1st International Conference on Advanced Multimedia Content Processing

AMCP '98

Conference Proceedings

Osaka University Convention Center
Suita, Osaka, Japan
November 9-11, 1998

Editors
Shojiro Nishio and Fumio Kishino

InvenTcl: A Fast Prototyping Environment for 3D Graphics and Multimedia Applications

Sidney Fels[1] and Kenji Mase[2]

[1] University of British Columbia***, Vancouver, BC, V6T 1Z4, Canada,
ssfels@ece.ubc.ca, http://www.ece.ubc.ca/~fels,
+1 (604) 822-5338, fax: +1 (604) 822-5949
[2] ATR MI&C Research Laboratories, Seika-cho, Soraku-gun, Kyoto, 619-02, Japan,
mase@mic.atr.co.jp, http://www.mic.atr.co.jp/~mase,
+81 774 95 1440, fax: +81 774 95 1408

Abstract. This paper describes *InvenTcl* which is an interpretive version of Open Inventor, a 3D graphics toolkit. To create InvenTcl, the Open Inventor toolkit is "wrapped" inside the interpreter Tcl/Tk and [incr Tcl]. To wrap InvenTcl the Open Inventor header files are parsed to create [incr Tcl] interpretive objects with the same names as objects in Open Inventor. Additionally, window event management, non-objects and object bindings are included and managed by InvenTcl. The advantages of InvenTcl include: script-able and direct manipulation of 3D objects in an Open Inventor scene, easy prototyping of 3D graphics and animation, low bandwidth communication of 3D scenes and animations (using scripts), and easy integration of 3D graphics with other media for fast prototyping of multimedia applications.

1 Introduction

There have been many 3D graphics packages and libraries available such as PEXlib [8], OpenGL [4], Open Inventor [9], GKS-3D [13], PHIGS [12]; however, they are usually precompiled toolkits, and thus not well-suited for fast prototyping new ideas, rapid experimentation with 3D scenes, or easy extension and integration with user-defined code. In addition, they often lack the combination of two useful modes of interaction; direct (mouse-click) mode and command line or script-based mode. Open Inventor falls into this category of 3D graphics packages.

InvenTcl extends Tcl/Tk/[incr Tcl] by providing interpretive access to Open Inventor. InvenTcl provides a window for creating, displaying, animating, and interacting with 3D objects. This is achieved by combining the Open Inventor C++ library [21, 9] with the Tcl/Tk library [18, 20] and the [incr Tcl] libraries [14]. By embedding the library of Open Inventor in Tcl/Tk, a high-level 3D scene interface is created with which objects can be manipulated on the fly via device

*** Sidney Fels was a visiting researcher at ATR MI&C Research Laboratories when this research was done.

interaction as well as command line interaction. As a result, we accomplish both, 3D scene access to Tcl and interpretive access to Inventor. Prototype applications can be programmed entirely in InvenTcl; using Tk for the traditional 2D interface widgets (buttons, text widgets, pull down menus, etc.) and InvenTcl calls to Open Inventor for creating and displaying 3D scenes and implementing direct manipulation within the scene. Further, embedding other media toolkits into Tcl/Tk provides similar access flexibility and integration.

Open Inventor provides an objected oriented view of 3D graphics. The objected oriented view of 3D graphics programming is well suited to an interpreted access model. For this reason, Open Inventor was chosen over a function based model such as OpenGL for the basis of InvenTcl.

The simplicity of using InvenTcl makes it very suitable for both novice and expert users of 3D graphics toolkits and C++ programming. The potential uses for InvenTcl include: multimedia prototyping, VR prototyping, 3D graphics education, 3D GUI prototyping and scientific visualization. In section 3, we describe our use of InvenTcl to create a VR prototype of an architectural walkthrough of our laboratory. We also have used InvenTcl for teaching concepts of 3D graphics. Students without formal 3D graphics training have been able to learn to make relatively complex 3D scenes within a few hours, including, animating a sheet and exploring animating a walking robot. In another project, we were able to link a musical research project [15] with InvenTcl to display 3D representations of gesture space. Linking the two systems, developing suitable 3D graphics, and getting useful results took less than one hour (with no recompiling necessary). The flexibility of Tcl as a "glue" language makes InvenTcl a powerful toolkit for the scientist wanting to connect research code to InvenTcl for 3D visualization of their results.

Several people have created other 3D graphics extension to Tk [19, 11]. These applications are using the low-level OpenGL [17] library. Other researchers have created interpretive 3D toolkits such as Alice [5] and Obliq-3D[16]. These approaches are similar to InvenTcl, however, they use either their own 3D graphics toolkits or interpreter rather than a pre-existing one. InvenTcl is the combination of two popular systems, thus leveraging the work (and support) of them. Another approach is taken in SWIG [3] where one can wrap additional C-code around an existing C/C++ function to create a new Tcl-command. Applying this program to the Open Inventor library would result in a globalisation of all methods of each class in the object oriented library, which would be problematic. Our approach keeps the object oriented feature of Open Inventor, but also tailors some higher level commands instead of using the basic Inventor methods. Interestingly, if SWIG is used to generate Python commands the object oriented structure should be maintained since SWIG supports object oriented code for Python. An early version of InvenTcl can be found in [6]. Compared to using VRML, InvenTcl has the advantage of using Tcl/Tk as the master shell, making it easy to bind in new applications without recompilation.

