A Graph-Based Design of an Open Database for 3D Models with Versioning

Wei Ke and Lap-Man Hoi

***Abstract*—We present a graph-based method for illustration and specification of the data structures to persist 3D models. We define the mapping from the object graphs to the relations in a relational database. Our ORM scheme fully opens the structures of the underlying relations so that the data can be easily manipulated and used by other tools and under other schemes. By this method, 3D models can be accumulated from large amount of contributors in the long run, and used by various types of applications, such as real-time rendering and geometric searching. Our graph notation and ORM scheme support arrays and variants, which are essential in the specification of complex data structures. Algorithms are provided to illustrate how the mappings are implemented, as well as how the database can be used. An independent versioning system is implemented on top of the 3D model database, showing the flexibility and advantage of this graph-based method. We implemented a minimal 3D model database following this design, and present the evaluation of the performance of loading models from the database.**

***Index Terms*—Object-relational mapping, class graph, 3D model, versioning, open database.**

I. Introduction

Many virtual reality systems involve rendering of city scenes, such as in urban planning and digital reproduction of historical city areas [1]. These VR systems require a large number of 3D models of buildings, streets and other elements, even for a relatively small city region. The work for construction these models can span over several projects or phases. Therefore, a database for storing and accumulating the models is required. It is obvious that the 3D model database must be independent to the applications that make use of it. Also, the structure of the database should be open in a way that it can be easily adopted by future applications.

3D models, including meshes, materials and texture maps, are naturally represented as objects and the relations between these objects [2]. These structures can be well represented by class graphs with the ease of understanding, a rigorously defined semantics [3] and a transformation theory [4]. On the other hand, in terms of other data applications, such as searching of features and re-composition of data components, tables of relations in a traditional relational database are more efficient. In order to take the benefits of expressing structures in class graphs and storing data as relations, we provide

1) a set of graph notations for the specification of the data structures of 3D models, and
2) an open graph-to-relational mapping scheme for persisting the object graphs, which are instances of the class graphs, to a relational database.

With the open mapping scheme, model data stored in the database can be manipulated independently to the structures of the models. Fine grained data features can be extracted, indexed and even recombined into newer structures. This provides the flexibility for future applications of the existing model data. It is very important that we can make use of those ever accumulated 3D models from past hard work, especially in smaller educational projects with limited resources and yet a city-scale large scene to deal with.

In addition, our class graph design method allows the introduction of new structures to the database without breaking existing structures and data items. It is inevitable that we need to keep adding new structures as the system evolves in the long run. Easy migration from the old dataset to the new dataset is essential. Based on our graph method, we demonstrate

3) a versioning and naming scheme on the top of the model database with tagging capability, and the scheme is also able to simulate a conventional file system hierarchy.

Versioning and tagging [5] is a critical feature in a long running 3D model database for rendering city scenes. There are regular updates to an existing model. There are also different versions of the same model, so as to present the chronological evolution of an area. Our versioning scheme fully illustrates the principle of using standalone structures to implement new features without affecting any existing data.

A. Related Work

There is a number of literatures on general object-relational mappings [6]-[8]. They focus on the transparent persistence of objects to relational databases, with data intentionally viewed only as objects. Applications that want to use the data have to turn them into objects, limiting the flexibility and re-composability. Also, they do not have a close correspondence to the graph notation used to design the data structures. For 3D model databases, several functional issues are addressed in [9], with the context of GIS. It focuses on what model features are stored and how they can be retrieved. The use of intuitive objects and graphs in the design method is not discussed. In [10], a graph database is actually used to simplify the generation of GIS test data and virtual cities. A general survey and comparison of graph databases is presented in [11]. However, directly using a graph database to store 3D models limits the types of applications, for relational database are far more sophisticated and ubiquitous.

B. Outline of Structure

Manuscript received December 5, 2015; revised March 11, 2016. The research present in this article is funded by Project 043/2009/A2 of The Science and Technology Development Fund of Macau S.A.R.

The authors are with the Computing Program, Macao Polytechnic Institute, Macao S.A.R., China (e-mail: wke@ipm.edu.mo, lmhoi@ipm.edu.mo).

The remaining of the article is arranged as follows. We present the graph notations for various types of structures and how they are mapped to relations in Section II, along with some examples of 3D model structures. We emphasize the formulation of arrays and variants. In Section III, we demonstrate how the versioning and naming scheme can be added independently to the model data. We also have a brief discussion of the way to add non-model data as annotations to the database. Next, we give the algorithms in Section IV to import and export 3D models from/to conventional model files. Finally, we study the caching of a partial database as a subgraph for off-line execution of applications in Section V, before we respectively evaluate and conclude in Section VI and VII.

