A Language for Creating and Manipulating VRML

Terence J. Parr*
Parr Research Corporation

Timothy F. Rohaly†
ORC Incorporated

Abstract

Documents written in two-dimensional markup languages such as HTML are easy to create with word processor-style editors. Creating VRML[1] documents, however, is complicated by its three-dimensional nature. VRML can be generated from existing 3D graphics editors or new applications can be written specifically for constructing VRML scenes. In this paper, we briefly introduce a prototype interpreted language called IVL (originally developed as a general Open Inventor[15, 16] prototyping language) that allows VRML models and scenes to be interactively developed and manipulated with simple English-like descriptions. IVL uses public domain code and can run on a wide variety of platforms—any platform supporting C++ and Open Inventor.

As a demonstration of an application of this language, we describe work on the development of a "metaverse"[14]—an interactive, multiuser, networked 3D world.

1 Introduction

Using a 3D graphics editor to construct VRML scenes is not always effective. Aligning objects and placing them relative to other objects can be difficult. We have built a prototype language, called IVL, that allows many scene graphs to be built quickly and easily, and manipulated across a network with simple English-like commands. We have made IVL network aware—all file references apply equally well to URL's, so for instance it doesn't matter where the parts of the scene graph are stored or where the command files are stored. Similarly, IVL can deliver its results locally or across a network using the HTTP protocol.

Because IVL is an interpreted language, scenes are created and modified as commands are executed. Our prototype implementation uses a modified Open Inventor-based VRML browser to display the scene graph.

Although IVL was developed for Open Inventor scenes, we have adapted and expanded it for use on VRML scenes. Consequently, we have tried to keep a close association between the names and conventions used in IVL and VRML whenever possible. For example, to create a scene containing a ball on top of a box, one would write

```
ball = Sphere        // Create named instance
box  = Cube
draw box             // Render box
draw ball above box  // Transform & render ball
```

All objects are placed relative to a previously-drawn object and, unless specified otherwise, are placed with bounding boxes touching; thus in this example, the ball is sitting directly on the top of the box. Distances between objects can be specified in "world coordinates"; e.g.,

```
draw ball 3 above box
```

places the ball 3 units above the box.

In this paper, we describe the IVL language definition and implementation then explore the applications made possible via VRML and IVL. The applications of IVL discussed in this paper is still under development, but we present our ideas and directions here to foster growth in interactive VRML development by other researchers.

2 The IVL Language

The IVL language is designed to be a rapid prototyping language for VRML/Inventor graphics programming.

*1535 Francisco St #9, San Francisco, CA 94123, parrt@parr-research.com
†603 Fox Bow Dr. Suite B, Bel Air, MD 21014, rohaly@ocnus.com

IVL programs are much shorter than functionally-equivalent C++ implementations and require no compilation as the language is interpreted—that is, commands are executed and the result apparent in the viewer window immediately.

IVL commands resemble English rather than a typical programming language. This feature makes it easy for even non-programmers to assemble and manipulate scenes.

IVL has four basic capabilities:

1. *Defining new graphical object classes.* New classes can be defined as any existing scene graph and can be read from a file or from a URL. The scene graph can be either a VRML graph or an Inventor graph. These new classes augment the predefined, atomic classes.

2. *Defining new graphical objects.* Objects do not exist until they are instanced. The programmer must create objects before drawing anything.

3. *Drawing objects in the browser window.* This amounts to modifying the scene graph being viewed by the browser. All objects except for the first are drawn in relation to another already-drawn object.

4. *Moving objects according to data files.* This is a special case of keyframe animation. The programmer can "attach" a data file with an x, y, z coordinate list (one `SFVec3f` per line) which is used as a motion path. The relative coordinates are placed in a `SoTransform` attached to the graphics object and repeatedly updated to simulate motion.

