RN/12/01
6 February 2012

Research
Note

Revision Control Database for 3D Assets

Jozef Doboš and Anthony Steed

Telephone: +44 (0)20 7679 3308
Fax: +44 (0)20 7679 1397
Electronic Mail: {j.dobos, a.steed}@cs.ucl.ac.uk

Abstract

Even though revision control has been successfully deployed for text-based files for many years, it does not efficiently map to 3D assets. In this research note we propose a non-linear concurrent revision control system for 3D assets. Scenes are represented as collections of polymorphic objects alongside their corresponding history in a database, which map existing revision paradigms to a scene graph manipulation. Our open source implementation leverages a NoSQL database and a 3D asset import library. Similarly to existing version control systems it works independently from editing software and does not require any knowledge of user edits. We demonstrate the feasibility of our framework by performing concurrent 3D model modifications.

Keywords: 3D database, NoSQL, MongoDB, asset storage, 3D modeling, revision control, asset management

Department of Computer Science
University College London
Gower Street
London WC1E 6BT, UK

1 Introduction

The construction and maintenance of large 3D models has long been a problem in the graphics community. With recent progress in graphics processing units, there is a desire for more detailed and more extensive models. Whilst engineering or general scientific visualisation has long provided massive datasets that are procedurally generated, in domains ranging from heritage through to games, there is a growing need to maintain massive 3D models that might be edited by multiple users concurrently.

The main motivations for this work are the drawbacks of the current dominant paradigm where 3D models are stored as files on a file system. Each user loads a particular model into an editor or mesh editing tool such as Autodesk Maya or Blender, makes some changes and then re-saves the whole file. This means the unit of access to the scene is the file. An individual file might be edited dozens or hundreds of times a day. If two users want to edit the same part of the scene they need to access the same file, but without extra supporting infrastructure, managing this access and resolving conflicts becomes difficult. Furthermore, deciding how a large scene should be partitioned into multiple files is a tricky decision. At one extreme the scene might be one file, but then machine memory is a strict limit and having multiple users edit is almost impossible. At the other extreme individual objects or meshes in the scene might be single files. In this case managing the scene ensemble becomes an important issue: how do edits that involve two objects get managed and how are the different object versions kept in synchronisation? On top of this, files themselves do not provide any management of the history of edits, so some other mechanism is needed on top of this. Revision control and asset management systems are common approaches to this (see §2).

There is a need for a centralised yet flexible system for managing 3D assets. This system would need to support multiple users, distributed access, flexible locking of pieces of models, efficient storage, control of revisions of the model, interface with existing packages and not enforce any particular modeling paradigm on the users.

Our proposal is to exploit recent developments in database technology to construct a single service that provides the needs outlined above. We propose using a NoSQL database management system to store complete scenes and their revision histories. Such databases avoid rigid table structures and tend to be optimised for large read/write operations [Membrey *et al.*, 2010]. We can thus easily store 3D models in the database, but further, because the database is very flexible, we can store other relationships over the models such as semantic data and even revisions. Once the scene is in a database access is through a query language and is implicitly supported in a distributed manner. We can add various types of locking to objects to support concurrent editing. Finally, we can use all the standard support that databases provide such as data integrity and robust storage (see §2 for more details).

Integrated revision control and editing has recently been studied for image editing [Chen *et al.*, 2011]. They provide a plug-in to an image editor that can track user actions and then deal with multiple revisions. Our work targets a different type of asset, 3D models, but also it is our belief that we should not, initially, rely on any specific 3D modeling tool, but deal with 3D files external to the editor. Whilst we can have a very efficient system with several advantages by integrating directly as a plug-in to an editor (see §6), we also deal with the problem of loading assets from files and comparing them to an existing database. Thus we start by supporting normal file-based editing, but with the advantages that we can interface with a database to manage the complete model.

Therefore, the main contribution of this note is the demonstration of the feasibility of a framework for a unified revision control of 3D assets using a database system. In demonstrating this we show:

- How 3D assets can be managed within a NoSQL database (specifically MongoDB).
- How a revision history system can be built on top of that database.
- An interface for loading models into the database, querying the database, exporting files, doing differences between versions and previewing models.

We demonstrate our system on moderate sized examples because this is a proof of concept. We feel confident in arguing that our approach would be suitable way of establishing a large-scale collaborative 3D development and visualisation. Our tool is already useful for management and integration of a moderate sized (few million) polygon models, in that multiple versions can be stored in one place, and different users can access and model remotely. Given that the type of database we are using is proven to scale to massive services, the potential long-term benefits are a scalable way of having large number of editors on a single model, or even open crowd-sourcing of models.

The rest of this research note is structured as follows. In Section 2 we discuss the wide range of related work on databases, network protocols and asset management. In Section 3 we outline the concept of a 3D database and give an overview of the interaction between it and the editing tools. In Section 4 we give details of our prototype implementation and design decisions made to cope with typical 3D model assets. In Section 5 we give details of typical user stories of interaction with the 3D database and in Section 6 we give an evaluation where we discuss the support achieved for the user stories, and implementation challenges of scaling the system.

2 Related Work

2.1 File Systems and Databases

An important debate, which we do not have space to do justice here, is the relative merits of using file systems versus databases for storing large data. As a simplification, a file system provides a mechanism to store, locate and retrieve data (files) on local or remote devices. A distributed file system provides transparency of access to files when the actual storage device is hosted on different servers, whereas a local file system provides fast access to local devices [Carrier, 2005]. Whether a file is stored on a local or a distributed device has a potentially huge impact on retrieval and access speeds, but distributed devices are commonly used to provide data reliability. Common issues with file systems include how to support user control of access, how to support atomic or concurrent edits and so on. File systems usually come with some access control method and various types of mechanisms for managing large storage (share names, symbolic links, etc.).