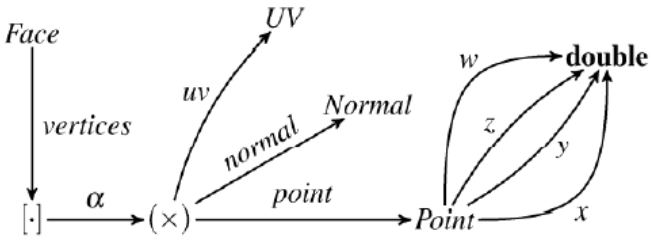

Fig. 1. The *Face* and *Point* classes.

II. Class Graphs and the Object-Relational Mapping

As in most OO languages, an object is an aggregation of values and references to other objects. Values are data items that are embedded directly in an object, and references are just pointers or identities that refer to other objects. Value types include common primitive data types, such as **int**, **double**, **decimal** and **string**. The structure of an object is defined by its class, represented by a directed graph, called a *class graph* [3]. A field in class C with name a and of type T is illustrated as an edge $C \xrightarrow{a} T$. If T is a value type, it does not have outgoing edges. Otherwise, T may have further outgoing edges to other types. Thus, a graph can represent several classes. For each class C, its structure is defined by those reachable edges and nodes from C. To prevent a graph from being too large, we draw classes separately when needed. For example, Fig. 1 shows the structures of class *Face* and *Point*, while class *UV* and *Normal* are not defined in this graph.

The objects of each class are stored in one table, where each object occupies a row and each field of the object takes up a column. Each object has an identity, represented by a value of **serial**, which is a 64-bit unsigned integer that is unique in a table, and associated with the row of the object. A field of a value type is embedded directly in the object. For a field of a class type T, the identity of some object of T is stored instead. For example, the *Point* class illustrated in Fig. 1 corresponds to the following Table I structure.

TABLE I: The Point Class Illustrated in Fig. 1 Corresponds to the Following Structure

Point :					
id	*x*	*y*	*z*	*w*	←field name
serial	**double**	**double**	**double**	**double**	←field type

A. Arrays and Tuples

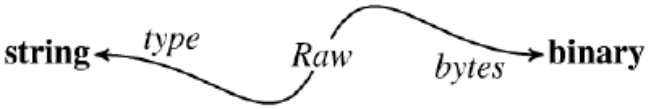

Fig. 2. The *Raw* class.

Besides named classes and value types, there are two unnamed class types, the array type and the tuple type.

We use the [·] symbol to represent the array type in a graph. Since the array type is generic [12], whose element type can vary, we use a type variable, in Greek letters, to label the relation between the array and the element type T, $[\cdot] \xrightarrow{\alpha} T$. This notation $G \xrightarrow{\alpha} T$ also applies to other generic types, when we instantiate a generic type G by substituting a concrete type T for the type variable α. In the graph definition of a generic type G, the present of a type variable α is thus represented by $G \xrightarrow{\alpha} \alpha$.

We use the (×) symbol to represent the tuple type. Usually, the components of a tuple each have a name, just like the fields in a class. This component name labels the edge from the tuple to the component type. For example, $(\times) \xrightarrow{m} T$ means there is a component with name m and type T in the tuple.

There is an array in each object of the *Face* class, as illustrated in Fig. 1. We store all arrays of the same element type in a table. Each array in the table has an *array id* and each element has an *element index* within the array. For a tuple type, we do not create a separate table as we do for a named class. Instead, we level the components of the tuple up, as if they are the fields of the enclosing class. The following Table I shows the table structure for the array of tuple (*Point*, *Normal*, *UV*), which is the type of the *vertices* field of *Face*.

TABLE II: The Table Structure for the Array of Tuple (*Point*, *Normal*, *UV*)

[(*Point*, *Normal*, *UV*)] :					
id	*index*	*point*	*normal*	*uv*	table to ← look up
serial	**int**	**serial**	**serial**	**serial**	
		Point	*Normal*	*UV*	

The objects of the *Face* class are stored in another Table III and refer to the corresponding array id in the above Table II.