In addition, this particular IVL implementation is capable of writing out the scene graph at any point and can read in this graph (or any other VRML ascii file, stored locally or at some URL) as a new graphical object class. Further, programs may be stored in text files and executed as if they had been typed in manually, allowing programmatic control of the scene graph. Applications can effect animations or control the "behavior" of objects in this manner.

2.1 Predefined Objects

IVL has predefined atomic types that the programmer may use to create graphical objects. These types represent all the VRML nodes derived from `SoShape` and exclude those nodes derived from `SoVertexShape`. IVL types are given the same name as the corresponding VRML node.

- `Cone`. A unit cone centered at 0,0,0.
- `Sphere`. A unit sphere centered at 0,0,0.
- `Cylinder`. A unit cylinder centered at 0,0,0.
- `Cube`. A unit cube centered at 0,0,0.
- `AsciiText`. A unit height, zero depth 2D text object whose `string` field must be set before it can be drawn.

Additional atomic types can be added easily. We have extended the VRML set of primitives to include:

- `Pyramid`. A unit pyramid centered at 0,0,0.
- `Torus`. A unit torus centered at 0,0,0. The ratio of diameters must be specified before it can be drawn.
- `Text3`. A unit height, unit depth 3D text object whose `string` field must be set before it can be drawn.

IVL also provides "environment" nodes in the form of light sources. The light sources can be placed, transformed, and colored in the same manner as shapes.

- `DirectionalLight`. A light source of parallel rays travelling in the -Z direction.
- `PointLight`. A light source with rays eminating radially.
- `SpotLight`. A light source producing a cone of light in the -Z direction.

2.2 Creating New IVL Classes

More interesting than the predefined objects, IVL allows the creation of new classes by simply giving IVL the name of a file or URL which holds a scene graph. Since IVL allows Inventor scene graphs to be read in, this is a way to extend the basic set of allowed nodes in VRMl. That is, one can model with NURBS-based shapes or any other Inventor-supported node, as still be able to manipulate these objects and generate valid VRML as output. IVL allows you to name the type of these objects for future reference, much like the VRML concept of `DEF`. For example,

```
class Pear = "http://fruit.org/bartlett.wrl"
```

This ability to define classes from pre-existing scenes allows complex scenes to be created and manipulated just as easily as simple scenes. The ability to reference VRML files through their URL, couples with publically-available repositories of VRML models, makes IVL a modeling tool. IVL also supports `gzip` compressed files, useful for reducing download times.

2.3 Creating New IVL Objects

Graphical objects are instanced by assigning a new identifier to a class name. For example,

```
a = Cone; b = AsciiText
```

where the ';' is required only if two commands are to be specified on the same line.

Objects of user-defined classes are actually references to a single scene graph read locally from a file or over the net from a URL. These graphs could be very large such as a high resolution model of a building or a city block.

The `AsciiText` class requires that you assign its `string` attribute to the actual text you want to display.

```
b.string = "something to say"
```

2.4 Drawing IVL Objects

Once an object has been defined, it can be displayed at 0,0,0 by simply saying

`draw` *object*

This is normally (but not necessarily) done only with the first object. All other objects should be drawn relative to a previously drawn object. For example,

```
lostair  = Cube
histair  = Cube
draw lostair
draw histair above lostair and behind lostair
```

The object to draw we shall call the "drawn object" and the object to which the drawn object is relative to what we shall call the "relative object." For the second draw command, `histair` is the drawn object and the `lostair` is the relative object. The form of any modifier is

n modifier relative-object

where the optional "*n*" indicates a distance in the direction of the modifier, with direction defined from the viewer's point-of-view using the coordinate system defined in the VRML specifications. There is only one drawn object per `draw` command. Multiple modifiers are combined with either an "and", a comma, or nothing at all in between. For example,

```
draw d to the left of a, above b and behind c
```

The following "positional phrases" or modifiers are available.

1. `above`. Place the drawn object directly above the relative object. In other words, make the bottom of the drawn object at the same height as the top of the relative object. Notice that due to the possible existence of other modifiers, the drawn object may not physically touch the relative object.

