Less Is More[1]: The Power Of Simplicity

Jean-Francis Balaguer[2]
artemedia online

ABSTRACT

VRML has partially failed to provide an effective platform for the development of rich 3D media experiences over the web. The main reason lies in *VRML*'s lack of flexibility and expressive power to adapt to unanticipated use. We propose to make *VRML* evolve towards a new extensible and flexible run-time environment based on a prototype-instance object model and indirect constraints. The flexibility of the object model allows for the addition of new, unforeseen behavior and easily supports the creation of one-of-a-kind objects. Constraints provide a means to explicitly declare relationships between objects that are automatically maintained. Indirect constraints can dynamically compute the variables on which they depend during their evaluation. These concepts fit well with *VRML*, and a working prototype implementation could be easily constructed using a public domain toolkit.

CR Categories and Subject Descriptors: I.3.7 [Three-Dimensional Graphics and Realism] Virtual Reality; D.1.5 [Object-Oriented Programming]; D.2.m [Rapid Prototyping]; D.3.3 [Language Constructs and Features] Constraints, Inheritance.

1 INTRODUCTION

The last year has been important in *VRML*'s short history. The specification has remained stable, and browsers such as *CosmoPlayer* have reached a reasonable level of maturity. This situation has allowed authors to seriously start using *VRML* for their web applications, which has helped identify the problems and limitations of *VRML*.

Interestingly, *VRML* has seldom been used for the purposes for which it was designed, i.e. scalable multi-user immersive environments on the Internet. Instead, authors have attempted to use *VRML* for an extremely wide range of applications, from games, visualizations, and sophisticated animations to very tiny web page elements like ad banners and 3D menus.

The lesson learnt is that *VRML* failed to provide an effective platform for the development of rich 3D media experiences over the web, and many applications could not go much further than the proof-of-concept level. The problems are twofold. First, *VRML* integration with other media elements on the web page relies on the *EAI* [13], which seriously lacks features. Second, even though composability and extensibility were two requirements of its initial design, *VRML* lacks flexibility and expressive power to adapt to unanticipated use. In particular, the programming of behaviors using events and *Script* nodes is fairly limited.

We believe that *VRML* should evolve towards an extensible and flexible run-time environment with enough expressive power to allow the easy authoring of a wide range of interactive graphics applications. Our approach is based on a small number of powerful concepts that can be easily combined to tailor and extend *VRML*'s run-time environment to meet the particular needs of an application. The core of our proposal is a prototype-instance object model chosen for its simplicity and flexibility. The object model is integrated with indirect constraints to explicitly declare relationships between objects that are automatically maintained. In the following sections, we introduce these concepts and analyze why they are more adequate for interactive graphics applications than more conventional object-oriented approaches. Then, we present how this programming paradigm fits with the design of *VRML*.

2 CONCEPTS

2.1 Prototype-instance Object Model

There are two predominant paradigms for objects in object-oriented programming: the class-instance paradigm and the prototype-instance paradigm (a.k.a. the delegation paradigm).

In a class-instance system, all objects are instances of a class (*is-a* relationship), and classes can inherit from other classes (*kind-of* relationship). When an object is sent a message, the method used to respond to that message is the version of the message in the object's class. If the class does not contain a suitable method, then the method used is the one defined in the nearest superclass. A search for a method can be performed at run-time (dynamic binding), or at compile time (static binding).

In a prototype-instance system, there is no distinction between instances and classes; any instance can serve as a "prototype" for other instances. An object is defined by a set of named slots (also called fields or instance variables) that may store either state or behavior. Slots can be added or removed when needed. Objects are created by cloning an existing object, which is referred to as the prototype of the newly created object. When an object receives a message, it looks first in itself. If no suitable method is found, the search continues by searching the object's prototype. That is how inheritance is modeled in prototype-instance systems.

---

[1] Mies van der Rohe

[2] artemedia online, rue de Morges 24, 1023 Crissier, Switzerland. E-mail: balaguer@artemedia.ch, www: http://www.artemedia.ch

The tradeoffs between a class-instance system and a prototype-instance system are subject to debate. In general, prototype-instance is seen as being more suited for exploratory programming, whereas class-instance systems seem to provide better support for guaranteed reliability [5][22].

Benefits of Prototype-Instance Systems:

Easier to learn. A prototype-instance system has fewer concepts to learn. Unlike a class-instance system, which has two types of relationship that describes how objects share behavior and state, a prototype-instance system has only one: "inherit-from". Hence, prototype-instance systems are easier to explain and understand [4][5][26].

More flexible. A prototype-instance object model is a strict superset of class-instance models, since it can model class-instance relations when classes and instances cannot model the prototype-instance relation without additional constructs [10]. This makes prototype-instance systems more flexible, in that the programmer is not bound to the class-instance paradigm and can use more open-ended constructs [5].

