References


An instruction to create the geometry of a new book can now be sent,

\[
\text{BkSlf1[1][1].Create( FreeBookIndex );}
\]

and the entries are mapped into the new book, using the new book’s Insert methods:

\[
\begin{align*}
\text{BkSlf1[1][1][FreeBookIndex].Insert( AuthorName, ‘‘Author’’ );} \\
\text{BkSlf1[1][1][FreeBookIndex].Insert( Address, ‘‘Address’’ );} \\
\text{BkSlf1[1][1][FreeBookIndex].Insert( AuthorPicture, ‘‘AutPic’’ );} \\
\text{BkSlf1[1][1][FreeBookIndex].Insert( Content, ‘‘Content’’ );}
\end{align*}
\]

6 Conclusions

We have considered the problem of productively displaying multitude of data originating at various sources: files, databases, the WWW and others. This multitude of data flows in and need be visually scanned and understood. This is done in a virtual environment consisting of familiar objects: bookshelves, tables, phones etc. One task to perform is the mapping of the data onto an internal scene database, this is done using a specialized mapping language. The scene database itself need be maintained over time. Its contents are visualized in a virtual environment. Each database object is associated with a geometry that is defined using a script language. This geometry, super-imposed with data (such as text, sound, image, and video) is then displayed on a physical device (head-mounted display or a computer terminal) subject to various placement constraints. The system offers interactions with displayed objects using a six degrees of freedom mouse and other I/O devices (e.g. a glove). The same scene may be concurrently fed from various concurrently executing mapping language programs as well as users’ interactions.

The abilities to incrementally and intelligently map data into, and manipulate data from within, a virtual environment are the essence of this work. The ability to efficiently handle distributed geometry, by pre-distributing a collection of parametrized objects is another key feature. We hope that we successfully demonstrated the capability of mapping data of text/pictures/movies/tables to geometry and that we successfully hinted on its potential. We are certain that the level of interaction that can be expected in such an environment will greatly increase the productivity of many data processing application.
(b) Scene Display. This module is responsible for the efficient display of the currently visible scene. It constructs a synthetic image of the scene as seen from the current viewing direction and position. The user can, interactively, affect the scene.

5 Example

Consider the following sequence that maps a data table of scientific papers in Computer Science on top of books in the virtual environment. Each paper has associated with it the following attributes: 

- AuthorName, Address, YearOfPublication, AuthorPicture, and the paper’s Content.

The following command creates a bookshelf with two selves in one column:

```
BkSlf1 = BookShelf( 1, 1, 1, 1, 2 );
```

Once the scene objects are defined, the following mapping commands will be executed at the GDB.

For each row r in Table < Papers >

Sorted by AuthorName

where YearOfPublications > 1960 do

Map row r Add

r.AuthorName onto BkSlf1[1][1][].Author correlated
r.Address onto BkSlf1[1][1][].Address
r.AuthorPicture onto BkSlf1[1][1][].AutoPic
r.Content onto BkSlf1[1][1][].Content

These commands are translated into the following sequence of statements in the scene database language which is internal to the SDB component. First, for each qualifying row, scanned in the prescribed order, the row’s column are read into internal variables, say AuthorName, Address, Content and AuthorPicture. For AuthorPicture, the internal variable will contain the name of a gif file that will contain the actual picture. Similarly, for Content, the name of a Postscript file is stored. Then, we need to identify a free slot for a book in the bookshelf so it can Add a new book. This is done by the getNext methods:

```
FreeBookIndex = BkSlf1[1][1].getNext( 0 );
```
(a) Objects Fetching Tool. This component is responsible for fetching objects and scenes from remote sites.

(b) Predefined Parametric Objects. A predefined library of parameterized objects that can be "instantiated" into real SDB objects.

(c) Scene Objects Storage. This is a collection of scene objects. Together they prescribe the scene to be displayed. These objects may be equipped with interactions and other methods.

(d) Object Updating. This component creates, deletes and updates objects of the SDB. Such activity may trigger scene updating actions.

(e) Scene Updating. Updating activity may trigger actions by this component. It is responsible for interacting with the GCK and instructing it via commands in the script language to affect display changes. An important component here is the constraint rule set that is used in constraining objects placement and mutual interactions in the scene.