There are several issues with using a file system to support large-scale modeling. The first is revision control. Whilst revision control systems are very popular, they aggregate edits at a per-file level. This means that when assets are changed a whole file is changed. If the model is stored as ASCII, it may be that the edits can be applied on a per-line basis. This type of edit is well supported in revision control systems that originated in supporting source-code. However, the scale and type of edits mean that the ASCII changes might be quite large and pervasive across the file, and in any case many 3D model formats are binary. In the worst case a very minor edit on a file might mean a whole new version needs to be tracked. Thus when creating a file which one expects to place under a revision control, it is important to get the size of the file correct: does one make every object a file, or should it be regions of a model? The choice is important because whatever is done, somewhere a description of how the files fit together must be stored. It is common to have a scene description separate from individual models. There might be an independent description format, though some formats such as X3D [Brutzman and Daly, 2007] can recursively include files and thus can serve both roles. Another problem with files is that certain modeling operations will require loading of several files and editing programs tend to only load one file at a time, or if they support multiple files, this requires careful management (e.g. XRefs in Autodesk 3ds Max). This then leads to the second issue which is that the file system itself does not provide facilities to track assets. Such information must be stored in some other form. Thus as we discuss later in this section, a range of asset management systems has emerged to manage the roles of both revision control and asset tracking. The third issue is access control. This is complicated by the operating system that the file system is run on. Unix-like permissions or access control lists can be useful for managing per-file data access, but these can be difficult to manage, especially when remote access is supported [Tanenbaum, 2009].

There are several potential advantages of a database-based approach. First, the database provides a central

point of control. The database itself might support multiple servers, caching, etc., but logically there is one point where the data and control come together. Second, databases are naturally designed to be used either locally or over a network. Third, the unit of access is under the control of the database designer. This means there is some flexibility in how one stores data: it could be a binary blob per-mesh or as a single entry per-vertex. The choice here is something we return to later. We can integrate into the database sophisticated user access control and can create locks on objects. Finally databases support some or all of the ACID principles [Connolly and Begg, 2004], so can preserve data integrity.

There are arguments against using a database approach. The main one is that loading from a file could be faster than loading from a database. Few file formats are described in a way that can be directly memory mapped; most require some form of single pass or multi-pass parsing to construct a consistent in-memory representation suitable for further processing. Also this does not mean that database access cannot be fast. The database could store binary data that can be loaded into memory and treated as parts of files for parsing or as arrays for passing to a rendering API.

2.2 Asset Management and Version Control

Management of graphics assets is an important facility in many industries. Whilst some guidelines and descriptions of best practices exist [Jacobsen *et al.*, 2005, Austerberry, 2006], there are few standards. The general process can be split into two main tasks: storage and tracking. Storage would typically refer to named files on a file system. Tracking would refer to keeping a history of the names and metadata associated with the files. This might include using a strict naming convention or a fixed file system hierarchy. Optionally, the storage facility might include a revision control system to reduce the amount of data stored. In some cases the tracking facility might use a database to store file asset histories as done is some games companies. If, for example, PerForce is used to store files and Oracle to track history and maintain a scene, creating game levels thus involves querying the database to get a set of assets, copying these locally and then processing before loading onto a target console or a PC.

Some commercial tools provide asset tracking. For the games industry tools such as Temerity Pipeline or Alienbrain provide digital asset management and other facilities [Jacobsen *et al.*, 2005]. Unity3D, for instance, offers *Asset Server* which is a networked service for delivering assets that can utilise various revision control systems for permanent storage [Unity Technologies, 2010]. High-end CAD also provides similar functionality, for example Dassault Systemes' Enoviva [Dassault Systemes, 2011] and Bentley's AssetWise [Bentley Systems, 2011].

For relatively small projects a simple revision control system is a popular option. Especially in a demonstration that includes source code, there is a temptation to put everything in one repository. However, revision control systems of the type that are commonly used (e.g. Git [Loeliger, 2009] or Subversion [Pilato *et al.*, 2008]) are better at managing text assets than they are at binary assets. They use difference tools (e.g. simple *diff*) to manage changes. Thus they tend to be extremely good at managing and integrating source code changes, but not binary assets [Hunt *et al.*, 1998].

Whilst these works are very promising they are either not open, are difficult to implement, require logging of the user actions or deal with scenes at an inflexible, per-file, level. Our proposal is to create one database with the whole scene and the revisions to the scene. The database unifies access and control. Potentially it can save very significant storage over an equivalent system using file-based revision control. It is also extensible to include all types of assets including images, materials, shaders, etc.

2.3 Editing Control

A problem related to asset control and dealing with revisions is how users can manage different versions of a file so that they can collaboratively edit [Dourish, 1995]. If the changes take place on different parts of the file (e.g. a text document), then the intention of both edits can be preserved by merging the edits. The Google Docs application is a good recent example of this type of editing [Dekeyser and Watson, 2006]. In

general, preserving intentions of edits is more difficult with binary data or more complex data structures. If a 3D model was stored in a binary format, it is unlikely that two sets of binary changes to the file could be merged successfully.

One approach is to track the editing operations that the user makes on a file, with the expectation that a record of the operations can be used to inform the merge process [Wäsch and Klas, 1996]. A similar concept has been realised by [Chen *et al.*, 2011] recently for managing 2D image editing.

Initially presented by [Kurlander and Feiner, 1992], we regard a sequential collection of commands in a graphical interface an *edit history*. VisTrails Provenance Explorer for Maya, unlike its open source counterpart for scientific workflow management, offers a proprietary plug-in solution for Autodesk Maya software [VisTrails, Inc., 2012]. Despite capturing the edit history within its MySQL database and allowing the users to share them over the Internet, the actual 3D assets have to be stored locally. To visualise changes across various versions this program provides a basic differencing in a side-by-side preview. In contrast, a very recent work by [Denning *et al.*, 2011] integrates an edit logger inside Blender modeling package. By clustering edits via several layers of regular expressions they enable interactive playback of modeling history. This system, however, deals only with simple meshes and covers only a subset of modeling techniques.

Whilst we do wish to preserve intentions of users in their editing operations, we also need to deal with the wide range of editing tools that are used for 3D models. Thus we have decided not to explore the route of recording 3D editing operations and instead deal with the fact that we will be importing and exporting files via a database. Edit tracking is an avenue for future work as once we start using plug-ins to connect directly to the database, we can use edit tracks to scope changes (see §6.2.2).

