often with simple text values. This is for debugging software components and performing specific data-gathering experiments.
3. *Modular extensibility:* It should be relatively easy to add a visualization component for an additional or unanticipated sensor/actuator. The architecture and API should be designed in such a way that additional programmers unfamiliar with the inner-workings can program, test, and run their own modules without delving into or recompiling the main source tree.

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

4. *Independence from the robot vendor's basic control UI:* Our existing tool[3] was built on top of our vendor's basic sensor/interface visualization package. Building our own extensions involved source-code modification of the vendor's source tree, requiring us to maintain our own branch of their code. Every vendor upgrade required a porting of our objects and re-modification of their source tree. This is particularly difficult if source releases lag behind binary distribution.
5. *Platform portability:* Our mobile robot software all runs under the Linux operating system, but it is often convenient to run user-control applications from a workstation with an outdoor window view of our experimental area, or possibly from a personal OS on other machines.
6. *Network transparent architecture:* The system should be reasonably broad-minded about the location of networked objects, both the display and the data streams. Robot services may be run on a local machine or accessed across the network without any appreciable difference to the user.
7. *Move beyond the flat 2-D representation of the current system:* Most of our robot data is inherently 3-dimensional. Given the ultimate goal of the system - building photo-realistic 3-D models of a campus - it would be nice to visualize pieces of the model as they become available in addition to the basic robot-oriented data streams.
8. *Straightforward methods for moving the robot:* Both an "immediate mode" driving interface and a facility for sending complex command batches containing high-level motion and sensor commands are needed.

2 Related Work

User control of complex hardware requires mapping a small number of input devices and actions to a potentially large number of robot sub-systems, in addition to traditional view manipulation. Unless sophisticated input devices are used - frequently research projects in themselves - standard mice and keyboards are the only tools available, especially for portable software and laptop applications. In some cases, mapping standard PC input devices is straightforward and intuitive. However, some mouse operations may cause radically different actions for different actuators on the robot when invoked in the 3-D pane. Lazloo et al[6] examine mapping various joint and force actions to a workstation mouse and discrete actions to keystrokes, allowing the user to execute complex movements in a simulated animated agent after a short period of training. The "creatures" may be fanciful or simulations of living systems, such as a running human or cat. While the simulations are restricted to 2-D for the purposes of animation and live interaction, the principles are useful in a 3-D environment nonetheless, particularly if motion is logically restricted to a ground plane.

Backes et al[1] demonstrate a system for controlling their Mars Polar Lander, dealing with multiple input intentionality resulting in an arbitrated one-to-many mapping between input modalities and actuator contexts. Controlling many aspects of their rover with simple standard controls involves the user re-mapping inputs from one task to the next. The multi-level interaction with the probe's several systems allows the user to control the task with varying degrees of granularity. Supplemental windows contain lower-level representations

Figure 1: Our three RWII/Mobility-compatible robots. The large robot in the middle/back is our ATRV2 annotated in detail in figure 2. The smaller robots in the foreground are a pair of Pioneer1's.

of data as well as stored mission plan sequences. The rover's *WITS* user application also provides a robust, secure framework for collaboration by many geographically separated users. Storage and transmission of command batches for high-level mission control between multiple remote user sites facilitate collaborative mission planning.

Latency and bandwidth issues abound in space and planetary applications. Backes et al[1] and Hirzinger et al[5] demonstrate strategies for overcoming these problems in teleoperation using limited visual feedback and predictive modeling. Low-refresh camera images are overlayed on a model of the robot and work cell generated from motion encoders and pose-prediction based on user commands.

Zaad and Salcudean[4] perform a rigorous analysis of the effects of resource limitations on a medium-bandwidth teleoperation system. Bandwidth considerations can drastically affect the kind of visualization a UI application can effectively produce. Unexpected network latency or bandwidth saturation can result in unintentional dangerous robot actions at the other end of the line.

Presence is an issue in virtual reality applications. Avatars, head-tracking, and motion-simulation devices are effective for enhancing the user's feeling of presence, as demonstrated by Usoh et al[10]. They explore different motion modalities such as flying (data glove, joystick), virtual walking (treadmill), and real walking around the experimental space. The degree to which natural motions affect the virtual world in ways that mirror real-world expectations enhance the user's experience of presence in the virtual world dramatically. Ideally, the greater the sense of presence, the more accurately and naturally the user can

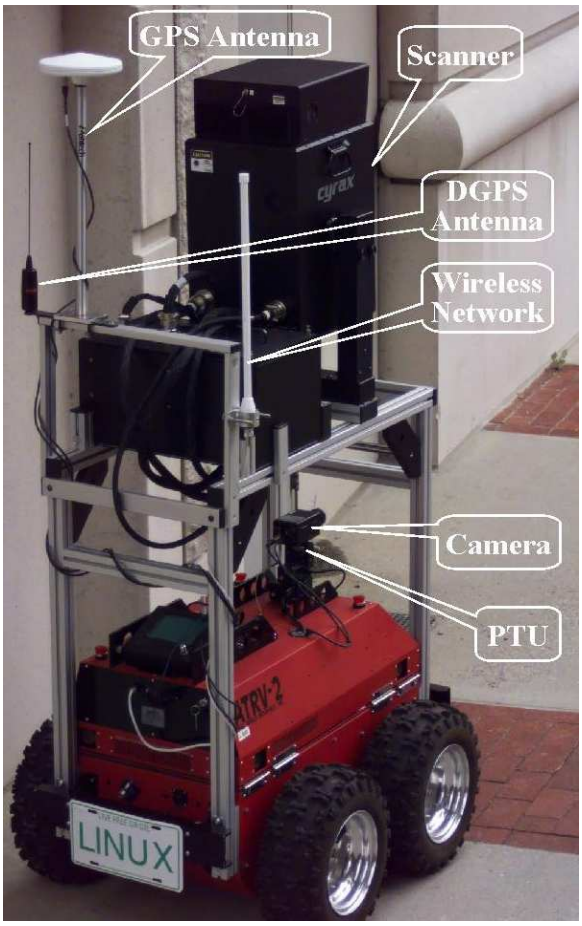

Figure 2: Our primary ATRV2 robot with all its hardware mounted.(Image courtesy of Atanas Gueorguiev [3])

control the virtual world or remote hardware.

The multiple-view paradigm where users can switch instantly between a number of camera viewpoints at will is very effective for driving video games and works well in teleoperation too. With the complexity of multiple robots and possibly multiple view contexts per robot, the user has to be able to easily switch between views. Worldlets implemented by Elvins et al[9] are effective for accelerating the user's search for a particular perspective. Users recognize viewpoints much more quickly if they see a snapshot of a view in a manipulable worldlet than simply a name or 2D image. In an environment with many such waypoints, visual cues are more effective for user recognition, particularly for locations with which the user is not already familiar.

