Collaborative Dataset Building: Action-Based Modification and File-Based Data Management

Connor Wray, Armaan Bindra, and Shane Allen
Advanced Team Project 2014
St. Olaf College
1500 Saint Olaf Ave, Northfield, MN 55057
wray@stolaf.edu, bindra@stolaf.edu, csallenii@gmail.com

Abstract

As members of a research team implementing a 3D modeling pipeline for paired 2D images, we designed and managed the team's underlying dataset. Tasked with quickly deploying a data management system that allowed for many users to simultaneously generate new data using a local client and add it to the globally available dataset, we chose to use the standard file system to hold our information. To meet the requirements of developmental software we created a dataset manager to update the data files based not on swapping and overwrites, but on action logging followed by global processing of those actions. This strategy proved highly flexible, quick to implement, and robust at providing data recovery. Future applications are explored for large collaborative projects as well as other developmental or research dataset development.

Introduction

The creation of the Internet has led to cascading new innovations in the computer science world over the past 20 years. The web has revived client-server applications, as "the cloud" becomes the trendy new place to host applications, it has forced upgrades to the telecommunications infrastructure which allows more data to be shared and collected than ever before, and incredibly it has paved the path for global collaboration on an unprecedented scale. As the size of our computers shrink and connectedness increases, these kinds of phenomena are pushing themselves into new spaces. One of these directions is towards an increasingly small boundary between the virtual world and reality. By this I mean that through devices such as Google™ Glass and the use of virtual reality applications we are pushing the virtual world on top of our own, but also that we are making attempts to increasingly represent our world virtually. Two-Dimensional mapping has become one of the most used Internet services and as camera technology becomes more prevalent, we can expect that 3-Dimensional representations of places and things increase in both prevalence and popularity.

The issues preventing this type of widespread 3D model building are significant. For the duration of this project our team has focused on maintaining and supporting the dataset necessary for other researchers to practice building 3D models, retrieve data for optimization of the modeling pipeline, as well as provide the flexibility for other programmers to modify the systems underlying data structures as necessary to further the project. This task presented our team with several difficult issues. First, a traditional database requires substantial effort to design and implement -- we were to have a functional management program within a week so that other research could proceed. Second, the database had to be able to handle a large number of concurrently operating users. Finally, as this dataset was being used in ongoing research, it needed the ability to revert to previous states and be robust in the face of improper data and bugs from the model's editor (a client-facing editing application). We will now discuss our resolution of these issues critical to project success, but before we do, let's examine some of the relevant preliminary literature in the field that guided our implementation and discovery.

Literature Review

There are a few considerations that need to be undertaken while building a quality database. A database should be able to efficiently implement storage, updates and retrieval of data. It should be dependable, and the data should have high integrity to promote users' trust in the data. Finally, a database should be adaptable and mutable to new requirements that might not have been previously expected. Two possible approaches to storing data, that needs to created, read, updated and deleted, are using conventional files and relational databases (Whitten 1986)

Designing a database system using conventional files is comparatively easier as this approach often involved tailoring the database to work specifically for a single application whereas a relational database usually has a much more complex design that is aimed to be compatible with various different applications and information systems. However, using a file-based system can easily lead to a lot duplication of data items in multiple files. Additionally, the file system approach is not entirely secure since the users often have more direct control over the data and, thus, is more prone to damage during modification (Singh 2009).

An example of multi user creation of data that is ubiquitous today is the crowdsourcing of geospatial data using social networks and web technologies (Heipke 2010). Collecting data like this over time on a large scale could be overwhelming and take some surveyors and researchers' lifetime, however, new advances in technology used in handheld devices are very capable of gathering geospatial data of this caliber. This means that several amateurs can now successfully implement a job that was previously carried out only by professionals. A crowd sourced data collection by amateurs might raise questions concerning the credibility of the datasets. However, Heipke tackles this issue with the assertion that since repetition is widespread in such datasets, errors are more likely to be outweighed by accurate data, thus, resulting in a highly reliable dataset.

File Based Dataset

After reviewing the possibility of using a relational database or the standard NTFS file system to manage our data, the decision was made to use the file system. While this flies in the face of standard client/server type data management, it critically allowed us to meet of challenge of quickly (within several days) having a system capable of permitting simultaneous dataset editing and provided the flexibility to change the underlying model data structures as research progressed. In our specific instance of providing a 3D model, we began using three distinct types of data files each kept in a file tree indexing its unique identifying information. These three data files formed a tripartite graph of the information necessary to render a model from a pair of standard images. We found that as we progressed through the process of building the pipeline necessary to create, edit, optimize, and display the model we needed two additional types to hold additional information about the locations of contours in 3-dimensional space. Thus, we validated our choice to go with the flexible file system approach as it turned out that the flexibility of modifying the underlying data structure quickly and during operation was critical to furthering the project.