2.4 Networked Protocols

Related to storage and control of assets is the way in which they are transferred over the network. A database implicitly supplies a query language and this can usually be accessed over the network. The support of distribution of graphics over networks is itself a large research area [Steed and Oliveira, 2009]. Of particular interest are systems that pass complete scene descriptions alongside scene edits. Systems such as Distributed Open Inventor [Hesina *et al.*, 1999] supported distribution of a full scene graph and subsequent real-time edits. A related concept, the use of a scene graph as a data format to inter-mediate between different applications, was presented by [Zeleznik *et al.*, 2000]. Support for live editing and viewing across a network was also implemented in the Uni-Verse system that used the Verse streaming protocol [Brink and Steenberg, 2011]. Uni-Verse supported sharing of 3D geometry, HDRI textures and shaders.

Each of these network protocols demonstrates that changes can be propagated, but this is only one problem in distributed editing. However such protocols could be useful in future work that considers the bandwidth requirements of database transfers.

2.5 Spatial Databases

A final area of related work is that of spatial databases [Rigaux *et al.*, 2001]. These databases originate from the geographic information systems domain. They store geographic features (points, lines, areas, etc.) and data about those features (heights, names, and so on). Whilst they can store a wide variety of data they are structured to provide access on a per-feature basis. For example one might have individual road centre-lines accessible independently in a database. A few commercial and open source databases support spatial data directly, in particular Oracle Spatial [V. Kothuri *et al.*, 2012] and PostGIS for PostgreSQL [Obe and Hsu, 2011].

Compared to spatial databases, our proposal is focused on distributed access to and editing of 3D geometry suitable for 3D content generation pipelines. Spatial databases also focus on 2D geometry and metadata, not 3D models and visual properties. They do not support the type of revision histories that we need.

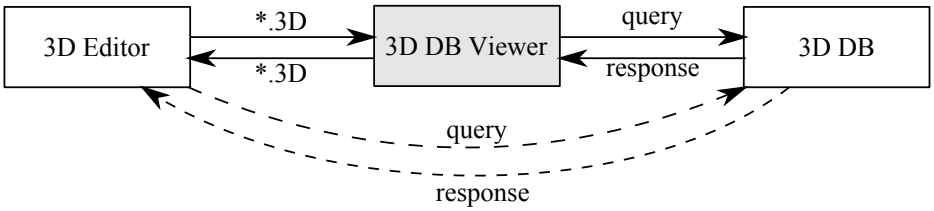


Figure 1: Conceptual system overview. 3D modeling package (left) exports a 3D file locally which is loaded into our 3D DB Viewer (middle). This acts as a front-end to the 3D database (right) and facilitates the revision control functionality. In the future, the 3D DB viewer could be bypassed by implementing a plug-in based database connection inside a 3D editing software, see dashed lines. For implementation details, see Figure 3.

3 System Overview

As shown in Figure 1, our revision control framework for 3D assets facilitates a connection in between the standard modeling packages and a dedicated centralised database for 3D assets, *3D DB* for short. Unlike distributed versioning systems such as Git, we focus our efforts towards a centralised model, because having a copy of an entire history of a large 3D scene across every involved computer would become infeasible. This follows examples set by the modeling and games industry, where centralised repositories store databases of hundreds of gigabytes of data, see §2.2. A single storage location simplifies data management but also enables a quick way of visualising the latest changes in the editing process.

Standard modeling packages edit and export a 3D file to a disk. This is subsequently loaded into a *3D DB Viewer*, which displays currently selected revision from a database as well as the locally modified file. In addition, it automatically detects updates and conflicts and commits the selected ones to back the database as a new revision. Furthermore, it supports branching and tagging and can query any full or partial revision from the DB.

In the future, we plan to bypass the need for a stand-alone viewer and provide a plug-in solution to connect to a 3D DB directly. Advantages of such approach are discussed in §6.1.

3.1 3D Database

Spatial databases offer flexibility for generic spatial and proximity queries, however as already mentioned in Section 2.5, they introduce several limitations, the most obvious being a consumption of significant amount of storage space, hence reducing the performance of the underlying database. 3D files, on the other hand, group every scene or object into a single binary or plain-text representation trading flexibility for storage efficiency. What is more, individual files do not support sub-object queries and if editing an object, the whole file has to be loaded into modeling software.

To our advantage, large models tend to encompass several separate meshes with interconnected semantic meanings, such as the building example in Figure 2. This model consists of separate meshes for roof, interior walls, glass and so on. We exploit this natural partitioning of 3D assets and build versioning framework upon it. Treating each part of an asset as an individual binary "blob" with preserved relational information with regards to other components, we achieve the desired flexibility and find a suitable compromise in between file storage efficiency and querying potential of spatial databases. By design, such polymorphic representation supports all common constituents of 3D assets including meshes, transformation matrices, cameras, lights, textures as well as animations, bones and shaders. It should be noted that some of these can be extremely large, so we acknowledge that storing them would be non-trivial and leave it as an open research question for future work. Our implementation, for example, is currently limited to 16MB per-object, see §6.1.

3.2 Scene Graph Representation

For our purposes a scene graph (SG) is a directed acyclic graph (DAG) in which individual nodes are linked in an interconnected structure of logical, hierarchical and spatial relationships. Unfortunately, there is no commonly accepted scene graph structure and different implementations assume various simplifications and restrictions of scene/node hierarchy. Therefore, we impose no restrictions on parental relationships of individual nodes (which can be of any arbitrary type) as long as individual objects appear in local coordinate system with an associated path that leads to a global transformation. Such assumption facilitates automatic edit merging as discussed in §5.1.

3.3 Revision Control System

Alongside a linear history path, a non-linear revision control system allows for branching and merging to occur, effectively creating a DAG. Therefore, if a scene graph can be stored in a database, so can be a revision history. In this graph, each node signifies a single incremental revision storing delta changes and can be either of type `trunk` to represent the main development path or `branch` for additional possible paths. This DAG stores edit histories in a similar way to other versioning systems such as SVN.

All parents of a node in a revision history define a dependency relationship and indicate where versions of documents, i.e. the individual binary constituents of a 3D asset, from SG that were not updated in that revision can be found. The smallest unit of change in our system is a *polymorphic binary document*. Even though a single vertex change in a mesh would make the system store another version of its binary document, we still consider it a delta increment, because it is only a partial change on the whole SG, hence not requiring to copy the entire original 3D asset over again. This design decision offers compacted storage and simplified data access implementation. Nevertheless, disadvantages do exist and are further discussed in §6.1.