3. GCK - Geometric Construction Kernel

(a) Geometry Constructor. This component is responsible for actually forming the objects in the displayed scene. Given a functional description of the geometry, such as in Figure 2, an explicit representation to the geometry is created as a set of polynomial surfaces and/or polygonal objects.

(b) Geometry Fetching Tool. This component is responsible for fetching geometries, from remote sites with the help of the Object Fetching Tool of the SDB.

(c) Script Language’s Parser/Interpreter. This component interprets commands from the SDB Scene Updating component are written in the script language or other sources such as remotely fetched scripts. These commands are interpreted by this component and activate the Geometry Constructor.

4. SID - Scene Interaction and Display

(a) Interaction Processor. This component handles interactions with the user at the level of event detection and classification, possibly from a glove or another multi axes input device. The actual handling of these events is done at the SDB Scene Updating Component.
handles the interaction with the geometry such as mousing selection of a book, a user action that is reported to the database for a reaction. The geometry is maintained and updated in the SDB while all interactions with the external world is conducted with the help of the GDA module.

We now explain in detail each of the components of Figure 4.

1. GDA - General Data Access

(a) File/DB/WWW Access. This component is responsible for obtaining data to be mapped. Its sources are traditional database systems, the WWW, user files and data obtained from the SDB.

(b) Mapping Language Parser. This component processes mapping programs written in the mapping language and transforms them into a sequence of object manipulation commands that are interpreted by the Object Updating component of the SDB.

(c) Scene DB Access. This component is responsible for obtaining information from the SDB. It may be used by the File/DB/WWW Access component or by the Mapping Language Parser.

2. SDB - Scene Database
The data to be mapped onto the geometry can be a picture, a text file, an animated movie or even a voice recording. Each of the constructed objects in the scene might be required to present this data. For a file cabinet or a bookshelf, one can simply place the data in a file or a book, respectively. A book or a file can present a picture as one of its pages, possibly texture mapped on the surface representing the page. Movies can also be played using texture mapping capabilities, while text can be integrated into images and similarly treated.

Once the user is immersed into the environment, he can interact with it. The result of the interaction is application and object dependent. Objects placed in the scene must support a set of reactions to users’ actions. For example, a mouse click on a lamp might turn it on/off. A mouse click on a phone might initiate a (video) phone call via the network. Grabbing a page from a book might denote copying or moving the object in that page. Placing a grabbed object in a waste basket, printer, or fax machine will mean the obvious action.

Imagine the application of a virtual museum. A scene can be constructed according to one’s preferences that might assign a hall for each artist. Attributes of walls will contain http anchors to images or icons of images across the WWW. A visitor to this virtual museum will select doors to open in order to enter new halls and might click iconified painting so as to get the full size pictures.

More appealing is the fact that sharing your museum with others is easy. Anyone can create an http anchor in their GCK script to your museum and attach it to their own museum through a new corridor (“breaking” the wall and creating a new door). The http anchor to your museum will point to your GCK script describing your museum. By fetching and executing your museum’s GCK script in remote sites, your museum can reappear virtually everywhere, and efficiently so. The end result of these capabilities is the creation of a distributed museum that is totally transparent to the visitor.

This points to the problem of update propagation and subscriptions. A user might be interested in being informed about changes in the sub-scene that he/she obtained from a remote site.

4 Implementation

Our current implementation contains three major modules that can be seen in Figure 4. The three modules are the GCK that handles the geometry construction and the geometry interaction, the scene database, SDB, and the general data accessing module, GDA. The GCK handles the construction of the geometry from either its own scripting language’s scripts, predefined parametric objects, or simply geometry fetched from remote sites with the aid of the other modules. This GCK module also
this is not always the case. Portions of the geometry and/or data objects can be tagged as invisible at times. For example, uninteresting books might be tagged as hidden and will not be displayed even though they continue to be part of the geometrical database, as is the content of a closed book.

