A Framework for Generating High-Fidelity, Interoperable Urban Terrain Databases

Chuck Campbell, Kevin Wertman
Applied Research Associates, Inc.
Orlando, FL
ccampbell@ara.com, kwertman@ara.com

Julio de la Cruz
Army RDECOM/STTC
Orlando, FL
Julio.Delacruz@us.army.mil

ABSTRACT

Historically, ground-warfare simulation programs have developed project-specific terrain generation systems. These stove-pipe systems satisfy a single program's requirements, involve a lot of manual editing, and employ little verification during processing. The government has invested millions of dollars developing runtime databases covering the same geographic areas. The generated databases lack correlation due to different processing tools and techniques, resulting in fair-fight issues and visual anomalies when interoperating in a confederacy.

The effort described in this paper strives to solve these problems. The goal is a means to rapidly generate high-fidelity urban terrain databases using existing applications while removing dependence on any particular tool. In addition, the solution must have the flexiblibility to evolve as programs identify new requirements. The resultant capability must import from a wide variety of sources, clean and normalize source data to a consistent representation, and deliver a correlated dataset that meets the needs of a confederacy.

This paper describes the technical challenges involved with developing an adaptable urban terrain generation framework. We'll take a look at each how the components of the system interact and discuss problems, deficiencies, and bottlenecks encountered during development. Finally, we conclude with the current state of the system and to what degree it is meeting overall expectations.

ABOUT THE AUTHORS

Chuck Campbell is a Principal Scientist at ARA with over 15 years experience developing Semi-Automated Forces (SAF) software. Mr. Campbell was a developer and technical lead for Synthetic Natural Environment representation and reasoning on US Army programs including CCTT, WARSIM, and OneSAF Objective System. Mr. Campbell is the Program Manager for the Rapid Unified Generation of Urban Databases (RUGUD) high-resolution urban database generation effort. Mr. Campbell holds a Bachelor of Science degree in Computer Science from Indiana University and a Master of Science degree in Computer Science from the University of Central Florida.

Kevin Wertman is a Senior Scientist at ARA. For the last five years, he has been involved in research and development in the Synthetic Natural Environment field, early in his career as a member of the SEDRIS core team and later a developer and technical lead of the Framework Optimizing Rapid Generation of Environments (FORGE) terrain generation effort. Mr. Wertman is the Deputy Program Manager and technical lead for the Rapid Unified Generation of Urban Databases (RUGUD) high-resolution urban database generation effort. Mr. Wertman holds a Bachelor of Science degree in Computer Science from the University of Central Florida.

Julio de la Cruz is a Lead Principal Investigator for the Rapid Construction of Urban Terrain Database for Training Science and Technology Objective (STO) at the Simulation and Training Technology Center of the Army Research, Development and Engineering Command. Mr. de la Cruz was formerly at U.S. STRICOM where he was the Principal Investigator for multiple acquisitions of 6.2 Exploratory Development programs. Mr. de la Cruz holds a Master of Science degree in Engineering from Texas A&M University and a Bachelor of Science degree in Engineering from The City College of New York.

A Framework for Generating High-Fidelity, Interoperable Urban Terrain Databases

Chuck Campbell, Kevin Wertman
Applied Research Associates, Inc.
Orlando, FL
ccampbell@ara.com, kwertman@ara.com

Julio de la Cruz
Army RDECOM/STTC
Orlando, FL
Julio.Delacruz@us.army.mil

INTRODUCTION

Generation of correlated, high-resolution urban terrain databases is challenging. Source data comes in many formats with varying content. Runtime compilers accept different types of source data and manipulate it into formats optimized to meet particular needs, whether for visual, semi-automated forces, C4I, or other applications. And finally, data sets are massive even by today's computing standards.

Designing an infrastructure to support integration of different COTS and GOTS tools is also challenging. Each tool has unique capabilities. By design, they typically handle processing in isolation. We want to use the best of each tool in a collaborative environment.

Our research started at the heart of the problem, establishing a data model to represent source data in a consistent manner, and a mechanism to persist the data over time. From there we tackled how to provide and receive information to and from external plugin tools and track modifications. This design accounted for distributed and parallel processing. As the design progressed, we incorporated utilities to report status, collect and store information, and verify correctness. To increase usability, we developed a user interface to assist composition, execution, and analysis of terrain generation processes.