4 Prototype Implementation

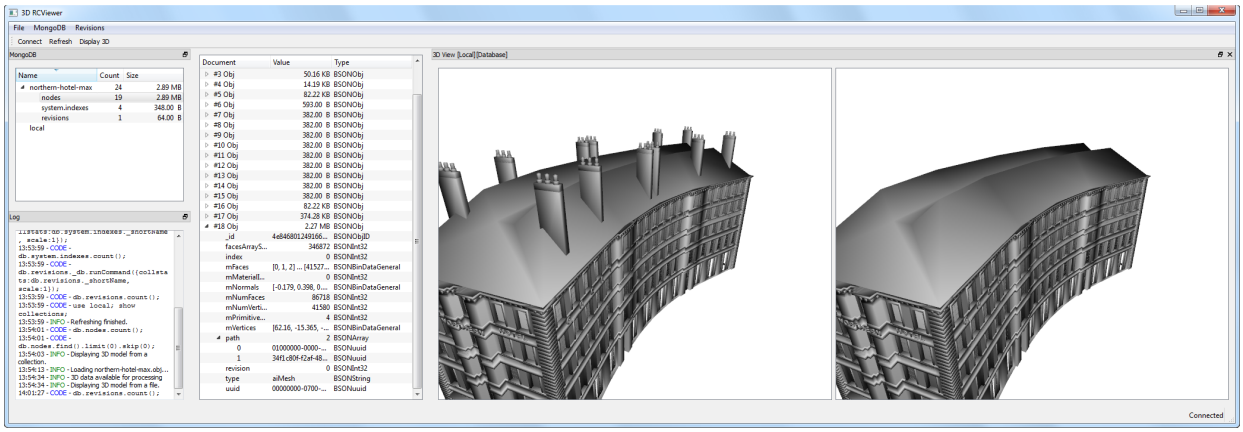


Figure 2: 3D Revision Control Viewer (3D RCViewer) GUI application enables DB inspection, retrieval of individual revisions as well as a side-by-side comparison of a locally modified file (left) and a DB revision (right). Each 3D object is stored in a single database with two collections (DB tables): `nodes` for scene graph and `revisions` for revision history graph. Changes are automatically detected and a list of modifications is displayed during the *Commit* dialog box.

Dedicated graph databases such as Pregel [Malewicz *et al.*, 2009] are suitable for large-scale distributed processing and utilise the *Map/Reduce* model [Dean and Ghemawat, 2008] by mapping tasks in parallel and contributing to a single reduction step that generates the results. However, we do not require complicated graph traversal computations, but rather an efficient and most importantly unified way of encoding various object-like data.

Unlike traditional relational database management systems, NoSQL databases store collections of structured data offering great flexibility and ease of access. Based on these advantages, we implemented our prototype as a user interface to a standard release MongoDB [Membrey *et al.*, 2010] instance. MongoDB is an open source C++ NoSQL database that utilises *Binary JSON (BSON)* [bso, 2012] documents for storage, querying and data transmission. By using C data types, BSON objects can easily be decoded making the database highly responsive. In addition to standard data types, BSON offers several binary entries which are a suitable representation for large arrays of vertex coordinates, normals and faces. In addition, MongoDB provides full indexing, replica databases, auto-sharding and Map/Reduce functionality as well as enterprise support and even geospatial indexing making it an attractive choice for production environments. Wordnik, for example, is an online database of relationships (such as being a synonym, antonym etc.) of English words which stores a directed graph of over 12 billion documents with an average retrieval of 60ms in MongoDB [Wordnik, 2010]. Despite all the aforementioned benefits, using MongoDB is not a functional requirement of our framework and can be replaced by some other suitable database if desired.

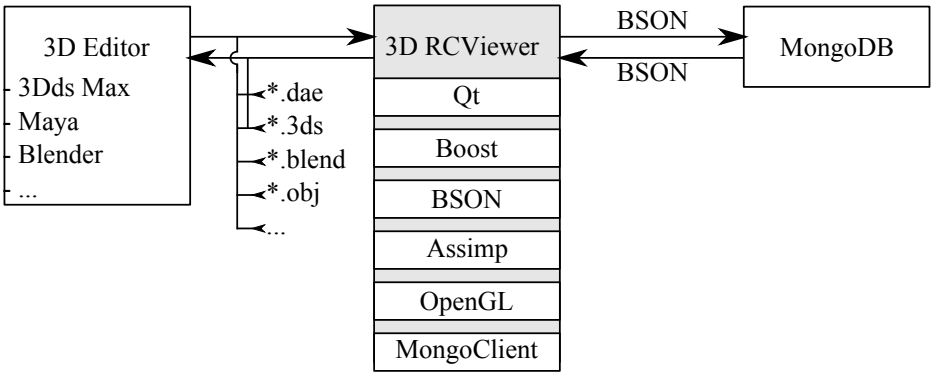


Figure 3: Implementation details overview. Standard 3rd party 3D editor exports a file locally which is subsequently imported into our 3D RCViewer. This in return communicates with MongoDB through BSON-based queries and responses. Please note that Assimp currently exports only Collada (*.dae) and 3ds Max 3DS (*.3ds) file formats, but imports many more.

4.1 Database Structure

Despite the schema-less approach of NoSQL databases, certain assumptions about common attributes have to be made. MongoDB consists of databases and collections (tables) storing potentially nested BSON objects.

For simplicity we propose to store documents belonging to a scene graph in a collection named `nodes` and documents belonging a revision history in a collection named `revisions`. Hence these collections store a single directed acyclic graph each, making the read and write implementation reusable.

To be able to identify each node in a graph within the database we propose to assign universally unique identifier (UUID) [Leach *et al.*, 2005] (a 128-bit number) to every component of a 3D asset including transformations, meshes and the like. Each document has to further carry the revision number it belongs to. In relational databases terms, a `UUID` together with a `revision number` represent a composite primary key on the `nodes` collection, so that each node can contribute exactly once to exactly one revision.

In order to represent a DAG in a flat collection of documents, we considered several common ways of storing graphs in databases:

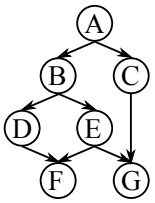
Parental and child links. Storing information either about immediate parents or children of a node in a tree or a graph requires recursive hierarchical querying to retrieve an entire sub-graph from the database. Unless there is a direct server-side DB support for this type of queries, such access would be very expensive.