RWII's *Mobility Object Management* (MOM) interface application formed the basis for our earlier user interface. MOM enumerates the available Mobility/CORBA services, allowing the user to launch different views for the various interfaces supported by each object. Data can be viewed as vectors, as interpreted status strings, and for many types, as 2-D graphical displays. Individual objects may be examined at several levels of detail, and using the 2-D displays, the robot may be teleoperated with the actuator interface views. MOM

is a very useful, but architecturally limiting system for testing and debugging the robot software and hardware using the network over which the automated systems will have to run.

3 Application and Development Framework

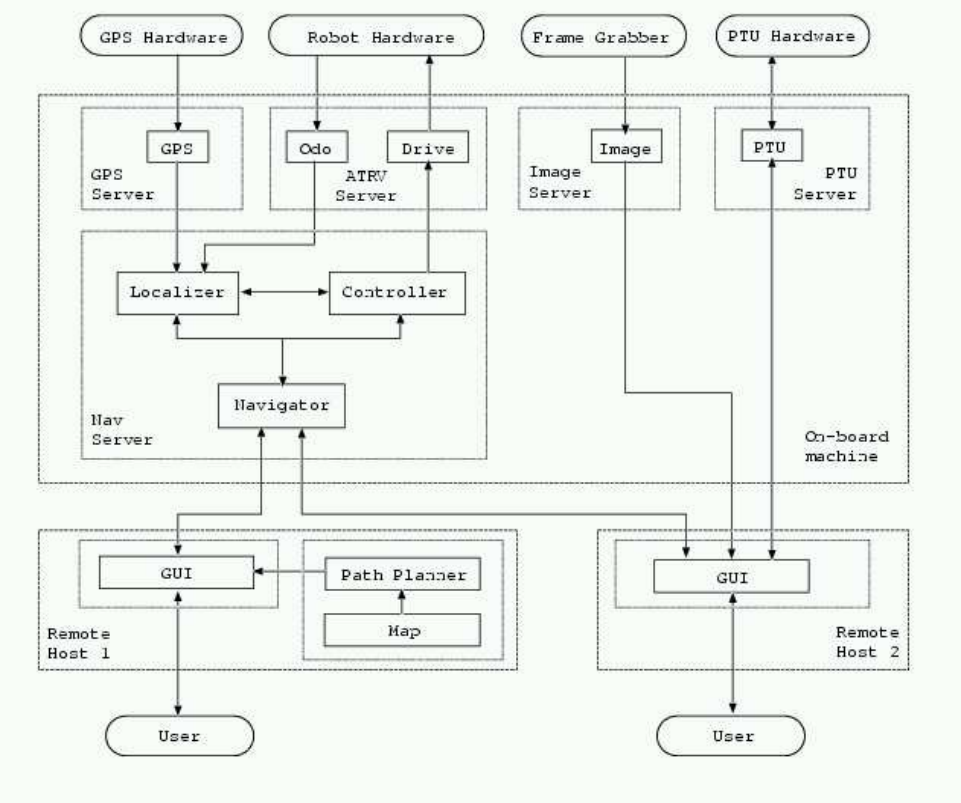

Figure 3: The AVENUE framework diagram, layered from top to bottom. the GPS, ATRV, Image, and PTU servers are all independent Mobility processes which communicate directly with the robot hardware. The UI described here corresponds to the bottom rectangles. Communication lines, represented as directed arrows, are provided by CORBA using Mobility data types. (Image courtesy of Atanas Gueorguiev [3])

3.1 AVENUE Architecture Overview

RWII's *Mobility*(tm) software is the lowest layer in our architecture (Figure 3). Designed specifically to talk to their robots, Mobility's comprehensive API includes data types for dealing with low-level robot data such as odometry, sonar and actuator hardware, as well as C++ and Java bindings for using the API over CORBA. RWII also supplies self-contained server processes which communicate with each popular hardware component they configure with their systems, taking care of all basic services for each logical part.

The ATRV server handles the robot motors, sonars, and system status via RWII's standard *rFlex* controller unit. The rFlex talks to the robot's basic hardware over its integrated proprietary real-time network. The motors, odometry, sonars, joystick, power-control, and host computer console are all under the purvue of the rFlex, which then communicates with the host computer over a RS-232 serial port. The other "Server" boxes (Image, PTU, etc.) in Figure 3 are independent Mobility processes which communicate with each independent auxiliary hardware component via individual serial ports. All servers export their data and control API interfaces over CORBA. Starting up the robot involves booting the rFlex subsystem, the host computer, starting the CORBA naming service, and then starting the Mobility servers on the host machine.

Our own software is layered on top of the Mobility services. The *NavServer* runs as another independent process on the robot computer, gathering the localization data from the other servers in its *Localizer* module. The localizer registers and fuses data from the robot's odometry, GPS and camera sensors to arrive at an accurate estimation of the robot's location and pose. Accurate location and pose data are critical to the laser scan integration steps in which all the views of a single building are fused along with camera images to complete the solid, texture-mapped model. A detailed explanation of the localization system can be found in Gueorguiev et al '00 [3]

The robot needs to know where to go to take building scan. This information is computed by a view planner[7] (not shown in the diagram) using a simplified representation of the environment to optimize the number of scans required to build a complete solid model.

The path planner is fed a Voronoi diagram based on a simplified 2-D map, and outputs paths from one scanning view location to the next. The Voronoi diagram is generated from the 2-D map of polygonal obstacles and simplified to remove all segments for which an endpoint lies inside an obstacle. The vertices of the diagram are used as navigation endpoints. The robot moves to the nearest Voronoi vertex, follows the diagram to the vertex nearest the target destination, and then moves off the diagram to the final target.[2] The user may also choose navigation targets and have the path planner calculate a route between them. The planned path is then executed and monitored by the NavServer which contains the localization component.

The NavServer accepts batch commands from a mission plan or from the user interface. It feeds commands to the controller module and monitors the progress and status of both the controller and localizer. The NavServer uses a shared format (which may be written to disk) for transmitting, loading and saving batches of high-level navigation commands across the network and between runs of the system. Experiments can be easily repeated and transferred among several robots with similar hardware configurations. Once all the scans of a building have been acquired, the complete set is processed and merged together to form a solid model [8].