Our immediate goal with InvenTcl is two-fold; one, provide an interpretive version of Open Inventor, and two, develop a complete 3D Tk canvas widget

version which will behave in a similar fashion to the current 2D canvas widget. The interpretive version of Open Inventor is geared to the novice and expert Open Inventor programmer and has the following advantages over directly using C++ linked with Open Inventor's libraries:

- script-able and direct manipulation of objects in a scene
- easy prototyping of 3D graphics and animation;
- easy prototyping of GUIs for interacting with 3D scenes
- low bandwidth communication of 3D scenes and animations (using scripts).
- easy integration of 3D graphics with other software

A 3D canvas widget can be built on-top of InvenTcl for the Tk programmer to shield them from the details of Open Inventor and make access have the look and feel of Tk.

The power and flexibility of InvenTcl is a function of embedding a C++ based 3D graphics toolkit inside an extensible interpreter. InvenTcl provides a strong demonstration of the fruitfulness of this direction of research and development. This interpreter philosophy used for InvenTcl should be applied to other toolkits to allow both compiled and interpreted modes to be available for the developer. Further, the interpreter should be an extensible one so that users can integrate 3D graphics (and other toolkits, such as MET++[1]) into their own applications easily. Wrapped in this way provides highly interactive, easy to use, and extensible toolkits.

This paper describes how we converted Open Inventor and some of the key points we addressed. In all likelihood, for future conversion of graphics toolkits some of the same issues will be important, thus, in addition to describing a useful tool, this paper can also be used as a reference for wrapping other 3D graphics or multimedia toolkits.

The first section of this paper describes how the Open Inventor libraries were wrapped. A simple example is provided to show how InvenTcl is used. The second section of this paper discusses an example application created with InvenTcl. The application is an architectural walkthough (and walkthrough builder) connected to a person tracker system and database system. While the application may not be particularly interesting on its own, what is significant for this paper is that the application was written in 6 days by one programmer with only medium expertise in 3D graphics. The entire application was written in InvenTcl, thus, required no C++ code or compilation. Finally, the future directions of InvenTcl and some conclusions are made.

2 Making Open Inventor Interpretive

To create InvenTcl, the Open Inventor libraries need to be wrapped so that all the objects, their methods and public fields are accessible from the Tcl shell. Further, to make the interpreter version appear similar to the C++ version the object hierarchy and naming convention needs to be maintained. To integrate with the Tcl/Tk environment the 3D objects should be "bindable". That is,

objects in the 3D scene can be bound to Tcl scripts which execute according to some user interaction such as a mouse button press. Finally, any non-object parameters accepted by the Open Inventor library functions need to have a representation in [incr Tcl] so that they can be passed as arguments. In summary, the five main requirements are:

1. Convert all Open Inventor classes to [incr Tcl] classes, including: methods, public fields, and static functions.
2. Integrate Open Inventor's event management loop into Tcl/Tk's event management loop.
3. Add support for 3D object binding so that interaction events, such as mouse and keyboard events, will call Tcl scripts.
4. Convert any non-object arguments, such as, enum types, arrays, and FILE pointers, accepted by Open Inventor object methods to [incr Tcl] objects to allow for run-time checking and value passing.

Most of the work to wrap Open Inventor is done automatically by parsing the header files of Open Inventor. The parser automatically converts all the classes and methods. Some parts are converted by hand, including some of the non-objects and event manager. A block diagram representing the necessary parts to create InvenTcl is shown in figure 1.

2.1 Converting Open Inventor Classes to [incr Tcl]

A program called Itcl++ [10] was used as the starting point for converting the Open Inventor class structure into [incr Tcl] classes. This program parses the header files of the class libraries and creates [incr Tcl] class structures. The [incr Tcl] class structure created provides the ability to instantiate objects and call methods of each object from the interpreter. The class inheritance structure is maintained in the [incr Tcl] class structure. Methods use run time checking of argument types. We have enhanced Itcl++ to also provide access to objects' public fields. When an object is created in [incr Tcl] the [incr Tcl] class constructor calls some C++ code which actually calls the *new* C++ operator to instantiate the Open Inventor object. The instantiation in C++ returns a pointer. This pointer is associated with the [incr Tcl] name assigned to the object created in [incr Tcl]. Thus, an intuitive way to think about the relationship between the interpreter and Open Inventor representations of objects is that InvenTcl provides string names as pointers to objects and Open Inventor uses integer pointers to objects. InvenTcl maintains the relationship between the two.