2. `below`. Make the top of the drawn object even with the bottom of the relative object.

3. `behind`. Make the front of the drawn object at the same depth (into the screen) as the back of the relative object.

4. `in front of`. Make the back of the drawn object at the same depth as the front of the relative object.

5. `to the left of`. Make the right side of the drawn object even with the left side of the relative object. The "`the`" is optional.

6. `to the right of`. Make the left side of the drawn object even with the right side of the relative object. The "`the`" is optional.

A uniform default offset can be set independently for each of these modifiers if it is desired to position objects so that the bounding boxes do not touch.

2.5 Object Attributes

The scale and rotation of any object can be altered by setting the appropriate field. The material properties color can be set for any object. Attributes of an object may be set either before or after the object is referenced; e.g.,

```
a = Cone
a.diffuseColor = [.2 .35 .4]
a.scaleFactor = [ 1 1 3]
a.rotation = [0 0 1 20]
```

The following assignments are currently possible for all objects, with the intention of supporting all attributes found in the current version of VRML:

obj.`ambientColor = [x y z]`
obj.`diffuseColor = [x y z]`
obj.`specularColor = [x y z]`
obj.`emissiveColor = [x y z]`
obj.`shininess = value`
obj.`transparency = value`
obj.`texture = "filename | URL"`
obj.`scaleFactor = [x y z]`
obj.`rotation = [x y z r]`

The following assignments are currently possible for text objects only.

obj.`string = "string"`
obj.`font = "fontname"`

2.6 Animating IVL Objects

Any object that has been drawn (inserted into the scene graph) can be animated—that is, an object can be translated along a path via a series of relative offsets stored in a file. Each line of the data file is an x,y,z coordinate such as:

```
0.000000 0.400000 0.000000
0.000000 0.500000 0.000000
0.000000 0.600000 0.000000
0.000000 0.700000 0.000000
...
```

where the increment is typically small so that the motion will not be jerky. Here, the object is slowly being raised in the y direction. Each tuple is currently read and the object translated regularly every 50 milliseconds. This delta can easily be changed in the IVL interpreter.

The form of the command to do this is currently: `translate`*obj* `with`*"xyzdatafile"*

We are currently working on the addition of roll, pitch, yaw and scale data.

2.7 Saving the Scene Graph

The entire scene graph may be saved in either VRML 1.0 ascii or `gzip`-compressed VRML 1.0 ascii form. This is primarily useful as a mechanism to create new complex graphical object classes defining the parts separately and assembling the complex classes hierarchically. The form of this command is:

```
save as "obj.wrl"
```

or

```
gzsave as "obj.wrl.gz"
```

(Note that the inclusion of `gzip` is very important since VRML files are highly compressible in general, and since many applications are limited by the bandwidth required to download these VRML files). Scene graphs can then be read in as single object classes via:

```
class Obj = "obj.wrl"
a = Obj      // make an instance
draw a       // draw the dude
```

Scene graphs may be written out such that their center of mass is 0,0,0 when read back in (a `Transform` node is added to the graph before writing it out). This is useful if the saved scene is going to be read back in as an object.

2.8 Executing Scripts

Files or URL's containing valid IVL command sequences may be executed via

```
exec "script.ivl"
```

2.9 Examples

This section provides a series of eight examples that illustrate many of features of the IVL language. The identifiers in these examples have been kept simple, but the full range of C identifiers is allowed.
Draw a cone with with two balls and a cylinder:

```
a = Cone
b = Sphere
c = Cylinder
d = Sphere
draw a
draw b to the left of a
draw c to the right of a
draw d above a
```

Draw the same objects as the previous example, but set the color of the objects and use more complicated position modifiers. Note the use of `and` and the comma for an English-like syntax:

```
a = Cone
a.diffuseColor = [.3 .2 .7]
b = Sphere
b.diffuseColor = [0 .5 .5]
c = Cylinder
c.diffuseColor = [.25 0 .1]
d = Sphere
d.diffuseColor = [.4 .2 0]
draw a
draw b to the left of a, above a and behind a
draw c in front of b
draw d to the right of b
```

