An Object-Oriented Approach To VRML Development

Curtis Beeson
Silicon Graphics, Inc

Abstract

The development of dynamic VRML worlds can benefit from object-oriented methodologies. This paper proposes a class hierarchy for modeling a physical simulation and illustrates several methods of realizing that object model. The implementations presented explore the uses of VRML prototype and Script nodes, Java, and the external authoring interface in describing behavior.

CR Categories and Subject Descriptors: I.3.6 [Methodology and Techniques] Languages; I.3.7 [Three-Dimensional Graphics and Realism]: Virtual Reality; D.1.5 Object Oriented Programming

Additional Keywords: VRML, Java, JavaScript, External Authoring Interface, Prototype, Script

1 INTRODUCTION

The Virtual Reality Modeling Language 2.0 Specification [1] seeks to create a scene description that combines geometries, links, inlines, sounds, images, and code managed in a hierarchical format. With the scene graph responsible for representing such a versatile domain of data, it is not surprising that management of the VRML scene graph is one of the most difficult problems facing the VRML author.

The VRML 2.0 Specification sought to create a node structure that would facilitate optimization for traversal and added behavioral mechanics such as routes, Script nodes, and an external authoring interface that a VRML viewer is to export to the Web browser. Regardless of the method that authors use to add behaviors to their scenes, they must structure and understand their scene graph. Either authors must create routes between the appropriate fields in the scene, or they must use a supported scripting language to retrieve the appropriate fields from the nodes of interest.

Unfortunately, most VRML files require very deep hierarchies to obtain the placement, articulation, inline, and level of detail hierarchies needed to develop complex scenes. Compounded with the many parameters required to describe geometries, it is difficult to examine a VRML file and understand the relevant structure. If the author has created a human figure that he wishes to articulate, he may be interested only in the *rotation* fields of the Transforms representing joints. Here, the many other fields and values in the scene are merely noise.

This complexity is magnified by the fact that compelling VRML content requires a wide range of skills; thus a world is frequently the product of a team. Worlds are often created by a group of individuals who specialize in modeling, code, images, or sound. If one member of a team is responsible for the code governing behaviors and another member of the team modifies the scene structure, the code can be made obsolete. Because the current Java and JavaScript access methods to the scene graph are based on fetching nodes and the fields of those nodes, any changes to the scene may break the code accessing that scene. Analogies exist for other specialists' interactions, each pointing to the fact that the behavior of complex worlds is precariously dependent on the scene graph structure.

1.1 Object-Oriented Design

When faced with similar issues in more conventional programming languages, developers have used object-oriented design to make systems more readily understandable and extensible. The term "object-oriented" is used to describe systems in which software is organized into a collection of objects that incorporate both data structure and behavior [4]. VRML seems particularly suitable to such methodologies, as it seeks to describe objects and events in a real or imaginary world.

The properties of object-oriented systems that are explored in this paper include abstraction, encapsulation, inheritance, and extensibility.

1.1.1 Abstraction

Abstraction is the act of dividing a system into discrete classes and determining the roles of each. Object classes should be as independent of implementation as possible. This approach creates a more intuitive understanding of how the system will solve the problem and how system components will interact.

1.1.2 Encapsulation

After an abstraction has been created, the developer determines what logic and data are associated with each class. The class should provide an interface that reflects the interactions described in the abstraction phase. This process of hiding the details of implementation from the outside world is known as *encapsulation*. The developer may then modify the implementation internal to the class without needing to modify the way other logic interacts with that class, keeping such changes localized.

1.1.3 Inheritance

In a well designed object model, classes are often general enough that they provide a wide range of possible uses within a system. Instead of creating one very complex class, it is sometimes useful to create a family of related classes, each encapsulating the logic associated with its particular use.

A subclass is a class that *inherits* the interface and functionality of its superclass and augments the class by adding to, or overloading,

curtisb@sgi.com http://reality.sgi.com/curtisb_engr/
Silicon Graphics, Inc., 2011 North Shoreline Blvd.
Mountain View, CA 94043

VRML 97, Monterey CA USA
Copyright 1997 ACM 0-89791-886-x/97/02 ..$3.50

the properties of that class. The resulting family of classes is often referred to as the *class hierarchy.*

1.1.4 Extensibility

The real benefits of object-oriented design become apparent when the developer needs to maintain the system. Incremental development has been improved due to the localization offered by encapsulation. In an ideal object model, modifications, optimizations, and fixes rarely cross class boundaries. When new functionality is added, the logic that interfaces with a particular object should be able to interface automatically with any subclass of that object. Thus, object-oriented methodologies can be beneficial for both incremental and radical changes to a system.

1.2 VRML Prototype Node

The prototype node in the VRML 2.0 Specification helps authors manage scene complexity by providing a method for defining higher-level objects. The prototype definition consists of an arbitrarily complex VRML hierarchy and a list of fields, eventIns, and eventOuts that constitute the interface to that hierarchy. Each field or event in the prototype's interface must be an alias of a field or event in its describing hierarchy. To use a prototype, it must be either defined in the file, or listed as an external prototype, in which case the definition is found at the specified URL.