Support for unanticipated sharing. Unanticipated sharing occurs when a designer would like to introduce new behavior into a system that does not already provide for it, and was not foreseen when the system was designed. Unanticipated sharing is supported best if new behavior can be introduced simply by explaining what the differences are between the new and existing behaviors. A prototype-instance object model accomplishes this by allowing new objects to re-use the behavior of existing ones without requiring prior specification of this relationship [22].

Support for one-of-a-kind objects. Each object can be customized with its own behavior. A unique object can hold the unique behavior, there is no need for a separate instance. In a class-instance system, there is no support for an object to possess its own unique behavior. The programmer must define a class that is guaranteed to have a unique instance, which is clumsy [4][5][26].

More concrete. Prototypes are more concrete than classes because they are examples of objects rather than descriptions of format and initialization as in class-instance systems. They are more suited to represent the kind of objects that are typically dealt with in 2D and 3D graphics systems, where objects are perceived more as independent entities than as instances of abstract categories [5].

Drawbacks of Prototype-Instance Systems

Need for code optimizations. Prototype-instance systems tend to generate even more message sends than class-instance systems. However, the same level of performance of class-based languages is obtained with prototype-instance systems through compilation/interpretation techniques that are more complex than their class-based counterpart [26]. Moreover, graphics systems based on a prototype-instance object model have proven levels of performance similar to graphics systems based on the more conventional class-instance paradigm [1][17][32].

Concreteness. While the concreteness of prototypes seems natural for things like windows or 3D shapes, it seems more unnatural for inherently abstract entities as integers, or for data structures such as stacks and queues [4][5].

Flexibility. The same flexibility that makes rapid prototyping possible makes it easy to modify prototypes inadvertently [4][5].

2.2 Constraints

Constraints provide a means to explicitly declare relationships between objects. In graphics systems, the most popular form of constraints is one-way dataflow constraints. A one-way constraint is an expression that computes the value of a slot. If the expression references slots of other objects, then when the value of any of the slots changes, the expression is re-evaluated automatically, which may trigger in turn the evaluation of additional constraints depending on the computed value. If no slot has changed, the expression is not re-evaluated and a cached value is returned. Constraints are not limited to simple algebraic expressions but can contain arbitrary code[1].

Constraints promote a declarative style of programming, where relationships are declared once and then are automatically evaluated when necessary. The programmer is freed from the arduous task of invoking them at the appropriate times [8][16]. One classic drawback of constraint systems is their inability to handle cycles in the constraints. A trivial yet effective solution to that problem is simply to break the cycle during constraint evaluation as in [14]. That way, one can set up mutual dependencies between objects.

Constraint systems gain a lot of expressive power when they allow the dynamic computation of the objects to which a constraint refers [8][16][28]. This means a constraint, when re-evaluated, can not only compute the value to return but also which objects and slots to refer. This allows the specification of constraints such as "the bounding box is the union of the bounding box of the children". The constraint will be re-evaluated either when the bounding box of any children changes or when a child is added or removed. This ability to define "indirect constraints" allows constraints to model a wide array of dynamic application behavior, and promotes a simpler, more effective style of programming than conventional constraints [28].

When combined with a prototype-instance object model, constraints allow arbitrary delegation of values, not just from prototypes. Any slot can get its value from any slot of any other object through constraints. Therefore, the constraints can be used as a form of inheritance. However, constraints are more powerful than conventional inheritance since they can perform arbitrary transformations of the values. [5][16]

2.3 Graphics Systems

Prototype-instance object systems and constraint-based architectures have long been used in the field of graphics. However, very few systems have combined both approaches.

ThingLab [3] was the first system to integrate a prototype-instance object system and constraints. *Garnet* [18] was the first general-purpose user-interface toolkit based on a prototype-instance object model and constraints. *Garnet* was also the first system to introduce indirect constraints. Its follow-up *Amulet* [17] implemented the same programming approach as in *Garnet* but using *C++* instead of *Lisp*. *Amulet* introduced several improvements and innovations, including structural inheritance, control over the inheritance of slots, support for multiple constraint solvers, and a flexible demon mechanism for imperative reaction to object creation, deletion and modification.