Adjacency and incidence matrix. Representing graph connectivity as a two-dimensional boolean matrix with 1 for a node to node relationship and 0 otherwise requires complicated memory management of sparse matrices if dealing with very large graph structures.

Nested sets. Introduced by Celko [Celko, 2004], nested sets assign each node a set of two integers that represent boundaries of all of its children. In comparison to recursive hierarchical queries the sub-tree retrieval can be implemented using a single query. However, inserting a new entry into DB causes re-indexing all nodes' boundary information and is therefore expensive. This problem was solved by using real numbers (represented as quotients) to encode node positions within hierarchy [Hazel, 2008]. Unfortunately, this is only suitable for tree-like structures, where each node has at most one parent.

Array of ancestors and materialised paths. Storing in each node a full path from the root itself makes retrieval of entire sub-graphs very easy–all nodes with a given node in their path are its children. In a tree, there is always only a single path to any node, however, in an arbitrarily complex directed graph such as SG or revision history, each possible path from the root would have to be stored, producing unnecessarily duplicated data in the DB.

For instance, Wordnik database (see §4) stores directed graphs as two collections, one for nodes of the graph and the other for corresponding child links respectively [Wordnik, 2010]. Unfortunately, this representation does not allow single query graph traversal, but requires steps of recursive queries from one node to another.



A: [A]
B: [A, B]
C: [A, C]
D: [A, B, D]
E: [A, B, E]
F: [A, B, (D | E), F]
G: [A, ([B, E] | C), G]

Figure 4: An example of compressed materialised paths. Each node stores only one entry for all the paths from root (A) leading to it. Comma signifies logical `and` and vertical bar a nested `or` operator such that either of the other routes can be used.

In contrast, we modified the standard materialised paths model to prevent data duplication. In a path from the root to any node, if more than one possibility exists we store this information as a logical `or` operation on nodes, see Figure 4 for further explanation. Because we store no information about child connections in the documents, inserting a new node in part of a graph requires no modifications to existing entries in the database.

4.1.1 Revision Retrieval

In order to retrieve any revision from the history, only the latest instances of each individual node up to the selected point in time have to be requested. This can be achieved by querying unique document instances with matching revision number. If not present, the first of the immediate predecessors in descending order has to be returned and unrelated intermediate branches skipped. Assuming the knowledge of all possible revision paths to achieve given revision (this information is stored in each revision node), Algorithm 1 will return a list of the latest individual nodes from `nodes` collection that belong to a requested revision number.

Instead of requerying the database over the network multiple times, MongoDB offers a server-side JavaScript evaluation. Hence, the function outlined in Algorithm 1 can be stored on the server and evaluated efficiently. Unfortunately, MongoDB does not allow complicated $distinct$ function calls making it impossible to construct a single query that would find all the latest nodes belonging to a certain revision,

Algorithm 1 Revision Retrieval

Require: $revisions$ /* array of possible revision paths */
  $nodes \leftarrow distinct(\text{'uuid'})$
  for all $node \in nodes$ **do**
    $list \leftarrow find(\text{'uuid'} = node \wedge \text{'revision'} \in revisions)$
    $\ldots \leftarrow sort(\text{'descending'})$
    $\ldots \leftarrow limit(1)$
  end for
  return $list$

though such functionality has been requested. To retrieve only a sub-graph of any revision, one has to further search through a list generated by Algorithm 1 for all occurrences of `UUID` that is to be the root of the retrieved sub-graph.

4.1.2 Document Deletion

If a document in SG is to be deleted, a BSON object with an element of type *NULL* and the same `UUID` is inserted in the new revision. In addition, the entire sub-graph of the deleted document has to to be checked and if any document has no other path leading to it from the root, it also has to be recursively deleted. Similarly to revision retrieval, this can be executed through a server-side JavaScript evaluation.

4.2 3D RCViewer

Our front-end GUI for 3D database interaction and revision control (see Figure 2) is written in C++ and a cross-platform UI framework *Qt* [Blanchette and Summerfield, 2008]. We use the *Open Asset Import Library (Assimp)* [Schulze *et al.*, 2012] to convert common 3D file formats such as Collada, Blender 3D, 3ds Max 3DS, Wavefront Object and similar into a file-format independent in-memory scene representation. We can easily convert between this representation and that required in BSON. The GUI facilitates all revision control functionality including modification detection and commits. For a full list of library dependencies see Figure 3.

4.2.1 Data Comparison

Due to several of the stored data items being represented in a binary form inside the BSON document, it is easy to perform a byte-by-byte memory comparison on these fields to see, whether they are identical or different. If, for example, in a mesh object the number of vertices does not match the head revision, it is immediately rejected as modified. However, if the counts are the same, then the binary representations have to be compared, which is still a reasonably fast operation.

5 Use Cases

Users of our system can store 3D scenes in the `trunk` as well as several `branch` history paths. Trying to commit changes on a document which is at least one revision behind the DB, hence out of sync, results in either conflicting or non-conflicting edits.

5.1 Non-conflicting Edits

Given the assumption that all documents in the scene graph are stored in their local coordinates with associated parental transformations that lead towards global world coordinates (see §3.2), separate edits of related documents can be merged automatically as shown in Figure 5. Suppose that User_1 checked out the entire building, doubled its height and therefore displaced the `roof`. Also suppose that User_2 checked out only a subset of the building, added chimneys to the `roof`, however, was not aware of height updates. Because modifications to the `roof` document are independent of its position within the scene, User_2 can still commit using the implicit auto merge functionality.