Draw a big flat red sheet with a blue and a green cylinder on it:

```
base = Cube
base.diffuseColor = [1 0 .0]
base.scaleFactor = [ 6 .2 6]
fl = Cylinder
fl.diffuseColor = [0 0 1]
fr = Cylinder
fr.diffuseColor = [0 1 0]
draw base
draw fl above base
draw fr above base to right of fl
```

Define a new object class. Note the use of distances in the modifiers so that the objects are not butted up against each other:

```
class ConeBall = ``coneball.wrl.gz''
a = ConeBall
b = ConeBall
draw a
draw b 2 to the left of a
```

Draw a staircase, save it and draw another staircase right next to it:

```
a = Cube
b = Cube
c = Cube
draw a
draw b behind a
draw c above b
save as ``step.wrl''
class Step = ``step.wrl''
d = Step
// Note that d will be down 1 unit
// from a.  This is normal since the
// new object, d, has a center
// at 0,0,0; the old object is
// centered at 0,1,0
draw d to the left of a
```

Draw a cone and two objects which are animated according to data in files d and d2:

```
a = Cone
b = Sphere
c = Cube
a.diffuseColor = [.25 .25 .75]
b.diffuseColor = [1 0 .5]
draw a
draw b above a
draw c to the left of a
translate b with ``d2''
translate c with ``d''
```

Put some text up:

```
a = AsciiText
a.diffuseColor = [.937 .517 .08235]
a.string = "Eat at Joe's"
draw a
```

Robot from page 58 of The Inventor Mentor[15]:

```
// Define a leg
thigh = Cube
thigh.scaleFactor = [ .6 1.1 .6]
calf = Cube
calf.scaleFactor = [ .5 1.1 .5]
foot = Cube
foot.scaleFactor = [.4 .4 1]
draw foot
draw calf above foot and -1 behind foot
draw thigh above calf
save as ``leg.iv''

// Construct robot
head = Sphere
head.scaleFactor = [1.5  1.5  1.5]
body = Cylinder
body.scaleFactor = [2.5  3  2.5]

draw head
draw body below head

class Leg = ``leg.iv''
lleg = Leg
rleg = Leg
draw lleg below body and -2.25 to the left of
     body and -3.1 in front of body
draw rleg 0.5 to the right of lleg
```

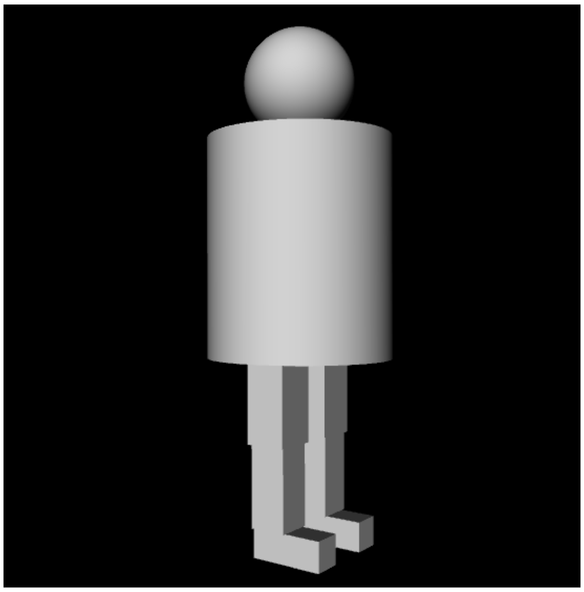

Figure 1: View of "Robot" from the IVL viewer

3 Implementation

The IVL language is implemented using PCCTS, the Purdue Compiler Construction Tool Set[12, 11, 13, 2]. PCCTS uses the IVL grammar to generate C++ code with calls to the Open Inventor class library. This code is compiled to create the IVL interpreter. To build IVL from scratch requires Open Inventor, PCCTS, and a

C++ and C compiler. Thus, IVL is a good match for the public-domain SDSC browser[3] which has the same requirements (*sans* PCCTS).