TABLE III: The Objects of the *Face* Class

Face :	
id	*vertices*
serial	**serial**
	[(*Point*, *Normal*, *UV*)]

TABLE IV: The *Face* Class

Face :			
id	*vertices*		
serial	[(×)]		
	point	*normal*	*uv*
	serial	**serial**	**serial**

However, some database systems support arrays and tuples as native types, where an array or a tuple can be stored in a field directly, simplifying the above mapping. For such a database system, the *Face* class is stored as simple as follows

in Table IV.

B. Raw Data

Some data files are stored directly without knowing their internal structures. These files are interpreted by programs outside the database system. We store these files each as a block of raw bytes in a table with an additional (mime-)type annotation, to help find the external handlers. The structure of *Raw* is shown in Fig. 2.

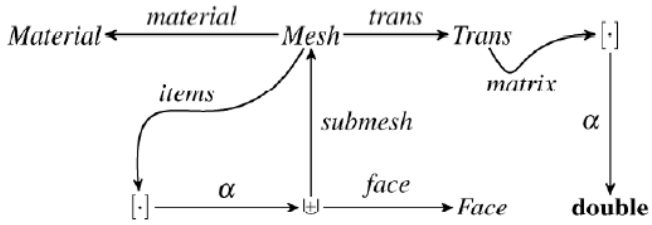

Fig. 3. The *Mesh* class.

C. Variants

A class of objects or a type of values may have several forms, such as internal nodes and leaf nodes in a tree, or real numbers represented in different formats. This is well-known as a variant type or a disjoint union. We denote a variant type which has a variant field with name v and type T by $\uplus \xrightarrow{v} T$ in a graph. Since we refer to objects through references, and references to different variants occupy the same amount of storage space. We partition the reference space (**serial** space) so that objects of different forms (variants) have references in different disjoint ranges. For example, as shown in Fig. 3, a mesh contains a list of items, and each of the items can be a face (leaf node) or a submesh (internal node). The faces and the submeshes are stored in two tables, where the *id* of a face is in a different range from the *id* of a submesh. The Table V structure of the *Mesh* class is given below.

TABLE V: STRUCTURE OF THE *MESH* CLASS

Mesh :				
id	*items*		*trans*	*material*
serial	[⊎]		**serial**	**serial**
	submesh	*face*	*Trans*	*Material*
	serial$^1/_2$	**serial**$^2/_2$		
	Mesh	*Face*		

We write **serial**$^m/_n$ for the range [|**serial**| × $(m-1)/n$, |**serial**| × m/n). If an item reference is in **serial**$^1/_2$, we know it is a reference to a submesh, and we search for it in the table of the *Mesh* class. Otherwise, it must be a reference to a face, we then search for it in the table of the *Face* class.

To partition the **serial** space, we must make sure the range of a class is unique in all the variant types it participates. If two variant types have an intersection, all the types in the two variant types must be considered in the partitioning. This forms an equivalence relation between types, and all the types can be partitioned into a number of disjoint sets using the *union-find* algorithm [13]. The types in one of the disjoint set share a single **serial** space. This information can be used to partition the **serial** space either evenly by the number of types in the set, or by also taking into consideration the weight of each type involved.

D. Variant Lifting

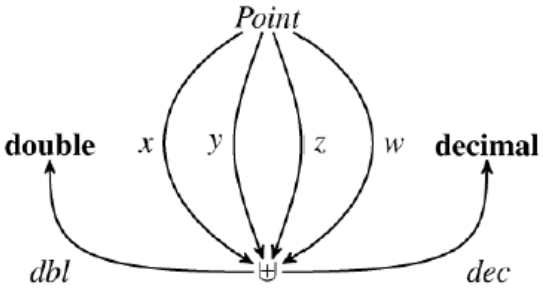

Fig. 4. The multi-format *Point* class.

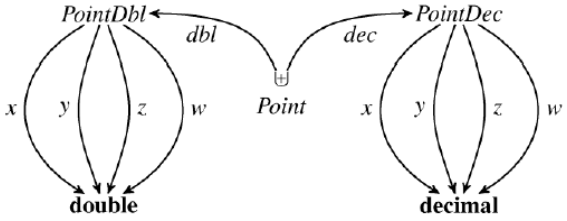

Fig. 5. The lifted multi-format *Point* class.