By choosing how to divide scene graphs into reusable prototype nodes, the author has created an abstraction. Because functionality in VRML is encoded in the nodes, the act of creating this prototype has encapsulated both the data and functionality of its describing scene graph. By exposing only those fields that the author intended through the prototype's interface, he has created an object that may be reused more intuitively.

Addressing a problem alluded to earlier, the author could create a prototype definition with the scene graph of a fully articulated human figure, exposing only the rotations of the joint Transform nodes. The author may now create several human figures, instantiating each as if it were a single primitive node in the VRML Specification.

1.3 VRML Script Node

The Script node allows authors to add behaviors to the scene. Script nodes contain programmatic logic that translates input events into output events. By routing events from sensors to the Script node, and from the Script node to elsewhere in the scene graph, the author can add customized functionality to their world.

The Script node contains a list of eventIns, fields, eventOuts, and a URL at which the logic can be found. In order to be compliant, browsers must support at least Java and JavaScript in the body of a Script node [1]. In order to use Java in the body of the Script node, the URL must point to a Java class that inherits from the Script class. By overloading methods such as initialize(), processEvent(), processEvents(), and eventsProcessed(), the developer defines the desired actions of the Java Script node.

1.4 External Authoring Interface

The VRML Specification defines a Java interface that a browser is to export so that the scene graph can be manipulated by external Java applets [2]. The Browser class allows modifications to the scene by providing methods to add and remove routes, to fetch nodes, and to read or write field values in those nodes.

It is sometimes necessary to use the external authoring interface instead of Script nodes to describe the desired behavior. If the author wishes to create an HTML page with controls that modify the VRML scene, the external authoring interface is the only existing method for effecting the scene graph.

2 PURPOSE

Research on the topic of applying object-oriented approaches to VRML development began during the development of a site to describe chaos and complexity. The goal was to create a Web site at which visitors could experimentally discover the fundamentals of chaos theory. By exploring interactive systems such as gravitational simulations and flocking algorithms, the experimenter could analyze how simple objects can display a complex aggregate behavior.

VRML is the ideal medium for this type of simulation, as it allows full examination of the system. The experimenter will be able to navigate a star system filled with gravitational bodies or walk around a room filled with flocking agents. When creating the logic necessary to develop such a system, it is in the author's best interest to create reusable assets.

This paper explores methods in which the author can combine the encapsulating abilities of the prototype and the behavioral mechanics offered by VRML to create reusable objects. Three methods are presented, and the limitations of each are discussed.

3 CLASS HIERARCHY

The first observation is that in both the gravitational and flocking experiments, the author desires a three dimensional geometry representing every body in the experiment. In both cases, the author also desires that the bodies obey some simple Newtonian laws of physics. Lastly, a desired feature in each experiment is that each body plot its course through time.

Applying object oriented methodologies, the author creates a class of object, called Newtonian for instance, that encapsulates the attributes of mass, velocity, position, and force. This class also encapsulates functionality for updating velocity and position based on the force exerted on the mass and for plotting the path of the body through time.

The author can then create Attractor and Flocker as subclasses of Newtonian. The Attractor and Flocker need only encapsulate how to determine their forces based on a gravitational attraction or swarming tendencies respectively. If Attractors are attracted to any Newtonian, the author can create worlds mixed with Attractors and Flockers, each interacting with the other despite their differences.

Lastly, the author defines a Container class to manage the Newtonians. This class facilitates the traversal of each Newtonian so that it can evaluate on each time change and maintains the list of Newtonians instances that each subclass of Newtonian requires to perform its interactions.

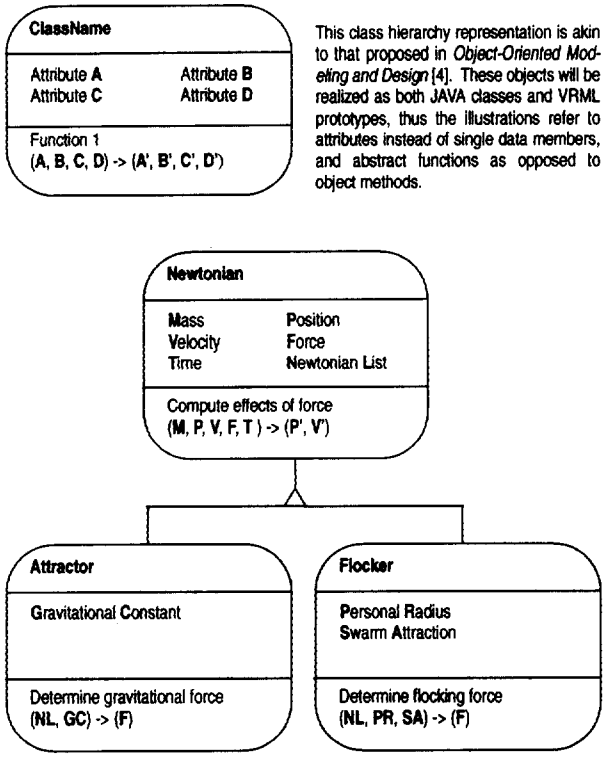

Figure 1: Newtonian Class Hierarchy

This object model implies that the code required to update velocity and position based on force and time is localized to the Newtonian class. When using numeric approximations for solving differential equations, the author must often try number of methods, as more accurate methods tend to be slower. Because of this clean encapsulation, the author can change the solution methods for the force equation in the Newtonian class without any modification to its subclasses or to the Container class.