In this section, we give a brief description of PCCTS and outline the IVL implementation structure.

3.1 PCCTS

Despite the parsing power of $LR/LALR$ algorithms, e.g. yacc[9], programmers often choose to write recursive-descent parsers by hand to obtain increased flexibility, better error handling, and ease of debugging. PCCTS contains ANTLR, a public-domain parser generator that combines the flexibility of hand-coded parsing with the convenience of a parser generator. ANTLR has many features that make it easier to use than other language tools. Most important, ANTLR provides *predicates* which let the programmer systematically direct the parse via arbitrary expressions using semantic and syntactic context; in practice, the use of predicates eliminates the need to hand-tweak the ANTLR output, even for difficult parsing problems. ANTLR also *integrates* the description of lexical and syntactic analysis, accepts $LL(k)$ grammars for $k > 1$ with Extended BNF notation, and can automatically generate abstract syntax trees.

PCCTS is totally public domain—it has no legal restrictions on its use or incorporation into commercial applications. Versions of PCCTS are available for Unix, DOS, Windows, OS/2, Macintosh, and NeXT platforms[4].

3.2 Program Structure

We have split the IVL interpreter into two main components (and two processes):

The IVL parser accepts commands from standard input and modifies the scene graph and other internal data structures. This component is run as the main process. Objects that are to be animated are added to a list shared by this process and the viewer process. Since SGI's implementation of Open Inventor is not multi-thread safe, we cannot modify the scene graph directly from within the parsing process.

The IVL viewer is attached to the scene graph and updates the viewer window accordingly. Every 50 milliseconds, this process looks to see if any objects have been added to the list of objects to animate. All animated objects are updated by this main loop "timeout" callback. In place of the viewer, IVL can deliver the results directly to a VRML browser via the HTTP protocol.

Figure 3.2 illustrates the basic block diagram of the IVL application.

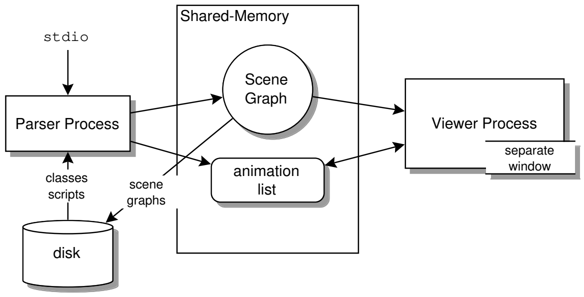

Figure 2: Implementation of IVL code

3.3 Source Code

The grammar specification for the IVL interpreter and a version of IVL pre-compiled for various platforms will be put in the public domain[5]. We also anticipate making the source code for the IVL interpreter publicly available under the same terms and conditions as PCCTS.

4 Applications

In this section our purpose is to demonstrate the use of an interpreted language to control aspects of a 3D interactive space through creating, instancing, placing, and modifying VRML nodes.

Our first example is simply the IVL interpreter on a web page[6]. This page has an HTML `textarea` in which the client can type in IVL commands; the resulting scene graph is returned to the client's VRML browser.

The second application is a little more ambitious. We would like to view VRML scenes as part of an integrated, seamless space which one could travel through while maintaining relation to the previous space. Currently in VRML, clicking on a hyperlink *replaces* the current scene with a new one. This leads to an effect sort of like the transporter in Star Trek; you are instantaneously "beamed" from one scene to the other without experiencing the distance in between. With this paradigm, the spaces are not associated, rather they are completely independent with no sense of attachment to each other. In the virtual world we wish to create, there is a sense of unity since the spaces all have enforced physical relationships to each other. So, I could walk out of my front door and be on someone else's mountaintop for example, but I would be able to turn around and see my house. Note that this does not preclude the use of hyperlinks to jump to other parts of the world—we have just added the relationship between spaces which was previously lacking.

