Argus Visualization System

by

Eric D. Cohen

Abstract: *The Argus Visualization System (AVS) displays a real-time rendition of the Argus robot. Argus is superimposed over a campus map, and positioned according to data supplied by its navigation system. All of the major degrees of freedom are rendered in the scene based on their real-time state. The AVS can run either locally on Argus, or remotely, with a client residing locally on Argus forwarding navigation data to a remote render server. The AVS introduces unprecedented remote monitoring capability to Argus, while assisting the operator in locating data collection nodes.*

Contents

1 Introduction

1.1 Argus

What follows is a very brief introduction to the Argus platform intended only to facilitate understanding of the Argus Visualization System (AVS). For a more complete description of the system, the reader is encouraged to visit the website of the MIT City Scanning Group (http://city.lcs.mit.edu/).

The Argus robot is a device used to collect geo-coordinated image data. Argus is deployed by an operator who wheels it to the desired location to take a "node" of data. To gather images, Argus uses a high-resolution digital camera. During data collection, the camera pivots on a rotating frame attached to a telescoping tripod. To associate images with location, Argus employs an extensive array of sensors. These sensors include a GPS system, an inertial measurement unit (IMU), and wheel encoders. What Argus lacks, however, is a means of visualizing its deployments and data collection operations.

1.2 Argus Visualization System

The *Argus Visualization System* (AVS) is designed to provide a real-time 3D or 2D rendition of the Argus robot. Argus is rendered in a scene using position data taken from its various navigation sensors (GPS, IMU, and wheel encoders). The AVS supports two different modes of operation. The first mode of operation is remote mode, where the render server is running separately from the deployed Argus hardware. Remote mode is useful for monitoring the progress of data collection from the lab or other remote location. In the second mode, the render server is running directly on the Argus platform during deployment. This mode is useful for the operator attempting to locate a new node for data collection, and for monitoring the proper operation of the Argus sensors.

2 Design Overview

The AVS is designed around the OpenGL graphics library. This library provides a simple, generic, platform-independent interface to the graphics hardware. The Argus objects are rendered within the OpenGL clipping volume, containing a standard Cartesian coordinate system.

All objects are placed according to absolute coordinates, and animation occurs by moving objects to new absolute coordinates. A campus map is texture-mapped onto the X,Y plane, with Argus and the Argus icon superimposed over it for reference. Translational motion of Argus and the Argus icon is accomplished by moving the map in the opposite direction of translation. Rotational movements are carried out directly on the appropriate object(s).

2.1 File Formats

Since a requirement of the render server was to avoid the use of the Inventor libraries, the Argus model had to be manually converted into OpenGL primitives. To expedite development, a small external library was compiled into the code to allow the AVS to import Alias Wavefront OBJ files. Using a conversion utility, the Inventor model was converted to OBJ format. Then, the external library was used to import the OBJ files.

Prior to conversion, it was necessary to manually partition the Inventor model into logical Degree's of Freedom (DOF). DOF's represent individually controllable objects, such as the swivel camera or telescoping tripod. Each DOF was removed and put into its own Inventor file. This allows each DOF to be imported as a separate object. Each object can subsequently be translated, rotated, and rendered independently.

2.2 Networking

The AVS is based upon the traditional client/server model. In this case, the server is the component that renders and displays Argus, and the client is a small program running on the Argus platform that sends navigation data to the server. The system is based on the standard socket network interface, and uses the TCP protocol to exchange data.

2.2.1 Server

On the server side, a thread is started on initialization that listens on a selected port for incoming connections. When a connection is received, the

server waits for navigation data to be sent from the client. As data is received, the server updates the state variables for each DOF. Upon receiving the entire update, the server schedules the display windows to be redrawn in the render thread.

Although synchronization should be implicitly enforced by scheduling a redraw only at the conclusion of a state update, the server locks the object state mutex for added protection against race conditions.

2.2.2 Client

The client runs directly on the Argus platform, and simply sends data captured from the navigation system (or keyboard simulator) to the render server. The client also contains the external interface to the AVS.

2.3 API

The entire API to the AVS resides on the client side. There are no publicly available functions on the server side. The function naming scheme is simple and logical. All functions are prefixed with the "avs" identifier. The next item indicates the type of transformation (rotational or translational), and the last item indicates whether the transformation is relative to the origin or the current position of the object. For object-specific functions (Table 2), the name of the object to which the function applies precedes the transformation type identifier. It is worth noting that all object-specific functions, such as avsCameraHeadRotateAbs, are merely wrappers around the more general functions (Table 1) that apply to all objects, such as avsTranslateAbs and avsRotateRelative.