The user interface (AvenueUI) lets the user monitor the autonomous progress of the robot and send immediate-mode commands such as *STOP* or *RESUME* and send batches of high-level mission commands to be executed as well. As indicated in Figure 3, the UI communicates with the NavServer for high-level control and the lower-layer servers for live data feeds. Multiple UIs may be running simultaneously, though there is no direct communication between them. This lets multiple users monitor the progress of the robot without interfering with each other. Multiple users can each control their own robot while

[2]http://www.cs.columbia.edu/~psb15/projects/path_planner

monitoring the progress of the whole team.

3.2 AvenueUI Architecture

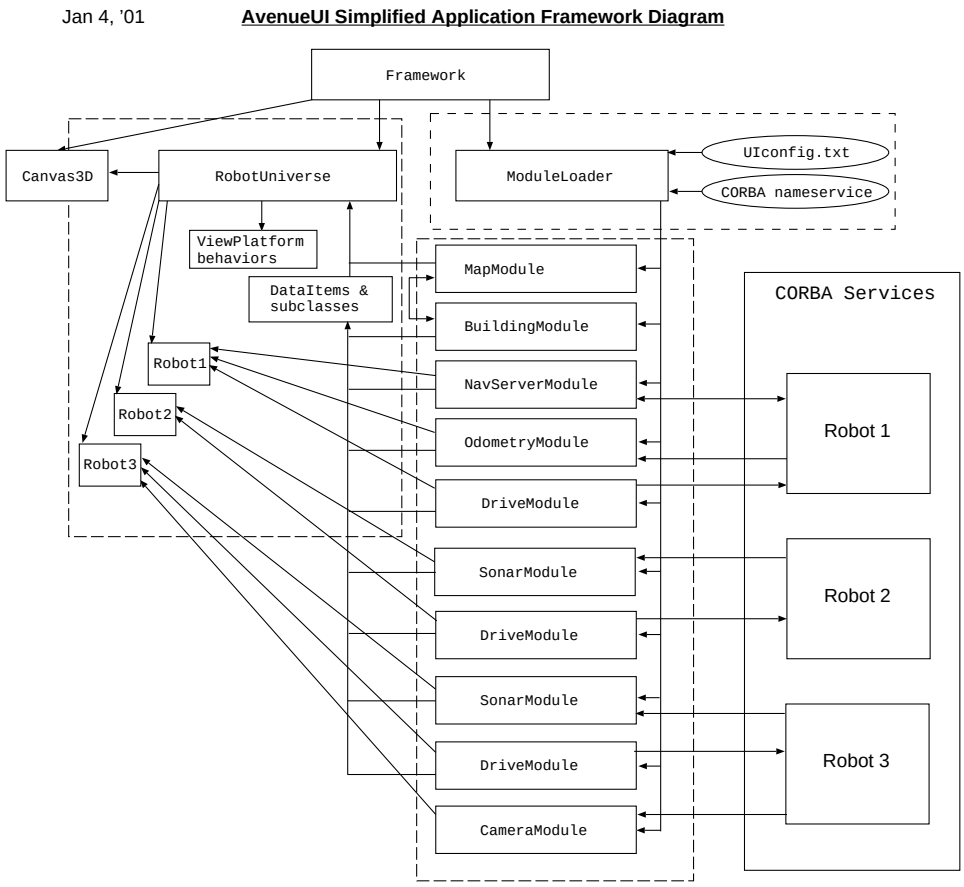

Figure 4: A simplified AvenueUI framework diagram. The boxes are object instantiations of classes, the ovals are auxiliary inputs to the system, and the arrows roughly indicate data and communication flow. Note the shared information between the MapModule and BuildingModule. Dashed boxes indicate logical object groupings.

In pursuit of the goals mentioned earlier, we opted to continue to use Java for reasons of portability, support, and development experience in the lab. Java3D was chosen over straight GL as the basis for 3-D visualization because of ease of integration and it's object-oriented nature. Since the Mobility(tm) robot control platform is already network aware and CORBA-ready, we used the vendor's hooks through the free Java JacORB implementation provided with their package. Most development took place under the Linux operating system.[3] The UI framework is designed in an object-oriented fashion both due to the development platform of choice, as well as the requirement for easy extendibility and future

[3]Primary development platform: Intel P-II/400, Linux 2.2.x, Riva TNT-1 graphics card, Java-1.2.2 (Blackdown), XFree86-4.0.1, 10/100Mb LAN and 2Mb IEEE-802.11 wireless LAN

development. The framework is logically divided into three main sections, all of which are "contained" within the *Framework* class (Figure 4):

1. The *RobotUniverse*, a subclass of the Java3D *VirtualUniverse*, which contains the scene graph, which in turn links to all the geometric primitives that are rendered, *MouseBehavior* subclasses, and all *ViewPlatforms*. The RobotUniverse maintains a list of all configured *Robots*.
2. The *ModuleLoader* which reads the config file at runtime, instantiates the described *Robots*, their associated *SensorModules*, extra ViewPlatforms, and sets all related options. The ModuleLoader also contains the ORB reference and related helper routines.
3. A collection of *SensorModules* which in turn point to their associated Robots, if applicable and all their geometry, data, and views. Modules are instantiated for each Robot that requires them.

The system contains a single virtual universe into which all objects are rendered if it is appropriate for them to be realized in the environment. A single ModuleLoader controls the contents of the universe by reading an external user-defined configuration file. The configuration file describes the critical parts of the system, including a definition for each Robot to be manifested and the sensor/actuator complement on each one. Java's Reflection API is used to load classes and instantiate objects dynamically at runtime. Options governing the behavior of the various SensorModules are set within the config file as well, and many are editable by the user within the running system. Once the config file is completely loaded, the ModuleLoader starts up all the SensorModules together and passes control to the user.

The UI supports an arbitrary number of Robots within resource limits. Each Robot has its own set of SensorModules with their own configuration-specific options. Robots may be of different types with different sensor configurations. Figure 4 shows three instantiated Robots with their independent SensorModule sets, hence the duplication of the *DriveModule* and *SonarModule* boxes. Mixing robots and environments is easy, and we use our ATRV2 and Pioneers together in our lab for testing, and the ATRV2 alone outside. Robots need not be compatible with the Mobility(tm) platform, though currently all our mobile platforms are. Modules can fetch data from local files, devices, or network sockets as necessary. No explicit restrictions are placed on module communication. Modules may also share data such as maps cooperatively.

Aside from the main rendering thread, the activity in the system is centered around the SensorModules. Most (but not all) SensorModules are associated with a single robot, if appropriate. Figure 4 shows the *MapModule* and *BuildingModule* as Robot-independent, but sharing configuration information (top of modules group). Modules all run their own threads, continuously if necessary. Data acquisition and rendering are both usually implemented in each module's time-delayed thread loop. Modules are each given control over their own data acquisition threads. Generally, a module will poll it's networked data object at a user-definable interval and update its displays and scene graph geometry accordingly. Certain modules, such as the 2-D map or building-rendering module, simply exit their threads once their data is loaded and attached to the scene.