Overloaded methods Many methods in Open Inventor are overloaded. In InvenTcl we use run-time type checking of arguments to perform the necessary casting so that the appropriate method signature is used. Due to the interpretive nature of InvenTcl, type checking must be performed at run time to correctly implement the overloaded methods. In contrast, the original scheme used by Itcl++ used different method names to deal with method overloading. In this

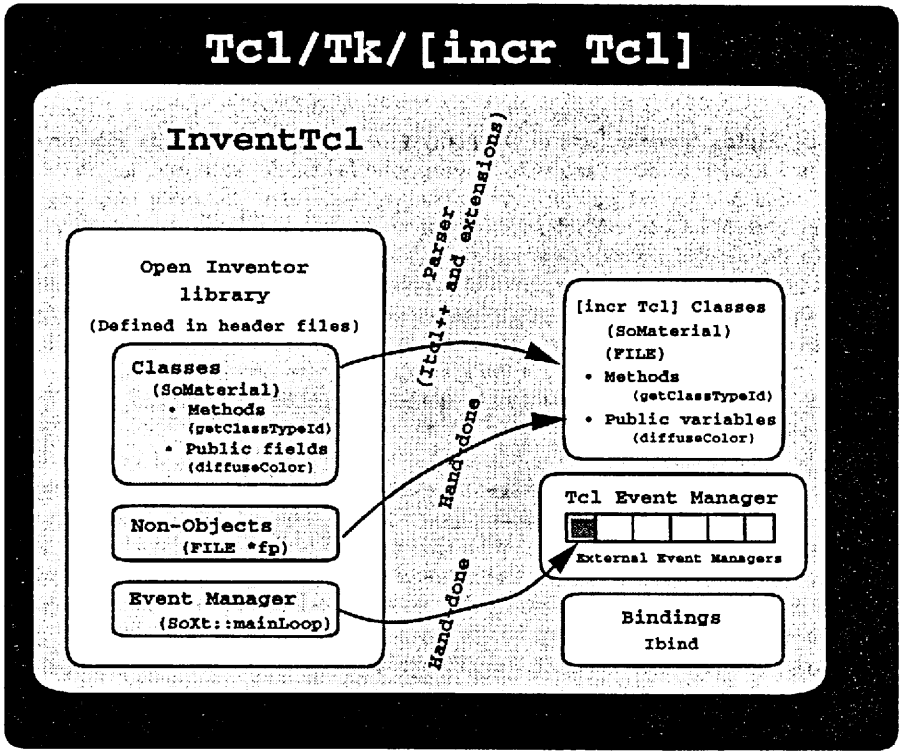

Fig. 1. Block diagram showing relationship of InvenTcl to Open Inventor and Tcl/Tk/[incr Tcl] and how InvenTcl is created from Open Inventor header files. InvenTcl is the set of interpreted classes connecting Open Inventor classes and [incr Tcl], the event manager embedding inside of Tcl/Tk and the Open Inventor object to Tcl binding mechanism.

scheme, for each argument signature a unique method name was adopted. The method name was the name of the overloaded method with a version number appended to it. For example, if a method **setValue** is overloaded with an **int** or a **float** argument, two methods are created in the interpreter called **setValue1** taking an **int** argument and **setValue2** taking a **float** argument. We found this scheme very difficult to work with from a user's point of view. It was often the case that the user could not remember which version of the method to use and had to always refer to the online help to figure it out. By performing run-time checking we eliminated this problem so that, for example, only one method called **setValue** was created.

Public Fields Some objects in Open Inventor provide public fields. We have modified Itcl++ to allow access to these public fields. For example, an *SoMaterial* node has several public fields, including *ambientColor* which is of type *SoMFColor*. When an *SoMaterial* node is created in C++ the *ambientColor* field is also created and can be accessed with a pointer reference. We provide access to this field from the interpreter by creating an object for the *ambientColor* field and associating it with a public variable in the [incr Tcl] object. Thus, the public variable in the *SoMaterial* [incr Tcl] object contains the name of the instantiated *ambientColor* [incr Tcl] object, and thus, can access the field. Access to the actual values maintained by the field are available through the methods provided.

The only modification to the parser that was needed was to make sure that for each public field of an instantiated object a call to the [incr Tcl] functions to create an [incr Tcl] object was made. The creation of the [incr Tcl] object assigns a name and associates the pointer with that name. Public fields play the role of members variables in an object. Open Inventor is consistent in only providing fields instead of simple member variables. As discussed in [21], one reason Open Inventor was designed this way was to provide consistent methods for setting and getting values. This design feature is extremely valuable when making Open Inventor interpretive.

Static Member Functions Finally, there are some public static functions associated with Open Inventor classes which have associated public procedures in the [incr Tcl] classes. For example, the *SoXt* class has a static function *init*. The static functions are provided for in the [incr Tcl] objects.