In the field of 3D graphics, many systems have used constraints to maintain dependencies between objects [7][8][11][25][27][32]. *VB2* and *TBAG* both use multi-way constraints maintained by *Skyblue* [21], a local propagation multi-way constraint solver. *VB2* was the first 3D graphics toolkit to use indirect constraints to model the dynamic behavior of interactive applications. *UGA* [32] was the first 3D interactive graphics system to combine a delegation system and one-way constraints but does not integrate general indirect constraints. The *i3D* system described in [1] is a *VRML* browser whose core 3D toolkit has been re-implemented as a 3D extension of *Amulet*. The *i3D* toolkit

[1] Depending on the constraint maintenance algorithm, there might be some restrictions on side-effects allowed while evaluating constraints [16].

is the first 3D interactive graphics toolkit based on a prototype-instance object system combined with indirect constraints. The *i3D* toolkit makes some innovative use of the features of the *Amulet*'s object system, in particular to support sub-graph sharing using copy on demand and per-slot inheritance control at run time (see section 3.2.4).

2.4 Prototype-Instance Paradigm and Graphics

An interactive graphics world is an exploratory kind of environment, amenable to easy, flexible, incremental creation and modification. Rather than abstractions, the world is composed of actual concrete objects, which have a visual representation on the screen. Users see the object they manipulate as independent objects rather than instances of abstract types. This approach is much closer in feel to prototype-instance than to class-instance paradigms.

Previous analyses of prototype-instance systems suggest that delegation is more suited for graphics [4][31]. Conner and van Dam in their thorough analysis of graphics systems [5] demonstrate that most object-oriented graphics system try to support the kind of flexibility of the prototype-instance programming paradigm while being implemented in a class-instance language. Each system tries to bypass the limitations of the language object system through its own particular constructs. The result is that these systems make limited use of the language's features while usually not being able to support the same level of flexibility as a true prototype-instance object system. The reason for this commonly used approach lies in a form of conservatism and dogma: as stated by Wisskirchen in [31], class-instance models are more standard. In fact, one of the reasons why the developers of *Garnet* decided to stop *Garnet* and start *Amulet* was that they believed their work would gained better acceptance if it was implemented using *C++*, *C++* being more politically correct than *Lisp* [15].

When combined with a prototype-instance object system, indirect constraints provide a means to tie together objects in the world by declaring relationships between their variables. This aspect is essential to interactive applications, to easily specify how the modification of objects' attributes impact the state of other objects. Also, constraints add to prototype inheritance a means to explicitly control how objects in the world share information, as for example in the inheritance of graphical attributes along the branches of a graph, or in the context dependent retrieval of attributes' default values, common features of object-oriented graphics toolkits. Since constraints are automatically re-evaluated by an algorithm whose behavior is clearly defined, the programmer is freed from the tedious and error-prone task of invoking the method by hand as it is done with callbacks.

3 A PROPOSAL FOR THE EVOLUTION OF VRML

A *VRML* world is composed of a set of built-in node instances that are arranged in a directed acyclic graph. Nodes are defined by a type, attribute-value pairs, the list of events the node can send to notify its changes, and the list of events it can receive to modify its internal state. A specific syntactic construct allows packaging of a graph into a new type of node with a specific interface of attributes and events. This mechanism is called PROTOtyping.

3.1 Modeling Scene Graph

The modeling scene graph is probably the most advanced part of *VRML*. Carefully designed from the *Inventor* [23] and *Performer* [20], the *VRML* scene graph presents most of the features of modern 3D graphics APIs. However, the *VRML* scene graph suffers from some complications, which are related to the execution model and the need for optimizations.

Field, exposedField, eventIn, eventOut. We believe that the existence of four separate field classes is an unnecessary complication for browser implementers and world developers. The motivation to introduce a difference between exposedFields and fields is only to allow for optimizations; fields cannot be modified at execution time, their value is used only at creation time and do not need to be preserved. The result is that most *VRML* geometric primitives are defined using fields that cannot be changed at run-time. Having one class of fields that can be modified and queried at run-time is largely enough, does not presuppose how the nodes will be used by world developers, simplifies the implementation of browsers, and limits the number of concepts one has to learn to develop *VRML* worlds.

Bindable nodes. Bindable nodes are global attributes and their position in the scene graph does not have a meaning. The active node is the one that is currently bound. *Viewpoint* nodes being a type of bindable nodes, only one view of a *VRML* world can be active at a given time. For the sake of simplification, we believe that bindable nodes should be abandoned and a *View* node be introduced instead with references to the *Background*, *Fog*, *NavigationInfo* and *Viewpoint* nodes used by this particular view. The *View* node should also specify whether stereo rendering is used.

Rendering effects. There are a number of rendering effects not currently supported by VRML that should be supported in the future. Material transparency and object colors should always be combined with the texture regardless of the number of components. More control should also be given over the rendering quality of textures, a major limitation when VRML is used in multimedia installations based on powerful graphics hardware. In addition, it should be possible to render IndexedLineSet and PointSet nodes using lights and textures as for any other geometry node.