The extensibility of the object model becomes apparent when the author chooses to add introduce a new object class to the simulation, perhaps by adding a Spring object to connect any two Newtonians together. To do this, the author could create a new class capable of imparting a force on Newtonians based on its spring constant, the distance between the Newtonians, and the spring's rest length. The author can now create a spring between any two nodes that inherit from Newtonian, perhaps pushing attracting bodies apart or tethering a flocking agent to a heavy weight.

4 NESTING VRML PROTOTYPES

In the first method proposed for realizing the class hierarchy, the author uses the encapsulating abilities of the prototype and behavioral abilities of JavaScript to create suitable VRML nodes. A Particle prototype is defined and used to create a Newtonian prototype capable of describing a mass acted upon by a force. Attractor and Flocker prototypes are then created, each containing a Newtonian instance in its describing hierarchy. By processing events and propagating them to their encapsulated Newtonian prototype, the Attractor and Flocker prototypes build on the functionality provided by the Newtonian.

4.1 Particle Prototype

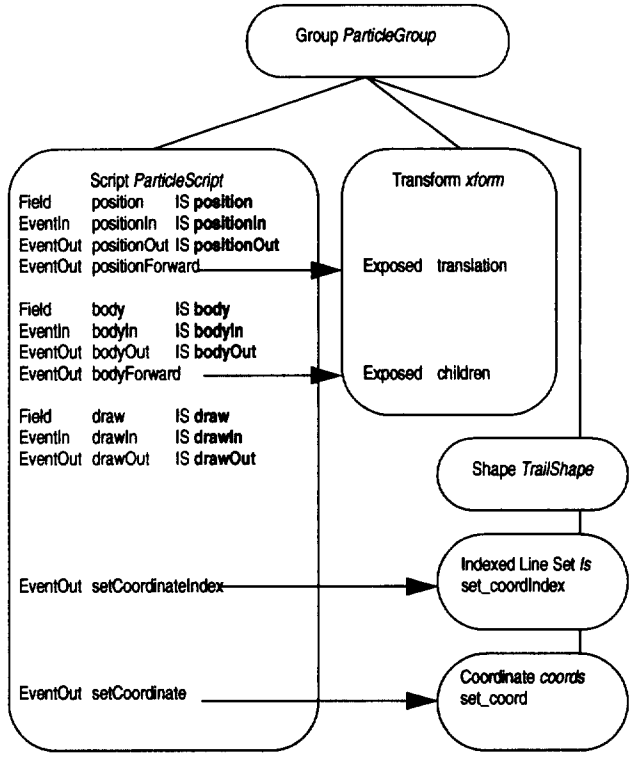

Figure 2: Particle Prototype Implementation

The Particle prototype encapsulates a Script node, a Transform, and a shape that describe a body that can create a path that tracks its changes in position. Its interface includes an eventIn, a field, and an eventOut describing the attributes of position, body, and whether or not the trail should be extended on the next positionIn event.

This separation is necessary because Script nodes do not support exposed fields. To provide the same functionality, each attribute used by a Script node needs an eventIn to be triggered by external stimuli, a field to store the value, and an eventOut to propagate changes elsewhere in the scene. Technically, because the code in the Script can access the last value propagated through an event, the field is not needed. This separation serves to simplify the distinction between storage of the attribute and propagation of the change in that attribute.

The Body attribute could have been described by an exposed field, as ParticleScript currently does nothing with the value except propagate it to the children of the Transform. Particle does not need logic to react to changes in its child list, but related prototypes may. Because their body attribute will alias to a field, an eventIn, and an eventOut on a Script node, Particle is defined in the same manner. External logic can therefore trigger *bodyIn* event on any member of this set of similar prototypes. This type of adherence to a common prototype interface is what will provide a type of prototype subclassing.