Unlike objects of classes, data of value types are stored directly in a table, this makes storing variants which contain values difficult, since different types of values may occupy different amounts of space. As shown in Fig. 4, each of the coordinates of a point can be stored in two formats, a double precision floating point of fixed size of space, or a sequence of digits of variable size of space. We cannot mix such two formats in one column of a table. To handle this case, we choose to *lift* the variant type for each coordinate up to the *Point* class itself, that the class is split into two variants, one contains coordinates all in the **double** type, the other contains coordinates all in the **decimal** type, as illustrated in Fig. 5. This transforms the variant type to a class type, thus objects of the class can be split into two tables and accessed via references of the same storage size, as described earlier, although we sacrifice the flexibility of mixing different formats for the coordinates within one point. The tables of the variants are shown below.

TABLE VI: THE TABLES OF THE VARIANTS

Point ⊎:					
id	*x*	*y*	*z*	*w*	
serial$^1/_2$	**double**				(*PointDbl*)
Serial$^2/_2$	**decimal**				(*PointDec*)

With the class graph notation that supports arrays, tuples and variants, we can represent the structures of other complex elements, such as *Material*, *Normal* and *UV*, of 3D models in graphs. The graphs can then be mapped to a relational database according to the mapping scheme. The table structures are completely static that the data items are fully open to other applications.

III. VERSIONING AND NAMING

Objects, such as 3D models and textures, often have multiple versions, for example, a series of updates, or a collection of dialects of the same model. We call such an object with different versions a *versioned object*, and each

version is a real object. A versioned object is just an identity associated with all the real objects that represent the different versions. We use the *VerId* class to represent versioned objects, and this class has no field but an *id* of type **serial**. Thus, each versioned object is identified by a *versioned id*.

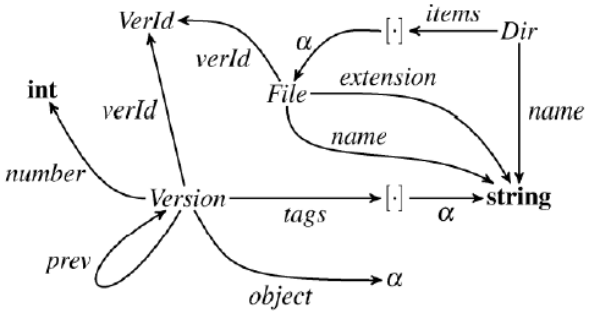

Fig. 6. The *Version*, *File* and *Dir* classes.

The versioning scheme manages an association between a versioned id and a real object as a *version*. Each version has a version number that is unique to the versioned id, a reference to the immediate previous version, and a set of string tags. While the version number tracks the update history, the set of tags can be used to select among the dialects of a versioned id. Fig. 6 shows the structure of the *Version* class. Given some tags and a versioned id, the object of the most up-to-date version can be determined by Alg. 1.

Alg. 1: Selecting an object by tags and a versioned id

Input *tags*: [**string**]; *verId*: **serial**
Output *object*: **serial**

$V \leftarrow \{v: Version \mid tags \subseteq v.tags \wedge verId = v.verId\}$
find $u \in V$ **where**
  $u.number = \max\{u'.number \mid u' \in V\}$
if *u* is found **then**
  $object \leftarrow u.object$
else
  object is undefined

Similarly, the naming scheme associates a (file) name and a (file) name extension with a versioned id, and this association is called a *file*. To support file paths, which are common to organize files in hierarchical structures, we treat a file name as a sequence of components split by the path separator '/'. A directory is just a mapping from a file name prefix to the list of files with this prefix. A table of directories can be built solely from the table of files, as shown in Alg. 2. This table of directories serves just as a cache. For database systems that support finding a range of data items, the list of files in directory D can be retrieved simply by finding the range of file names between

D+'/'+min(**string**) and D+'/'+ max(**string**),

and the table of directories is completely unnecessary.

Alg. 2: Building the directory cache

for $f \in File$ **do**
  $P \leftarrow$ split *f.name* by separator '/'
  $p \leftarrow$ empty
  for $p' \in P$ **do**
    $p \leftarrow p$+'/'+p'
    find $d \in Dir$ **where** $d.name = p$
    if *d* is found **then**
      update *d* in *Dir* with
        $d.items \leftarrow d.items \cup \{f.id\}$
    else
      add a new *d* to *Dir* with
        $d.name = p \wedge d.items = \{f.id\}$

While, within the database, data structures can refer to each other internally through object ids, which are unique numbers of type **serial**, we can force objects to only be accessed from the outside world through files and versions. In this case, if an object is to be retrieved externally, we must assign a file to the object. Versioning is automatic for objects of the same file.