The proposed modifications to the modeling part of the *VRML* scene graph are very close to the modifications suggested in the *EMMA* proposal [12]. We chose to preserve dynamic lighting and all geometry nodes in order to preserve the compatibility with the existing *VRML* modeling graph. This aspect is important since most 3D authoring tools are now able to export (and some to import as well) their models as a *VRML* scene graph. Breaking compatibility would imply a year or more before these tools can again import/export their models natively as a *VRML* scene graph. We chose also to preserve the *NavigationInfo* node. Most users still lack experience with 3D navigation and we feel it is important that *VRML* provide a standard set of navigation techniques.

3.2 Run-time model

The *VRML* run-time execution model is based on ROUTEs and *Script* nodes. ROUTEs are used to directly connect a node's eventOut to an eventIn of some other node for the propagation of value changes. *Script* nodes are used to incorporate small programs in *VRML* worlds. They implement their behavior using a reactive model where the *Script*'s eventOuts are modified in response to changes of its eventIns. *Script* nodes are then tied to other objects in the world using ROUTEs. They are the only programmable part of *VRML* and hence, the only means to implement the behavior of the world.

One limitation of this model is that it does not allow the extension of existing nodes with new behaviors, which makes it very difficult to support applications that do not fit in the initial design. PROTOtypes provide a means to encapsulate data and behavior to define a new type of node, but there is no way to define a new PROTOtype as an extension of an existing PROTOtype using inheritance. Similarly, PROTOtypes cannot inherit from a built-in node type. The lack of inheritance complicates the implementation and maintenance of standard PROTOtype libraries.

Another problem lies in the *VRML97* specification itself, which does not clearly specify the order of evaluation of events along an event cascade [12], yielding to execution inconsistencies between browser implementations.

3.2.1 Prototype-instance object model

We believe that in order to go beyond the current limitations of the *VRML* programming model, we should redesign it using more solid foundations, i.e. an object-oriented programming paradigm. There have been several proposals in the past to make *VRML* more object-oriented [2][6][19]. However, these approaches have all been based on classes and instances. We believe that a prototype-instance object model is more suited for *VRML*. As stated in previous sections, this object model is more adequate for graphics because of its greater flexibility and its ability to define and modify instances at run-time.

Fortunately, the design of *VRML* fits well with a prototype-instance object model. Nodes are defined by a type and a set of named fields. Node instancing is equivalent to copying the node's prototype and setting its fields to the desired values. Providing the set of built-in nodes simply consists of supplying the set of built-in prototypes.

Adding a prototype-instance model has a number of benefits for *VRML*. First is the support for inheritance; nodes can inherit from other nodes and in particular from built-in prototypes. Second, objects advertise their fields to other objects; field traversal as well as query for field existence and type are available. Third, fields can be added to or removed from any object at any time to extend its data and behavior. This flexibility of the object system will allow *VRML* to be extended or modified to support applications unanticipated by the initial design.

3.2.2 Indirect Constraints

The current *VRML* event model does not allow the extension of existing or new prototype nodes with behaviors defined at run-time. Moreover, the event model fails to provide enough expressive power to effectively describe the relationships between objects. We propose to abandon ROUTEs and *Scripts* and to introduce the support for indirect one-way dataflow constraints instead. This means that relationships between objects are defined by installing into a node's field a formula that computes its value based on the value of others fields. As stated by Myers et al in [16], since constraints can contain arbitrary code, they might be considered to be like methods, and be used to define the operation of objects. Constraints can therefore be used to extend the behavior of existing prototypes or to define that of new prototypes.