Consider the bookshelf constructed in Section 3.1. One can map a File into one book in this BookShelf using the command (executed within the SDB, see below)

```plaintext
Bs[1][3][2][17].Insert( GifImageobj, "Content" );
Bs[1][3][2][17].Insert( Authorobj, "Author" );
```

The predefined Insert method allows the mapping of the object GifImageobj to the “Content” slot of the 17th book in the second BookShelf of the third BookShelf column. The same book can be assigned an author name using the “Author” tag. Geometrical processing of these Insert commands would include the creation of the new book’s geometry at the proper shelves (if none exists) and the placement of the GifImageobj and Authorobj in them. Each scene database object has a set of predefined methods. A bookshelf can have the methods Insert, Create, Delete, Replace, GetNext and Open. The GetNext finds the next empty spot for a book in the bookshelf, returning zero for an empty bookshelf. The Create, Delete, Replace and Open methods create, delete, replace, and open one book, respectively. Opening a book activates an animated sequence displaying the process of extraction of a book from the bookshelf, opening its front page, and presenting its content - the GifImageobj herein. The Open method also activates the book object itself, making it ready for further browsing of the interacting user.

While animation design is a complex problem, in general, it is simplified in this case due to the ability to pre-design the animation beforehand. That is, every object in the scene will include, among other attributes, an animation attribute that, once activated, performs a known task. A file cabinet object will animate the opening of a (mouse selected) drawer. A door/window will open/close once clicked. A book will be extracted and its front page will be exposed once selected.

Therefore, this set of motion commands can be planned beforehand. The GCK also supports animation by supporting animation curves, curves that are parameterized by time and prescribe the exact orientation and position of the animated objects as a function of time. A door will have a rotational animation curve representing its rotation. A drawer will have a translational animation curve representing its sliding motion. During the animated interaction, this time parameter is stepped forward, activating the animated sequence.
3. Fetching of remote objects (using, for example http anchors).

The third option is attractive mainly if the mapping is done in the WWW environment. An http anchor can point to a predefined object in a remote site at a low level as a list of polygons, or alternatively, it can point to a GCK script that, upon execution, creates an instance of the desired object.

Each and every object in the scene is given a unique name. For instance, the bookshelf in Figure 1 (i) was constructed using the command

\[
Bs1 = shelves(1, 2, 1, 3, 4);
\]

where the parameters (in order) define the (three) dimensions of the bookshelf followed by the definition of three columns with four shelves each. Any access to this particular bookshelf, in further mappings or interaction, will reference \( Bs1 \). Attribute modifiers can be attached via an \texttt{attrib} command and orientation can be selected via homogeneous transformation specification. For example:

\[
Bs1 = shelves(1, 2, 1, 3, 4) * rx(30) * tx(3) * ty(1);
\text{attrib}(Bs1, '\color', red);
\]

is asking to rotate the \( Bs1 \) bookshelf 30 degrees in \( X \), translate \( Bs1 \) in \( X \) three units, and then translate \( Bs1 \) in \( Y \) one unit. This same bookshelf is also assigned the \texttt{red} color.

### 3.2 Maintenance, and Interaction

Once a scene is constructed, data need to be mapped to it. The distinction between the different stages in which the geometry of the scene might be affected is now clarified:

1. **Scene construction time.** Here, the scene is graphically constructed based on a given GCK script (based on the different object definitions). No data is displayed, or graphically attached, to the geometry at this time.

2. **Mapping time.** At this time data is attached to the objects in the scene database. This triggers actions that may result in changes in the displayed scene. At this point the basic scene geometry is already defined and one can associate displayed data with the displayed geometrical entities. Displayed data can be text files, image files, animation sequences, etc. Methods activations can also affect the display.

3. **Display and Interaction time.** While it is typically the case that a constructed geometry can be displayed, and attached data can be viewed with the geometry once stages 1 and 2 take place,
but can be quite difficult to construct if only the finest polygonal approximation to the surface is available.

The scripting language of the GCK allows one to attach attributes to each object in the scene. These attributes can serve as modifiers to the basic behavior of the geometry. For instance, such an attribute can specify the color of the specific instance of the object, or alternatively, can prescribe the correct orientation in space of a new instance, relative to the canonical orientation of the original predefined object.