Table 1: General API Functions

Function Name	**Arguments**	**Purpose**
avsTranslateAbs	Object, Translation Vector	Translates an object to new absolute translational coordinates
avsTranslateRelative	Object, Translation Vector	Translates an object to new relative translational coordinates
avsRotateAbs	Object, Rotation Vector	Rotates an object to new absolute rotational coordinates
avsRotateRelative	Object, Rotation Vector	Rotates an object to new relative rotational coordinates

Table 2: Object Specific API Functions

Function Name	**Arguments**	**Purpose**
avsRotateCameraHeadAbs	Scalar Float	Rotates the camera head by the specified amount
avsRotateCameraHeadRelative	Scalar Float	Rotates the camera head by the specified amount (relative)
avsRotateUpperFrameAbs	Scalar Float	Rotates the upper frame by the specified amount
avsRotateUpperFrameRelative	Scalar Float	Rotates the upper frame by the specified amount (relative)
avsMoveTelepostAbs	Scalar Float	Extends or retracts the tripod by the specified amount
avsMoveTelepostRelative	Scalar Float	Extends or retracts the tripod by the specified amount (relative)

2.4 Camera Views

The render server provides two camera views; one is two-dimensional orthogonal, and the other is three-dimensional perspective. Each view occupies its own window, and both can be observed simultaneously.

2.4.1 2D View

The two-dimensional camera view (Figure 1) allows the user to quickly ascertain the relative speed (by observing the rate the Argus icon appears to be moving relative to objects on the map), heading, and position of Argus at a glance. Argus is represented by a simple polygon-based arrow, with the tip pointing in the direction corresponding to the front of Argus. Z-axis (up and down) data is ignored in this mode. The result is similar to the view one would perceive looking directly down on Argus.

Moreover, the 2D mode is orthogonal, meaning that there is no perspective. The result is uniform perceived size and velocity regardless of the distance of the arrow from the camera position. The camera always looks in the North (top) direction in this mode.

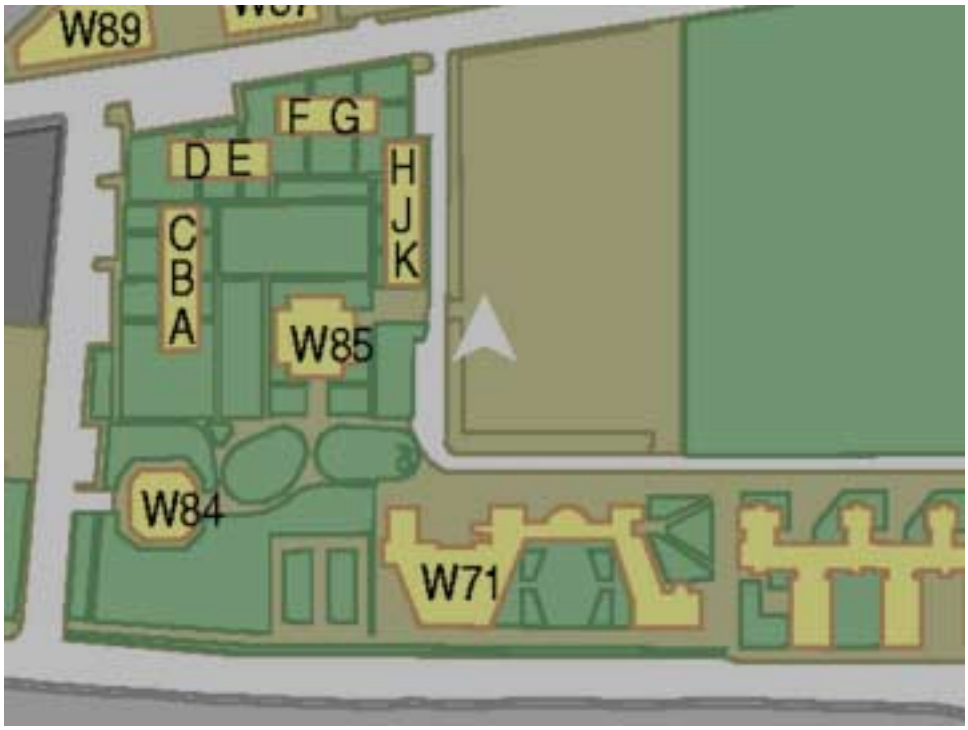

Figure 1: 2D view of Argus.