IV. Importing and Exporting 3D Models

Model data must be imported into the model database. A model is a collection of material groups, and the texture maps referred to by the materials are separately imported. Models and texture maps are the two kinds of top level data. The top level data are referred to by file names when they are imported. In this section, we give the algorithms based on the previously illustrated structures, to show how the model databases can be built by importing from model files, and how models can be exported back to model files. We adopt the commonly accepted Wavefront `.obj` file format as our base for 3D models [14]. For simplicity, we omit the structures and algorithms for materials, which contain too many trivial fields.

A. Binary Files

The content of a binary file is stored as a byte array in the *Raw* table, and the file name is stored in the *File* table, associated with a versioned id. A new version is created in the *Version* table linking the versioned id with the corresponding byte array. Alg. 3 shows how to associate a file name and a versioned id with an object id, and Alg. 4 imports the binary data.

Alg. 3: Importing a file

Input *name*, *extension*: **string**; *id*: **serial**

// add a new file
find $f \in File$ **where**
  $f.(name, extension) = (name, extension)$
if *f* is not found **then**
  add a new *w* to *VerId*
  add a new *f* to *File* with
    $f.(name, extension, verId) = (name, extension, w.id)$

// add a new version
find $v \in Version$ **where** $v.verId = f.verId$
if *v* is found **then**
  add a new v' to **Version** with
    $v'.(prev, number, tags, verId, object) = (v.id, v.number+1, \varnothing, f.verId, id)$
else
  add a new v' to *Version* with
    $v'.(prev, number, tags, verId, object) = (0, 1, \varnothing, f.verId, id)$

Alg. 4: Importing a binary block

Input *name*, *extension*, *mimeType*: **string**; *bytes*: **binary**

// add a new binary block
add a new *b* to *Raw* with

b.(*type*, *bytes*) = (*mimeType*, *bytes*)

```
// add a file and version
```
call Alg. 3 with *name*, *extension* and *b.id* to import the file

B. Numeric Tuples and Faces

There are many numeric tuples in a model, such as points and normal vectors. These tuples are stored separately from the model, and referred to by their ids. In the database, the ids are stored along with the tuples, while in memory, the ids are indices into the arrays of the tuples. When we import the tuples from an array into the corresponding database table, we need to establish a map from the array indices to the ids in the table. This map is used when importing other data that refer to the tuples by ids. Alg. 5 shows the procedure to import an array of points. Other numeric tuples are handled similarly.

Alg. 5: Importing an array of points (tuples)

Input *P*: [(**double**, **double**, **double**, **double**)]
Output M_P: **int** ↦ **serial**

$M_P \leftarrow \emptyset$
for (*x*, *y*, *z*, *w*) = $P[i] \in P$ **do**
  add a new *p* to *PointDbl* with
  p.(*x*, *y*, *z*, *w*) = (*x*, *y*, *z*, *w*)
  add an entry $i \mapsto p.id$ to M_P

The components of a face are indices of numeric tuples. When importing a model, these indices are local to the imported data, so we need to convert them to global indices of the database. See Alg. 6.

Alg. 6: Importing a face

Input M_P, M_N, M_U: **int** ↦ **serial**; *V*: [(**int**, **int**, **int**)]
Output *id*: **serial**

$V' \leftarrow \emptyset$
for $(j_{point}, j_{normal}, j_{uv}) = V[i] \in V$ **do**
  $V'[i].(j_{point}, j_{normal}, j_{uv}) \leftarrow (M_P[j_{point}], M_N[j_{normal}], M_U[j_{uv}])$
add a new *f* to *Face* with *f.vertices* = *V'*
id ← *f.id*

C. Models

A 3D model consists of meshes (faces), materials and texture maps. We separate the latter two because of that texture map files can be manipulated independently. We import the texture maps as binary files, followed by the materials, then the arrays of numeric tuples, finally the groups of faces with each group associated with a material. Alg. 8 details this procedure. We can then call Alg. 3 to associate a model with a file name.