An additional, unintended, positive consequence presented itself as a result of choosing to store the dataset within the file system throughout the course of the projects development. It was previously stated that robustness and recoverability were priorities in our design and the choice to store data as files allowed all numeric and string data to be represented in the files as ASCII characters. Thus, all data files were human readable and easily amendable. The ability to easily check data integrity and view exact locations of corruption provided decreased down time when bugs in the client or data management software were discovered. We will cover specific instances of failure and recovery in a moment, but it must be stated here that having human readable data files is of significant benefit for dataset upon whom developmental work is being performed.

Action Based Modification

Once the decision was made to store out data as files in a standard file system, the issue of updating them as many concurrent users edited modeling information was the next issue to address. The solution we arrived at derives itself from a non-destructive photo and video editing techniques pioneered to prevent original item data loss. This system is action or modification based updating, where the original image is stored and additionally, all user modifications are stored along side of the original. Thus, with a click of a button, it is possible to remove all (or some) previous actions and revert to an earlier -- possibly original -- state. We adapted this system to help us achieve two of our main goals. First, the idea of processing user updates as actions meant that concurrent users would submit action lists, which we will call action logs, to the database management program which would then apply those updates to the relevant data files. This solution permits many users to work on an image simultaneously and still have all (non-conflicting) updates saved to the data set without overwriting, which is a common file system problem.

Secondly, as with the original incarnation with photo editing, our action based system allows for extensive reversal of modifications and therefore satisfies the constraint on our system that we must provide a robust recovery solution. Unfortunately, we put this recovery hypothesis to the test in a substantial fashion during the project upon a massive client side error, and in general it has allowed for a restoration of the dataset to usable conditions with small, to no, amounts of data being lost. However, the question remained, how do we apply the lists of actions to our dataset once users submit their logs? To perform this function we created a Python scripted backend manager to accept incoming logs, que them according to time of submission, and process them individually by updating the files according to a predetermined set of acceptable actions.

Considering that all of our users resided on the same file system, we chose to use the file system as our log submission vehicle. A drop box was created and hard coded into the modeling client application, and when a user chose to save their actions to the global dataset, their cumulative actions were printed into a file that was dropped into the submission queue. This method was elegant in that it required very little time to implement, held all the advantages of human readable files for debugging, provided a queue of exceptionally large length, and allows an essentially infinite number of users to save at around the same time. Each of these benefits was more important to us than the negative effect of not having a two way communication channel open between the client and the server to provide feedback, or the obvious security issues of allowing users to drop a file of any type into the targeted submission folder.

Once the files had been placed in the submission queue they were prioritized by the data server in order of age, with the oldest having the highest priority. Upon choosing the file with the oldest priority, the server would read in the selected log of actions performed by the identified user and perform each action in order, on the data files. There was a predetermined list of actions that could be recorded in the submitted logs, this list was coded into both the server and client applications, and any lines of the action log not in conformity with the preset standards would be ignored. This allowed the server to continue running in the event of improperly formatted expressions. After a log had been completely processed it's original log file was removed from the queue, and the process was repeated on the next oldest log file available

It should be noted that the choice of python to be the language in which this server was written could be criticized for it's lack of speed and inefficiency of memory management, but these factors were offset by the ease with which we were able to implement the program. Starting with no existing code, we were able to create a functioning database management script within a week. This allowed the rest of the research team to begin assembling data and retrieving it for optimization and further research purposes. The speed of implementation was critical for our project and the scripting strategy we utilized produced this desired outcome. Additionally, as you will see in the following results section, the speed of the file operations and processing in python did not prove to be excessive. And with some further action log optimizations, we were able to decrease processing times significantly. The rapidity with which we have been able to deploy and modify the solution exceeded all negative consequences from deployment on our scale.

Results

We were able to collect 2 types of quantitative measurements that clearly show the results of deploying our action log, file based data storage approach. First we will examine the size of the data set that collaborative users were able to create during the project's research period. The following table displays the number of unique objects corresponding to each different data file created during the brief two-week window available for building our data set.

Date	Corners	Tiles	ImgInfo	Poly3
1/16/2014	1869	2396	66	142
1/17/2014	2535	3153	87	387
1/21/2014	2535	3153	87	387
1/22/2014	3189	7492	633	725
1/23/2014	3189	9644	1269	725
1/26/2014	3480	10134	1395	996
1/27/2014	3930	10208	1399	1324
1/30/2014	5427	13919	1405	1778

Table 1: The Size of the Dataset over two weeks

As is evident from the data. A large number of each data type could be created by our users in a fairly short amount of time. The interaction between the client and data server was quick and enabled rapid prototyping of modeling images. Our strategies clearly provided a way for the client's users to perform the desired tasks and sync those updates with the globally available data files. Another set of quantitative statics of particular importance to our team was the speed at which we were able to process action log files. While the queue based set up meant that speed was not mission critical, delayed processing could prevent users from seeing the most up to date information and is still undesirable, especially during heavy use periods.