In summary, the geometry can be constructed with the aid of the following sources:

1. Predefined parametric objects (these already have associated scripts).

2. Execution of scripts (written in the scripting language of the GCK).
can be installed in remote sites and further reduce the amount of data to be transmitted to as little as over 10 kilobytes of uncompressed data, including predefined animated behavior such as a file cabinet opening or lifting the receiver of a phone. Needless to say, transferring this entire geometry to remote sites as low level geometry would necessitate the transmission of millions of polygons, hence we gain a reduction in data size of approximately two to three orders of magnitude. Figure 3 shows a partial view of one of the 22 offices, namely the office of the first author. This entire office was instantiated using the following GCK commands:

```gck
room490 =
    list( room( 2, 3, 2.5, list( list( "s", 0.6, 0, 1.4, 2, TRUE ) ),
             list( list( "n", 0.4, 1.7, 1.6, 2.3, TRUE ) ) ),
        instance( "Board" ) * rotz( -90 ) * trans( vector( 2, 1.8, 1.2 ) ),
        instance( "FileCabinet" ) * rotz( 90 ) * trans( vector( 0.3, 0.1, 0 ) ),
        instance( "Shelves" ) * rotz( 90 ) * trans( vector( 0.2, 0.7, 0 ) ),
        list( instance( "Table" ),
             instance( "Computer" ) * trans( vector( 0.2, 0.25, 0.7 ) ),
             instance( "Chair" ) * rotz( 90 ) *
             trans( vector( 0.65, -0.3, 0 ) ) ) *
        rotz( -90 ) * trans( vector( 1.3, 2.4, 0 ) ),
        instance( "Sofa" ) * rotz( -90 ) * trans( vector( 1.5, 1.2, 0 ) ),
        list( instance( "Table" ),
             instance( "Phone" ) ) * rotz( -90 ) *
             trans( vector( 0.3, 0.4, 0.7 ) ) ) *
        trans( vector( 0.8, 2.4, 0 ) ) ) *
    trans( vector( 6, 4, 0 ) );
```

assuming the proper predefined geometry of (board, filecabinet, shelves, table, computer, chair, phone, and sofa) is readily available. The `room` function creates the walls of the room with as many doors and windows as needed. Herein, two openings in the walls are defined in the northern ("n") and southern ("s") walls.

Having the full control on the geometry and on the way the geometry is constructed, opens the way to various optimizations that are difficult to perform otherwise. For example, freeform parametric surfaces are typically converted into an auxiliary polygonal approximation representation for the purpose of display. The transmission of the polygonal representation implies that huge amounts of data must be retransmitted. It also hinders the exploitations of useful techniques such as multi-resolution (levels of details) display. At various viewing distances, the same freeform object may be displayed using a different polygonal approximation. The closer the freeform object is, the finer the polygonal approximation must be. This multi-resolution representation of a freeform surface is easy to construct if the freeform surface is available (as a (piecewise) polynomial or rational function)
\begin{verbatim}
cup = function( size ):
  body: cross: axis: handle:
  cross = cbspline( 3,
    list( ctlpt( E3, 0, 0, 0 ),
      ctlpt( E3, 0.5 * size, 0, 0 ),
      ctlpt( E3, 0.55 * size, 0, 0 ),
      ctlpt( E3, 0.55 * size, 0.2 * size ),
      ctlpt( E3, 0.7 * size, 0, 0.25 * size ),
      ctlpt( E3, 0.9 * size, 0, 0.9 * size ),
      ctlpt( E3, size, 0, 1.28 * size ),
      ctlpt( E3, size, 0, 1.3 * size ),
      ctlpt( E3, 0.9 * size, 0, 0.92 * size ),
      ctlpt( E3, 0.7 * size, 0, 0.27 * size ),
      ctlpt( E3, 0.55 * size, 0, 0.22 * size ),
      ctlpt( E3, 0.55 * size, 0.02 * size ),
      ctlpt( E3, 0.5 * size, 0.02 * size ),
      ctlpt( E3, 0, 0.02 * size ) ),
    list( KV.OPEN ) ),:
  body = surfRev( cross ):
  axis = cbspline( 3,
    list( ctlpt( E3, 0, 0, 0 ),
      ctlpt( E3, 0.5 * size, 0, 0 ),
      ctlpt( E3, 0.7 * size, 0, 0.5 * size ),
      ctlpt( E3, 0.15 * size, 0, 0.5 * size ) ),
    list( KV.OPEN ) ),:
  handle = sweepSrf( circle( vector( 0, 0, 0 ), 0.1 * size ) *
    scale( vector( 0.3, 1, 1 ) ),
    axis,
    vector( 0, 1, 0 )):
  return = list( body,
    handle * trans( vector( 0.8 * size, 0, 0.6 * size ) ) );
\end{verbatim}