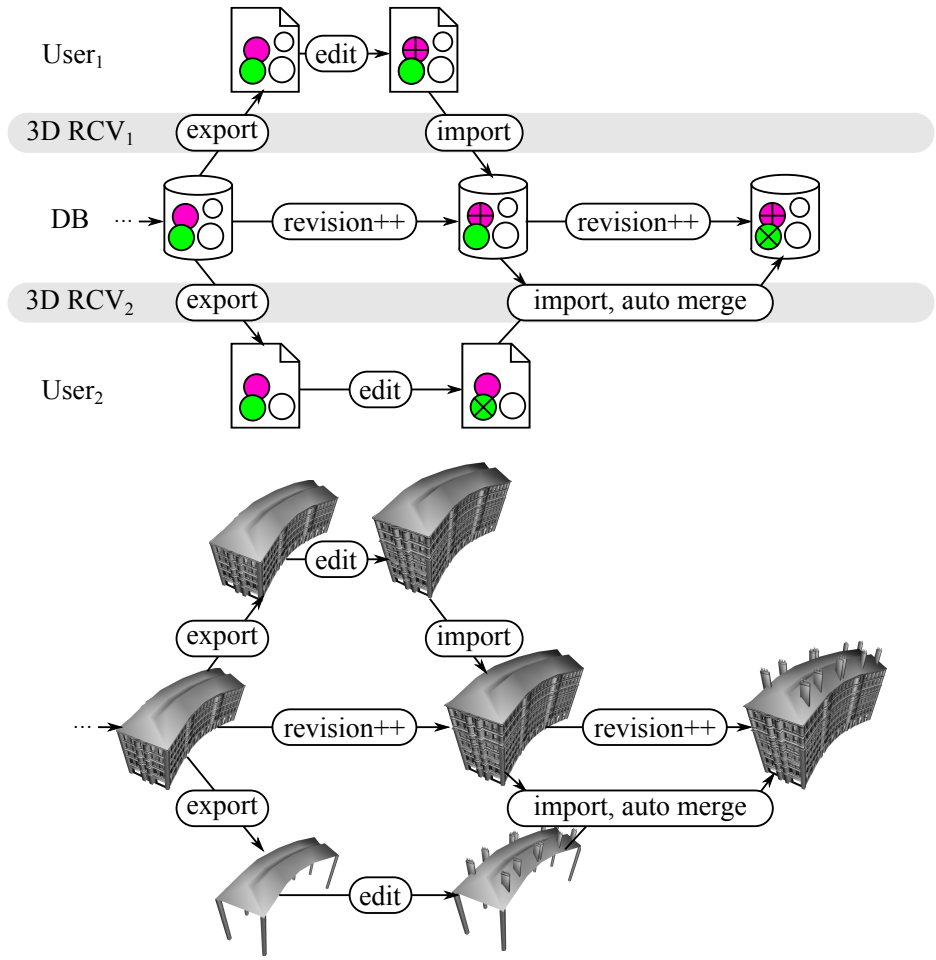


Figure 5: An example of non-conflicting edits resolved by automatic merge. Schematic representation (top) shows how purple (global transformation of `roof`) and green (`roof`) documents are in parental relationship which ensures that the merged result will satisfy both edits. Graphical representation (bottom) shows individual stages of the 3D geometry.

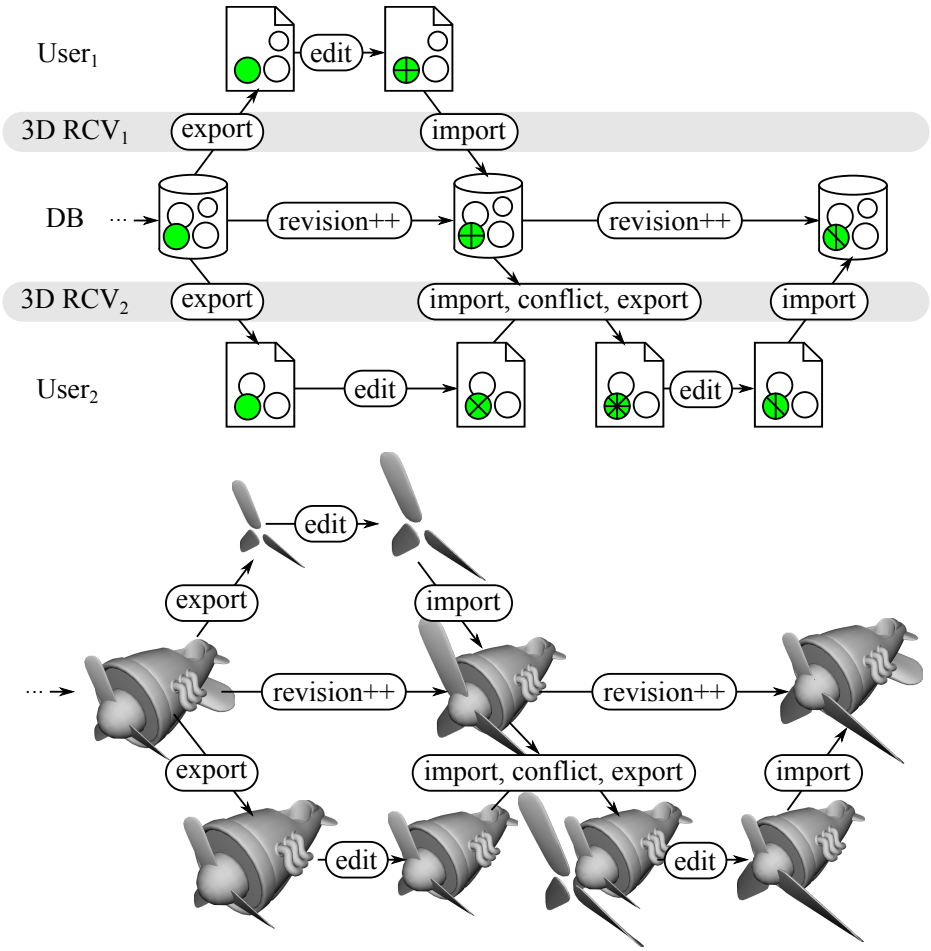


Figure 6: An example of manually resolving conflicting edits. $User_2$ can either keep local changes of conflicted propeller edits or the head revision or as shown in this example, export both conflicted propellers into a file, resolve conflicts manually and commit again. The same can be achieved when merging from a branch to the `trunk` or another `branch`.

5.2 Conflicting Edits

If changes have been made on the same document the conflicts have to be resolved in one of following ways:

1. Export conflicted meshes or the entire asset and edit the conflicts in an external 3D modeling tool.
2. Preserve local changes.
3. Preserve head revision such as revert does in text-based versioning systems.

An example of the first of these is depicted in Figure 6. Similarly, for the other two cases, a user can instead of exporting the conflict simply decide which of the two revisions to keep.