Typical use of the modeling client that we employed generated an action log of, on average, around 13000 lines. What follows is data from a run of an action log of 13054 lines through our server:

```
Found File and Started Processing /home/cg/palantir/w/submit/allencs
Completed Processing of /home/cg/palantir/w/submit/allencs in 31.5934860706 seconds
```

As you can see, the result is a run time of approximately 31 seconds to run through all of the actions required to bring the data set up to date with the client modifications. While this is in no way extreme, we implemented a preprocessing optimization program to bundle similar actions together with the idea of increasing performance. What follows is a run of the same action log as above, but with the preprocessor optimizing the log:

```
Found File and Started Processing /home/cg/palantir/w/submit/allencsActionLog
BEFORE PREPROCESSOR: 13054 lines
AFTER PREPROCESSOR: 2473 lines
Completed Processing of /home/cg/palantir/w/submit/allencs in 3.79432296753 seconds
```

The results of the preprocessing make a meaningful improvement to the server's performance, bringing the time down by over 87%. This set of optimizations made the use of python as a language choice no hindrance compared to other parts of the project written in faster compiled languages. There is a final set of results which cannot be truly quantified, but was of critical importance to our implementation. This result is our record of robustness in the face of interaction with other buggy application elements under development and our ability to recover data when incorrect actions were sent to the database. Despite many other parts of the application under development, our server did not suffer from any crashes after the first week of debugging and beta operations.

Unfortunately we were forced to implement recovery operations several times during our project's development, as improper actions were being logged by the client side application and sent to the server to be processed. This led in one case to all image data accidentally being deleted and in another case improper file identifiers being used (thus resulting in modification of improper files). To untangle these methods we were able to find the offending action log files and run modifying scripts to correct the errors and then re-run the action logs through the server application to make the proper data adjustments. This ability shows the power of an actions based approach. And while the ability to completely recover from this kind of situation cannot be quantified it allows the dataset users to have faith that their intended actions will be reflected in the data even if a bug occurs and have confidence using the data provided, one of our highest priorities for this project.

Conclusion

While the choice of data management strategy for our project diverges from the modern standard, it proved to be very adept at meeting our performance goals while being feasible within our constraints. With that said, we do have several recommendations for future implementations of this -- or a similar -- strategy, especially on a larger scale. First, we would recommend an object-based approach to managing the file data types within the server program. Spreading the parsing of ASCII strings into many functions spread throughout our code proved to be the cause of many file corruption bugs. Consolidating these kind of operations into object methods would offer a single location for all string modification, thus lowering the chance of error. As we approached the end of our project we had the opportunity to write an object oriented server script and begin testing, but the program was never swapped with the operational server. We would recommend starting with the object oriented approach from the beginning as it proved much more reliable in initial testing and is a highly convenient way to think of the data files within the action based modification scheme. Secondly, for deployment on a large scale, it is possible to predict that a run time of 3 seconds per action file is still too long. Therefore, writing the portions of the server responsible for performing the file operations in a faster compiled language -- the authors of this paper would recommend c++ to also benefit the goal of moving to an object oriented scheme -- to reduce processing time on a per action log basis. Finally, within our highly coordinated project we did not have to worry about collaborators either intentionally or unintentionally making conflicting modifications to similar locations in the dataset. These conditions were controlled through careful organization. However, in another deployment, we see the necessity of providing some degree of mutual exclusivity to editors, to prevent either duplicate or conflicting work from being performed. Implementing some form of an image check out system to prevent multiple users from modifying multiple images at the same time would be the simplest option available, but extension of this concept to more sophisticated locking techniques is also possible.

Using files as a choice of data store may be a controversial method given the advanced relational database tools available today, but using files to prototype developmental datasets and underlying structures is flexible and quick. Additionally, moving from traditional file overwriting to action-based modification provides flexibility, robustness, and reliability that more saving operations cannot provide.

References

Heipke, Christian. 2010. "Crowdsourcing geospatial data." *ISPRS Journal of Photogrammetry and Remote Sensing* 65 (6): 550-557.

Singh, S. K. "File-Oriented System versus Database System." *Database Systems: Concepts, Design and Applications*. New Delhi: Published by Dorling Kindersley (India), Licensees of Pearson Education in South Asia, 2009. N. pag. Print.

Whitten, Jeffrey L., Lonnie D. Bentley, and Thomas I. M. Ho. "11." *System Analysis*

Design Methods. St. Louis,MO: Times Mirror/Mosby College Pub., 1986. N. pag. Print.