4.2 Newtonian Prototype

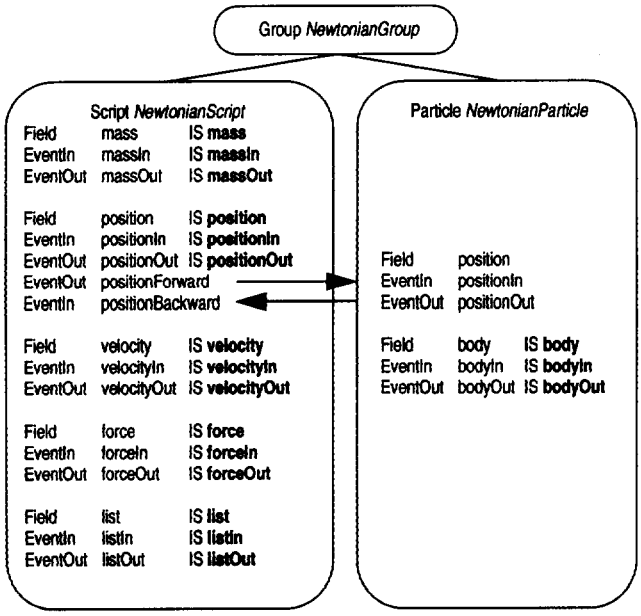

Figure 3: Newtonian Prototype Implementation

The implementation of the Newtonian prototype consists of the NewtonianScript and a Particle prototype instance. Newtonian builds on the abstraction offered by Particle, adding functionality for solving the force equation. To accomplish this task, the Newtonian interface exposes fields and events for mass, position, velocity, force, and body. Again, each attribute requires an eventIn, a field, and an eventOut.

The Newtonian prototype has an MFNode attribute that maintains a list of Newtonian (or subclass) prototype instances. This is not required for the Newtonian's functionality but will be required by both the Flocker and Attractor prototypes. The *listIn* event is defined to do nothing in the Script node. The field and events are provided so that Attractor and Flocker may inherit a common interface for their list of Newtonians to act upon.

NewtonianScript updates its internal storage of mass, position, velocity, or force on an eventIn, and propagates this change by sending the new value through the corresponding eventOut. As is common in simulations, behavior is a function of time. On a timeIn event, NewtonianScript solves the force equation and updates its representation of velocity and position. Subclass prototypes will therefore be responsible for setting the force attribute before the Newtonian's evaluation to achieve their characteristic behaviors.

The position value is propagated to the Particle prototype so that the Newtonian's body can appear at the appropriate location. The position and velocity eventOuts are then triggered, making the values readable by logic outside the prototype implementation.

4.3 Subclass Prototypes

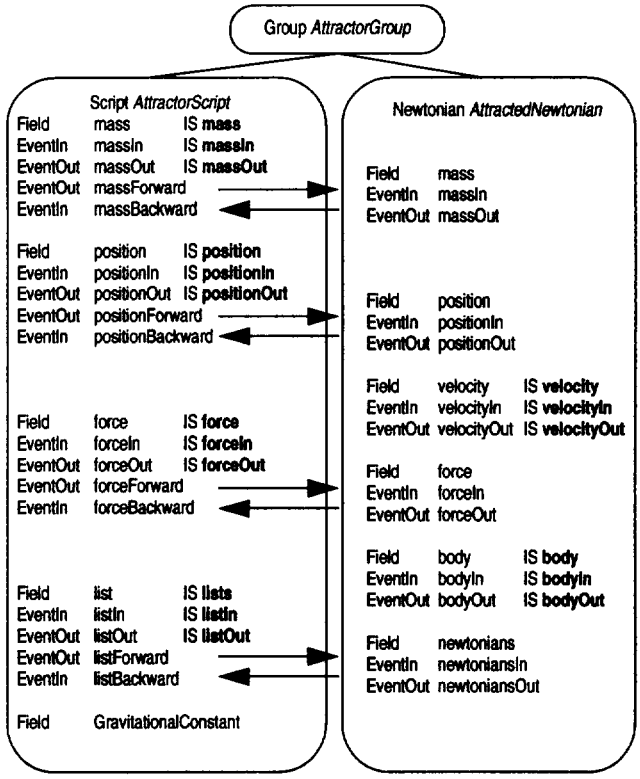

Figure 4: Attractor Prototype Implementation

Each subclass prototype interface includes at least the same fields and events that the Newtonian did, as if it inherited them. It is important to do this so that any external logic trying to interact with an Attractor or Flocker can trigger the same eventIns that the Newtonian had, allowing it to interact homogeneously with all subclasses of Newtonian. Unfortunately, there is no syntax in VRML to extend another prototype's field list, so the author must maintain the subclass prototype interfaces manually.

Any fields not needed for the added functionality of the subclass prototype are aliases of the fields of the Newtonian prototype instance. Those attribute fields and events required for the subclass's behavior are aliases of the subclass Script. Any changes to attributes that are local to that subclass update the subclass Script's local storage, and must then be *forward propagated* to its encapsulated Newtonian prototype. A *massIn* event changes the *mass* field in the subclass Script, and is then forward propagated through the *massForward* eventOut to the Newtonian prototype, where the *mass* field of the Newtonian's Script is also updated. Without forward propagation, the subclass prototype would perform its calculations based on the correct mass, but the NewtonianScript within the Newtonian prototype would evaluate the effects of force with its default mass, and the calculation would be incorrect.

