Shooting A Moving Target—
An Experience In Developing A Production Tracking Database

Shiro Kawai
Square USA Inc.
55 Merchant St. Suite 3100
Honolulu, HI 96813, USA
shiro@squareusa.com

Abstract

How can you track complex information flow for an organization rapidly changing its structure and workflow?

This is the problem we've been facing since we established Square USA Honolulu Studio in 1997 and began to work on a 100% computer-generated feature film. This is a new studio trying to do something new, and things change very quickly and sometimes drastically.

In the film production, the most important thing is to get the final film-out image, and it doesn't matter how you get it. The consistency of workflow is sometimes compromised by the deadline of image delivery.

To cope with this situation, we've been using a Lisp-based object oriented database (OODB) from the beginning of the production. Lisp's flexibility allows us to change internal data structures quickly while maintaining the compatibility with the other parts of the production, which turned out to be the key requirement for such a fluid structure.

We implemented a client-server architecture on top of the OODB and defined a query language between them. No matter how the internal definition of schema changed, we could keep the old query interface as well as the new one so that the old client tools would work. Also it allowed our programmers to work in parallel, without making them stop and update their interface all at once.

1 Introduction

3D computer graphics (CG) technology is rapidly advancing. It is less than a decade since we were amazed by the movies featuring a liquid-metal-cyborg or genetically reproduced dinosaurs, and now we don't even raise an eyebrow at seeing a long-deceased actor appear in a new movie with whole new lines of dialogue.

Nevertheless, it is still a major challenge to create the amount of quality images required for a feature animation film by CG. We have seen only a few of them since the first full 3D computer generated feature film, Pixar's *Toy Story* (1995).

Square USA Inc. launched its first full CG movie project "Final Fantasy" in 1997. Unlike existing full CG animations, we aim to produce photo-realistic images entirely in CG, including human characters. After a year of research and preproduction, we've been in production for about two years by now, and are expecting to release the film in 2001.

It was clear from the beginning that we needed some sort of integrated database system to track large amounts of data related to the production. What we had in mind was the followings: A movie is divided into sequences, and a sequence is divided into shots. A shot contains characters, sets, props and effects. Artists were assigned to those elements. Schedules were created based on those assignments. So far, it sounds rather simple. In fact, it was not.

Although we had quite a few experienced peo-

ple from the visual effects studios, computer graphics studios and the traditional animation industry, it was not trivial to figure out how to manage this project. In the course of preproduction we found neither the traditional animation workflow nor visual effects workflow would fit to our goal.

So we've been undergoing trials and errors to find a better way. One thing we've certainly learned is that everything may change. And our database system has been required to support a process in which the specification would never be complete and consistent.

Now let me share the story about how we cope with this situation using Lisp. In the next section, I explain various challenges arising in CG film production and identify the issues our system shall address. In section 3, I describe our early attempts and findings from it. In section 4, I share the technical details of our system and how it overcame the challenges. In section 5 I summarize what we learned and what we're looking forward to.

2 Challenges in the CG film production

The basics of 3D CG animation is fairly straightforward. These days you can even pick up off-the-shelf software and create a minute-long CG clip by yourself with your home PC.

It works like this: You simulate live-action film shooting. You create a 3D representation of actors and stages, using polygons or parametric surfaces with appropriate textures. You lay them out and animate them. Given camera and light positions, you calculate the image of the scene from the camera's viewpoint (this calculation is called *rendering*). And boom! You get a 3D computer-generated animation.

Although the underlying theory is much the same for a CG feature film production, there are some issues which make the latter a little bit complicated.

Quantity. In our production, it is not unusual to have a 3D model taking up to 100 megabytes of data. Each model may consist of up to hundreds of files of subparts and textures, and a scene is assembled of several dozens of such models. Several versions of the data must be kept on the server, so the number of files and the disk usage easily explode. Usually a whole scene's data doesn't fit in the processor's memory, so we need to split it into a number of layers — usually from 5 to 30 layers—render them separately and composite them together.

Interdependency. Lots of elements depend on other elements in the scene: e.g. the animation data depends on the character to be animated. Some elements may be shared by different shots. Some elements depend on a specific version of the other element. We also need to keep track of which version of elements are actually used for rendering of a certain layer.

Parallel development. You can never make an animation film with a small staff. Our studio has over 100 artists and 40 support staff (programmers and systems engineers) working in full throttle to deliver the images. A scene is composed of a whole bunch of files made by several artists, and they have dependencies with each other. Some elements are shared by lots of shots. If someone updates one file, it may affect ten other artists in unexpected ways.