2.2 Integrating Open Inventor's Event Manager

A typical Open Inventor program performs some initialization, creates the scene graph and sets up any user interaction mechanisms before entering an infinite event management loop using the *SoXt::mainLoop* function. The event management loop loops forever handling any Open Inventor events which need to be serviced. This structure is modified in InvenTcl since the interpreter must also check for Tcl/Tk events too.

To solve this problem we embedded Open Inventor's event management inside of Tcl/Tk's event manager. Thus, when an Open Inventor event is found by Tcl/Tk in the main event queue, Open Inventor is called to manage the single event and return control to Tcl/Tk's event manager. In this way, events for Tcl/Tk and Open Inventor are handled appropriately.

From the user's point of view, nothing has changed. That is, in InvenTcl when they want to start Open Inventor's event handling they issue the *SoXt::mainLoop* command. But, rather than starting the infinite loop, InvenTcl installs the embedded version of the event handler and returns control to the interpreter. This is illustrated in the example below in section 2.5.

2.3 3D Object Binding

One of the powerful user interaction capabilities offered in Tcl/Tk is the ability to dynamically bind Tcl scripts to Tk widgets[1] which are triggered by events (which typically come from the user). We have implemented the same binding mechanism in InvenTcl for 3D objects. That is, in InvenTcl the user can bind Tcl scripts to 3D objects which trigger on user events such as mouse button clicks, mouse movements or keyboard presses.

The function created for performing the binding is called *Ibind*. The Ibind command expects four arguments: name of the object being bound, the name of the head of the scene graph which contains the object, the user event to watch for and finally the Tcl script to execute. For example, if we have an object called *cone* which is in the scene graph with *root* as the top we can do the following:

```
Ibind $cone $root <Button1> {puts "InvenTcl is great!"}
```

This command will make it so that when the user presses mouse button 1 the string "InvenTcl is great!" is printed.

To add this functionality, we add a generic call back node to the top of the scene graph that is triggered on any user generated event. This callback node retrieves the user event and the path where the event occurred and checks a list of all bindings to see if any of them should trigger given this information. The Ibind command also allows the current mouse position and the name of the [incr Tcl] object where the event occurred to be passed to the Tcl script. When the *Ibind* command is executed the appropriate binding is added to the list.

This binding mechanism is extremely useful. Prototyping different user interfaces can be achieved very quickly since developers can immediately try different types of user interactions. Remember, the Tcl scripts can be *any* Tcl scripts, thus, one can control: 3D graphics, all the Tk widgets, the Tcl interpreter or any "glued" in application such as video, audio or text toolkits. This mechanisms for implementing this event based binding provides the necessary infrastructure for developing the high-level synchronization mechanisms needed in a multimedia toolkit, such as found in [1].

[1] Tk widgets are graphical elements such as buttons, sliders, canvases, etc. which are used for creating GUIs

2.4 Converting Non-object Structures

Open Inventor is mostly consistent in its objected oriented approach, however, there are a number of non-object structures which can be used in Open Inventor. Generally speaking, these structures are used as input parameters to some methods. The main structures are: enumerated types, arrays (1D, 2D, 3D and nD), FILE pointers, and function pointers. For each of these some appropriate [incr Tcl] object (or access method) was created[2].

Enumerated Types For enumerated types, we provide a static procedure of the same name to access the value. For example, the class *SoNormalBinding* has an enumerated type called PER_FACE. In C++ this is accessed using

```
x = SoNormalBinding::PER_FACE;
```

where as, in InvenTcl there is a static procedure called PER_FACE which returns the enum value. The interpreter version of the above C++ code is:

```
set x [SoNormalBinding::PER_FACE]
```

Arrays For arrays, the original version of Itcl++ provides a 1D array object with values of basic types: **int**, **float**, **char**, and *SbBool* and methods to set and get their values. The types can be short or long, signed or unsigned where appropriate. We have extended this support for InvenTcl to include 2D, 3D and 4D arrays. Currently, we are still implementing support for arrays of complex object types as well as n-dimensional arrays.

File Pointers We have created a *FILE* [incr Tcl] class which associates an [incr Tcl] name with a file pointer. An object of this class can be used as the input argument for methods which require a file pointer argument. During initialization, we create FILE objects for the standard I/O file pointers: stdin, stdout and stderr. These *FILE* objects are referenced by the Tcl variables **stdin**, **stdout** and **stderr**.

The next section shows a simple example which demonstrates how InvenTcl works.

2.5 A Simple InvenTcl Example

This example shows some of the most basic features of InvenTcl. The example covers the main aspects necessary to create 3D graphics and setting up user interaction from Tcl to Open Inventor and from Open Inventor to Tcl using InvenTcl.

The example consists of creating an active 3D cone as shown in figure 2. If the user clicks button 1 on the mouse the message "running a Tcl script" appears. Also, a simple GUI is created with Tk for changing the colour of the cone. Remember, all this code is typed in directly (or sourced from a file) within the Tcl shell. The full source code is shown in figure 3.