2.4.2 3D View

The three-dimensional camera view (Figure 2) shows the fully rendered Argus platform, with perspective. The camera always looks down on Argus from an angle, and looks toward the top of the screen. The 3D view allows one to observe the various different DOF's in action. For instance, when the Argus camera is extending and scanning, the camera on the rendered model extends and moves accordingly.

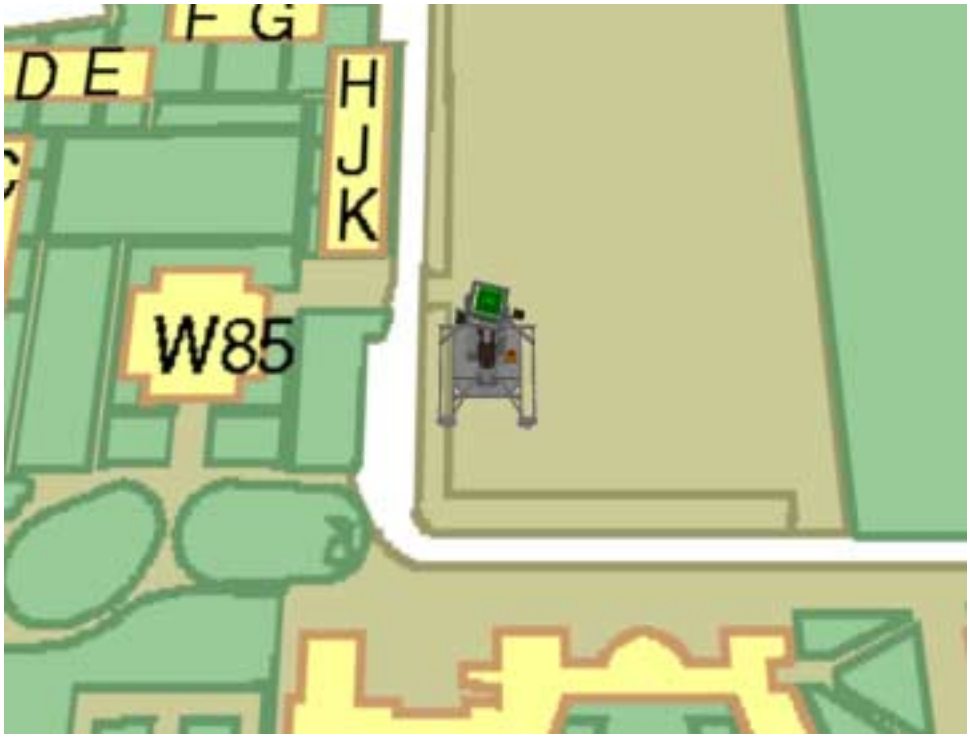

Figure 2: 3D view of Argus.

2.5 Object Hierarchy

With each pass through the render loop, Argus objects are rendered according to their position in an object hierarchy, depicted in Figure 3 as a directed, acyclic graph. This allows objects to move implicitly with respect to their parents. For example, when then camera frame rotates, the camera body should also rotate. A flat (non-hierarchical) object model would require the state variables of the camera body to be explicitly modified, in addition to the state variables of the camera frame.

The hierarchy is based around an array of *obj_state_t* data structures. Each element of the array corresponds to a single DOF object of the Argus model. The DOF's are assigned to indices of the array by assigning integers to the nodes of the graph in Figure 3. The root node is assigned index 0, and subsequent nodes are assigned by proceeding from top to bottom, and left to right.