Figure 2: The definition of the parametric type of the cup in Figure 1 (a). The body is constructed as a \textit{surface of revolution} via the \texttt{surfRev} constructor. The handle is a \textit{sweep surface} of an ellipse swept along the axis curve using the \texttt{sweepSrf} constructor. The handle is then positioned at the proper location via the \texttt{trans} attribute.

With the set of primitives and predefined object types one is able to express a new geometry very succinctly. A scene consisting of 22 rooms, a portion of our Computer Science building, was modeled to a great accuracy, including all accessories, such as desks, chairs, file cabinets, black boards, phones, computers, books etc. The exploitation of the GCK reduced the definition of this entire 22 rooms' complex to a little over 10 kilobytes of geometry instantiation commands and a little over 50 kilobytes of the predefined primitives (similar to the cup in Figure 2). This set of scripts of predefined geometry
Figure 1: A sequence of predefined parametric objects that can be constructed at remote sites with a single line stroke. See also Figure 2.
We have a local GCK (geometry construction kernel) that inputs scripting language commands and generates geometry. This GCK is installed in every site that supports three dimensional geometry, much like the wide support of image display tools (for example using SGI's rgb format for image manipulation tools). Unlike images, however, that are difficult to regenerate, one can prescribe a three dimensional scene in a very compact way, either parametrically or constructively, yielding very high compression rates. The GCK is, in essence, a solid modeling tool that has no graphical user interface - it consists of a core of geometric functionality driven by a scripting language. With the GCK, an object can be fetched from remote sites or loaded in from local predefined geometry, can be created and manipulated, and cached for future processing.

One can define a scene using predefined parametric types, object types that are expected to be useful everywhere and are provided with the GCK. As complex as these predefined types might be, one can request their creation at a remote site with a single line of the scripting language. Figure 1 shows images of some predefined object types to be supplied with the GCK that we use (which is based on a freely available modeling package, IRIT [IRIT]). Each and every one of the objects in Figure 1 might be represented using tens of thousands of polygons. Nonetheless, a single instantiation of a predefined parametric type is sufficient, when the predefined type itself is described using the GCK script language. Figure 1 also shows the commands that were used to create these objects.

Figure 2 shows the commands in the GCK script language that were used to create the cup in Figure 1(a). Even if no such predefined cup object exists, sending a GCK script prescribing a cup object as is defined in Figure 2 is several orders of magnitude more efficient than sending thousands of polygons approximating this same cup.

While predefined parametric objects require a single line to define, complex geometries can also be constructed by exploiting the modeling capabilities of the GCK scripting language. Clearly, transmitting a script representing several Boolean operations on basic primitives is more efficient than transmitting the end, gigantic, polygonal approximation result. Similarly, sending the curves (i.e. mathematical equations) that construct a freeform surface is more compact than sending the surface itself or its polygonal representation. Alternative sources for geometries can be http anchors pointing to remote GCK scripts or even at low levels of geometry: polygon or pixel. If the user insists on getting millions of polygons from a remote site, this should be allowed.

\[\text{The scripting language is different than the mapping language. The latter is used to specify the mapping of data onto objects, the former defines the displayed geometry of an object.}\]
- The architecture must be open, allowing an end user the ability to modify existing objects, adding his own objects, and even constructing whole new scenes that one can connect to. All these operations must be feasible without compromising efficiency.

- Preserve the ability to create, process, and affect the lowest level of the geometry - i.e. a single point or polygon.

Because the sizes of geometrical databases tend to explode as scenes become more and more complex, and since efficiency was recognized as a prime goal, the option of transferring millions of polygons across the network was rendered impractical. VRML [VRML] was proposed as one standard of three dimensional geometrical data transfer across the network. In VRML one describes a scene in a hierarchical manner. The language is mainly concerned with the display activity itself. VRML lacks freeform surface representations\(^1\), such as (piecewise) polynomial (B spline) Bezier surfaces. Further, VRML requires the transfer of the entire geometry in explicit form, allowing for a high order geometry representation of very simple primitives such as cones and cylinders.