[2] Function pointers have not been implemented yet.

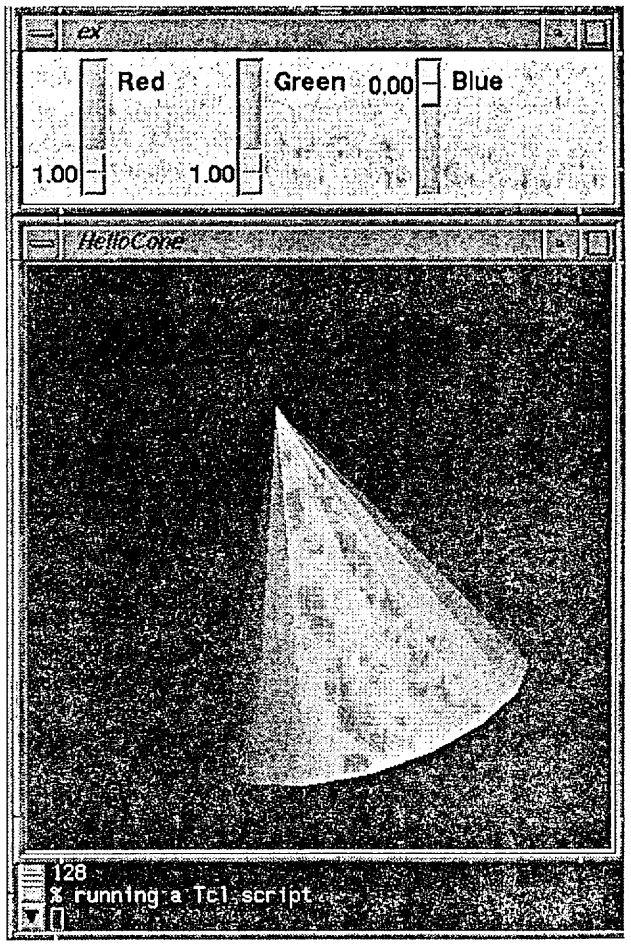

Fig. 2. 3D cone example with Tk sliders to manipulate colours. The top widget shows the GUI for manipulating the colours of the cone. The middle window shows the cone and the bottom part of the figure shows the Tcl shell. Notice, that we are viewing this right after the user clicked mouse button 1 on the cone.

```
1. set window [SoXt::init "HelloCone" "HelloCone"]

2. set root [SoSeparator::Constructor]
   $root ref

   # make camera and light to see the cone
   set camera [SoOrthographicCamera::Constructor]
   $root addChild $camera
   $root addChild [SoDirectionalLight::Constructor]

3. set coneMaterial [SoMaterial::Constructor]
   $root addChild $coneMaterial

4. set cone [SoCone::Constructor]
   $root addChild $cone

   set ra [SoXtRenderArea::Constructor $window]
   $ra setTitle "HelloCone"
   $camera viewAll $root [$ra getViewportRegion] 1
   $ra setSceneGraph $root
   $ra show
   SoXt::show $window

5. SoXt::mainLoop

6. set coneColour [lindex [$coneMaterial configure -diffuseColor] 2]
   set red 0.5
   set green 0.5
   set blue 0.5

   # procedure to connect cone colour with sliders
   proc changeColor {val} {
      global coneColour red green blue
      $coneColour setValue $red $green $blue
   }

7. toplevel .ex
   scale .ex.r -from 0 -to 1 -resolution 0.01 -label R -variable red \
       -command changeColor
   scale .ex.g -from 0 -to 1 -resolution 0.01 -label G -variable green \
       -command changeColor
   scale .ex.b -from 0 -to 1 -resolution 0.01 -label B -variable blue \
       -command changeColor
   pack .ex.r .ex.g .ex.b -side left

8. Ibind $root $cone <1> {puts "running a Tcl script"}
```

Fig. 3. Code to draw a cone with Tk sliders to adjust its colour. The binding is set so that if button 1 is pressed when the cursor is on the cone the message "running a Tcl script" is displayed.

Referring to numbered parts in figure 3, here is an explanation of some of the important features of the example. The numbered parts are explained below.