3.3 Networking

Most communication takes place via CORBA objects. RWII's Mobility(tm) platform is centered around CORBA, and the UI framework provides some facilities for dealing with CORBA objects so that modules don't need to do it themselves. There is only one ORB instance in the system, and the Name Service is set at a single point in the config file, set by the user for the given network environment. While both *push* and *pull* methods of data acquisition are available via the CORBA interfaces, pull is used because it allows the application/user to limit the data transmitted over the network to avoid saturation and maintain responsiveness. Modules may also open sockets at will, or invoke any other sort of communication channels. Because all CORBA communication takes place over TCP/IP, the communicating processes may run either on the same host, as is often the case for our Pioneer robots, or on several separate machines.

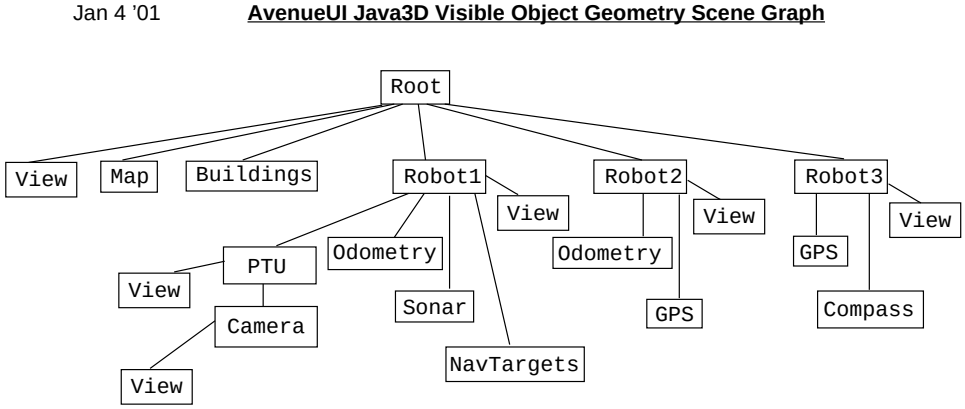

Figure 5: A diagram of the Java3D scene graph. Each box is an entity which creates visible data in the 3-D world. Sub-children are robot transform-dependent sub-trees. Traansformation matrices are inherited along the branches of the tree.

3.4 Scene Graph

Java3D uses a scene graph rather than a matrix stack-based state machine to organize primitives in the virtual universe. Nested branches in the scene graph inherit the geometric transforms attached to their parent nodes. As a result, scene data layouts are logical and mesh very cleanly with the Java object model. The structure of the Java3D scene graph is leveraged to provide robot-location dependent rendered sensor geometries. Modules may attach their scene branches to the root of the scene graph if they use absolute coordinates (MapModule) or to a specific robot's branch so that their data is rendered with a consistent location and orientation relative to the robot. Figure 5 shows three robots with several module geometries attached to each. The *Map* and *Building* boxes are Robot-independent geometry branches attached directly to the RobotUniverse's root *Locale*. The Robots' transform matrices are updated as their locations and orientations are set by modules which handle the relevant data. The scene graph closely mirrors the structural hierarchy of the actual robot hardware. The development API allows SensorModule geometries to be nested within each other as well as directly on a Robot, simplifying complex dependent

configurations like arms, or in our case, a camera mounted on a PTU, mounted on the robot.

3.5 Robot Location Arbitration

Much of the data in the system is rendered as geometry relative to the robot in question. Accordingly, the Robot class maintains an authoritative location, heading, and appropriate transform matrix in the Java3D scene graph under which all visible robot-relative data is attached. More than one module may set location data on a single robot, so an arbitration scheme is used to ensure that the most accurate data is used as the authoritative robot position. Modules which set the robot position and heading all accept a "priority" config-file option by which the user defines the accuracy precedence of the data delivered by the modules.

In the case of our ATRV2, we have odometry, differential GPS, and a higher-level localization system[3] which fuses odometry, GPS, and vision data. Clearly the localizer has the best idea of where the robot is, so it is assigned the highest priority. GPS and odometry are something of a toss-up. Under ideal conditions, the differential GPS gives centimeter accuracy, but in practice the number of satellites our antenna can see is limited by the very surrounding physical architecture that we want to model. The Robot has a definable priority timeout, set by the user in the config file, which is based on the expected update frequencies of the various location-setting modules. Our GPS updates at roughly 1Hz, odometry is polled at 5Hz (we can change this), and the localizer is also polled at about 5Hz.

The simple degradation function takes location, heading, and priority as arguments, and operates as follows:

```
if (modulePriority >= lastUsedPriority)
           then { set location & heading }
if ((modulePriority < lastUsedPriority) && (timeSinceLast > timeout))
           then { set location & heading }
otherwise
           { ignore }
```

This provides graceful degradation of the visualized robot's location. All arbitration takes place via the shared Robot object, so the modules do not need to be aware of each other and can be used or ignored as necessary for development and experimentation. By the same token, if the highest-accuracy service should fail, location/heading will be automatically set from the next available data after the timeout. If/when more accurate data arrives again, it will once more become authoritative. An additional feature of this behavior is that modules may be commented out of the config file without requiring physical robot reconfiguration on the part of the user.

It is important to note that this arbitrated data is *not* fed back to any of the robot control software. Only visualization is affected by this behavior. The UI does not make any behavioral decisions for the actual robot beyond what the user specifies explicitly. Since a version of the path-planner is instantiated in the UI at this stage, it must be invoked by the user, and all the resulting navigation targets are created as editable batch commands along with the manually-specified targets. Batch commands are only sent to the robot for execution explicitly by the user.

3.6 Development

New sensor, actuator, and other data support is added by subclassing the *SensorModule* abstract class. The SensorModule class provides the API required by the framework for managing dynamically loaded components. SensorModule subclasses are not known to the framework until read from the user config file at runtime, thus enabling rapid robot re-configuration without code compilation.

New modules inherit a rich set of framework methods and fields which greatly speed development and system integration. Among these are default options, option parsers, live option-update callbacks, thread support, nested scene graph support, and data management utilities. As the capabilities of our robots evolve, support for new features are often generalized into the SensorModule API.

The framework also provides a basic *DataItem*, which is already a subclass of the Java3D *Shape3D*, to simplify data tracking and matching visibly manifested geometries to real-world data. Classes for managing variable and semi-fixed ageable sets of these data types are also provided. Some modules create a variable number of DataItems, adding more as new data comes in, and deleting old ones when they expire, if applicable. Others use a fixed number of DataItems and cycle their data through them in a queue arrangement, moving items around as necessary and switching individual node visualization on and off as necessary.