Workflow variation. Some shots may have their own technical challenges and require special treatment. This creates exceptions in the studio workflow. Actually, you need to deal with lots of such exceptions. Sometimes you need to try different options to find out the more efficient way.

Long development period. The production goes on for a few years. The software versions to create the data, the workflow, and file formats, all change during the production. One day a producer may come to tell you to fix a scene which was rendered a year ago, and you need to retrieve all the data from the archive, along with the versions of software which were used to work on the data.

Heterogeneous environment. Artists choose computers which fit their needs. At least we need to

support Windows PCs, SGI/IRIX, and Linux PCs.

Note that we found out some of these issues only after trial and error, since they arise only in full CG film production.

In visual effects production, the work can be divided into groups of shots in a way that they don't have many interdependencies. Also the live-action shots provide some visual elements, so the number of elements aren't as big as the full CG case.

On the other hand, in traditional 2D animation, each cell is drawn specifically for certain shots and not many elements are shared. Also it has a defined workflow historically.

In the full CG production, however, you have several times larger number of shots, elements and layers than the visual effects production, have elements shared among shots or layers, and have different variation of workflow per shot. You can see how much the complexity increases

To address those issues, we need a database system which fulfills the following requirements.

- Accommodating the rapidly changing workflow.
- Allowing parallel development.
- Allowing smooth transitions across changes while keeping backward compatibility.
- Clients working on heterogeneous platforms.

3 The first version

During the preproduction and the early stage of the production, we implemented a couple of versions of the production database. We chose Franz Inc.'s AllegroStore[1] Lisp-based object oriented database system as an infrastructure of our system.

AllegroStore implements a persistent object class on top of ObjectStore[2] from eXcelon Corp, with a nice integration with CLOS. You can define a persistent object just by specifying the metaclass `persistent-standard-class`. Persistent objects

```
;; Defining persistent object class <user>

(defclass <user> ()
  ((name  :allocation :persistent
          :type       string
          :initarg    :name)
   (email :allocation :persistent
          :type       string
          :initarg    :email))
  (:metaclass persistent-standard-class))

;; Open database "mydb.db"
(with-database (db "mydb.db")
  ;; begin transaction
  (with-transaction ()
    ;; make a persistent object
    (make-instance '<user>
       :name  "Shiro Kawai"
       :email "shiro")
    ;; traverse all objects of class <user>
    (for-each*
      #'(lambda (user) (display user))
     :class <user>)
  )
)
```

Figure 1: Defining and using persistent objects in AllegroStore

are created, referenced, modified and deleted inside a special form `with-transaction` which takes care of the atomic operation of the transaction. See figure 1 for the example.

When you change the definition of persistent classes (schema), AllegroStore converts the old instances on demand, using `change-class` feature of CLOS.

Why OODB? We expected a lot of design changes and a rather small number of transactions. Traditional relational database systems (RDBs) might have been good for processing large number of transactions, but tweaking SQL for a changing specification didn't seem like a good idea.

Indeed, OODB was very handy when we needed to change many-to-one relationship to many-to-many relationship or vice versa, which happened quite often. In AllegroStore you can have a slot keep a set

[1] AllegroStore is a registered trademark of Franz. Inc.
[2] ObjectStore is a registered trademark of eXcelon Corp.

of pointers to the persistent objects, and the system takes care of referential integrity so that you can have the reverse pointer as well. Changing between many-to-one relationship and many-to-many relationship simply consisted of changing a flag in the slot definition.

In our first version we used HTML as the user interface, since any client platform could access it as far as it had a browser installed. We wrote a daemon server process on top of AllegroStore which received queries in the form of `QUERY_STRING`, and returned HTML documents (see figure 2(a)).

Lisp worked well for rapid prototyping of the server. One good thing was that we could patch the running server by just telling it to load the recompiled module. It was very useful since a service interruption would impact the production.

However, it had a downside. The server was written as one monolithic program and any change or addition of features went into the server source. And alas, Lisp programmers are so precious these days that all the change requests had to be processed by one or two programmers and eventually this became a bottleneck. We realized that we needed to find a way to allow other programmers and script writers to add functionality to the server.

Another difficulty we faced was that sometimes we needed to work on specifications which were logically inconsistent. For example, one document said the relation should be many-to-many, but another specification assumed it was one-to-many and indeed some part of the production was working based on this specification. Or one section in the production assumed a certain condition was true between some data, while the other section created the data which didn't meet the condition. Although this may sound funny, indeed it works as far as the data is locally consistent. Some data needs to be consistent only to finish a certain shot, and then is archived away. It's important not to stop the production just to convert the data to be consistent.