Many of our initial concepts and goals are operational. We have laid the foundation. Still, much work remains to achieve our overall goals. We now report the status of our efforts.

MUDM

A data model lies at the core of any terrain generation. The Master Urban Data Model (MUDM) defines all features and attributes required to represent urban environments suitable for military training simulations. Attributes have a label (name) and a value. The runtime software uses the MUDM to associate attributes to features and values to attributes. The runtime structures accommodate these associations so that the data dictates the layout of internal structures. This data-driven approach means the data model can evolve and adapt to new requirements without requiring software modifications.

We chose the OneSAF Objective System (OOS) Environment Data Model (EDM) (Miller, 2002) as a basis for the MUDM. The OOS EDM contains a rich feature set and attribution definitions for urban environments. We augmented the OOS EDM with structural information for buildings and tunnels to support physics-based weapons effects modeling. The software did not change to support storage and retrieval of the new attributes, substantiating the flexibility of the approach to easily accommodate new requirements.

The selection of the OOS EDM as a starting point implies using the SEDRIS Environmental Data Coding Standard (EDCS) (Birkel, 1999) as the data dictionary. The National Geospatial-intelligence Agency (NGA) uses the Feature Attribute Coding Catalog (FACC) for their data model. SEDRIS developed the EDCS to address gaps between FACC and modeling and simulation training requirements, and provides a mapping between FACC and EDCS. The data dictionary chosen isn't critical to a data-driven architecture, since the software processes the data generically.

The MUDM facilitates data verification. Each attribute has associated default, minimum, and maximum values. The verification module checks imported data values against defined ranges and reports anomalous data. We chose to snap anomalous data to the nearest bound to most closely preserve original data. Other options include setting values to their default or continuing without modification while logging the detected inconsistency.

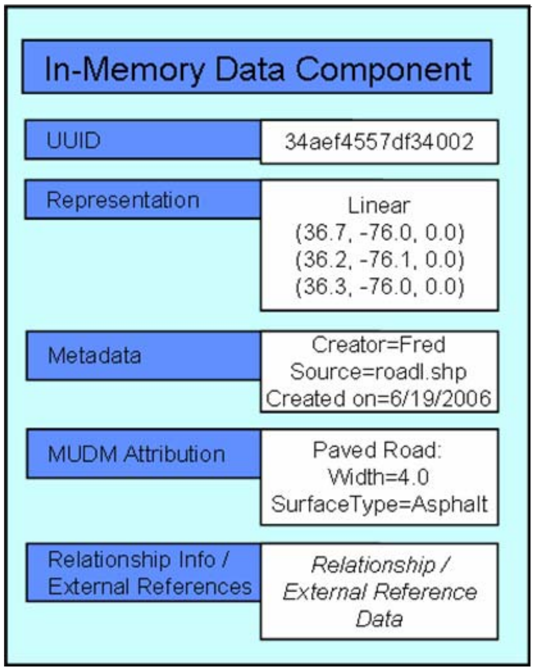

Figure 1. The in-memory data component is capable of holding any type of source data.

In-Memory Data Component

We desired a single in-memory data structure to simplify moving data through the framework. This data structure holds multiple representations of information. It contains fields for metadata, attribution, and a physical representation such as the points of a line, elevation posts in a grid, texture, imagery, or feature model. Figure 1 displays some of the fields in our modular data component. The data structure also houses a globally unique identifier to facilitate specific object lookup. Later, we identified the need to maintain relationships and external references between objects. The internal data structure's modular design of the made this a simple matter of adding new fields. None of the existing interfaces required modification.

We also needed a layered data manager to store and track all in-memory data objects. This approach encapsulates the burden of memory management and object retrieval away from the rest of the framework. This implementation has basic hierarchical layers and a quick indexing mechanism to retrieve objects from them. Figure 2 illustrates the hierarchical structure of the layered data manager. All data that flows through the framework resides in this data manager construct. Hiding storage details behind the data manager interface affords us the option of implementing caching mechanisms to reduce the number of objects in memory at any given time without affecting the rest of the framework.