1. This piece of code calls the static procedure to initialize the scene data base.
2. This piece of code creates a root node to head the scene graph.
3. This piece of code creates a material node to control the material properties of the cone. This node is used below to create a GUI to allow the user to dynamically change the properties of the cone.
4. This code creates the cone and puts it in the scene.
5. This code starts the Open Inventor event manager. The embedding of Open Inventor's event manager inside of Tcl/Tk's event manager is discussed in section 2.2. The critical point here is that control is returned to the Tcl interpreter so that further manipulation of the scene graph can occur while the user sees the current scene graph.
6. This set of code defines a procedure which manipulates the material node in the scene graph. The material node's colours are changed by setting the *coneMaterial*'s public field *diffuseColor* values to the current values of the global variables: red, green and blue. The red, green and blue values are manipulated by the Tk sliders and the changeColor procedure is called whenever a change in one of the values occurs.
7. This section of the code creates the Tk sliders and connects changes to the sliders to the *changeColor* procedure and the values of the red, green and blue variables.
8. Finally, this piece of code shows the Ibind command being used. The command makes it so that when the user clicks mouse button 1 and the cursor is on the cone the Tcl script, {puts "running a Tcl script"} is run which prints the message. The result of the click can be seen at the very bottom of figure 2. This piece of code demonstrates how Open Inventor objects can be connected to arbitrary Tcl scripts. The scripts could execute more graphics commands, Tcl commands, Tk commands, operating system commands or any application code that has a Tcl front end.

This simple example demonstrates some of the main features of InvenTcl. However, what may not be obvious from the example is the high level of flexibility that InvenTcl provides. InvenTcl code can be modified *on-the-fly* to suit the developer's needs and can be integrated with other applications easily. The next section briefly describes an example application created completely using InvenTcl which takes advantage of many of the features of InvenTcl.

3 Example of using InvenTcl for Prototype Walkthrough

To illustrate the power and ease of using InvenTcl we created an example application for demonstration during our laboratory's open house. The application we created was an architectural walkthrough and walkthrough builder. The walkthrough was also connected to an active badge system from Olivetti. The active badge system tracked visitors to our laboratory using infrared badges and badge

sensors. The walkthrough would show the current location of all the visitors in the laboratory. The laboratory set up was not going to completed until the night before the actual open house so the map of the floor would not be available until just before the open house started. For this reason, the walkthrough had to have a walkthrough builder which would allow the developer to change the layout of the walls very quickly.

The entire system was programmed in six days by one programmer with only medium expertise in graphics programming and no experience with walkthroughs. The programmer did not use any C++ code or compilation[3]. The entire application is a set of Tcl files which are sourced to run the application.

The system consisted of two main parts, a 2D map layout and editor and a 3D view of the floor. The 2D map editor was built using Tk widgets and is shown in figure 4. The 3D view used the InvenTcl objects and is shown in figure 5. When a line was drawn on the 2D map a corresponding wall was displayed in the 3D window. The wall in the 3D window was active so that the developer could click on it to popup a Tk based wall editor. This editor allowed the developer to specify properties of the wall including colour and texture. The 2D and 3D views of the floor were always kept in sync by using the binding mechanisms in Tk and InvenTcl.

The 2D map editor allowed the system to be reconfigured easily to suit the dynamic environment of the preparations for our open house. On the night before open house, when the final layout of the floor of the laboratory was finalized, the final map was drawn in, photographs taken of the walls (for texture mapping) and the complete walkthrough was ready for the next morning.

The other aspect of the system that was required was to include a representation of all the visitors in the lab detected by the active badge system. The system used both a 2D and 3D representation of each visitor. The 3D representation was a doll whose position could be dictated by the active badge system. One doll is shown in figure 5. The position of the visitor was determined by the active badge system. Position information was exchanged as Tcl scripts using a client/server relationship with a Tcl based position server. By making InvenTcl a Tcl server any application can connect to it and send InvenTcl scripts to control graphics. In this way, it is easy to connect Tcl extended application running on a different machine to InvenTcl to take advantage of the 3D graphics capabilities.

A late breaking requirement was that each visitor was going to be assigned a cartoon character representation as a representation of their own personal tour guide agent. This was done as part of the C-Map project [7]. Each visitor's 3D doll was required to display the character on its face. The assignment of the character was done at a registration desk. This information was communicated using the same Tcl client/server mechanism used with the active badge system; that is, once the assignment is made, a Tcl command is sent from the registration server (written in Java) to the Tcl client which then updates the 3D graphics.

[3] Of course, an application developer can use C++ in combination with InvenTcl for creating applications.

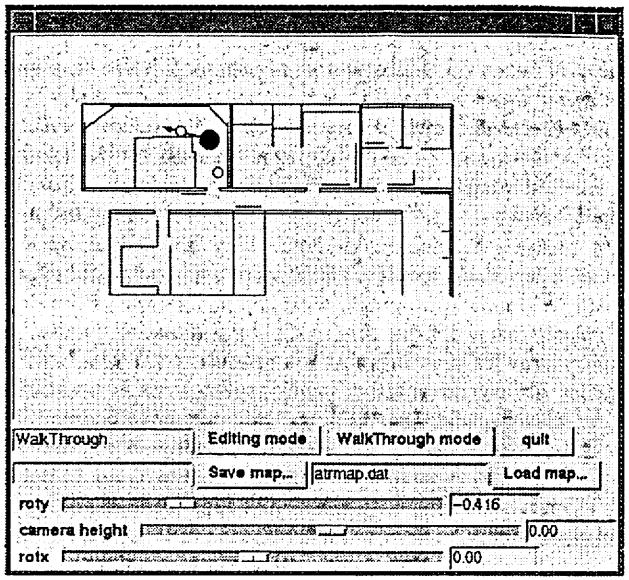

Fig. 4. 2D GUI for laying out, editing, saving and reading maps. There are also controls on the GUI for manipulating the viewpoint of the scene. The large red dot is the current viewing position and the arrow indicates the direction of view. The small yellow dots indicate visitors in the lab.

