Implementing Virtual Robots in Java3D using a Subsumption Architecture

Nathan Smith, Christopher Egert, Elisabeth Cuddihy, Deborah Walters

Department of Computer Science and Engineering
State University of New York at Buffalo
Buffalo, New York 14260-2000
[npsmith | egert | ecuddihy | walters]@cse.buffalo.edu

Abstract

New web-based technologies for 3D graphics such as Java3D provide tools for creating autonomous virtual reality agents for educational, entertainment, and simulation software. Such agents need to be responsive and reactive to their environment in order to maintain a steady appearance of behaviors. An extension of a subsumption control architecture, from robotic agent control literature, can provide the basis for these requirements. This paper discusses issues in the design of a virtual reality robot implemented in Java3D using a modified subsumption control architecture and the problems that may be encountered with 3D platforms that provide virtual reality development environments. The virtual robot we describe is able to successfully perform a set of behaviors within its environment. Virtual sensors and actuators are implemented within Java3D, allowing the virtual robot to sense and interact with its environment. Overall performance of the system is taken into consideration.

Introduction

The potential for creating interesting virtual reality based agents has become more feasible with the introduction of various high-level 3D graphics standards for web-based applications. Virtually embodied autonomous agents can be created using these tools for education, entertainment, and simulation software. The difficulty in creating such agents is providing a robust control architecture which is capable of maintaining the appearance of continuously believable, life-like behavior. An interesting autonomous agent needs to be responsive and reactive to its environment. Furthermore, it should select actions that appear intelligent with regard to its situation. This poses the problem of how to create an autonomous agent that "thinks fast on its feet."

Much work in robotic control architecture research has been devoted to creating fast, reactive agents that are capable of responding to their environment swiftly and in a manner that appears intelligent. The subsumption architecture [Brooks85, 90] was specifically designed to create a reactive control system for mobile robot. It is a computational architecture that allows tight coupling of perception to action via a substrate of computational incremental layers. The substrate consists of networks of augmented timing finite state machines [Brooks90]. Thus, the agent is defined by a set of behaviors it exhibits and by the relationship between those behaviors. The appeal of this architecture is that it doesn't rely on planning or complex internal knowledge representation in order to make decisions on what to do next. Instead, the control architecture continuously selects actions for the agent to perform based on the agent's immediate perceptions of its environment and on the agent's current internal state. This allows the agent to react to its environment and provide a steady appearance of life-like behavior.

Using virtual reality to develop this control architecture is quite advantageous even though subsumption architectures have seen little use in applications involving virtual, rather than physical, robots. Virtual robots can be rapidly and inexpensively reconfigured to create new ones. Similarly, environments can quickly be redesigned to embody different sets of interconnections. Virtual sensors can be created to provide different sensory experiences, such as image capture and processing as a virtual analog to video capture and analysis techniques currently being explored at a hardware and firmware level [Lorigo97]. Brooks has argued against the possibility of an adequate simulation of a physical system claiming that complex agent behavior is a reflection of a complex environment and that virtual reality is not rich enough to provide this level of stimulus [Brooks91]. Although this viewpoint is valid for the development of real world robots, if an agent's purpose is to ultimately function within a virtual environment, then the most appropriate place for its development is in virtual reality.

This paper describes issues in using Java3D to develop a modified subsumption architecture for controlling a virtually embodied, rather than physically embodied, autonomous agent. We describe the implementation of a virtual reality robot with insect-like behavior. The virtual robot is designed to incorporate a simple set of behaviors that enables it to wander freely on a playing field but never wander off of the field, avoid other objects while wandering, and return to a nest area after wandering for a sufficient amount of time.

VRML vs. Java 3D

After making the decision to use a high-level 3D platform for a virtual environment and virtually embodied autonomous agent, it is important to choose one that would support the implementation of the subsumption architecture. Most particularly, the platform needs to support simulated sensory devices and simulated robotic actuators. Two packages worth consideration are the Virtual Reality Modeling Language (VRML) [VRML97] and Java3D [Sowizral97].

VRML's main advantage is that it is currently the de facto standard for web based 3D visualizations. VRML allows for easy definition of geometric shapes in hierarchical groupings and it also provides many advanced 3D graphics functions such as lighting models and surface materials. VRML allows for simple interactions between a user of a virtual world and the various objects within the world, such as clicking on an object to activate an object's script. VRML is currently well supported with various end-user browser and modeling programs, simplifying the task of both creating and viewing virtual worlds.