In order to get the most out of these example applications, some browser modifications are necessary (although they do work with stock browsers). While our original application was based on the Inventor `SceneViewer` application, we have begun to migrate the code into the public-domain SDSC browser so that people can download our modifications and try them.

4.1 Networked IVL

Our first example is a scene-construction application which uses an HTML form to pass IVL commands to a server. The server constructs the scene and returns it to the client as a VRML file. This application can be used as a VRML compliance tester (returns an error if the instanced scene doesn't comply with VRML 1.0 specifications), as a world-builder (for those without a scene assembler), and a format converter (since IVL will accept not only VRML but Inventor and DXF and several other formats as input scenes, but will only output VRML).

As described, this works with all browsers, but is suboptimal since all commands are sent, and the entire scene reloaded from the server each time the form is submitted. Browser modifications are necessary if one wants to execute just one IVL command at a time (after all, this is the advantage of an interpreted language) and if one wants just incremental updates to the scene. With the current protocols, the entire scene is downloaded from the server to the client upon each modification. We have been experimenting with the Netscape server push feature in order to send out just those parts of the scene graph which need to be modified. Since IVL maintains an internal naming scheme which is shared via the VRML file, the browser client then simply executes a `replaceChild()`, an `addChild()`, or a `deleteChild()` method on the scene fragment once it has been received.

4.2 A Metaverse

In order to accomplish our goal of a metaverse with the current tools and protocols, we chose to use a client/server model. All information about the world, including hyperlinks, is stored on one server. Clients request the information, which is transmitted using HTTP and displayed. The client keeps track of the current viewpoint and generates commands in the IVL language which tell the server which parts of the world to transmit. Navigation and rendering is entirely up to the client—the server simply sends scene descriptions (including links and inline information).

If we concede that we are going to have problems with bandwidth and rendering speed, we must implement features in our world which act to minimize these problems, else the world will quickly become unusable. To this end, we *must* define levels of detail properly so that things don't show up if the viewer is too far away (map will give an overview, so nothing will be missed) and will show up as low detail until the viewer is close. The metaverse application by default wraps all added nodes and URL's in an `LOD` node which does this. Texture maps are forbidden (stripped out automatically) except at highest level of detail.

Users access the metaverse home page to build permanent structures in the metaverse. A password protection scheme is used to ensure only the owner of a structure or a link can modify it. Bounding Box checks are performed to make sure structures do not intersect. Since the IVL parser for the metaverse is running continuously, it maintains its current state in file written via `SoIdleSensor` when CPU permits. This allows for recovery of the (dynamic) world should the server crash.

An icon, called an avatar, is used to represent the physical location of each client in the metaverse, and the server push feature is used to send current location to all other clients. Avatars are removed when the client exits or when a browser idle time limit is exceeded. Users are allowed to link their own avatar, or one is provided by default. We do not as yet provide for interactions between avatars or between avatars and scene objects (other that collision detection). Avatars simply share the same space at the same time.

4.2.1 Behaviors

Everything internal to the metaverse will be handled through Open Inventor code, so we can utilize Inventor engines to animate a Sun, Moon or any other constant-motion object. Actually, we can allow people to instance engines or any Inventor feature we want, as long as we can *deliver* VRML as output. Input is handled by piping VRML files through a script which changes the VRML scene into Inventor format and "flattens" the scene by recursively pulling down all referenced (`WWWInline`) scenes. The input is then read into the metaverse application using a standard `SoDB::read()` method. Output is piped through `vrmllint` before being returned to the client to ensure that it conforms to the current VRML specifications.

While the internals of the world are implemented in Inventor, the delivered scene descriptions adhere strictly to the current VRML standard.

Unlike many other multiplayer application, in this metaverse 'bots (automatons) are *encouraged.* Example IVL scripts to drive 'bots are be provided.