Let's take a simple example to show the power of constraints over some more standard callback mechanism as proposed by others [12]. Let's assume that a world developer wants to extend *VRML* with the ability to maintain hierarchical bounding boxes. This functionality might be needed, for example, to implement a scale manipulator so that it could be able to position its handles around the object it manipulates. For that purpose, the developer will add two new fields, *minBox* and *maxBox*, to the grouping, shapes and geometry nodes, which represent respectively the lower, left, back and upper, right, front of the box. Constraints installed in these fields will compute their value according to the semantics of the node. Let's assume we have an *OnInit* section in the *VRML* file programmed using an *ECMAScript* like language. To make the example simpler we also assume that geometry nodes maintain their bounding box, and that the world is composed only of *Transform* and *Shape* nodes.

```
DEF shapeMinBoxFormula SFVec3f Formula {
  // self refers to the instance executing the
  // constraint. Accessing the geometry field
  // installs a dependency on that field
  var geometry = self.Get("geometry");
  if (geometry == null) {
    // dependency on geometry.minBox
    return geometry.Get("minBox");
  } else {
    return new SFVec3f(0,0,0);
  }
}

DEF shapeMaxBoxFormula SFVec3f Formula {
  // Similar code to the min formula
}

DEF transformMinBoxFormula SFVec3f Formula {
  // dependency on the children field
  var children = self.Get("children");
  if (children.length == 0) {
    return new SFVec3f(0,0,0);
  } else {
    var minVec = new SFVec3f(
      MAX_FLOAT, MAX_FLOAT, MAX_FLOAT);
    for (var i=0; i < children.length; i++) {
      var child = children[i];
      // dependency on child's minBox field
      var childMinVec = child.Get("minBox");
      minVec[0] = min(minVec[0], childMinVec[0]);
      minVec[1] = min(minVec[1], childMinVec[1]);
      minVec[2] = min(minVec[2], childMinVec[2]);
    }
    // dependencies on translation, rotation,
    // center, scaleOrientation, scale fields
    var m = new Matrix(
      self.Get("translation"),
      self.Get("rotation"),
      self.Get("center"),
      self.Get("scaleOrientation"),
      self.get("scale"));
    return Matrix.transform(minVec);
  }
}

DEF transformMaxBoxFormula SFVec3f Formula {
  //similar code to transformMinBoxFormula
}

OnInit () {
  // Here, we install the fields and formulas
  // for the Shape prototype.
  // Setting a non existing field, automatically
  // installs the field in the prototype.
  Shape.Set("minBox", shapeMinBoxFormula);
  Shape.Set("maxBox", shapeMaxBoxFormula);
  // From now on, any instance of Shape
  // will compute its bounding box.

  // Here, we install the fields and formulas,
  // for the Transform prototype.
  Transform.Set("minBox",transformMinBoxFormula);
  Transform.Set("maxBox",transformMaxBoxFormula);
  // From now on, any instance of Transform
  // will compute its bounding box.
}
```

Since indirect constraints are able to automatically determine the variables on which they depend when they are evaluated, they are able to transparently handle the dynamic nature of the graph. In that example, a *Shape*'s bounding box will be recomputed if the geometry's bounding box changes or if the geometry node is replaced. In that latter case, the constraint will automatically install a dependency on the new geometry's bounding box.

Similarly, the bounding box of a *Transform* node will be automatically recomputed if the transformation changes, if children are added or removed, or if any of the children's bounding boxes changes.

Programming the same behavior using callbacks would be much more complex. For example, to implement the bounding box behavior on a *Shape*, a callback should be installed on the current geometry's bounding box. To make provision for changes of geometry, another callback must be installed on the shape's geometry field to remove the bounding box callback from the old geometry and install it on the new geometry. It would be even more complex for the *Transform* node.

Constraints are inherited. Therefore, when a constraint is installed in a prototype's field, newly created instances as well as existing instances[2] will exhibit the new behavior. Conversely, when a constraint is uninstalled, instances will cease to exhibit the behavior. This aspect is important to add and remove transient behaviors.

It is useful to allow for multiple constraints in one field when the field's value might be computed different ways. Only one constraint is used to compute the value at a given time, but the computing constraint may change depending on the system state.

It is interesting to note that backward compatibility with ROUTEs can be preserved with constraints. In effect, ROUTEs can be viewed as one-way equality constraints and fan-in as multiple equality constraints in a field.

3.2.3 Demons

Demons are special procedures that can be registered with an object's fields to react to their value changes. Demons are useful to implement frequently occurring, autonomous object behavior. A demon's action is a procedure of any complexity that is executed sequentially, that may create or destroy objects, add or remove constraints, attach or detach demons, or set the value of fields. Demons are executed in their order of activation and need to be interleaved with constraint evaluation. A demon can be registered with multiple fields for an action to be performed in response to the value change of different fields. Similarly, multiple demons can be registered with a single field, for multiple actions to be performed when the field's value is modified. Like constraints, demons are inherited. Therefore, when a demon is attached to an object's field, newly created instances as well as existing instances[3] will exhibit the demon's behavior.

3.2.4 Simulated Sharing and Multiple Parents

Constraints rely on the invoking instance to find the variables on which they depend. Therefore, it is essential that walking along the scene graph branches be possible from any node. In VRML, only walking down the graph branches is possible. This is due to the DEF/USE mechanism that allows node sharing. Since a node can be referenced several times in the graph, it may have multiple parents. Walking up the scene graph seems impossible as the relevant parent cannot be determined [24]. Therefore, node sharing prohibits maintaining or querying any node information that depends on the position in the graph, such as the global transformation or inherited attributes. In the Inventor toolkit [23], this problem is addressed using path objects that describe how to go down the graph to reach the node. This approach is not satisfactory since it requires structural knowledge of the scene graph. Graph manipulations are very likely to invalidate the path objects and queries cannot be performed from within the shared node.