MouseBehaviors are developed as separate classes from the modules, conforming to Java3D's *Behavior* specification. The Java3D scene graph and Behavior API provides routines for performing geometry intersections along subsets of the scene graph and callbacks to handle picked items. When the user clicks into the 3-D space, the behaviors intersect the generated *PickRay* with the geometry objects of the scene graph for which they are responsible (as determined by the scene graph). The results can then be processed either within the Behavior itself or by the SensorModule that originally registered the Behavior. The Behaviors vary in complexity depending on what is required by the module. All ViewPlatforms (and *ModuleViewPlatforms*) share a standard set of translation and rotation behaviors which only modify the current ViewPlatform. By contrast, the Behaviors used by the SensorModules usually interact with the objects in the environment, or cause actions to be taken by the robot. Since many modes of interaction don't quite map from one module to the next, code re-use among behaviors is generally limited to cut-and-paste.

A complete module might consist of a primary SensorModule subclass, zero or more Behaviors, a custom DataItem subclass, and any number of support classes. Only the SensorModule subclass is specified to the system in the config file, leaving the rest of the class loading to the internal module package import statements and the Java classpath mechanism.

A new SensorModule is developed by first sub-classing the SensorModule abstract class. The programmer must provide an *initModule()* function to initialize the module and its options, at the minimum, before the module's main thread is started. This tells the ModuleLoader what options are valid for the module, among other things, such as setting up visible geometry data structures, initializing windoids, input behaviors, and data queues. The Framework class uses this basic initialization information to properly construct the applications main menus. The *updateOptions()* callback must be overridden if live configuration updates are to function properly. Network connections, data retrieval, and geometry manifestation all occur inside the *run()* method which controls the module's primary thread. The user/programmer configures the new module into the system by adding a module dec-

laration line to the config file along with any necessary associated module option settings as defined by the module. As long as Java can find the new module in the user's *CLASSPATH* environment, the user need only run the application to launch the new module. No other special integration is required.

4 User Interface

4.1 Interaction

The user primarily interacts with the system via the Java3D Canvas window topped with the main menus. The user flies through the space by dragging the mouse in the 3D window. Sensor Modules may also accept input in the main 3D window via special Java3D MouseBehaviors. Unmodified mouse actions always control the current view, while keyboard-shifted mouse actions are directed to the currently "active" SensorModule. Modules are activated for 3-D user input by selecting them from a menu of registered input-accepting modules. Modules may also create independent windows in which to display numerical data (or anything else) and accept input. These input/output windoids [4] can be very useful for basic module development without requiring a potentially complicated or un-intuitive interaction with the 3D space.

Any given robot in the system has an arbitrary number of modules associated with it. Modules may take visualization or actuation input via the mouse in the 3D canvas. Clearly one standard set of mouse behaviors is inadequate for interacting with all the modules. Not only would the two degrees of freedom be insufficient for controlling the possible large DOF offered by the modules, but the input behaviors might be completely counter-intuitive. Consequently, behaviors are written on an as-needed basis.

The *NavServerModule* provides a *NavTarget* creating and picking behavior. When the NavServerModule is enabled for 3-D input, clicks onto the 3-D canvas are intersected with all NavTarget DataItems. If the mouse's PickRay intersects a NavTarget flag object in the 3-D universe (figure 6) then it can be dragged around on the ground for positioning. If no targets are in the path of the pick ray, then the ray is intersected with the ground plane and a new NavTarget created at that point. Currently the NavServer's target-creation behavior only allows creation of targets on the ground plane. The PathPlanner can be invoked between any two NavTargets, and the listbox interface to the NavTargets sends the NavTargets to the NavServer for execution as a batch.

The *DriveModule* converts the user's ground-plane-intersected clicks into a robot-centered polar coordinate system. The clicks are used to drive the robot directly in immediate-mode in the direction of the selected point while the mouse is down. This contrasts with the NavServerModule's batch target execution model. The DriveModule also maintains an independent windoid which allows the user to drive the robot in immediate-mode reliably regardless of the current view or 3-D input-active SensorModule.

The *PTUModule* uses a Cartesian coordinate system to allow the user to point the Pan-Tilt Unit within the limits of the device. The ViewPlatform configured by the PTUModule pans its view around the scene as the physical PTU moves on the robot.

[4] in this context, windoids are small, secondary application windows which are not necessary for the complete functioning of the system and can be shown or hidden at will.

4.2 Multi-mode Visualization

Methods and fields inherited from the SensorModule abstract class simplify and enforce consistency on the behavior and management of sensor modules. Many modules create their own small windows for displaying numerical data in addition to rendering appropriate events into the 3D universe. This multi-mode display lets the user pay attention to exactly what s/he is interested in. Menus on the main application window let the user hide and show both module windows and 3D data at will. This is extremely useful for larger data types such as complex building models and high frame-rate video streams which have a drastic effect on UI and network responsiveness.

The middle-top windoid in figure 6 is displaying the current X/Y location and heading as read directly from the robot's odometry. Immediately to its left is the NavServerModule's status window. In the upper-right corner, immediate navigation commands are collected as buttons in a control panel. The listbox to the left contains a text listing of all NavTargets managed by the NavServerModule. The list view lets the user modify the raw NavTarget data directly, specifying such parameters as Action Type, latitude, longitude, pan angle, etc. The list window also contains mission control commands which will plan paths between adjacent targets, load/save mission command batches, and send batch commands to the robot for execution. Further down the figure is the overall UI status. The crosshair windoid at the bottom right is the Cartesian drive panel for the DriveModule which lets the user easily drive the robot in immediate mode regardless of the current View or current 3-D input behaviors.

Modules use a simple logging facility to stream live data to framework-managed log files (when enabled). These data streams are also available in a default window for each module that can be called up from a menu by the user, regardless of whether or not the data is saved to disk. The framework provides a configurable means of data recording to aid in experiments. This conveniently puts data recording in a single location in the whole system rather than tweaking each individual low-level control component on the robot itself.

Figure 7 depicts the robot's-eye view of the world in figure 6 from the point of view of the real robot on the left of figure 6. The robot has nearly achieved NavTarget #10 in its planned path around the large planters in our experimental area on campus. The planters here are shown as polygon obstacles on the 2-D map projected onto the groundplane. Simplified preliminary extruded building models overlay the 2-D map in the distance. From this view the user can accurately drive the robot in immediate-mode, control the video camera's PTU (not shown), adjust or add new navigation targets, and modify the view relative to the robot's position.