6 Discussion

6.1 Limitations

By design, the smallest unit of change in our prototype is a BSON document where one BSON document contains an array that might be used for vertex coordinates, indices, texture coordinates etc. This is efficient if the likely access is a collection of documents, but not if edits were very localised. For example, if a single vertex was repositioned in a mesh object, the entire document would replace previous version in the next revision. This level of granularity might not be suitable for all projects, however, it would certainly be possible to support multiple types of object representation within the same database. Thus conceptually the database interface could represent small edits as operations, and the implementation of this within the database would depend on the structure of its collections (tables).

An important implementation issue is that as of version 1.8, MongoDB has an increased upper limit of 16MB per-document. This means there is a potential need to split larger meshes into smaller components not exceeding this restriction. This should occur rarely as many modelers and file formats have size limits that are lower than this. A more serious problem with a schema-less database is a lack of data validation on insertion, unless specified as a unique key. Therefore, for a production-ready version of our system it would be necessary to build a server-side daemon that would act as a DB interface, facilitating the revision control instead of relying on a GUI implementation.

Some obvious functionality is not yet implemented, but is relative minor extensions now that the base functionality is working. At the moment, we focus on mesh vertex properties, texture coordinates and normals. There is no reason why the model we have should not extend to texture assets, other material descriptions and shader descriptions. Further, one can imagine that application-level information could be stored in a flexible way.

6.2 Database Interaction

Currently 3D editors interact with the systems through the transfer of files via the 3D RCViewer tool. In this section we discuss the advantages and disadvantages of file support, describe the scope for developing other direct interfaces and discuss other facilities that the database itself might provide.

6.2.1 File-based Interaction

The interaction between the 3D modeler and the database takes place through the import and export of files via 3D RCViewer. We have succeeded with decoupling the modeling from long-term storage, as files are now considered to be only temporary representations to facilitate interchange with a variety of editors. This file-based interaction will remain a useful facility because the range of 3D editors and other tools that generate meshes (e.g. procedural generation) is vast.

As we are interacting with file representations our revision control system needs to recognise the changes efficiently. That is, it must be able to detect that a set of nodes that is being imported matches existing nodes and makes thus revisions. At the moment we rely on the fact that the imported asset files preserve revision metadata in a form of `UUID` and a `revision number` for each component. However, there is no guarantee that an editor will do this, and users will always be able to modify this data if they wish. Thus, there is an interesting matching problem. We can search by matching nodes against those in the database and against all previous versions. This is speeded up by relying on data sizes to reject matches and because matching is a pure binary comparison. To go further we would need to be able to match arbitrary meshes against each other looking for those that are approximately equivalent. This is an interesting research problem in itself. So far we would have to rely on visual comparison through the interface, but this can perhaps be automated for a large class of incoming meshes. This difficultly could be reduced dramatically by using a spatial keying extension (see §6.2.3)) or circumvented by direct editor connections to the database (see §6.2.2)).

6.2.2 Other Interfaces

As suggested in Figure 1, the 3D RCViewer is not a necessary component to use a 3D database of the type we are proposing. The editor could itself connect to the database directly. This interface could be implemented using the plug-in frameworks provided by many modern editors such as Autodesk 3ds Max and Blender.

Plug-in access would be able to exploit all the metadata that was available in the database, such as UUIDs, access control and revision histories. In particular, on pushing assets back into the database we can rely on knowing exactly what point the asset was retrieved from and its revision number. Thus the user does not need to be aware of versioning. Of course some form of conflict resolution could still be necessary, but we can avoid it by using access control (see §6.2.3). This fits extremely well with the direct connection model because the editor can request locks as and when required, and thus users can be assured that as long as the lock is retrieved, they are able to edit.

6.2.3 Other Services

Although we have discussed the potential benefits of access control, it is not currently supported. It is not difficult to add to the database, but it is difficult to preserve in the round-trip to the file store. The benefits of access control are most apparent with direction connections from editors to the database as mentioned in the past section. We can imagine supporting read and write locks, and also hybrid types of locks that enforce geometric constraints of some types.

Another feature to extend the 3D database is the facility to search for assets via a spatial query; this is reminiscent of the role of spatial databases (see §2.5). To achieve this we would provide some sort of spatial hierarchy such as a bounding box hierarchy. This could be used both for general queries (“fetch all objects within this region”, “fetch all objects adjoining given object”), but also to facilitate detection of matches when other metadata has been lost. MongoDB has native geospatial indexing which could potentially support this. Geospatial indexing also suggests that we could provide web-enabled interfaces that exploit other web services for features such as online visualisation.

7 Conclusions

We have presented a novel approach for storage and revision control of 3D assets in a database. We believe that the unification of storage and control in a database can provide significant benefits over the common practice today which uses standard version control systems and asset tracking.

We created a 3D database in MongDB. We had to address issue of storing hierarchical scene graphs in such a database, and we had to select the size and types of elements stored. On top of this database we created a revision control system that allows branching and merging. This allows concurrent edits on the

database. In creating our database we decided to support 3D editing using the broadest range of tools; thus we support import and export of files in standard formats. We have demonstrated how we can now support common user stories such as merging conflicted edits.

There are many avenues for future work as this is a proof of concept prototype. Some of these are discussed in §6 in the context of limitations and potential benefits of our approach. Our intention is to build this in to a robust tool that can be used by the community, and thus it is our intention to make our work open source in the near future. This will hopefully facilitate crowd-sourcing of models for various purposes. In the next release, we will expand the code base by implementing a plug-in based database access for one or more standard 3D modeling packages and will enable a locking mechanism.

Acknowledgements

We would like to thank the Foresight team at Arup for sponsoring this research.

References

[Austerberry, 2006] D. Austerberry. *Digital asset management*. Focal, 2006.

[Bentley Systems, 2011] Bentley Systems. Assetwise, September 2011. `http://www.bentley.com/en-GB/Products/assetwise/`.

[Blanchette and Summerfield, 2008] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt4*. Prentice Hall, second edition, February 2008.

[Brink and Steenberg, 2011] Emil Brink and Eskil Steenberg. Uni-verse main page, September 2011. `http://www.uni-verse.org`.

[Brutzman and Daly, 2007] Don Brutzman and Leonard Daly. *X3D: Extensible 3D Graphics for Web Authors*. Morgan Kaufmann, first edition, May 2007.