One obvious and straightforward solution would be to abandon the sharing of nodes. However, sub-graph sharing is a powerful modeling construct and it should be preserved. An original solution to this issue has been implemented in the *i3D* toolkit. We refer to it as *simulated sharing*. The technique exploits the ability of *Amulet*'s object model to modify the inheritance rule for slots [14]. *Amulet* supports four inheritance rules: *inherit*, *local*, *copy*, and *shared*[3]. *Inherit* is the default rule; a slot is always obtained from the prototype unless it is overridden in an instance. With *local*, the slot is not inherited and only exists for the prototype. With *copy*, the slot is copied into instances at creation time so later changes to the prototype will not affect instances. With *shared*, the slot is shared by the prototype and all its instances. In *i3D*, the nodes' slots are defined with the default *inherit* rule. Slots that corresponds to a *VRML* field are marked so that they can be identified. Nodes maintain a *parent* slot as well as extra slots that depend on the node's position in the graph such as a *globalTransform* slot for *Transform* nodes.

When a node is referenced more than once in the scene graph, it is automatically promoted to be the root of what is called a *share group*. The inherit rule of all slots corresponding to a *VRML* field is changed from *inherit* to *shared*, and an instance of the node is created and installed in the graph in lieu of the node itself. If the node has children, a *share group* is created for each of the children. Since all *VRML* slots are shared by all the instances of the *share group*, *VRML* sharing behavior is obtained. However, because each reference to the node in the graph is using a different instance, a *parent* slot can be maintained as well as extra slots that depends on the position of the node's reference in the graph. With regard to *VRML*, a *share group* is treated as a unique instance and destroying one instance will destroy and remove all instances from the graph.

3.3 PROTOtypes

VRML PROTOtypes allow authors to define new types of nodes. PROTOtypes' instances can later be inserted in the *VRML* scene graph. A PROTOtype is defined by an external interface and a hidden implementation built using an internal graph of nodes. Values are mapped from the interface to the implementation using the IS syntactic construct.

The main advantage of PROTOtypes is that they allow the definition of reusable components. However, PROTOtype definition suffers from the limitations of the event/Script run-time model, which is used to implement its behavior. Moreover, the lack of inheritance strongly complicates the implementation and maintenance of standard libraries of PROTOtypes.

In our proposal, PROTOtypes inherit from other nodes' prototypes (either built-in or user-defined) and use indirect constraints to implement their behavior. PROTOtypes inherit the interface of their parent prototype which do not need to be described again unless some fields must be overridden. Let's take an example to make the PROTOtype mechanism more concrete. Let's assume that we want to define a new type of Shape that can be selected if a TouchSensor is active on that Shape.

```
PROTO SelectableShape EXTENDS Shape [
  field SFNode touchSensor NULL
  field SFBool selected FALSE
  // other fields are inherited from Shape and
  // do not need to be described
] {
  DEF findSensorFormula SFNode Formula {
    // Go up the hierarchy to find a touchSensor
    // active on that Shape
  }
```

[2] Unless the field has been overridden in the instance.

[3] The *shared* inherit rule is referred as the *static* inherit rule in the Amulet's manual. However, we find *shared* more explicit so we use it instead.

```
  DEF selectedFormula SFBool Formula {
    var sensor = self.Get("touchSensor");
    if (sensor == null) {
      return FALSE;
    } else {
      var geometry = self.Get("Geometry");
      if (geometry == null)
        return FALSE
      else {
        var active = sensor.Get("isActive");
        var current = self.Get("selected");
        if (active) return !selected;
        else return selected;
    }
  }
  OnInit {
    // install the constraint. self refers to the
    // new prototype
    self.Set("touchSensor", findSensorFormula);
    self.Set("selected", selectedFormula);
  }
}
```

The *selectableShape* inherits from *Shape* and therefore behaves exactly as a standard *Shape*. Some behaviors have been added using constraints so that it can keep track of its selection state. The *OnInit* code section is called when the prototype is defined and installs the constraints. Similarly, optional *OnCreate*, *OnCopy*, and *OnDelete* code sections exist to define the special operations when a prototype's instance is created, copied or deleted. Some new PROTOtype could inherit from *SelectableShape* to add some feedback behavior for example. Note that the PROTOtype does not rely on a *Shape* instance for its implementation as with current *VRML* PROTOtypes. Compound PROTOtypes are simply defined as instances of *Group*, and the implementation nodes are declared as children.