Alg. 7: Importing a mesh

Input M_P, M_N, M_U, M_A, ($j_{material}$, *matrix*, *S*)

$id_a \leftarrow M_A[j_{material}]$
add a new *trans* to *Trans* with *trans.matrix* = *matrix*
$S' \leftarrow \emptyset$
for each face $f \in S$ **do**
  $id_f \leftarrow$ call Alg. 6 with M_P, M_N, M_U and *f* to import the face
  add id_f to S'
for each submesh $s \in S$ **do**
  $id_s \leftarrow$ call Alg. 7 with M_P, M_N, M_U and *s* to import the submesh
  add id_s to S'
add a new *ms* to *Mesh* with
  ms.(*material*, *trans*, *items*) = (id_a, *trans.id*, *S'*)
id ← *ms.id*

In the algorithms, we use M_P, M_N, M_U, M_A to denote the maps from an index to a point, a normal vector, a UV mapping and a material, respectively. These maps are generated in the importing of the corresponding model components.

Alg. 8: Importing a model

Input texture map files, points, normals, UVs, materials and meshes
Output *id*: **serial**

call Alg. 4 to import the texture map files
$M_A \leftarrow$ the map from a material index to a material *id*
M_P, M_N, $M_U \leftarrow$ call Alg. 5 to import the points, normals and UVs
$S \leftarrow \emptyset$
for each mesh *s* **do**
  $id_s \leftarrow$ call Alg. 7 with M_P, M_N, M_U, M_A and *s* to import the mesh
  add id_s to *S*
add a new *mo* to *Model* with *mo.meshes* = *S*
id ← *mo.id*

D. Exporting

Model data must be exported from the database into memory before they can be processed by a program. A model consists of a list of meshes, each of which has a material, a transformation matrix and a collection of faces and submeshes hierarchically structured in a tree. The vertices of the faces and the materials of the meshes are separately stored in arrays. The faces and the meshes respectively refer to the vertices and the materials by array indices. Also, several materials can share some texture maps, therefore, texture maps of a model are stored in an array and referred to by array indices similarly.

Alg. 9: Exporting a mesh

Input *id*: **serial**
Output (*mat*, *fs*, *sms*, *ps*, *ns*, *uvs*)

fs, *sms*, *ps*, *ns*, *uvs*, M_P, M_N, $M_U \leftarrow \emptyset$
find *ms* ∈ *Mesh* **where** *ms.id* = *id*
for $id_f \in ms.items \mid id_f \in Face$ **do**
  find $f \in Face$ **where** $f.id = id_f$
  $f' \leftarrow \emptyset$
  for (id_P, id_N, id_U) ∈ *f.vertices* **do**
    for (*id'*, *M*, *T*, *A*) ∈ {(id_P, M_P, *Point*, *ps*), (id_N, M_N, *Normal*, *ns*), (id_U, M_U, *UV*, *uvs*)}
    do if $id' \mapsto * \notin M$ **then**
      find $r \in T$ **where** *r.id* = *id*
      add $id' \mapsto |A|$ to *M*
      append *r* to *A*
    append ($M_P[id_P]$, $M_N[id_N]$, $M_U[id_U]$) to *f'*
  add *f'* to *fs*
for $id_s \in ms.items \mid id_s \in Mesh$ **do**
  s ← call Alg. 9 with id_s
  add *s* to *sms*
find *tr* ∈ *Trans* **where** *tr.id* = *ms.trans*
mat ← *tr.mat*

To export a model, we need to export the materials, the vertices and the meshes. As we export a mesh, we accumulate those vertices that are referred to by the faces of the mesh, together with the maps from *id*s to array indices. Vertices are not shared by the faces in different submeshes. The

construction of a mesh is recursively done. Alg. 8 lists the steps, where we respectively denote the matrix, faces, submeshes, points, normal vectors and UV mappings as *mat*, *fs*, *sms*, *ps*, *ns*, *uvs*.

We export a model by constructing the materials and the meshes, and by accumulating the arrays of shared vertices, materials and texture maps.

V. Caching

To run an application off-line, or in the case to speed up the loading of data, we need to cache the data retrieved from the database and package them for saving in the local storage. With the locally cached data package, the application can run in the same way, at least with only a few parameter changes, as if the data are still retrieved from the database. The packaging and delivering of the cache data are obvious. The only problems are to figure out what data need to be cached, and how the originally successful queries can be redirected to the cached package in the off-line executions.

An application issues queries through resource locators, which are actually text strings [15]. Since resource locators are interpreted, it is very convenient to associate the data retrieved from the database with the resource locator in the interpreter. When caching is required, we build a map from resource locators to exported models, and other data records can also be handled similarly. Then the map can be stored in the application requesting the local copy of the cache. When the application runs off-line, the resource locators are redirected to the local copy of map, and the associated data records are returned as the results.