We have taken a different approach that exploits a local Geometric Construction Kernel (GCK) at each and every end user’s site:

- Scene database. We have a scene database which describes the scene. This is an object database where objects are equipped with both data, appearance (i.e. geometry) and actions (methods). In contrast, VRML objects are purely displayed geometrical entities. Thus, we have an advantage in the (much higher) level of abstraction in which we view the scene.

- Powerful script language. The actual geometry construction activity is done via a script language. This script language is more powerful than VRML and it enables a lot more freedom in expressing geometries succinctly, as will be demonstrated shortly.

- Pre-staging useful objects. Each client (i.e. user) is equipped with a universally known pre-defined set of parametric objects (e.g. a file cabinet with slots). Each such object is associated with a script describing its geometry. This creates a known “mini-world”, or ontology, composed of objects whose definition need not be moved around, and that are very useful for data mapping applications. This universal database will be periodically augmented as new such object types are defined.

\(^1\)A freeform is a shape that cannot be represented using boolean combinations of finitely many simple forms such as cubes, cones and cylinders.
originating from the same row. It can also be useful in scene maintenance activity. Every object member has a correlation component. Correlation components are not considered in determining duplication.

An interesting aspect of the Map statement concerns concurrency control. It is possible that at the same time there may be a number of concurrently operating mapping programs affecting a scene. Also, the user may be affecting the scene interactively, or possibly several such users may interact with the scene simultaneously. It seems that globally enforcing serializability will be the wrong thing to do. Still, at the level of a single Map statement, atomicity seems reasonable. One way to address the issue is to partition the scene objects into those requiring serialization and those that do not. This can be done in conjunction with transaction blocks within mapping programs. This issue deserves further work.

We have not described the Scene Organization subset of the mapping language. This subset deals with creation of (displayable) objects, specifying placement and orientation of objects in the scene, stating constraints on object placement (e.g., minimum distances between objects).

3 Scene Construction, Maintenance, and Interaction

A scene is a collection of geometrical entities. A scene may be static, continuously changing or occasionally changing. For example, a scene can be a museum tour, a moon walk through, or driving on the highway. As hinted by the scene’s types, interaction is expected to play a major role within a scene: by walking/driving in the scene and by selecting and interacting with the different objects of the scene.

3.1 Constructing the scene

The scene is created using a solid modeling tool, i.e., a tool to construct and represent the scene’s geometry in a computer based display. Several concepts directed the approach that was selected:

- The scene construction should be done efficiently, for both local users as well as remote ones, to enable the exploitation of this tool across the network, as a WWW geometry engine server.

- The scene should allow for interaction and hence it must support animated objects and dynamic behavior.
For each row <r> in Table <T>  
{sorted by <column-list> {ascending, descending}}.  
where <qualification> do  
a sequence of semicolon separated statements od;  
The semantics is to examine table <T> one row at a time. If the row satisfies <qualification>  
then the body of the for statement is executed. If the sorted-by construct is present, the rows  
of the table to which the row <r> belongs, as indicated by an enclosing For statement,  
are scanned in this sort order. The <qualification> is a boolean combination of simple  
comparisons, e.g. <r>:Sal > 100000 & <r>:Name = 'Smith'.

- The basic decision statement is the If statement whose form is simply:  
  If <expression> then <statement>  
or If <expression> then <statement> else <statement>.  
Here, <expression> may involve both <row-reference>'s and <object-reference>'s.  
If the <expression> involves only <row-reference>'s then it is said to be one directional  
and bi-directional otherwise.