3.4 Other issues

Our approach tends to make *VRML* evolves towards a programmable run-time environment rather than a scene description language. In this paper, we have concentrated our discussion on a new flexible and powerful run-time model based on a prototype-instance object paradigm and indirect constraints. However, there are a number of issues that still need to be addressed to make *VRML* fully functional:

Interaction. In *VRML*, sensor nodes inserted in the scene graph are the source of events related to user actions. This approach works fairly well but has limitations. First, there is no way to react to keyboard input. Second, the pointing device is the mouse and there is no way to add support for other devices. Third, drag and drop behaviors cannot be implemented since sensors do not identify the shapes over which the interaction is started or finished. Finally, programs cannot access the definition of the viewing volume, the global transforms of objects and the collision detection algorithm.

Programming language. Our programming approach is based on indirect constraints and demons. We did not talk much about the programming language. In effect, we did not want to overload the discussion with language wars. In fact, we tend to support a similar approach as the one promoted in the *EMMA* proposal [12]. A small built-in language based on *ECMAScript* provides the basic services to create objects, to manipulate the objects' fields and their values, and to add constraints and demons. Other APIs are provided either built-in or as a library for services like network access, mathematical calculations, collision detection, standard constraints, etc. These libraries can be programmed in other more efficient languages.

Integration with other media. Currently, the integration of *VRML* with *HTML* is done through the *EAI* [13]. One problem with the *EAI* is that the set of proposed features is far from being complete. In particular, the fields of a node cannot be traversed and queried for their name, type or value. One has to know what the name and type of a field to be able to access its value. We believe that we should look for integration with *XML* in the future and abandon the *EAI*. Again, we should look for maximum flexibility. In particular, important features are the ability to query and set the values of fields and to attach data streams to node fields.

3.5 Impact On Browser Implementation

The most difficult task for developing browsers that support the new run-time model, will be the implementation of the object system and the constraint maintenance algorithm. These techniques are well-described in the literature: for prototype-instance object systems, references [4][14][16][26] are good starting points while reference [29] contains a detailed description of the algorithms for one-way indirect constraint evaluation. Instead of doing all the work by themselves, developers might want to use some public domain implementation. One candidate system is *Amulet* which is a public-domain toolkit that runs on *Windows*, *MacOS*, and *Unix*. *Amulet*'s object system [14][16][30] provides all necessary features to support the proposed run-time model and would be an ideal basis for the rapid implementation of a working prototype. *Amulet*'s source and documentation can be downloaded from the *Amulet*'s home site: http://www.cs.cmu.edu/~amulet/.

4 CONCLUSION

We have proposed and evolution of *VRML* towards an object-oriented run-time environment. Unlike other authors, we have chosen to use a prototype-instance programming paradigm combined with indirect constraints. This approach fits well with the design of *VRML* and providing the set of built-in nodes simply consists of supplying the set of built-in prototypes. Indirect constraints are used to specify the behavior of objects and to explicitly declare the relationships between objects. Indirect constraints are automatically maintained freeing the world developers from the tedious and error-prone task of invoking them by hand as it is done with callbacks. Demons are used to implement frequently occurring autonomous object behavior using a reactive model. PROTOtypes inherit from other nodes' prototypes (either built-in or user-defined) and use indirect constraints to implement their behavior. The flexibility and expressive power of this approach allow world developers to tailer and extend *VRML*'s run-time environment to meet their needs. This is especially important to allow *VRML* to effectively cope with unanticipated applications.

We have given few details about the exact syntax for the definition of PROTOtypes and constraints. The next step is clearly to write a detailed proposal for the evolution of the *VRML* syntax. Then, a working browser prototype should be implemented so that world developers can experiment with these features.

ACKNOWLEDGEMENTS

I would like to thank Enrico Gobbetti, Russell Turner and Magnus Kempe for their useful comments and suggestions.

REFERENCES

[1] Balaguer J.-F., Gobbetti E. (1995) i3D, A High-Speed 3D Web Browser. *Proc. 1st VRML Symposium*.

[2] Besson C.A (1997) An Object-Oriented Approach to VRML Development. *Proc. 2nd VRML Symposium*.

[3] Borning A. (1981) The Programming Language Aspects of ThingLab; a Constraint-Oriented Simulation Laboratory, *ACM Transactions on Programming Languages and Systems*, 3(4): 353-387.

[4] Borning A. (1986) Classes versus Prototypes in Object-Oriented Languages. *Proc. ACM/IEEE Fall Joint Computer Conference*: 36-40.

[5] Conner D.B., van Dam A. (1992) Sharing Between Graphical Objects Using Delegation. *Proc. Third EUROGRAPHICS Workshop on Object-Oriented Graphics*, Champéry, Switzerland.

[6] Diehl S. (1997) VRML++: A Language for Object-Oriented Virtual Reality Models. *Proc. 24th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Asia)*.

[7] Elliot C., Schechter G, Yeung R., Abbi-Ezzi S. (1994) TBAG: A High Level Framework for Interactive, Animated 3D Graphics Applications. *Proc. SIGGRAPH*: 421-434.