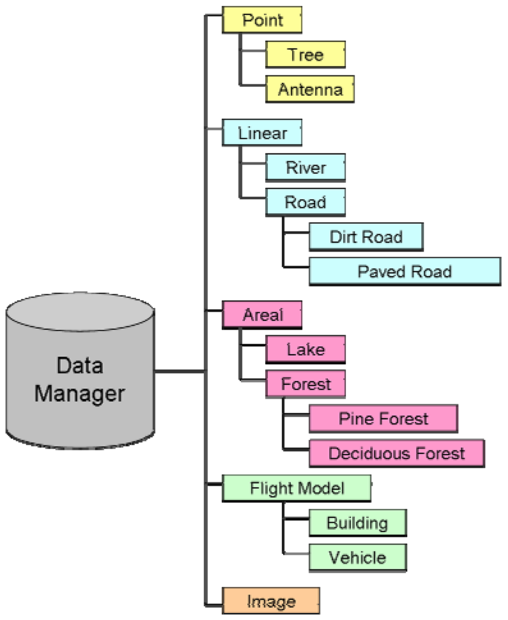

Figure 2. The layered data manager sorts and tracks all in-memory data components.

Master Urban Database

Terrain generation systems typically process huge amounts of data. Persistent storage is required to save interim data during processing. Efficient and robust data storage became our challenge. We initially explored two possibilities. The first involved creating a new optimized binary format in which to store environment data. The second entailed storing environment data in a relational database mechanism.

We did not have time to fully design and implement both solutions to determine the better solution. We decided to explore the relational database mechanism. The full relational database mechanism quickly proved cumbersome. Changes or additions of new data types would require re-engineering a new set of database table structures and storage of large imagery and elevation data would require preprocessing, such as chunking the raster data into smaller blocks of the binary data suitable for database storage. Conversely, a complete binary format would require reading and writing to physical storage at all stages, even simply to determine if the data met some criteria that was being fetched from the MUDB. In addition, binary storage would require significant up-front design and testing of the new format before we could even determine if it would be scaleable enough to use for the MUDB.

We chose to implement a hybrid storage system for the Master Urban Database (MUDB). We store the physical representation of data on disk in standard formats (see Table 1).

Table 1. Physical data representation is stored to disk in standard, open formats.

- Geographic Imagery – GeoTiff & ECW (with/without compression)
- Elevation Data – GeoTiff (with/without compression)
- Terrain Feature Data – ShapeFile
- Texture Data – PNG
- Building Structural Data – U2MG (Mann, 2004)

A relational database houses attribution, metadata, external references, relationships, and other indexing information such as spatial extents of the data component. This flexible approach uses the SQL (ANSI, 1992) language to build queries on the data stored in the MUDB. The relational database indexing mechanism makes it easy to build complex queries. We leverage optimized search techniques provided by relational database technologies to maximize performance.

We began with a simple MUDB design. It covered a segmented storage mechanism for indexing, metadata, and attribute information. As shown in Figure 3, the design calls for a separate component responsible for storing each set of data. This way, the MUDB stores all data in parallel and each component can optimize to the particular type of data it handles. As the implementation evolved, this approach allowed changes to portions of the design to work within the framework efficiently without rewriting the interface code. For example, we redesigned the metadata storage mechanism to record values once and reference the stored value from multiple data elements for which it applies. This redesign affected no other piece of the MUDB system.

The segmented storage design facilitated rapid implementation of multi-threaded storage and retrieval algorithms. Each thread stores/retrieves distinct segments of data from the different storage services. This alleviated bottlenecks by placing input/output on separate threads of execution. This allowed parallel retrieval operations, maximizing CPU usage and accelerating MUDB storage and retrievals.

We designed the MUDB physical format handler interfaces to avoid reliance on any particular format. The MUDB contains a standard format handler interface for integration to the framework. An XML-driven mapping mechanism associates each physical representation to a format handler. We have implemented format handlers for standard formats seen in Figure 4. The MUDB records the format handler chosen to store an object with the object indexing data. The MUDB uses this information to select the correct format handler to read the data. This allows continual upgrade to the underlying storage mechanism without affecting access to older MUDB content. For example, if we chose to replace our ShapeFile format handler with a SEDRIS transmittal handler for terrain features, we could update the XML mapping file to point all terrain feature data to the SEDRIS format handler. Subsequently, the MUDB would store all newly created data objects of that type using the SEDRIS format handler. However, the MUDB would read and write all previously stored objects with the ShapeFile format handler.