- The basic mapping statement is the map statement. The general syntax is:  
  Map row <row> {Add, AddMult, Delete, DelMult} <Map-stmt> {, <Map-stmt>}*;  
Where:  
  <Map-stmt> ::= <row-reference> onto <object-reference> {correlated {tableid | rowid | second | minute | hour | date}+}.  
The meaning of a Map statement is the following:

1. Add indicates addition in the sense of set-union, of the row component, as indicated by  
the <row-reference> to the <object-reference>. AddMult is similar to Add, in  
the sense of multi-set union, i.e. with duplicates. Delete indicates deletion in the sense  
of set-difference, of the row component, as indicated by the <row-reference> to the  
<object-reference>. DelMult is similar to Delete, in the sense of multi-set deletion,  
i.e. deleting one out of possibly many duplicates.

2. correlated means that the piece of data that is mapped “remembers” its table id, its row  
    id, or its mapping time. This data is kept in the scene database that describes the content  
of the scene. This correlation component may be used later on in locating pieces of data
A scene is composed of objects whose names appear in the scene’s definition file. Some of these objects may be indexed objects. For example, a book-shelf object may have 50 entries for books. Each object, and sub-object, may have object attributes. For example, the $i^{th}$ sub-object of a bookshelf object may have a content attribute. These attributes may be tested and operated upon. Each object or sub-object may also have defined operations (methods) which, when activated, result in actions on the displayed virtual reality object.

- An object reference is defined as:
  \[
  \langle\text{object reference}\rangle ::= \langle\text{object}\rangle | \\
  \langle\text{object reference}\rangle [\langle\text{indexing variable}\rangle] |
  \langle\text{object reference}\rangle . \langle\text{object attribute}\rangle |
  \langle\text{object reference}\rangle . \langle\text{object method}\rangle |
  \langle\text{object reference}\rangle . \text{correlated \{tableid | rowid | second | minute | hour | date\}+}
  \]

Each scene database object or any component of an object has a correlation tag associated with it. This tag is used to store information about the sources from which scene database objects, and components thereof, originated. For example, one may store the time the object was last updated or the table name whose data was used in updating or creating the object (or component).

Similarly, a row reference is defined as:
\[
\langle\text{row reference}\rangle ::= \langle\text{row}\rangle | \langle\text{row}\rangle . \langle\text{column}\rangle.
\]

An indexed object is either a set-object, a multi-set-object, an ordered-set-object, or an ordered-multi-set object. Ordered objects are basically arrays with slots; an ordered-set-object enforces a no duplicate policy. If the indexing variable is omitted, e.g. $O[]$, the meaning is that object $O$ is either an ordered-set-object or an ordered-multi-set-object, and some unused slot within $O$ is selected at run-time. This usually happens when we add information to the scene database and we don’t care where exactly it is added (a one directional mapping). If an indexing-variable, say $k$, is present, there are two cases. If $k$ has a value at run-time, i.e. it has been initialized and perhaps changed thereafter, that value is used. Otherwise, as in the case of omitting the indexing variable, $k$ is assigned a value at run-time that can later be used (exploiting a bi-directional mapping).

- The basic data scanning device is the For statement which may assign each row of the data source table to a scene database component. The syntax is as follows:
more comprehensible, and it also presents optimization opportunities. In this paper, we use a simple
looping language. This language may be seen as “syntactically sugared” logic based language with
update facilities. One problem with logic based update language mechanisms is that assignment is not
a natural construct. This forces the formation of semantics that is often not natural for programmers
to understand. Here, because we operate on displayable objects that can also be manipulated,
logical satisfaction leads to the immediate activation of object’s methods. In other words, we have
an operational model (similar to Prolog’s assert and retract predicates, rather than a model based
semantics).

The paper is organized as follows. In Section 2, we describe the basic features of the mapping
language. Scene construction, maintenance, and interaction are dealt with in Section 3. In Section 4,
we present an example and we outline our system implementation. We conclude in section 5.

2 Mapping Language Definition

In this paper we limit the exposition to flat tables, i.e. one whose elements are atomic data types.
Still, a cell of a table may contain various information types: an integer, a string, a gif file, a postscript
file, or an http anchor. We view files and http anchors as typed strings.