There are, however, numerous difficulties in using VRML as a basis for a virtual world that can support interesting autonomous agents. The first problem is the lack of a mature collision handling mechanism within VRML. Although VRML supports collision handling between the user's viewpoint and the scene objects, there is no support for object to object collisions. This creates a serious limitation when trying to implement virtual touch sensors for a VRML-based agent. The second problem with VRML is the relationship between the virtual objects and the VRML browsers. VRML is only a scene description language. In order to create a complex and dynamic environment with VRML, it is necessary to use the Java External Authoring Interface (EAI) [VRML98] to programmatically control the visual representation of the environment. Originally developed as an addendum to VRML, EAI does not have a tight enough coupling between the user and the browser to extract useful runtime information about either. Although it is possible to use EAI to determine property and coordinate information about objects in a scene, it is not possible to determine the user's viewpoint or to capture graphical frames from the point of view of a simulated visual sensor.

Java3D provides a purely object-oriented language-based approach for designing 3D system. Built as an extension to the Java language, Java3D offers a high-level Application Programming Interface (API) for 3D scene description and graphical control. In this sense, it offers some of the same advantages of VRML while also providing tight integration with a fully capable programming language. Furthermore, because it is a Java API, Java3D allows for a fully object-oriented approach to defining and controlling the virtual agent and its environment. It is capable of providing better integration of three-dimensional content, interface and events within the system. Java3D's sophisticated event model allows for interesting object interactions, such as object to object collections, as well as a unified interface between timer and scene change events. Java3D is also designed to take advantage of multi-threaded programming techniques, allowing for better performance from the implementation.

Although Java3D provides better mechanisms for the construction and control of a virtual environment, it does present some difficulties. The relative newness of Java3D versus VRML along with its complexity means that development of a system would have to include appropriate measures to train developers in Java3D's use. Also, due to the lack of 3D modeling programs capable of exporting to Java3D, initial environment designs would be simplistic. Java3D also suffers from a lack of support within web browsers. Many of these difficulties are rapidly being overcome. The release version of Java3D includes support for VRML models within a Java3D scene as well as Java plug-in support to include Java3D applets on the web.

Java3D VR-Bot Design

Two different approaches can be taken in the design of the virtual robot using Java3D. The first is a strong object-oriented model with object-level event handling. This approach may be taken in order to support hierarchical component abstraction within the system. At the highest level, separate objects encapsulate the functionality of the environment, obstacles on the field, and the virtual robot. This functionality includes methods to retrieve object status and to enact object actions. Any object in the scene is instantiated as a specialized case of a general base object. The virtual robot's construction consists of several sub-objects, including the body, sensor array, individual sensor elements, event handlers, and a visual platform binding for first person navigation of the world. The strong object-oriented model allows for easy reconfiguration of the virtual robot at the base class level and it also provides a useful learning tool, as many are only familiar with physical implementations of subsumption architecture and not Java3D. This model separates the virtual robot from its intermediate control layers, allowing development of different layers to proceed separately.

Although the object model's abstraction does prove advantageous for visualizing the hierarchy and dependencies within the system, it is less than optimal with regard to performance. This could be attributed to non-optimized event handling from the Java3D event system. The abstractions of individual handlers tend to be counterproductive to performance, and the dependency on events does not encourage the exploration of multi-threaded designs. In order to counter these effects, the second approach focuses on the development of a robust thread model to achieve a higher degree of parallel and asynchronous activity. The advantage of this new model is that more reliance is placed on native Java threads that can be managed and optimized manually, rather than Java3D threads to which application programmers have no access.

The major components of this model are the environment, the sensors, the modified subsumption architecture, and the robot. The subsumption architecture receives, as input, signals from the sensors whose states change as the environment changes. Signals coming out of the subsumption architecture are interpreted as commands to move the robot around in its environment. The environment consists simply of a few elements: a field, a set of obstacles, and a set of perimeter markers. The obstacles are there to test the robot's obstacle avoidance behavior, and the perimeter markers are there to test the robot's penning behavior. The robot's obstacle sensors are sensitive to every obstacle on the field, and whenever an obstacle is in a sensor's range, that sensor is made active. Any sensor whose field is empty of objects to which it is sensitive remains inactive. Similarly, the robot's perimeter sensors are sensitive to all the perimeter markers, and their activity status depends on the presence of perimeter markers within their range.