4.3 View Platforms

Multiple view platforms are supported on a per-module basis, and arbitrary view platforms can be defined by the user in the config file. For example, Prof. Allen's office window overlooks the outdoor area where most of our experiments are performed, and we have configured a view platform just outside his window. Certain SensorModules also provide their own view platforms. The camera and PTU module(s), for instance, provide a view that is attached to the Java3D scene graph at their own robot-relative branch. The view is carried along as the robot moves in the universe and the user has control over the view as constrained by the relative locations of the PTU and robot. From the viewpoint of the

camera the user can see the image overlaid into the universe in the appropriate direction at a fixed distance. When not "sitting" on the PTU along with the camera, the image is rendered into a 2-D windoid alongside the main 3D canvas window.

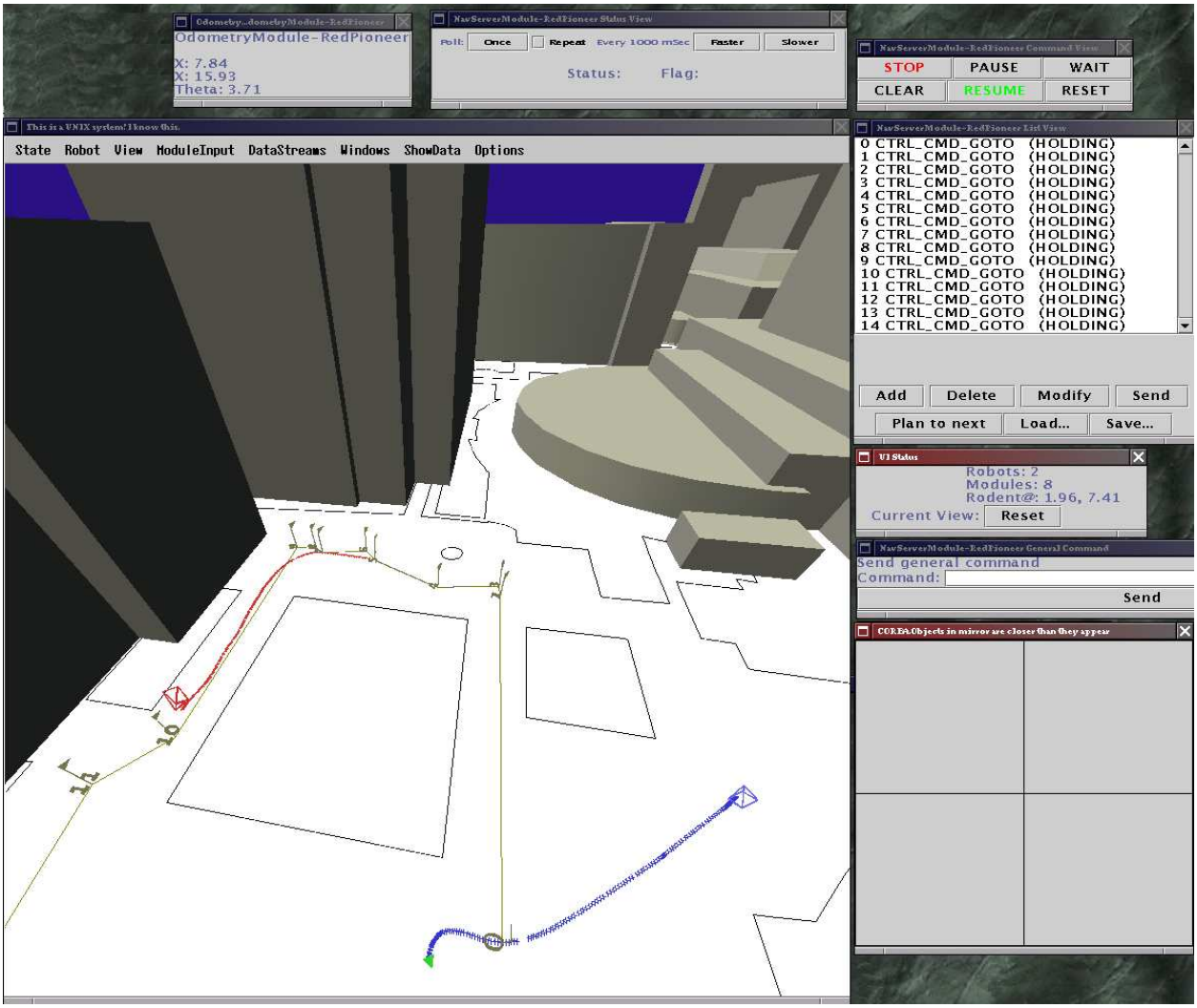

Figure 6: A full-screen picture of the UI including the main and supporting windows. Two robots track their way along the ground. The Red robot (small pyramid on the left) is following the planned path (triangular flags) around the large planter (black square outline). Preliminary low-res building models are overlaid on top of the 2D navigation map. The small triangle on the ground near the bottom of the 3-D window is our defined origin for the campus.

5 Experiments

The application has been used successfully for data gathering missions for other components of AVENUE development out on the campus. When outdoors we can either drive the robot from a workstation with a view of our experimental area or use a laptop with a wireless

connection. Usually we use the laptop on site. Since our laptop does not have built-in hardware 3D acceleration, we have to turn off the building visualization to get acceptable performance. Network bandwidth has not been an issue, however, as long as both the robot and laptop have a good line of site to a wireless base station. Pushing video data across the link does slow things down, but as with the building module, the user can turn off the video feed at will on the client end either in the config file or with the menus while the system is running. We found that our map registration, planned paths, and navigation were very accurate while conducting image-gathering experiments. The robot successfully and accurately executed our intended paths with the exception of having to be manually guided around two lamp posts which were not represented on our map. Sonar obstacle detection was not in use at the time.

During the development process we created a second indoor environment for the smaller mobile robots to ease testing. Once a simple 2D map was drawn and the associated Voronoi diagram generated for the pathplanner, the Pioneers were as happy inside as the ATRV2 was outdoors. Changing environments requires only a couple lines to be changed in the config file, though generally two separate files are maintained for simplicity. One useful cross-environment effect was simulating outdoors with the Pioneers in the lab. This lets us drive the robots outside the bounds of our wireless network coverage. While good for certain simulations, clearly sonar and video data do not match the simulated environment.

Extensibility by other programmers was a critical requirement of the UI system so as to avoid rewriting the application every two years or for every new piece of hardware. Armed with the Programming Guide, JavaDocs, and a few source-code examples, two students who did not participate in the design or development of the framework application were able to handily create two additional system modules - the building renderer shown in the screenshots, and the camera/PTU module which is not visible. It was a joy for the framework author to hear "Yeah, I had that problem too until I found it in the Programming Guide". One student reported that the framework part of the module development topped Mobility and Java3D as the easiest portion of development.