[8] Gobbetti E., Balaguer J.-F. (1993) VB2: a Framework for Interaction in Synthetic Worlds. *Proc. UIST*: 167-178.

[9] Gobbetti E., Balaguer J.-F. (1995) An Integrated Environments to Visually Construct 3D Animations. *Proc. SIGGRAPH*: 395-398.

[10] Halperin B., Nguyen V. (1987) A Model for Object-Based Inheritance. *Research Directions in Object-Oriented Programming*, the MIT Press.

[11] Kass M. (1992) CONDOR: Constraint-based Dataflow. *Proc. SIGGRAPH*: 321-330.

[12] Marin C. (1998) Beyond VRML. http://www.marrin.com/vrml/private/EmmaWhitePaper.htm.

[13] Marrin C. (1997) External Authoring Interface Reference. http://www.cosmosoftware.com/develo-per/moving-worlds/spec/ExternalInterface.html

[14] McDaniel R, Myers B.A (1995) Amulet's Dynamic and Flexible Prototype-Instance Object and Constraint System in C++. Carnegie Mellon University School of Computer Science Technical Report, no. CMU-CS-95-176.

[15] Myers B. (1996) *The Garnet FAQ*. http://www.cs.cmu.edu/afs/cs.cmu.edu/project/garnet/garnet/FAQ/

[16] Myers B.A, McDaniel R., Miller R., Vander Zanden B., Giuse D., Kosbie D., Mickish A (1999) The Prototype-Instance Object Systems in Amulet and Garnet. *Prototype Based Programming*, James Noble, Antero Taivalsaari and Ivan Moore, eds. Springer-Verland. To appear.

[17] Myers B.A, McDaniel R.G., Miller R.C., Ferrency A.S., Faulring A., Kyle B.D., Mickish A., Klimovitski A., Doane P, (1997) The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering*, 23(6): 347-365.

[18] Myers B.A., Giuse D.A., Vander Zanden B. (1992) Declarative Programming in a Prototype-Instance System: Object-Oriented Programming Without Writing Methods. *Proc. OOPSLA*: 184-200.

[19] Park S., Han T. (1997) Object-Oriented VRML for Multi-user Environments. *Proc. 2nd VRML Symposium.*

[20] Rohlf J., Helman J. (1994) IRIS Performer: A High Performance Multiprocessing Toolkit for Real-Time 3D Graphics. *Proc. SIGGRAPH*: 381-394.

[21] Sannella M. (1993) *The SkyBlue Constraint Solver*. TR-92-07-02, Department of Computer Science, University of Washington.

[22] Stein L.A, Lieberman H., Ungar D. (1989) A Shared View of Sharing: The Treaty of Orlando. *Object-Oriented Concepts, Databases and Applications*, Addison-Wesley.

[23] Strauss P.S, Carey R. (1992) An Object-Oriented 3D Graphics Toolkit. *Proc. SIGGRAPH*: 157-166.

[24] Strauss P.S. (1993) IRIS Inventor, a 3D Graphics Toolkit. *Proc. OOPSLA*: 192-200

[25] Sutherland I.E. (1963) Sketchpad: A Man-Machine Graphical Communication System. *Proc. Of the Spring Joint Computer Conference*: 329-346.

[26] Ungar D., Smith R.B. (1987) SELF: The Power of Simplicity. *Proc. OOPSLA*: 227-241.

[27] Upson C., Fulhauber T., Kamins D., Laidlaw D., Schlegel D., Vroom J., Gurwitz R., van Dam A., (1989) The Application Visualization System: A Computational Environment for Scientific Visualization. *IEEE Computer Graphics & Applications*, 9(4): 30-42.

[28] Vander Zanden B.T, Myers B.A., Giuse D., Szeleky P. (1991) The Importance of Pointer Variables in Constraint Models. *Proc. SIGCHI*: 235-240.

[29] Vander Zanden B.T., Myers B.A., Giuse D.A, Szekely P. (1994) Integrating Pointer Variables into One-Way Constraint Models. *ACM Transactions on Computer-Human Interaction* 1(2): 161-213.

[30] Vander Zanden B.T. (1998) The Design and Implementation of Amulet's ORE System. http://www.scrap.de/pub/openamulet/ore_essay.pdf

[31] Wisskirchen P. (1990) *Object-Oriented Graphics*. Springer-Verlag.

[32] Zeleznik R.C., Conner D.B., Wlocka M.M, Aliaga D.G. Wang N.T., Hubbard P.M., Knepp B., Kaufman H., Hughes J.F., van Dam A. (1991) An Object-Oriented Framework for the Integration of Interactive Animation Techniques. *Proc. SIGGRAPH*: 105-112.