- The table schema is defined using the Table command which specifies the columns and their
  content types. The syntax is:

\[
\text{Table <table-name> : }
\text{Column <column-name> : <column-type>,}
\text{...}
\text{Column <column-name> : <column-type> ;}
\]

- The Scene statement determines the scene database onto which the mapping is currently carried
  out. We assume that each scene is uniquely associated with a scene database. The statement
  is:

\[
\text{Scene <scene-name>}
\]

In general there may be a number of scenes we may want to map data to. The scene statement
serves to switch among scenes by telling the system which is the current scene.
Users today are exposed to large amounts of data from traditional databases as well as from the Internet [Inter]. This data needs to be displayed and conveyed in a user friendly way. This paper deals with the utilization of the virtual environment for the presentation of, and interaction with, selected pieces of data. The data may originate as a database query, a WWW query [KS95] or simply as a data file. We assume that the data is given in a tabular form. We also assume that each cell of the table may itself be a complex structure, a tuple, a set, or a table, recursively.

The virtual environment consists of scenes, each containing inter-related objects. These objects may model real world objects, e.g. an office table or a file cabinet, or imaginary objects such as the next generation spaceship. These objects may be compound, that is have sub-objects. These sub-objects are either named, as attributes of higher level objects, or are indexed numerically, e.g. the 50'th book on a bookshelf object. These objects may be inter-related, e.g. a fax machine object may be placed on top of a table object, a printer object may be connected with a wire object which itself may be connected to a computer object. Interaction with every one of the objects is made possible by defining the behavior of the object when it is subjected to various events. For example, mouse clicking on an object representing a phone might initiate a phone call. The virtual environment may also contain objects (e.g. corridors) that lead into other virtual environments which are defined elsewhere.

In mapping data to a virtual environment we are essentially mapping from one database (the table containing the data to be mapped) into another database (the one describing the scenes and their contents). The simplest mapping type to implement is the one directional mapping. Here, data is mapped into the scene without examining the scene content. For example, one can add a “folder” to a “file-cabinet” without examining whether a related folder already exists there. A more complex mapping type is the bi-directional mapping where mapping decisions are also based on the current scene content. For example, one may place an item on a “bill-board” only if a related item is not already in the virtual object “incoming-mail-basket”. If one would like to maintain a complex scene over time then bi-directional mappings are a necessity. Once data is placed, updated, or method-activated in the scene description database, actions may result which change the way the scene database is actually displayed. Such method activations may also be the result of user-data interaction.

For the mapping task, one could in principle use a general purpose programming language. We argue that a database programming language, one that is designed to handle large volumes of similar data, is more suitable for the task. Such a language makes scene understanding and management
Abstract

This paper describes a language and a system for mapping data onto a virtual reality environment. The data may originate as a database query, a WWW query or as a data file. We assume that the data is given in tabular form. Generally, each cell in the table may be either an atomic object, e.g., an integer or a LaTeX file (thought of as a typed string). Each cell of the table may in principle be a complete structure, a tuple, a set or a table, recursively; in this paper we shall only treat tables whose cells contain atomic values.

The virtual environment consists of scenes. A scene contains inter-related objects, e.g., a bookshelf. The mapping assigns content to scene’s objects, e.g., inserting a book with images into a bookshelf. The mapping may also induce scene manipulations, e.g., opening a door. The scene itself is internally represented by a scene database. The mapping language can directly operate on this database as well, so it can function as a scene maintenance mechanism.

The system exploits pre-defined useful high level parametric objects that are known everywhere, e.g., a file cabinet, a bookshelf, a phone. Moreover, it allows the definition of new parametric geometry with the aid of a distributed solid modeling kernel. This saves on network traffic and enables a universally known vocabulary. The system is designed to enable incremental changes to scenes (e.g., a new call for papers is attached to a bill board). It will also enable navigation from one virtual scene to another via anchors. Scenes are interacted with using a six degrees of freedom mouse and a head mounted display. The system is currently in advanced implementation stages, using the IRIT solid modeling environment [IRIT] and Silicon Graphics Performer.

1 Introduction

Virtual reality places a human being within an illusion of an environment by using specialized equipment, a viewing helmet, sensors to determine position and movement, and various input devices such as a 3-dimensional pointing and clicking device and/or a glove to manipulate virtual objects. Virtual reality applications include simulators of intricate environments such as flight and surgery, as well as design tools in architecture, mechanical engineering, civil engineering, electrical engineering (chip design) and many other fields.

*Research supported by Grant 8528-1-95 of the Israeli Ministry of Science and the Arts.