The subsumption architecture itself consists of four layers: a wandering layer, a penning layer, a homing layer, and an obstacle avoidance layer. The wandering layer's output directs the robot to randomly explore the field, the penning layer keeps it from wandering off the field, the homing layer causes the robot to return to a particular location, and the obstacle avoidance layer keeps it from colliding with obstacles.

The wandering layer has no input, since arbitrary wandering doesn't require any information. This layer chooses a direction and then moves in that direction for a random amount of time. After that time runs out, it chooses another direction and starts over. The penning layer has access to the perimeter sensor array. It polls these sensors, and whenever any of them becomes active, it maneuvers the robot directly away from the vector sum of all active sensor regions. Since only perimeter markers can activate these sensors, this has the effect of moving the robot away from the perimeter when it gets too close. The homing layer also has access to the perimeter sensor array. When activated, this layer causes the robot to move in a straight line until it encounters the field perimeter. It then follows the perimeter until it reaches a home station. This behavior would be useful if the robot was collecting objects and depositing them at a nesting location or if the robot had an energy source that needed recharging at a home station. The obstacle avoidance layer operates in an almost identical fashion as the penning layer behaves. Whenever any of the obstacle sensors become active, it maneuvers the robot in a direction that is slightly different than directly away from the vector sum of all active sensor regions. A slight random variation is added to this direction because of a special case. If the robot were to wander into a position directly between two obstacles, the obstacle avoidance layer would calculate their vector sum, find it to be zero, and tell the robot to stand still while suppressing the wandering behavior. Thus the robot would be trapped in that position indefinitely. The random variation in direction assures that the obstacle avoidance layer isn't continuously telling the robot to not move.

Technical Considerations

The development of this system has brought to light some Java3D issues and difficulties Java3D programmers are likely to face. Foremost of these is a problem involving collision detection. In the current implementation of Java3D, collisions can trigger events. The event model can report the initial entry of an object into a collision state, the removal of an object from a collision state, and the continuance of an object involved in a collision. One of the limitations occurs when two objects are in a collision state and a third one enters. Currently, the Java3D implementation will not indicate that a new collision has occurred. This problem has a direct impact on the implementation of our touch sensor since the use of a single sensor object can not indicate that multiple collisions has occurred. This makes it difficult to determine whether an object is in a sensor's field or not when many objects are moving in and out of it. In order to eliminate this problem, two measures must be taken. First, the sensor region must be divided in into smaller sub-regions, each to be allocated a smaller sensor. This decreases the likelihood of more than one object passing through a sensor region at a time. The quantity of sensors, configuration, and granularity can be adjusted in order to examine the capabilities of the sensor array. Second, each sensor contains information as to what objects in the scene it can sense. After any collision event, a sensor checks to see if it is intersecting any of the objects to which it is sensitive. This extra computation is expensive, and if this multiple collision issue isn't resolved in new releases of Java3D, it could very well defeat the purpose of using Java3D's own collision detection.

Java3D's collision model introduces another difficulty to agent development. Currently there is a lack of separation between the collision engine and the render engine. In the design of a system, it is desirable to defer as much of the interaction with the environment to the Java3D package. This allows Java3D to handle object collisions, since it can perform these operations in a device-independent manner. Although using these pre-built mechanisms simplifies the design task, it does not provide for an optimal model. The difficulty lies in the fact that the current Java3D implementation can not perform off-screen processing. This problem means that the limiting factor for an implementation is the speed of the graphics accelerator, as collision detection and object manipulation require that objects be physically painted to a display device. Not only does this create a bottleneck for simulation, but it also implies that a dynamic world can not continue to operate without a viewer present. To date, we have tested our implementation on two platforms: a Pentium II 300 system with a Matrox Millenium II graphics accelerator, and a Sun Sparc Ultra 60 with a Creator 3D option. On both platforms, profiling indicates that even though simulation timing is controlled by timer events, the graphics pipeline speed creates a ceiling for the computational speed of the system. It is our recommendation that current discussion in the Java3D interest newsgroup [Java99] continues to treat off-screen rendering as a high priority in the next release.