The indices alone, however, do not contain enough information to build the hierarchy. It is also necessary to include in each object element two additional integer values. These are the number of times to push the model view

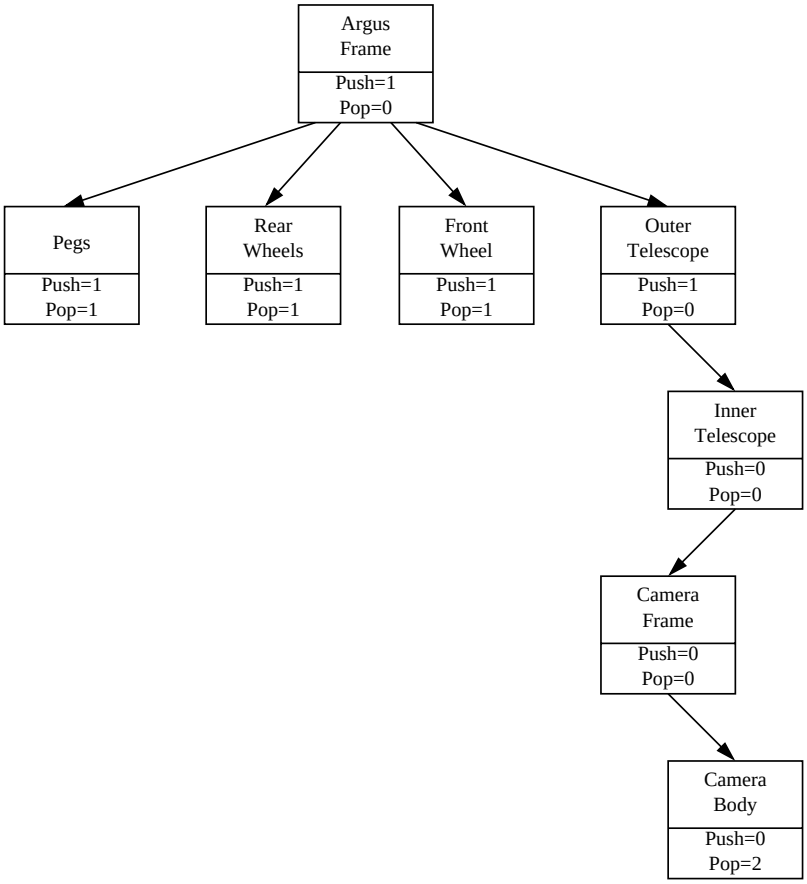

Figure 3: Object hierarchy tree.

matrix on entry of the object into the render loop, and the number of times to pop the matrix on exit of the object from the render loop. By pushing the model view matrix, it is possible to preserve the state of the parent node, while using that state as the reference system for the children. Once oriented in the scene, the children pop the model view matrix as necessary, restoring the parent state in LIFO order.

2.6 Render Loop

Each time a state variable changes, the *glutPostWindowRedisplay* function is called to schedule the display window to be redrawn by OpenGL. The draw function accomplishes this by looping through the object array, and rendering each DOF individually. The hierarchical constraints are satisfied by rendering the objects in ascending order, and by executing the appropriate

number of push and pop operations for each element. Although this system is more complex then simply hard coding each DOF into the loop, it has the advantage of being generic. As new DOF's and/or objects are added to Argus, no modifications need be made to the render loop. All that is required is the initialization of a new element object with items such as file name, number pop's, etc.

2.7 Texture Tiling

The campus map that Argus is rendered on must be texture-mapped onto a plane. However, the maximum texture map size is implementation-dependent (the value can be obtained with the glGetIntegerv function), with most sources recommending a size no greater then 256x256 texels. Since the campus map is 4096x1024 (over 12MB), it would not be wise to depend on being able to load the entire image onto a single surface. Instead, the texture map is broken up into smaller tiles, which are assembled contiguously to create the full map[1]. The tiles are composed of triangle strips, the optimal shape to work with in OpenGL.

Unfortunately, this method causes some problems when linear filtering is employed. In particular, the filtering algorithm attempts to average pixel values across the border of the texture map image. The result is subtle lines throughout the image, corresponding to the joining of individual tiles. The solution is to set the border of each tile one row or column inside the tile. When the linear filtering algorithm crosses the border for its last iteration, it gets data from the current tile, instead of extraneous data from outside the tile. It seems that this results in a severe performance degradation, however.

2.7.1 Software Frustum Culling

Although OpenGL provides automatic clipping for objects outside the viewing volume, it does not provide a function to determine if a particular object, after being drawn, will be visible in the scene. This can result in requests to draw objects even though they are completely outside the viewing volume. The objects will not be visible, but the act of attempting to draw them can decrease overall system performance. While this performance degradation is negligible for a scene with few small objects, it can become significant in a scene where many objects are drawn, many of which fall outside the viewing volume.