It is very important to our applications that we be able to uniquely identify each node. With avatars, for instance, we specify that articulations must have named `Transform` nodes with standard names. If there are two avatars in the world, each with a "Head", how can we distinguish between the two nodes named "Head" within our code? We would benefit greatly from modifications to the VRML standard which allowed us to reference these `DEF`-ed nodes within a URL. In the above example, we might want to refer to `Joe.Head` where `Head` would be a named node within `DEF Joe WWWInline{name "http://joe.com/Joe.wrl"}`. This would allow us to reference Joe's head independently of Bob's head for example. We do not think that this model of refering to nodes implies or rules out any particular implementation of behaviors that we have seen—it merely enhances the ability to manipulate VRML scenes in code.

4.2.2 Navigating the Virtual World

When navigating a metaverse, terrain following (collision detection with the "floor") is extremely important. Walking in a building for instance requires negotiating stairs, which can be difficult and unnatural with the current browser interfaces. Adding the constraint that the eyepoint is at a fixed height above any object makes going up or down stairs trivial, and requires only that there be something to walk *on*. Although having a hole in the floor through which one could "fall" down to a lower level would be allowed, we do not want this to happen unintentionally. Collision detection with "walls" and objects is also important. Navigating a building becomes much easier with this feature. Collision detection (performed with `SoRayPickAction`) is implemented as a modification to the browser, and is a switchable option.

While the action of the walk viewer is adequate for moving short distances within the metaverse, moving long distances requires a different paradigm. We introduce the concept of *stepping disks* for this purpose. The concept as we originally encountered it was in the novel *Ringworld*[10]. Briefly, stepping disks are hyperlinks which instantaneously transport the viewer to a new point within the world. These disks are physically represented as a square platform, three meters on a side. Just off each side is a one meter diameter disk with a hyperlink to the next set of disks in that direction. Stepping on the disk to the North for instance will transport you North to the platform in the center of the next set of disks. If these stepping disks are laid out in a regular grid, with "long distance" disks every tenth interval, navigation becomes quick and easy even in a large world.

This is one way that we can get away from the single server for the world geometry—we could impose a standard interface to worlds where they must use a stepping disk and conform to certain other rules. As long as movement is confined to the area surrounding the disks (and doesn't cross the region between two servers) we have the illusion of one large space.

Other aids to navigation include pre-defined view points for points of interest, a visitor center which serves as the default entry point, a directory service in visitor center (searchable HTML form), and a selectable image map in the visitor center with a miniature representation of the metaverse. Clicking on a point on this map transports the viewer to that location.

We also implement a hierarchical terrain in which only the local, viewable region is downloaded. Multiple levels of detail and inlines are defined in the terrain, and the browser is modified so that it only loads that section of the terrain which is visible. Additionally, the terrain is tiled so that a large world is possible with a relatively small VRML file. Stepping disks are defined as just one small VRML file, and included it in our hierarchical terrain so that the file is instanced once and only once on the client side.

4.2.3 Protocols

A primary concern for this design is whether the server can adequately handle the bandwidth once we include interactions between avatars. Indeed, our first preference for behaviors is not the client/server model, but a distributed model such as implemented by the Distributed Interactive Simulation (DIS) protocol[7]. We believe the DIS method of multicasting state information for each entity not only scales better with the number of entities (clients), but has the ability to be more general and allow a much greater degree of interaction and experimentation than the client/server model. The disadvantage is that behavior implementation is shifted to the clients, which are already burdened with a major rendering task. We feel however that the network is the current bottleneck to multi-user worlds and DIS uses the network more effectively. But in order to implement a world in which we and others could try out our ideas about behaviors, and in order to stimulate further discussion and work in this area, we decided to start with the easier client/server approach using HTTP as a protocol.

We have prototyped a DIS-subset multicast protocol to send IVL commands across the network. We intend to support both new protocol *and* HTTP, but we anticipate some functionality will only be available with the new protocol.