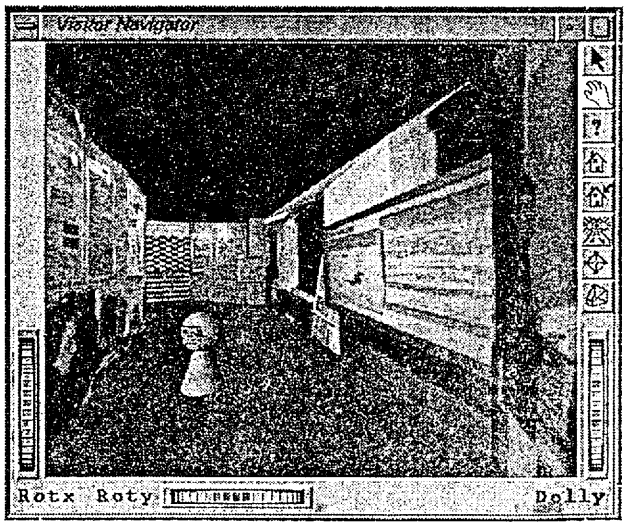

Fig. 5. 3D Open Inventor window displaying the lab floor.

Because of the flexibility and ease of modifying InvenTcl code this change was integrated in less than 2 hours.

One aspect of InvenTcl which was very useful for this application is the ease which different user interaction techniques can be tested. The developer of the system was not too familiar with walkthrough environments. Thus, he was unsure of how to allow a user to move through the 3D floor space. He was able to test out many different techniques of moving the user's viewpoint around using 1D sliders and 2D widgets. He was able to experiment "on-the-fly" with new ideas as he discovered them. This fast testing cycle led to him exploring many different alternatives. This exploration potential would be nearly impossible using a standard program/compile/debug cycle of development.

The conclusion reached from this example application is that InvenTcl is a powerful paradigm for creating and manipulating 3D graphics. By making the 3D graphics library interpretive development time is drastically reduced in creating complex worlds. The binding mechanisms provided by Tk for 2D widgets and the one provided by InvenTcl for 3D objects make creating GUIs simple and quick. These properties allow many different GUIs to be explored for determining which is best. This feature becomes even more important as different media toolkits are embedded since the interaction paradigms are less understood and GUI experimentation becomes critical (in contrast to the approach in [2]).

4 Future Work

There are two main areas for the future of InvenTcl. The first deals with improvements to the current system. The second area deals with expanding the concept of InvenTcl to include other communities.

With the current version of InvenTcl, a large portion of the Open Inventor libraries is accessible from the interpreter. The main areas of improvement are:

- implementing function and procedure callbacks,
- integrating Open Inventor's draggers and manipulators,
- using other interpreters such as Python[4],
- improving speed, and
- integrating Tk widgets and the Open Inventor window so they appear in one window.

To improve speed we have implemented an interactive mode switch which allows users to turn the interpreter on and off on demand to allow the Open Inventor event manager to run at full speed. However, when the switch is on the Tcl event manager is off, thus, preventing access to the interpreter.

There are other 3D toolkits available. It is hoped that this paper motivates 3D graphics developers to see the merits of having an interpretive version of other toolkits and create them. Further, the interpreter used should be extensible and matched to the structure in the toolkit, i.e., if the toolkit is object oriented the

[4] Python web address: `http://www.python.org/`

interpreter should support object oriented code. Such efforts[5] have been pursued, in particular, World Tool Kit (WTK) by Sense8 has been "wrapped" in Python. In extending InvenTcl, we plan to wrap Cosmo3D[6].

The approach of wrapping toolkits is general in nature. For example, by wrapping a toolkit such as MET++, a scriptable interface is achieved. Further, as Tk provides a GUI builder, various multimedia interaction techniques can be built and experimented with, including a visual programming environment [2]. The versatility of InvenTcl demonstrates the advantages of embedding toolkits inside of interpreters. Multimedia researchers and developers are in an excellent position to take advantage of the merits of extensible interpreters.

5 Conclusions

We had two main goals when creating InvenTcl; one, provide an interpretive version of Open Inventor, and two, develop a complete 3D Tk canvas widget version which will behave in a similar fashion to the current 2D canvas widget. We have achieved the first goal. This paper has discussed the implementation of the interpretive version of Open Inventor. Most of the objects and methods available in the Open Inventor library have been wrapped in Tcl/[incr Tcl]. Additionally, mechanisms have been created to allow 2D GUIs to directly control the 3D environment and for 3D user interaction in the Open Inventor window to call back to the Tcl interpreter.