[bso, 2012] BSON–binary json specification, January 2012. `http://bsonspec.org/`.

[Carrier, 2005] Brian Carrier. *File System Forensic Analysis*. Addison-Wesley Professional, March 2005.

[Celko, 2004] Joe Celko. *Joe Celko's Trees and Hierarchies in SQL for Smarties*. Morgan Kaufmann, May 2004.

[Chen *et al.*, 2011] Hsiang-Ting Chen, Li-Yi Wei, and Chun-Fa Chang. Nonlinear revision control for images. In *ACM SIGGRAPH 2011 papers*, pages 105:1–105:10, New York, NY, USA, 2011. ACM.

[Connolly and Begg, 2004] Thomas Connolly and Carolyn Begg. *Database Systems: A Practical Approach to Design, Implementation and Management*. Addison Wesley, 2004.

[Dassault Systemes, 2011] Dassault Systemes. Enovia, September 2011. `http://www.3ds.com/products/enovia/solutions/`.

[Dean and Ghemawat, 2008] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51:107–113, January 2008.

[Dekeyser and Watson, 2006] Stijn Dekeyser and Richard Watson. Extending google docs to collaborate on research papers. Technical report, The University of Southern Queensland, Australia, 2006.

[Denning *et al.*, 2011] Jonathan D. Denning, William B. Kerr, and Fabio Pellacini. Meshflow: interactive visualization of mesh construction sequences. *ACM Trans. Graph.*, 30:66:1–66:8, August 2011.

[Dourish, 1995] Paul Dourish. The parting of the ways: divergence, data management and collaborative work. In *Proceedings of the fourth conference on European Conference on Computer-Supported Cooperative Work*, pages 215–230, Norwell, MA, USA, 1995. Kluwer Academic Publishers.

[Hazel, 2008] Dan Hazel. Using rational numbers to key nested sets. *CoRR*, abs/0806.3115, 2008.

[Hesina *et al.*, 1999] Gerd Hesina, Dieter Schmalstieg, Anton Furhmann, and Werner Purgathofer. Distributed open inventor: a practical approach to distributed 3d graphics. In *Proceedings of the ACM symposium on Virtual reality software and technology*, VRST '99, pages 74–81, New York, NY, USA, 1999. ACM.

[Hunt *et al.*, 1998] James J. Hunt, Kiem-Phong Vo, and Walter F. Tichy. Delta algorithms: an empirical analysis. *ACM Trans. Softw. Eng. Methodol.*, 7:192–214, April 1998.

[Jacobsen *et al.*, 2005] Jens Jacobsen, Tilman Schlenker, and Lisa Edwards. *Implementing a digital asset management system: for animation, computer games, and web development*. Elsevier Focal Press, 2005.

[Kurlander and Feiner, 1992] David Kurlander and Steven Feiner. A history-based macro by example system. In *Proceedings of the 5th annual ACM symposium on User interface software and technology*, UIST '92, pages 99–106, New York, NY, USA, 1992. ACM.

[Leach *et al.*, 2005] P. Leach, M. Mealling, and R. Salz. A universally unique identifier (uuid) urn namespace. Technical report, The Internet Society, July 2005.

[Loeliger, 2009] Jon Loeliger. *Version Control with Git: Powerful tools and techniques for collaborative software development*. O'Reilly Media, first edition, 2009.

[Malewicz *et al.*, 2009] Grzegorz Malewicz, Matthew H. Austern, Aart J.C. Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 28th ACM symposium on Principles of distributed computing*, pages 6–6, New York, NY, USA, 2009. ACM.

[Membrey *et al.*, 2010] Peter Membrey, Eelco Plugge, and Tim Hawkins. *The Definitive Guide to MongoDB: The NoSQL Database for Cloud & Desktop Computing*. APRESS ACADEMIC, first edition, 2010.

[Obe and Hsu, 2011] Regina Obe and Leo Hsu. *PostGIS in Action*. Manning Publications, first edition, 2011.

[Pilato *et al.*, 2008] C. Michael Pilato, Ben Collins-Sussman, and Brian W. Fitzpatrick. *Version Control with Subversion*. O'Reilly Media, second edition, September 2008.

[Rigaux *et al.*, 2001] Philippe Rigaux, Michel Scholl, and Agnès Voisard. *Spatial Databases: With Application to GIS*. Morgan Kaufmann, 2001.

[Schulze *et al.*, 2012] Thomas Schulze, Alexander Gessler, Kim Kulling, David Nadlinger, Jonathan Klein, Mark Sibly, and Matthias Gubisch. Open asset import library (assimp), January 2012. `http://assimp.sourceforge.net/`.

[Steed and Oliveira, 2009] Anthony Steed and Manuel Oliveira. *Networked Graphics: Building Networked Games and Virtual Environments*. Elsevier, 2009.

[Tanenbaum, 2009] Andrew S. Tanenbaum. *Modern Operating Systems*. PHI, third edition, 2009.

[Unity Technologies, 2010] Unity Technologies. Using external version control systems with unity, August 2010.
`http://unity3d.com/support/documentation/Manual/ExternalVersionControlSystemSupport.html/`.

[V. Kothuri *et al.*, 2012] Ravikanth V. Kothuri, Albert Godfrind, and Euro Beinat. *Pro Oracle Spatial for Oracle Database 11g*. Apress Academic, January 2012.

[VisTrails, Inc., 2012] VisTrails, Inc. Vistrails for maya, January 2012. `http://www.vistrails.com/maya.html`.

[Wäsch and Klas, 1996] Jürgen Wäsch and Wolfgang Klas. History merging as a mechanism for concurrency control in cooperative environments. In *Proceedings of the 6th International Workshop on Research Issues in Data Engineering (RIDE '96) Interoperability of Nontraditional Database Systems*, pages 76–, Washington, DC, USA, 1996. IEEE Computer Society.

[Wordnik, 2010] Wordnik. 12 months with mongodb, October 2010. `http://blog.wordnik.com/12-months-with-mongodb`.

[Zeleznik *et al.*, 2000] Bob Zeleznik, Loring Holden, Michael Capps, Howard Abrams, and Tim Miller. Scene-graph-as-bus: Collaboration between heterogeneous stand-alone 3-d graphical applications. In *In Proceedings of Eurographics 2000*, pages 200–0, 2000.