The author may wish to change Newtonian so that it can modify its mass over time, perhaps to simulate radioactive decay. To handle such a change, the subclass Scripts have eventIn called *massBackward*. A route is created from the Newtonian prototype's *massOut* event into *massBackward*. This route is needed so that when handling this second eventIn, the logic sets the local field and triggers

the eventOut instead of forward propagating. This act of updating attributes from superclass to subclass will be referred to as *backward propagation.* Note that backward propagation is not needed for attributes that are not used at that level in the prototype hierarchy but are necessary even if the superclass does not yet modify the value of that attribute. This machinery aids localization. If the Scripts within the subclass prototypes already handle backward propagation, modifications made to Newtonian should be safe.

Forward propagation could be optimized through the use of the JavaScript interface for setting eventIns directly. If the subclass Script had an SFNode field that pointed to the superclass prototype (this can be done with DEF / USE), forward propagation could be implemented by setting the eventIn's value in the Script's logic. This is functionally equivalent to the Script setting its own eventOut and routing this eventOut to the superclass prototype's attribute eventIn.

Backward propagation could also be optimized, but it would require an SFNode field on the Newtonian prototype to maintain a reference to any surrounding subclass prototype, and the backward propagation eventIns for each attribute in the Script node would have to be part of the subclass prototype's exposed interface. Routing to these eventIns would confuse the logic of the subclass, so backward propagation should be achieved using routes, with all of the machinery encapsulated by the prototype.

The subclass prototypes function by catching the timeIn event, forward propagating a behavioral force to the encapsulated Newtonian, then forward propagating the *timeIn* event. In this way, the author is certain that the Newtonian prototype does not update velocity and position until after the subclass prototype has updated the force. When it is finished evaluating, the Newtonian triggers eventOuts for position and velocity. The Newtonian's *positionOut* event is routed to each subclass's backward propagation eventIn, and therefore updates the subclass Script's position and triggers the subclass Script's *positionOut* event.

4.4 Container Prototype

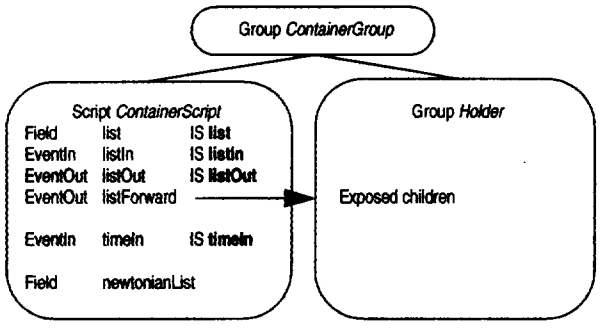

Figure 5: Container Prototype Implementation

Finally, the Container prototype is defined. The Container manages a list of nodes, some of which may be Newtonians. The Container's describing scene graph consists of a Group whose children may include Newtonian (or subclass) prototype instances and a Script node. Because the prototypes are under the Container, it can manage the propagation of both time and of the Newtonian list to each Newtonian in that list programmatically.

Any time a node is added to or removed from the list of the Container class, ContainerScript iterates through the child list and compiles a list of those prototypes inheriting from Newtonian. The result is cached in the *newtonianList* field, and is propagated to each member of that list. A route is dynamically created from the *listOut* of each Newtonian to the *listIn* of the ContainerScript, empowering each prototype to add or remove members to the simulation.

4.5 Analysis

The author has created abstractions that encapsulate the data and functionality associated with performing each class's task by nesting prototype definitions. By taking advantage of the event model, subclass prototypes are able to reuse the Newtonian's functionality.

Although this method yielded functional abstractions, it is difficult to implement. Attributes are duplicated in every prototype of the nested hierarchy. This causes data flow problems, as values must be forward propagated during initialization or during a subclass modification, and values must be backward propagated during any superclass set. Object interactions rarely decompose so cleanly, making the forward and backward propagation framework intractable.

VRML is an event-driven language because the execution of logic occurs as a result of an eventIn being triggered. Thus, execution flow is as dependent upon the event model as data flow. This complicates communication between VRML objects as well as execution within an object.

In more traditional object oriented languages, the subclass can redefine the methods of its superclass. To implement the analogy of "overloading" in VRML, each prototype would require an SFNode field whose value is the outermost prototype definition of the instance. This outermost prototype must receive notification any time an eventOut is meant to cause execution so that the outermost event handler receives the event. Combined with the logic for forward and backward propagation, even the simplest systems become unbearably complex using this method.

The problem is that nesting prototype definitions is not truly subclassing. In a real subclass, there exists one unique representation for an attribute or method. In this object-based VRML approach, nested prototypes are masquerading as a single object. Not only is the state of the object duplicated throughout the class hierarchy, but communication within the hierarchy of a single instance is asynchronous in nature and difficult to orchestrate.

5 PUTTING JAVA IN THE SCRIPT NODES

In order to address the difficulties in using nested prototype definitions to achieve functional inheritance, this second method of object oriented development takes advantage of the object oriented nature of the Java programming language [4]. The class hierarchy is defined by creating subclasses of the Java Script class. Prototypes are then defined to encapsulate the relationship between the Script node and the VRML scene graph.