4 The second version

To overcome the difficulties in the earlier attempts, we rewrote the whole system with two things in mind.

1. To allow programmers and script writers to implement their own functionality, while maintaining the database integrity.
2. To allow multiple versions of interface even which seemed inconsistent with each other.

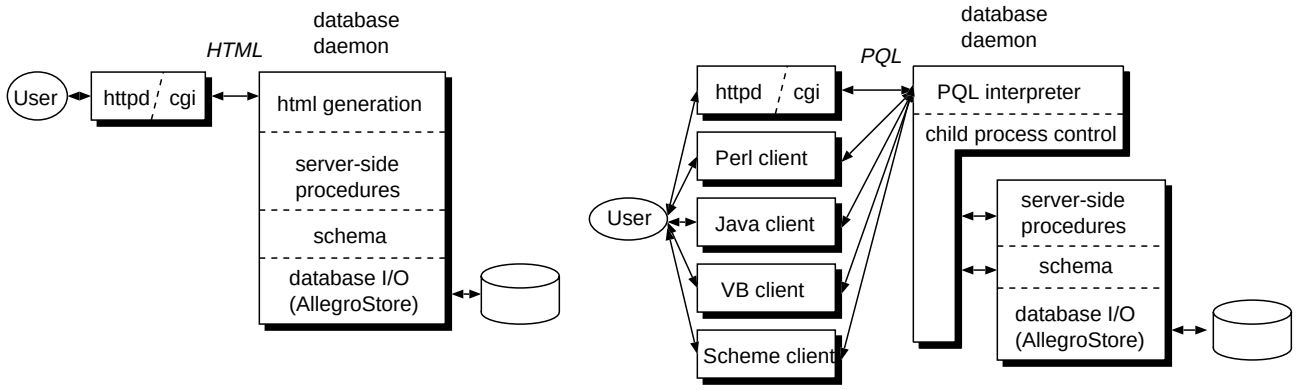
As a layer for application programmers, we created a query language "PQL" (Phantom Query Language, where "Phantom" was the codename of this system) which was functionally similar to the languages such as SQL or OQL[1] but has the Lisp syntax. The database team had responsibility for the integrity below the PQL layer, while the technical staff in the production can write their own client applications based on PQL.

On the server side, we clearly separated a schema-dependent part (schema definition and server-side procedures) from a schema-independent part (PQL interpreter engine, metaclass for schema definition). Schema definition interface allowed us to change schema dynamically while easily keeping the old interface.

4.1 PQL interface

PQL is a subset of Common Lisp to be interpreted in a "sand box" on the server. The dangerous functions such as file I/O are not available. Instead, a special form to issue a SQL-like query is added (see figure 3, table 1).

There are a couple of reasons why we implemented a proprietary language instead of using standard ones. SQL has its own strengths in the relational database operations, but is weak in treating aggregate types like lists and object references, and we'd lose the advantage of using OODB without them. OQL is, on the other hand, designed for OODB and powerful enough, but a bit overkill for our purpose. We had very limited time to implement it, and using Lisp syntax and supporting only Lisp native types was a trivial task.



(a). The first version (monolithic server)

(b). The second version

Figure 2: The structure of our database system. (a) In the early attempts, we tried to use HTML as an universal interface, and the server was monolithic. (b) The current architecture uses PQL as an universal API between clients and the server.

The users of our database are the artists, the coordinators, and the technical staff. They use their own choice of computers and software which serves best for their creative goals. Hence the database should support various platforms. The client libraries which have the PQL interface layer are currently available in Perl, Scheme, Java and Visual Basic. An interactive interface is also provided (see figure 4). This was important because we had a lot of staff who could write scripts, but few staff who could program in Lisp. By opening up this level of interface, people could work in parallel, to solve their own problems in the production.

4.2 Schema definition interface

On the server side we defined a metaclass on top of AllegroStore's `persistent-standard-class`, which enabled the following features.