However, when we review our graph-based design of the 3D model database, we can easily find out that there are many shared pieces of information. Saving a data record for each resource locator produces many duplicated data members. To address this space-efficiency issue, we further analyze the data records stored in the database. We find that each data record starts from a node in the object graph representing the database. This node is called the root of the object representing the data record. Instead of associating the entire data record with the resource locator, we associate the root node. At the same time, we maintain an in-memory subgraph of the entire object graph of the database as the cached dataset. This can be done when we retrieve the data records. We view a data retrieval as a navigation in the object graph, when we visit an edge of the graph, we add it to the in-memory subgraph. That way, when we finish retrieving the data, we have a subgraph that contains all the edges and nodes we need for the cache. We then save this map and subgraph for local off-line executions.

Alg. 10: Construction of the data access subgraph

Reference E: **serial** × **string** ↦ **V**

for each successful (**find** r **where** $r.id$ = requested id) **do**
  for each field $f \in r$ **do**
    add an edge ($r.id$, name of f) ↦ $r.f$ to E

We use an edge list to represent the subgraph. The list can be formulated as a map from an *id* and a field name to another *id* or a value, **serial** × **string** ↦ **V**, where **V** = **serial** ∪ **int** ∪ **double** ∪ **decimal** ∪ **string** ∪ **binary**. Arrays are expanded to edges from an *id* and an index to another *id* or a value, i.e., **serial** × **int** ↦ **V**, We show this process in Alg. 10.

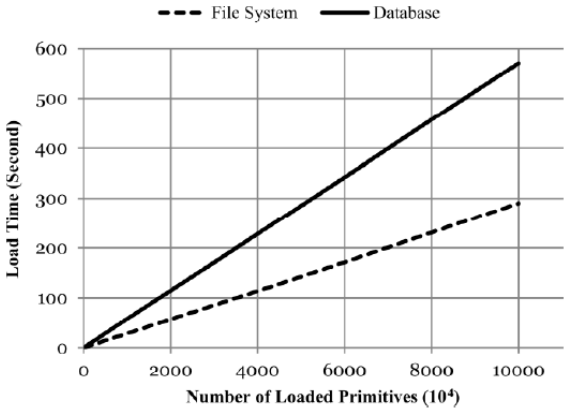

Fig. 7. Database vs. file system load time comparison.

With the data subgraph, the map from resource locators to data records is simply a map **string** ↦ **serial**. This map can be constructed when we interpret a resource locator successfully and have a result. Once these maps are stored locally, data records then can be retrieved by navigating the subgraph without any further access to the database.

VI. Evaluation

While a model database provides the convenience in indexing and collecting of 3D models, it has certain performance impact on loading models into 3D application programs, compared with direct sequential component read from model files. This is because the components of a model must be individually selected from the database. For this particular database design, when loading a model, data rows are mainly selected by their *id*s, which can be set as the primary keys of the tables. Therefore, logarithmic time of locating a requested row is expected, hence the overhead of selecting components should only have a near constant factor on the model load time, when the model dataset becomes very large.

We implemented a minimal 3D model database according to the graph-based design presented in this article, and compared the time of loading various numbers of complex 3D models from the database with that from a file system. The configuration of the database is listed in Table VII.

TABLE VII: Configuration of the Testing Platform

Database Engine:	PostgresSQL 9.3
Number of Tables:	17
Linux Version:	3.5.0-23 (gcc 4.7.2) Ubuntu 12.10
Processor:	Intel(R) Xeon(R) E5-2640@2.50GHz
Memory:	32GB
Total Primitives:	10^8

The database server was connected to the test computer via LAN. The test application loaded a series of model packages from both the database of 10^8 total primitives and the local file system, and measured the load times. The packages were of sizes up to 10^4 models each consisted of 10^4 primitives. The comparison result is shown in Fig. 7. We observe a constant factor performance hit with the database, and both load times

are linear to the size of the loaded model package.

Based on the experiment, as many VR applications are still using file systems to manage their 3D resources, we can expect that moving to our database will not introduce unacceptable negative effect on the model loading performance, while we can access the ever growing number of 3D models with almost no further maintenance cost, through using the model database.