5.1 Script Classes

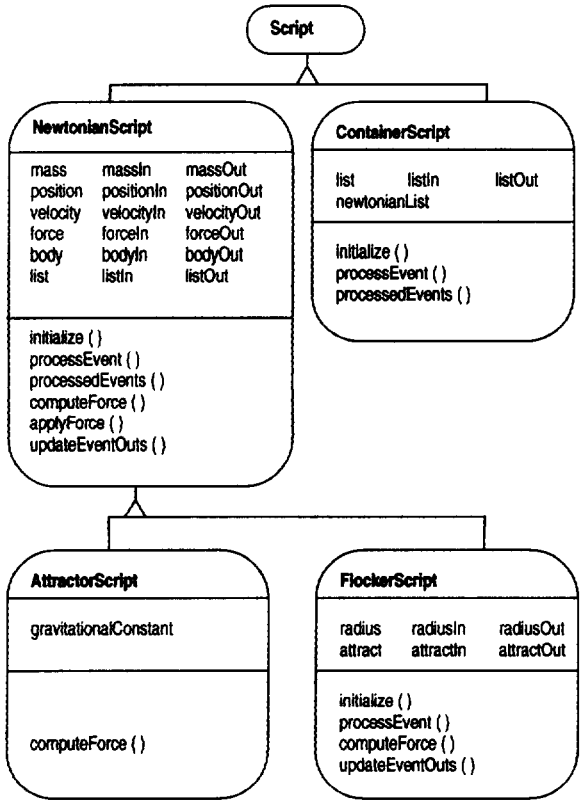

Figure 6: Java Script Class Hierarchy

The first class is NewtonianScript, which has an input, field, and output for each attribute on the class. The initialize() method caches fields and eventOuts so that NewtonianScript has access to the Java wrapper for each without searching each time they are needed. After this is done, field values are propagated to the eventOuts of the Script class, which correspond to the eventOuts of the Script Node. These are in turn aliased to the eventOuts of the Newtonian class prototype.

The processEvent() method is defined so that any eventIn sets the corresponding field value. No evaluation or propagation to an eventOut occurs until after all events have been triggered, and the processedEvents() method is invoked. ProcessedEvents() determines if time was one of the attributes that changed. If it has, computeForce(), applyForce(), and updateEventOuts() are invoked.

NewtonianScript does not modify the force attribute in computeForce(). This method is redefined by subclasses to determine the force on the Newtonian based on various behaviors. The applyForce() method is the one in which force and time are used to update velocity and position. Finally, updateEventOuts is called, propagating the field changes out of the Script, and therefore out of the Newtonian prototype.

Attractor needs no additional attributes, and merely overloads computeForce() to impart a force based on gravitational attraction. Flocker adds attributes for an agent's attraction to the center of the flock, and the agent's personal radius. The Flocker overloads initialize() processEvents(), and updateEventOuts() to add support for these new attributes, and overloads prepareValues() to impart a force based on flocking tendencies.

5.2 Prototype Definitions

With the behavior summarized in each VRML object's Script, the creation of the respective prototypes is simple. The Newtonian and Container prototype definitions are identical to those presented in the previous method, with the exception that the Script nodes point to their respective Java classes. The Newtonian subclass prototype implementations are similar to those of Newtonian, but Flocker augments its interface by adding its behavioral attributes.

5.3 Analysis

The system resulting from this approach is more robust, more efficient, and easier to understand that the previous approach. The author may now add subclasses to Newtonian by extending NewtonianScript, adding code for attributes unique to that subclass, and redefining computeForce() to impart a force based on the desired behavior. A prototype definition must then be written around the new Script, extending the interface of its superclass prototype.

This method does not suffer from the issues that the first had with communication within a class, but it is still dependent upon the prototype structure for communication between separate objects. The execution model within AttractorScript is easy to understand, but it must still iterate through a list of Newtonian prototypes to determine their locations. The Java code within a Script node is unable to directly access the Java object within another Script node. The code in AttractorScript may only communicate with the VRML scene graph using the Java scene interface. Thus, the class hierarchy must still be reflected in the prototype structure.

To illustrate this point, consider the result if each prototype named its positionOut event differently. The result is that the computeForce() methods of AttractorScript and FlockerScript could not find the 'positionOut' event of the Newtonian prototypes in their list to grab their positions. Each prototype that introduced a new name would require the author to rewrite these two methods adding a fieldname to look for.

Because communication between objects is performed through the prototype interfaces, execution must still occur as a function of the event model. The use of Java in the Script node alleviated the problems with communication within an object, but offers nothing to amend the difficulties of orchestrating execution and data flow between nodes.

6 JAVA AND THE EXTERNAL INTERFACE

In the last proposed method, Java classes are written that encapsulate the logic necessary to perform the physical simulation. A separate class hierarchy is then created to encapsulate the display logic needed to illustrate the system. Through the separation of simulation and display logic, the resulting objects are easier to understand and more extensible.