- The class and persistent slot became visible from PQL interpreter. Various extra slot options could be added to control visibility, access privileges, data validation, etc.
- A so-called 'virtual-slot' could be defined, which behaved as if it were ordinary an CLOS slot but

```
;; A form beginning with specific keywords is
;; interpreted as a query.
;; This form lists name, email and sections of
;; users who have email account.

(:select name email sections.name
 :from <user>
 :where (not (null (email-of this))))

;; The query can be intermixed with Lisp form.
;; This lists each section's name and the number
;; of users in the section.

(mapcar #'(lambda (name&users)
            (list (car name&users)
                  (length (cadr name&users))))
        (:select name users :from <section>))

;; A reader syntax is extended so that you can
;; directly refer to a specific object using
;; its unique key, using a syntax ?(class key)

(name-of ?(<user> "shiro")) ; get shiro's name
```

Figure 3: PQL examples

```
dbsh[1]> (:select name sections.name
---->     :from ?(<user>"shiro"))
(("Kawai, Shiro" ("RD")))
dbsh[2]> (length (users-of ?(<section>"RD")))
37
dbsh[3]> display plain
display mode is set to plain
dbsh[4]> (:select abbrev-name (users => length)
---->     :from <section>)
AN 29
AT 14
CT 1
CH 16
```

Figure 4: A sample session of dbsh, an interactive command-line interface for PQL. A simple query can be written as a dbsh script

when the program accessed the slot it triggered the associated hook function. When we updated the class structure, we left obsoleted slots as virtual slots instead of completely removing them, so that other parts of our architecture could still work. These virtual slots could be seen from PQL the same as the persistent slots. A virtual slot could also be used to create mutual references between objects, combined with AllegroStore's inverse function.

It turned out that the virtual slot feature was extremely useful, since we couldn't update large numbers of related tools scattered in the production at once in the required time. When we needed some major change, we first updated the schema and the core server, then gradually updated the other parts.

Figure 5 shows how the schema definition looks like. Users and sections have many-to-many relationship, and mutual reference is implemented using a virtual slot.

4.3 Experiences

I mention a few events which happened in the production to convince us that our approach was winning. I think modern RDBs also have a mechanism to handle these kind of problems, but the OODB approach is more straightforward.

Reader extensions
?*object-id*, ?(*class unique-name*) - object reference
var.slot - expanded into slot accessor
Query special forms
(`:from` *expr*|*class* [`:where` *cexpr*]) - from the set of objects specified by *class* (class name) or *expr* (Lisp expression), list of objects which satisfy *cexpr* is returned. `:where` clause can be omitted.
(`:select` *slots* `:from` *expr*|*class* [`:where` *cexpr*]) - from the set of objects selected by `:from` and `:where` clause, the list of values of *slots* are retrieved and returned.
(`:map` *forms* `:from` *expr*|*class* [`:where` *cexpr*]) - from the set of objects are selected by `:from` and `:where` clause, *forms* are mapped onto each of the objects and the list of results are returned.
Creating and updating special forms
(`:create` *initargs* `:class` *class*) - create an object of class `class` by initial arguments *initargs* and returns the reference of the created object.
(`:update` ((*slot expr*) ...) `:from` *expr* [`:where` *cexpr*]) - update objects selected by `:from` and `:where` clause.
(`:delete :from` *expr* [`:where` *cexpr*]) - delete objects selected by `:from` and `:where` clause.

Table 1: Brief summary of PQL basic syntax. Besides those special constructs, most of CommonLisp forms and functions are available.

4.3.1 A big change in a shot structure

Although a 'shot' was the most basic entity for the production, its definition and fundamental structure kept changing.

In a traditional animation films, a storyboard is broken down into shots in the early stage of the production and pretty much locked down, for animators should work from well-defined timing. On the other hand, in live action films, 'shot' refers to a film segment taken by a continuous camera roll. It is common to shoot one scene with several cameras, in which case a several shots are taken simultaneously, then they are 'cut' in the editing. Often the cut is fixed at the last moment.

At a certain time, our production wanted to try the mixture of those two styles. We had 3D geometries

```
(defclass <USER> (<phantom-base>)
  ((EMAIL
    :allocation :persistent
    :type       string
    :inverse email->user)
   (NAME
    :allocation :persistent
    :type       string)
   (SECTIONS
    :allocation :virtual
    :type <section> :set t
    :getter sections-of
    :setter (setf sections-of)))
  (:metaclass phantom-persistent-class))

(defclass <SECTION> (<phantom-base>)
  ((NAME
    :allocation :persistent
    :type string)
   (USERS
    :allocation :persistent
    :type <user>  :set t
    :inverse sections-of))
  (:metaclass phantom-persistent-class))
```

Figure 5: Example of schema definition interface. It implements many-to-many relationship between users and sections by an inverse function (`:inverse` in the slot *users* of class `<section>`) and a virtual slot (`:allocation :virtual` in the slot *sections* of class `<user>`).

inside a computer, so we thought we could simulate a process of live filming to some extent. So we added `<cut>` class to the database, and made a `<shot>` object have a list of `<cut>` objects. The length of shot was changed from simple integer slot into a virtual slot which calculated the sum of the lengths of the cuts. A few months later, it turned out that the process caused more confusion than it was worth, so the production decided to drop a concept of a cut. We changed back the shot length field to simple slot. During the process the change was invisible for most of the client applications which used shot slots; they could use the same PQL and need not be changed at all. The conversion of the data was handled by AllegroStore automatically, so all we needed to do was to rewrite a schema definition and to reload it.