Another important consideration in the development of an agent is that of interface and view perspective. During the development process, it is often very important to gain different visual perspectives of the environment. It is sometimes necessary to maintain an overhead view of the simulation field in order to view movements. It is also important to view the profile of the field in order to view collisions and representation issues. In other circumstances, such as simulating visual systems, it would be important to create a view from the perspective of the virtual robot. In order to support these different views, a generalized viewing model for the world must be implemented. The basis of this model includes a viewpoint bound to a null geometry, to act as a "disembodied" point of view. The null geometry of the viewpoint could either be bound to an event handler, such as a keyboard event handler, or to a portion of a pre-existing geometry. By generalizing the interface, one may switch between various perspectives, allowing for both first person and third person visualizations of the world with respect to the virtual robot.

Conclusion

It is possible to implement an autonomous agent that is capable of displaying interesting, reactive behavior in Java3D using a modified subsumption architecture. Java3D and the Java language platform are in a state of evolution, and many of the difficulties currently encountered due to the language are already scheduled to be addressed in future releases. Java3D provides for many of the elements required by simulated robot systems, including physicality through representational geometry, sensors through event handlers, and action via dynamic changes to the scene graph. Java3D's evolution means that it will be able to address the needs of web based 3D visualizations as well as handle the needs of distributed graphical simulations. Already, there are signs that both of these issues are being addressed, even in a limited fashion. As of this writing, Java3D applications can be run as Java applets over the web via the use of Java plug-in as part of the Java 2 runtime environment. Additional research into the use of distributed Java3D has been stimulated by groups working on shared data abstractions for Java, including the JSDT Development Project at SUN Microsystems [Burridge98].

Not only is Java3D appropriate for interests in artificial agent research, but it serves another purpose in that it allows developers to present their work in a manner that is easy to understand. The power provided by Java3D can give a student enough visual stimulation and interactivity to more effectively learn about the nature of subsumption architectures than through a less complex two-dimensional model. The idea of complex emergent behaviors can sometimes be difficult to grasp, but this system is capable of showing students the simplicity of the control architecture's components as well the complexity of the behaviors they produce. Also, Java3D immersive experiences can allow researchers to obtain more insight into their own systems.

There is still much that can be done with this implementation. More layers could be added to the subsumption architecture to give the robots a richer set of behaviors. Noise could be added to the sensors in order to test the robot's tolerance for bad data. Also, different layers of the architecture could be engineered to fail arbitrarily in order to view the effect such a situation would have on the robot's performance. Due to the virtual nature of this testbed, all of these changes could easily be made, and the effectiveness of the simulation would clearly show any results.

References

[Brooks85] Brooks, R., "A Robust Layered Control System for a Mobile Robot", MIT AI Lab Memo 864, 1985

[Brooks90] Brooks, R., "Elephants Don't Play Chess", Robotics and Autonomous Systems, Vol. 6, pp. 3-15, 1990

[Brooks91] Brooks, R., "Artificial Life and Real Robots", Towards a Practice of Autonomous Systems: European Conference on Artificial Life, MIT Press, Paris, France, pp. 3-10, 1991

[Burridge98] Burridge, R., Java Shared Data Toolkit User Guide Version 1.3, Sun Microsystems, http://developer.javasoft.com, 1998

[Java99] Java3D Interest newsgroup, http://java.sun.com/products/java-media/mail-archive/3D/index.html, 1999

[Lorigo97] Lorigo, L., Brooks, R., and Grimson, W., "Visually-Guided Obstacle Avoidance in Unstructured Environments", Proceedings of IROS'97, Grenoble, France, pp. 373-379, 1997

[Sowizral97] Sowizral, H., Rushforth, K. and Deering, M., Java3D API Specification 1.1, Sun Microsystems, 1997

[VRML97] VRML Consortium, ISO/IEC 14772-1 1997 The Virtual Reality Modeling Language (VRML97), http://www.vrml.org, 1997

[VRML98] VRML Consortium, Couch, J. and Marrin, C., External authoring Interface for VRML, EAI Working Group, http://www.vrml.org, 1998