With the AVS, it is often the case that many of the texture tiles fall outside the clipping volume. With the border cropping enabled, rendering of

[1] The algorithm is based on one presented in a course at SIGGRAPH 1996

these objects can take significant time and power, resulting in a poor frame rate. To prevent drawing unnecessary tiles, a test is performed within the render loop to determine if a particular tile will be visible in the scene. If so, it is drawn, otherwise, it is not. An additional bonus of this method that it allows for acceptable performance on machines with limited texture memory, since the tile elements are quite small, and it is not expected that many tiles will be visible at once.

The algorithm to carry this out, based on Mark Morley's excellent Frustum Culling tutorial[2], multiplies the current model view and projection matrices to extract the actual viewing volume (the projection matrix contains the parameters of the clipping frustum, and the model view matrix contains the orientation of this frustum). This operation is carried out once for each redraw of the campus map. Before each tile is drawn, the function polygonInFrustum is called with the vertices of the tile to be drawn as arguments and returns TRUE if the tile is entirely or partially within the viewing volume, and FALSE otherwise.

3 Future Work

Like any evolving project, there are a nearly infinite number of possible directions for future work on the AVS. Some of the more important short-term issues are outlined below.

3.1 Integration of Client

Currently, the client gets its navigation data from keyboard inputs of the remote machine. While this is useful for testing purposes, it does not represent data from actual Argus deployment. This requires integrating the client into the Argus code base. This should be a simple matter of tapping into the navigation system data stream, and updating the client state variables as necessary.

3.2 Argus Object Quality

As described in Section 2.1, the Argus frame and associated DOF's are converted from Inventor format to Alias Wavefront format by a third party utility. The resulting OBJ file is then converted into OpenGL primitives by a small library. There appears to be a significant quality degradation

[2] http://www.markmorley.com/opengl/frustumculling.html

from this process, however. Among other things, the winding on some objects is incorrect, small details are lost, and material properties are not fully preserved from the Inventor model.

The solution to this problem is to write a small utility to directly convert the Inventor file format into an OpenGL display list. This is a one-time conversion, and the resulting model can be compiled directly into the server or stored in a binary file to be loaded on startup. The process, although tedious, is tractable, as the Inventor format is publicly documented.[3] Additionally, the *ivfix* utility can be employed to convert the Inventor description format into a list of vertices, normals and indices. From this state, it is merely a matter of writing a Perl script to parse the file and output a display list.

3.3 Network Performance

The obvious use of the AVS is to remotely monitor an Argus deployment. Ostensibly, communication between the client (Argus) and server (the remote render server), would occur over the campus-wide 802.11b wireless network. Wireless networks provide mediocre peak performance, and poor quality of service in general. They are susceptible to spurious RF noise (such as that emitted by the Cogeneration facility), multi-path reflections, and signal attenuation through walls. Thus, to ensure acceptable performance, a more intelligent client/server system should be developed.

Unlike the current system, it should only send data that is changing, instead of re-sending all state variables on each update. Additionally, the size of the state variables can be decreased by a factor of two by packing floating point values into short (two-byte) fields, at the expense of losing accuracy. To compensate for a brief null in the wireless signal, a physics package could attempt to predict the orientation and location of Argus over small time periods, preventing jerky movements on the render server.

4 Logistics

The source code will be located in the CVS repository for the City Scanning Project (CVSROOT = /u5/city), under the 'city_base/src/avs' directory. Within the 'avs' directory, there are two subdirectories. The first is the 'avsserver' directory, which contains the code for the render server. The second directory is the 'avsclient' directory, which contains the code for the client. To compile the system, a new project should be created in Visual

[3] See http://www.sgi.com/software/inventor/PostScript/translator.ps

C++ for each. Both the client and server can then be compiled. Note that the server depends on both the OpenGL libraries and the GLUT libraries.

Although no provisions have been made for compilation on a Unix platform, it should be very straight-forward to port the server. Since the GLUT libraries are used, there should be no significant platform dependencies for the graphics module of the server. The networking module uses the winsock library, which is very similar to Unix sockets, so only minor modifications need be made. Like the server, the client is written using the winsock library. Since the client is little more then an RPC stub, porting to a Unix platform should be extremely easy.

5 Conclusion

The primary goal of this project was to provide a graphical representation (in both two and three dimensions) of the location of the Argus robot and the state of its various DOF's. The Argus Visualization System achieves this goal. However, the ability to render Argus locally is of limited utility. The real value of the AVS is the ability to monitor, remotely and in real-time, deployment of the Argus platform.