Later, the production somehow decided to keep a separate copy of shot objects for each stage of progress of the shot. Now we needed to have not only a single 'shot' object but something like 'layout-shot' and 'production-shot' and so on. (By the way, I suggested that it was not a good idea, but sometimes you need to let things go to see if they work or not). To maintain compatibility, I made the existing shot object have pointers to those sub-objects. Some slots like shot length became a virtual slot again, thit time with a procedure which dispatched the accessor to the appropriate sub-objects. Several months later the production decided not to continue that direction, and things got back to the old simple way. Again, for most clients the changes were completely transparent.

4.3.2 One-to-many or many-to-many?

A term 'element' is generally used to refer a group of things appearing in a shot and which need to be worked on as a unit. In traditional visual effect studios and animation studios, elements are almost the same as layers. In full CG films, however, there are lots of data which can be shared among different shots and layers and we wanted to treat such data as an element.

Thus, we first defined the relationship of shots and elements as many-to-many. However, as the number of shots and elements increased (we eventually had thousands of each), it became impractical to maintain the relationships. Sharing the same element in large number of shots also caused problems such as the concentration of disk access and undesirable propagation of errors in the shared element. So we decided to copy the element over to each shot, and the shot-element relationship became one-to-many.

The story wasn't over. After a while we found out that we did need to share an element in a group of a small number of shots. If a shot was split up into a few shots for editing purpose, they had very similar elements in there and copying the elements was just a waste. Now, shot-element relationship was many-to-many again, but we also tracked such a group of shots so that the relationship was manageable.

4.3.3 Optimization by server-side procedures

Besidesthe bare PQL interface, we could implement whatever procedure we needed and make it visible from PQL interpreter. It was something like a 'stored procedure' in the traditional RDB. The difference was that we had full access to the database from the procedure, since the procedure would be compiled into the server kernel.

Our rendering guys were complaining that a certain rendering check query took way too long to complete (close to a minute). The query traversed all the rendering jobs, accessed the list of commands issued to each job and retrieved the last command, and then checked if it satisfied a certain condition. There were several thousands of rendering jobs recorded in the database and each job had several commands in its list, so the process was time consuming.

First thing I did was to cache the last issued command for each job. It was simply done by changing the slot of the list of commands to a virtual slot so that a certain procedure could be invoked every time the command was issued to a job. Then I realized that I could even keep a pool of suspicious commands to be checked, by applying the condition when the command was issued. The rendering check query would look into that pool instead of traversing all the jobs every time. These changes were implemented gradually without breaking existing code, and once I thought it was working I told the rendering guys how to use the new feature.

5 Conclusion

People often ask "why Lisp?" or "why don't you use traditional RDBs/ distributed object systems?". It is a bit difficult to answer, since we have no experience of applying other systems to this type of production and cannot make a comparison. However, we can say those things in general, as the lessons we learned:

- The balance of integrity and flexibility is subtle but important. In our size of production, sometimes you need to allow one part of the production to work inconsistently with another, during a transition (and if things keep changing, it is always a transition period!) If you emphasize integrity and consistency too much, things cannot change in a timely manner. If you emphasize flexibility too much and allow people work their own way, you lose integrity. In my opinion, Lisp itself keeps a good balance between these, and fits well into this type of project.

- Instead of making people ask and waiting for you to solve their problem, build an architecture which allows people to work to solve their own problem. In our case, an open and well-defined scripting API (PQL) made our life much easier. And it is one of advantages of Lisp to implement a small language like PQL.

We keep improving our system. The production is still changing dynamically, whenever it faces a difficulty in achieving our goal, and the database system must follow it.

Recently, we became aware of a problem which should be solved for future productions. Currently, most data is passed as plain unix files and the production database only keeps meta-data. However, the size of data to be passed around is getting bigger, as we target more realistic images. Even when the artist want to change one parameter in a 3D scene, he/she sometimes needs to load 100 megabytes of scene data, wasting save/load time and network bandwidth. Some sort of system which manages finer granularity is needed. We're looking for ways to apply the object-oriented database technology to the issue.

References

[1] R. G. G. Cattell (Ed.), *The Object Database Standard: ODMG 2.0*, Morgan Kaufmann, 1997.