VII. Conclusion

We present a graph-based method for designing a long running open 3D model database aiming at accumulating 3D models for both current and future virtual reality applications, particularly those involving large scale city scenes. The entire database is represented as a large object graph following the structure defined by the class graphs of the database. We provide the graph-to-relational mapping for storing the model data in a traditional relational database. The mapping is fully open that the data in the relational database can be accessed independently by other applications. We design and implement the graph notation and the mapping scheme with the support of arrays and variants, and provide key algorithms to manipulate the data. We add the versioning and naming scheme using the graph-based method to show the flexibility. This scheme is simple but powerful enough to handle versioning and tagging, and is able to simulate the tree structure of a conventional file system. Many VR applications, such as 3D game engines, that use file paths to locate resources may find this naming scheme handy. For applications that need to run off-line of the database, we give a general method to cache the application traversed data as a subgraph, and the applications can use this method to store the cached subgraph locally for future off-line executions. We also use a minimal implementation of the database to evaluate the model loading performance, compared with a conventional local file system.

Future work. Based on our rigorously and completely defined graph notation, we can build graphical tools in the future to help users use the graph-based method visually. We also consider building tools that read the graph-based design and generate the tables in the relational database automatically.

References

[1] M. Portman, A. Natapov, and D. Fisher-Gewirtzman, "To go where no man has gone before: Virtual reality in architecture, landscape architecture and environmental planning," *Computers, Environment and Urban Systems*, 2015.

[2] K. McHenry and P. Bajcsy, "An overview of 3d data content, file formats and viewers," *National Center for Supercomputing Applications*, vol. 1205, 2008.

[3] W. Ke, Z. Liu, S. Wang, and L. Zhao, "A graph-based generic type system for object-oriented programs," *Frontiers of Computer Science*, vol. 7, no. 1, pp. 109–134, 2013.

[4] R. Bruni, Z. Liu, and L. Zhao, "Graph representation of sessions and pipelines for structured service programming," *Formal Aspects of Component Software*, Springer, 2012, pp. 259–276.

[5] A. Seering, P. Cudre-Mauroux, S. Madden, and M. Stonebraker, "Efficient versioning for scientific array databases," in *Proc. 2012 IEEE 28th International Conference on Data Engineering (ICDE)*, IEEE, 2012, pp. 1013–1024.

[6] J. Juneau, "Object-relational mapping," *Java EE 7 Recipes*, Springer, 2013, pp. 369–408.

[7] N. Kojic and D. Milicev, "A survey of object-relational transformation patterns for high-performance UML-based applications," in *Proc. 2015 3rd International Conference on Model-Driven Engineering and Software Development (MODELSWARD)*, IEEE, 2015, pp. 280–285.

[8] E. J. O'Neil, "Object/relational mapping 2008: Hibernate and the entity data model (edm)," in *Proc. the 2008 ACM SIGMOD International Conference on Management of Data*, ACM, 2008, pp. 1351–1356.

[9] W. Xu, Q. Zhu, Z. Du, and Y. Zhang, "Design and implementation of 3D model database for general-purpose 3D GIS," *Geo-spatial Information Science*, vol. 13, no. 3, pp. 210–215, 2010.

[10] T. Płuciennik and E. Płuciennik-Psota, "Using graph database in spatial data generation," *Man-Machine Interactions 3*, Springer, 2014, pp. 643–650.

[11] R. Angles, "A comparison of current graph database models," in *Proc. 2012 IEEE 28th International Conference on Data Engineering Workshops (ICDEW)*, IEEE, 2012, pp. 171–177.

[12] D. R. Musser and A. A. Stepanov, "Generic programming," *Symbolic and Algebraic Computation*, Springer, 1989, pp. 13–25.

[13] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein, "Chapter 21: Data structures for disjoint sets," *Introduction to Algorithms*, pp. 498–524, 2001.

[14] Appendix B1. Object Files (.obj), Advanced Visualizer Manual. [Online]. Available: http://www.cs.utah.edu/boulos/cs3505/obj_spec.pdf

[15] AssetManager, jMonkeyEngine Documentation for Advanced Users. [Online]. Available: Available: http://wiki.jmonkeyengine.org/doku.php/jme3:advanced:asset_manager

Wei Ke received the PhD degree from School of Computer Science and Engineering, Beihang University. He is an associate professor of computing program, Macao Polytechnic Institute. His research interests include programming languages, functional programming, formal methods, and tool support for object-oriented and component-based engineering and systems. His recent research focuses on the design and implementation of open platforms for virtual reality applications, including programming tools, environments, and frameworks.

Lap-Man Hoi got his bachelor degree in computer science at York University, Canada, and the master degree in internet computing at the University of London. He is now working as a researcher in Macao Polytechnic Institute. The main duties are researching in the gaming industry and teaching the computing courses.