InvenTcl leverages all the advantages of interpretive languages and brings them to bear on the Open Inventor toolkit. Thus, using InvenTcl it is possible to have:

- script-able and direct manipulation of objects in a scene
- easy prototyping of 3D graphics and animation;
- easy prototyping of GUIs for interacting with 3D scenes
- low bandwidth communication of 3D scenes and animations (using scripts) and
- easy integration of 3D graphics with other applications and toolkits.

The current version of InvenTcl[7] is available.

6 Acknowledgements

We are grateful for the contributions made by Kazuhiro Kawagoe, Tameyuki Etani, Silvio Esser, Armin Bruderlin, and Ryohei Nakatsu for their assistance with creating InvenTcl. We also thank other members of the C-MAP team, Yasuyuki Sumi, Nicolas Simonet, and Kaoru Kobayashi.

[5] PyWTK: `http://www.mic.atr.co.jp/~gulliver/PyWTK/www/`
[6] Cosmo3D: `http://www.sgi.com/Products/cosmo/cosmo3D/`
[7] InvenTcl: `http://www.mic.atr.co.jp/organization/dept2/inventcl/`

References

1. Phillipp Ackermann. *Developing Object-Oriented Multimedia Software - Based on the MET++ Application Framework.* dpunkt Verlag, 1996.
2. Phillipp Ackermann, Dominik Eichelberg, and Bernhard Wagner. Visual programming in an object-oriented framework. In *Proceedings of Swiss Computer Science Conference*, Zurich, Switzerland, Oct. 1996.
3. D. M. Beazley. Swig: An easy to use tool for integrating scripting languages with C and C++. In *Proceedings of Tcl/Tk Workshop, Monterey, CA*, July 6-10, 1996.
4. OpenGL Architecture Review Board. *OpenGL Reference Manual.* Addison-Wesley, 1992.
5. Randy Pausch et al. Alice: A Rapid Prototyping System for 3D Graphics. *IEEE CG&A*, 15(3):8–11, May 1995.
6. S. S. Fels, A. Bruderlin, S. Esser, and K. Mase. Inventcl: Making open inventor interpretive with tcl/[incr tcl]. In *Visual Proceedings of SIGGRAPH'97*, page p. 91, Aug 1997.
7. S. S. Fels, Y. Sumi, T. Etani, N. Simonet, K. Kobayashi, and K. Mase. Progress of c-map: a context-aware mobile assistant. In *Proceedings of the AAAI Spring Symposium on Intelligent Environments*, pages pp. 60–67, Mar 1998.
8. T. Gaskins. *PEXlib Programming Manual.* O'Reilly & Associates, Inc., 1992.
9. Open Inventor Architecture Group. *The Inventor Reference Manual.* Addison-Wesley, New York, 1994.
10. W. Heidrich and P. Slusallek. Automatic generation of Tcl bindings for C and C++ libraries. In *Proc. of the Tcl/Tk Workshop*, July 1995.
11. I. Hsu. Tksm a mesa/opengl 3d modeling widget extension for tcl 7.[45]/tk. In *http://www.isr.umd.edu/%7Eihsu/tksm.html.*
12. American National Standards Institute. *American National Standard for Information Processing Systems - Programmer's Hierarchical Interactive Graphical System (PHIGS) Functional Description, Archive File Format, Clear-Text Encoding of Archive File, X3.144-1988.* ANSI, New York, NY, 1988.
13. American National Standards Institute. *International Standard Information Processing Systems - Computer Graphics - Graphical Kernel System for Three Dimensions (GKS-3D) Functional Description, ISO 8805:1988(E).* ANSI, New York, NY, 1988.
14. M. McLennan. [incr Tcl]: Object-oriented programming in Tcl. In *Proc. 1st Tcl/Tk Workshop*, University of Berkeley, CA, USA, 1993.
15. A. Mulder, S. S. Fels, and K. Mase. Empty-handed gesture analysis in Max/FTS. In *Proceedings of Kansei - The Technology of Emotion, AIMI International Workshop*, pages pp. 87–91, Oct 1997.
16. Marc A. Najork and Marc Brown. Obliq-3D: A high-level, fast-turnaround 3D animation system. *IEEE Trans. on Visualization and Computer Graphics*, pages 175–193, June 1995.
17. J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide.* Addison-Wesley, New York, 1993.
18. J. K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, New York, 1994.
19. B. Paul. Togl: Togl allows opengl or mesa to render graphics into a special tk canvas. In *http://www.ssec.wisc.edu/%7Ebrianp/Togl.html.*
20. B. B. Welsh. *Practical Progamming in Tcl and Tk.* Prentice Hall, New Jersey, 1995.
21. J. Wernecke. *The Inventor Mentor.* Addison-Wesley, New York, 1994.