4.3 Future Work

Of course, in addition to linking everyone's VRML worlds together into one large apparent world, we think it would be interesting to duplicate functions found in the real world. But much remains to be done to accomplish the ideal of a metaverse. Some areas which we would like to address are:

- A "store", such as a furniture store, a hardware store, a car dealer, etc., containing models people can use to create or re-model their own space. This is a familiar and intuitive way to *organize* 3D information and models. If you need to find models of food, navigate to the local supermarket!
- Enforcement of collision detection, creation of "private" areas which are password protected?
- Sound, video and other types of data in the virtual world. IVL can also be used for all types of web data, each data type is just a separate object class. Much work needed here.
- Avatars. We would like to define standards for avatars, i.e. how to deal with articulated models in a real-time way. This is do-able with IVL, but requires a new protocol for good results.
- Post-it notes (HTML link with appearance of note) to allow leaving messages for and communicating with others.
- Many more—use your imagination.

5 Conclusions

We have implemented an interpretive language for the creation, manipulation, and linking of VRML scenes, and presented several example applications of this language. It is our hope that this effort will spur further development of VRML tools and applications and help to make the whole VRML effort a success.

For applications like the ones presented here, some extensions to the VRML specifications will be necessary. In particular, we mentioned multi-file `DEF/USE` as well as browser caching and client pull/server push features. To this end, we will distribute the browser source code to implement these features[8] and we will pursue these extensions in the public forums in the coming months.

References

[1] The VRML 1.0 language specification can be found at `http://vrml.wired.com/vrml.tech-/vrml10-3.html`.

[2] PCCTS is discussed in the usenet newsgroup `comp.compilers.tools.pccts`.

[3] WebView, a public-domain browser developed by the San Diego Supercomputer Center, can be obtained at `http://www.sdsc.edu/EnablingTech-/Visualization/vrml/webview.html`.

[4] PCCTS source code and documentation is available from `ftp://ftp.parr-research.com/` in the directory `/pub/pccts` and from a mirror site in the UK, `ftp://src.doc.ic.ac.uk/` in the directory `/computing/programming/languages/tools/pccts`.

[5] A prototype IVL grammar implementing the features described in this paper, and including a makefile and example syntax, can be obtained from `ftp://ftp.ocnus.com/pub/IVL`.

[6] We have set up a web page with a form interface to drive the IVL interpreter and return the VRML scene to you. This site is located at `http://www.ocnus.com/ivl.html`.

[7] A good introduction to DIS can be found at `ftp://www.cs.nps.navy.mil/` in the file `/pub/npsnet/DISIntro.ps`. This location also has much more information about DIS, including source code for implementing DIS in your own application.

[8] Source code modifications to the public domain SDSC VRML browser which allow it to interoperate with our metaverse can be obtained from `ftp://ftp.ocnus.com/pub/IVL`.

[9] S.C. Johnson. Yacc — Yet Another Compiler-Compiler. *Comp. Sci. Tech. Rep.*, 32, July 1975.

[10] Larry Niven. *Ringworld*. Ballantine Books, New York, 1970.

[11] Terence J. Parr and Russell W. Quong. Adding Semantic and Syntactic Predicates to $LL(k)$—*pred-$LL(k)$*. In *Proceedings of the International Conference on Compiler Construction; Edinburgh, Scotland*, April 1994.

[12] Terence J. Parr and Russell W. Quong. ANTLR: A Predicated–$LL(k)$ Parser Generator. *To appear in Journal of Software Practice and Experience*, 1994.

[13] T.J. Parr, H.G. Dietz, and W.E. Cohen. PCCTS 1.00: The Purdue Compiler Construction Tool Set. *SIGPLAN Notices*, 27(2):88–165, February 1992.

[14] Neal Stephenson. *Snow Crash*. Bantam Books, New York, 1992.

[15] Josie Wernecke. *The Inventor Mentor*. Addison-Wesley, 1993.

[16] Josie Wernecke. *The Inventor Toolmaker*. Addison-Wesley, 1994.