6.1 Functional Classes

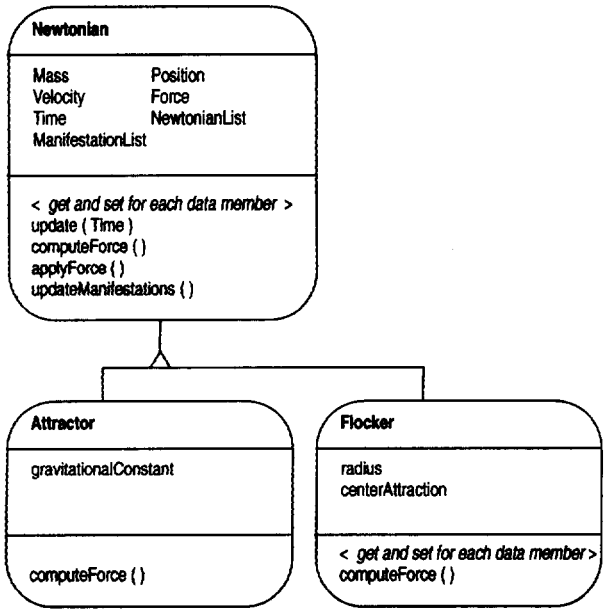

Figure 7: The Functional Class Hierarchy

The Newtonian class contains data members representing time, mass, position, velocity, force, a list of Newtonians to interact with, and a list of Manifestation objects responsible for the display of that Newtonian. The class contains access methods for each data member, and methods for performing the force calculation.

The setTime() method should not have the 'side effect' of changing position and velocity, so the update() method is introduced. Update() accepts a time value, and invokes setTime(), computeForce(), applyForce(), then updateManifestations(). ComputeForce() does nothing in the Newtonian base class, but is overloaded by Attractor and Flocker to compute forces based on gravity or swarming respectively. The applyForce() method is defined to apply the force to the Newtonian, updating the class's data members.

If a Newtonian (or subclass thereof) has an empty list of associated Manifestation objects, updateManifestations() will do nothing, and update() will merely cause the Newtonian to update its internal state. Otherwise, each Manifestation's show() method is invoked. Each Manifestation object may invoke the access methods of the Newtonian to query values, and may render them in any way that the author desires.

6.2 Manifestation Classes

The Manifestation class is responsible for making the properties of its associated Newtonian understood by the viewer. Subclasses of the Manifestation class encapsulate the logic necessary to render Newtonians to a Java drawing area, a VRML scene graph, or perhaps to Cosmo3D (Silicon Graphics' 3D Java library).

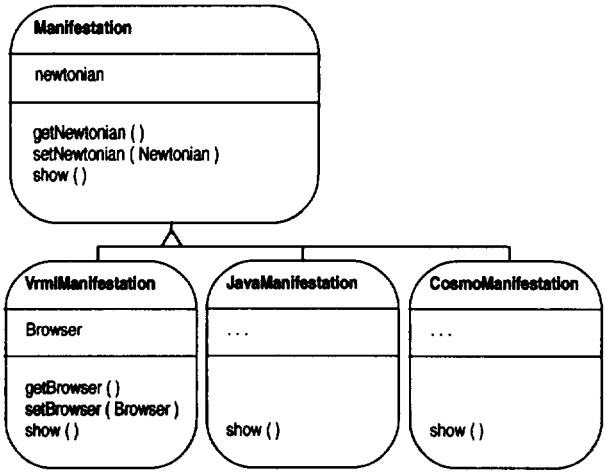

Figure 8: The Manifestation Class Hierarchy

Each subclass of Manifestation will contain the data members and access methods needed to maintain its associated Newtonian and the information for rendering to its particular medium. The draw() method queries the Manifestation's Newtonian to find salient information, and performs the logic unique to that subclass for displaying that information to the viewer. By allowing each Newtonian to have several Manifestations, we have made it possible to display the results of a single simulation in multiple windows, in several media.

VrmlManifestation's constructor creates an instance of the Particle prototype that will represent its associated Newtonian and adds it to the scene using the browser's scene access methods. Similarly, its destructor removes the Particle from the scene. The draw() method has only to get the position from its associated Newtonian, and send the value to the positionIn event of its Particle prototype. In this way, the VrmlManifestation class acts as liaison between the functional Java classes and the VRML scene.

6.3 Container Class

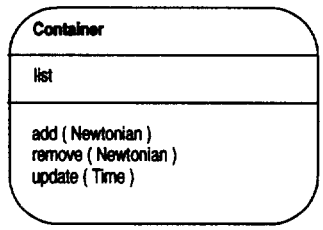

Figure 9: The Container Class

If the developer had wanted every Newtonian that was created to be part of the simulation, each Newtonian's list of peers could be shared across all instances of the class. The Newtonian's constructor could automatically append it to a global list of all Newtonians, eliminating the need for a Container class.