Paul Blaer developed the module which communicates with the Mobility V4Lserver interface which outputs a video data stream from a Video4Linux source. Both a regular digital video camera and an omnidirectional camera are used. The video stream is rendered to auxiliary windoids and may also be projected into the visible universe. (Figure 8)

Shlomo Hershksop developed the module which loads compressed geometry models into the visible universe and allows the user to position the models appropriately. Multiple models, transformations, viewpoints, and transparency are supported. (Figure 8)

When weather conditions make outside experiments impossible, we can test some subsystems (including the UI) either by using the indoor maps, or by raising the wheels to test navigation and multi-robot support. With the latest XFree86 drivers we found that remote display across the LAN to a 3D-enabled Sun workstation afforded surprisingly good performance as well.

6 Summary

This thesis has described an Application and Framework for a user interface to autonomous and semi-autonomous mobile robots. The focus was on usability for experiments with multiple mobile platforms both in the lab and outside on our campus and ease of continued

development. The user interface was presented, demonstrating several visualization modalities and contexts. The framework API was described at a high-level. Several basic experiments of intended use of the application were described, illustrating the successful adoption of the new interface and framework into the AVENUE system.

6.1 Future Work

AvenueUI was designed to be under continuous development as the requirements of our lab evolve. The Framework itself has settled, but module development continues. Modules to be developed include:

1. Sonar projections into the 3-D world
2. Cyrax scanner control, pending the release of the control specs by the vendor
3. Simple image renderer to map aerial photos to the groundplane

It is unclear whether or not a tighter coupling of the user to the robot's functions would be advantageous in this context. For example, the PTU could be mapped to head-mounted display for increased telepresense rather than simply teleoperation. On the other hand, too much telepresence and user-coupling may undermine the effectiveness of an autonomous system, especially if the user is competing with software components for control of the hardware.

A comprehensive interaction API for robotic arms on mobile platforms would be a powerful addition to the framework, allowing effective teleoperation such as described in [5] and [2]. Overcoming certain limitations which restrict many UI operations to the 2D XY plane (though not the robot's actual position) would further complicate the user's interaction with the system.

A simple snapshot capability for recording the state of the universe's geometry for use in a static scene viewer is on the TODO list.

There is currently no security infrastructure in the application. Access is restricted via TCP wrappers on the networked hosts. A basic authentication system is probably all that is necessary in our situation. Additionally, "sandbox" restrictions on loaded SensorModules to limit the damage a module can do to the framework could be handy, especially since a poorly written module can clog the system at load-time.

References

[1] P. Backes, K. Tso, J. Norris, G. Tharp, J. Slostad, R. Bonitz, and K. Ali. Internet-based operations for the mars polar lander mission. In *Proc. of IEEE Int. Conference on Robotics and Automation in San Francisco, California*, page 2025, 2000.

[2] S. DiMaio, S. Salcudean, C. Reboulet, S. Tafazoli, and K. Hashtrudi-Zaad. A virtual excavator for controller development and evaluation. In *Proc. of IEEE Int. Conference on Robotics and Automation in Leuven, Belgium*, 1998.

[3] A. Gueorguiev, P. K. Allen, E. Gold, and P. Blaer. Design, control and architecture of a mobile site-modeling robot. In *Proc. of IEEE Int. Conference on Robotics and Automation in San Francisco, California*, 2000.

[4] K. Hashtrudi-Zaad and S. Salucdean. Analysis and evaluation of stability and performance robustness for teleoperation control architectures. In *Proc. of IEEE Int. Conference on Robotics and Automation in San Francisco, California*, page 3107, 2000.

[5] G. Hirzinger, B. Brunner, R. Lampariello, K. Landzettel, J. Schott, and B.-M. Steinmetz. Advances in orbital robotics. In *Proc. of IEEE Int. Conference on Robotics and Automation in San Francisco, California*, page 898, 2000.

[6] J. Laszlo, M. van de Panne, and E. Fiume. Interactive control for physically-based animation. In *Proc. of the ACM Special Intrest Group on Computer Graphics and Interactive Techniques*, pages 201–208, 2000.

[7] M. K. Reed and P. K. Allen. Constraint based sensor planning. *IEEE Trans. on PAMI*, 22(12):1460–1467, 2000.

[8] I. Stamos and P. K. Allen. 3-d model construction using range and image data. In *Proc. of IEEE International Conference on Computer Vision and Pattern Recognition, South Carolina*, pages 1435–1440, 2000.

[9] D. K. T. Elvins, D. Nadeau. Worldlets, 3d thumbnails for wayfinding in virtual environments. In *Proc. of the IEEE Int. Conference on User Interface Software and Technology, San Diego, California*, pages 21–30, 1997.

[10] M. Usoh, K. Arthur, M. Whitton, R. Bastos, A. Steed, M. Slater, and F. B. Jr. Walking ¿ walking-in-place ¿ flying, in virtual environments. In *Proc. of the ACM Special Intrest Group on Computer Graphics and Interactive Techniques*, pages 359–364, 1997.

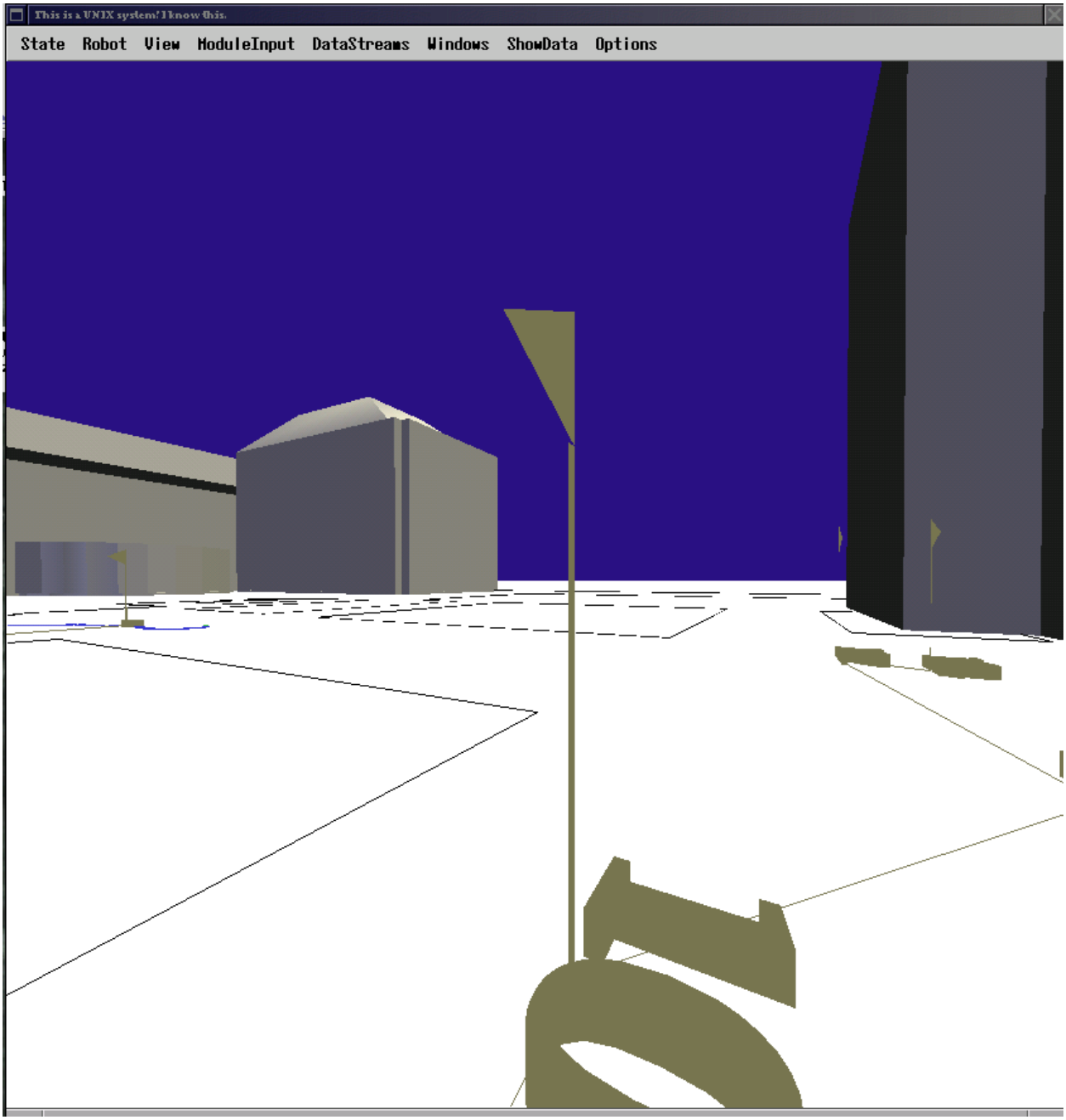

Figure 7: View from the Red robot's perspective (figure 6). Flags are navigation targets and the thin line between them is the intended path of the robot. The large numeral at the base of each flag is the navigation command index of that target.

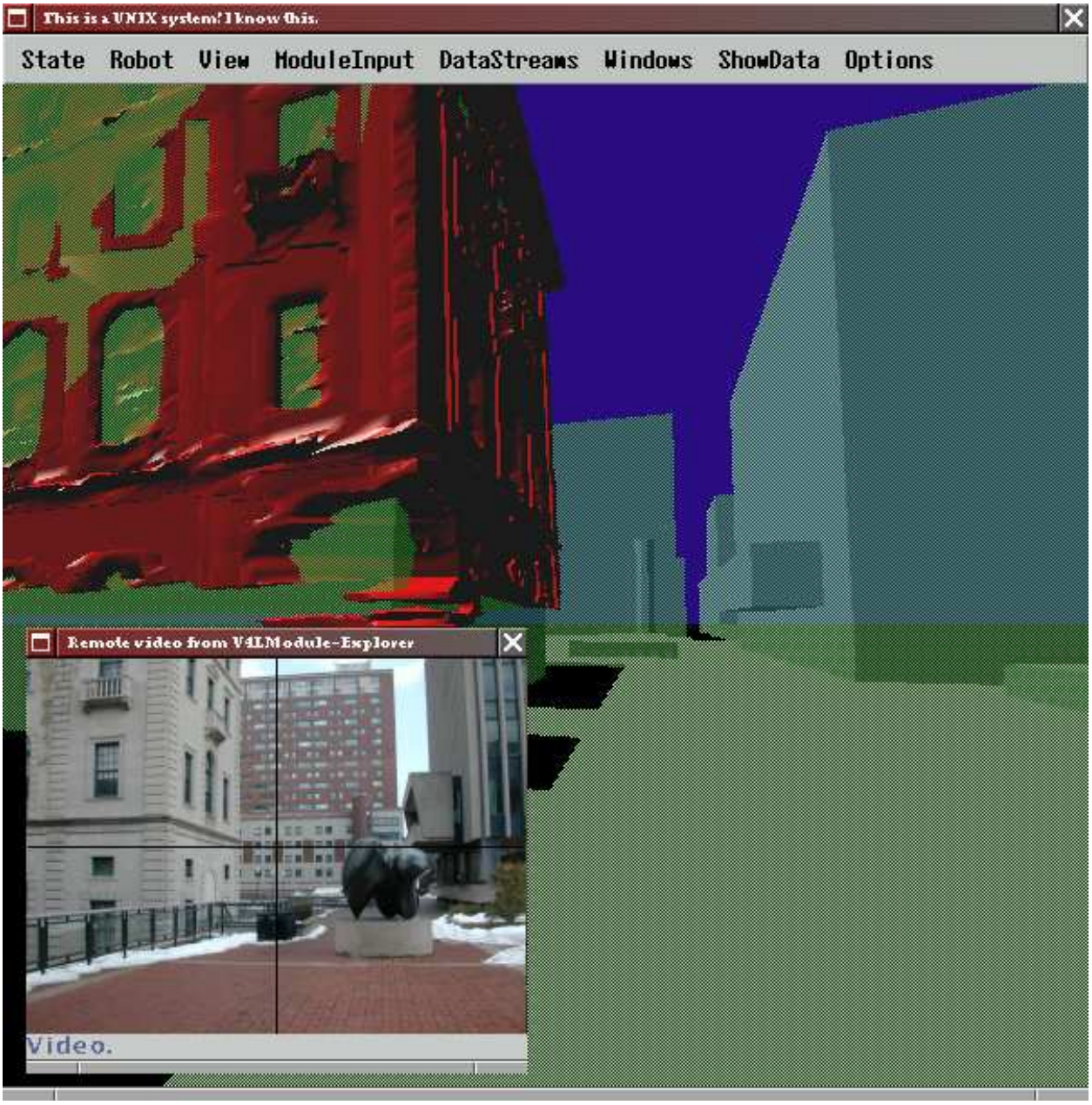

Figure 8: The scanned model of the Italian House on the Columbia campus set into it's location, overlapping the rough model which is part of the rest of the preliminary campus map.(left) Preliminary structures are semi-transparent for positioning and display purposes. The video feed from the robot's primary, front-facing camera is displayed in a separate window in the lower left corner.