By introducing the container class, however, a single executing applet can maintain several independent containers, each of which is an independent simulation. More interestingly, subclasses of the

Container could allow for more experimental designs, such as having two sets of Newtonians, each of which interacts with members from the other set, but not its own.

AddNewtonian() accepts an instance of a Newtonian as a parameter, and adds it to the Container's list of managed Newtonians. The Container then iterates through its managed Newtonians, calling the setNewtonianList() method on each so that they have a complete list of bodies to interact with. RemoveNewtonian() has similar logic for removing a body from the simulation. The update() method accepts a time value, and iterates through the list passing the new time value to the update() method of each Newtonian.

6.4 Execution

In order to execute Java code, the developer must create a class that inherits from Applet, and overloads methods such as init() or draw() to customize the object. The init() method is invoked when the object is first created, so it is the common place to put setup information.

In a simple simulation, init() begins by retrieving the browser object present on the page so that it may be passed to each VrmlManifestation object of a Newtonian. Init() then creates the Container object that will manage all of the members of the simulation. Several Newtonians, Flockers, and Attractors are then instanced, and the code calls their access methods to set initial conditions on each. Each Newtonian (or subclass) instance is then given a VrmlManifestation object, and added to the Container. Finally, the applet enters a loop in which time values are fed to the Container. These time updates are handed to each Newtonian in turn, causing their VrmlManifestation to animate the bodies visible in the VRML browser.

There is no structure imposed on the logic outside of the VRML scene graph. Other simulation applets could search the VRML graph for Newtonian, Attractor, or Flocker prototype wrappers around Particles and generate the simulation and initial conditions from the VRML file. In this way, the author could modify the experiment without recompiling Java bytecodes.

6.5 Discussion

This method of realizing the proposed object model is the most extensible presented in this paper. In previous approaches, the functionality was written in a VRML context and was therefore limited to that medium. By creating a distinction between the functional classes and the display classes, the author has created a valuable asset that can be reused for VRML or elsewhere. Java provides a clean inheritance model, and the procedural style of execution simplifies execution and data flow.

The property sacrificed when using this approach is that a prototype definition no longer encapsulates all of the logic needed to implement a Newtonian, Attractor, or Flocker. In either of the two previous approaches, these prototypes could be used by others if they simply declared the author's prototypes as external. In this approach, it is more difficult to determine where salient logic is contained. The logic for plotting a Particle's course through time is found in the prototypes of the scene graph, but in order to create a full simulation, they must copy the relevant and classes and simulation applet from the author.

7 FUTURE WORK

There were no VRML viewers mature enough to test many of the proposed techniques when this paper was written, so the Web site is a work in progress. The ideas presented in this paper are the result of exploring various methods of realizing one system. With each new problem comes a unique desired object model, and new interactions will undoubtedly suggest new roles for VRML prototyping, scripting, and applet to browser interactions.

Several of the complexities discussed in this paper inspire extensions to VRML, and these may someday find a home in the specification. One such change is the need for an inheritance sematic that would allow a prototype to build upon the interface of another. Another desirable feature is support for synchronization, particularly in the external authoring interface. Without the ability to temporarily disable rendering in the VRML viewer the system will be rendered when only some of the particles have been updated. These intermediate states are inconsistent, and it is unacceptable that they are displayed.

8 CONCLUSION

Authors can benefit from using object oriented methodologies when developing worlds. The author does not need to create an entire class hierarchy before concepts like abstraction become valuable. The Particle prototype is a simple abstraction that uses JavaScript, but it has proven to be invaluable throughout the implementations explored.

VRML was not written to be a full object oriented language, thus it seems that Java is the appropriate language in which to write an object's functional class hierarchy. Java classes may act upon the scene from within a Script node, or externally through the authoring interface.

Placing Java in the Script node has the advantage that the Script node's interactions with the scene graph can be encapsulated by a prototype definition, creating a reusable abstraction. Unfortunately, the VRML objects constructed in this way must still rely on the event model for interactions. Deveoping Java classes that act upon the scene graph using the external authoring interface can provide highly reusable assets, but lacks the highly encapsulated properties offered by the Script node embedded in a prototype definition.

9 ACKNOWLEDGMENTS

I'd like to extend my sincerest thanks to all of the members of CosmoWorlds, CosmoPlayer, and CosmoCode groups of Silicon Graphics Incorporated, Rikk Carey, Gavin Bell, and Dave Immel. Special thanks to Aaron Siri for his help prototyping the Java functionality, and Josie Wernecke for her work on this document.

10 REFERENCES

[1] Bell, Gavin, Rikk Carey, and Chris Marrin, *VRML 2.0 Specification* (Version 2.0, ISO/IEC CD 14772). 1996.
[2] Marrin, Chris, *External Authoring Interface Specification.* 1996
[3] Ritchey, Rick, *Java!*. New Riders Publishing, 1995.
[4] Rumbaugh, James, *Object-Oriented Modeling and Design.* Prentice Hall, 1991.