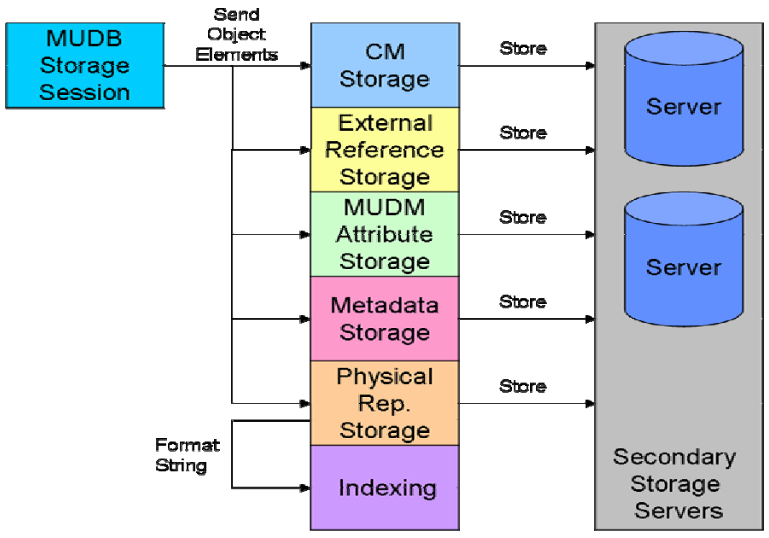

Figure 3. The MUDB implements a segmented storage design that allows for a multi-threaded storage engine

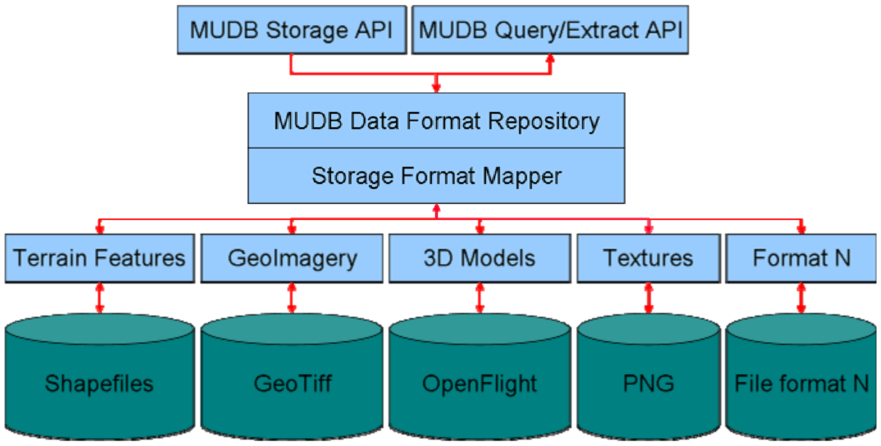

Figure 4. The reconfigurable format repository uses a variety of industry standard formats.

Our original format handlers were all disk based storage mechanisms; however, we had trouble with shared file handles in our ShapeFile format handler that wrote directly to disk. We replaced it with a format handler that instead wrote the binary data to a relational database. This change was transparent to the rest of the MUDB.

As our relational database experience grew, we began optimizing our storage techniques using *stored procedures*. Stored procedures save processing logic in the database server that triggers when data is stored to it. This optimization allows handling the entire configuration management system in stored logic on the database server. This approach offloads the costs associated with tracking earlier revisions of data components. Offloading all operations associated with storing revision data to the data server simplifies the MUDB storage implementation to only store updated information. This permits the stored procedure logic to identify and process revision tracking information automatically in the database server.

With relational database tables and revision tracking in place, we could begin building a complex MUDB query API on top of simple SQL statements. The search mechanism can use any segmented data in a query. We designed a simple API to build atomic query operations (bounding box query, MUDM component category query, etc), and extended that API to allow the logical combination of atomic search criteria to construct complex query operations.

Plugin Architecture

A plugin architecture provides a means to integrate COTS and GOTS tools. For terrain generation, plugin interfaces need to support data importers, automated and manual manipulation tools, and data exporters. Other useful capabilities include inspection tools, metrics collection, verification and validation, and metadata gathering.

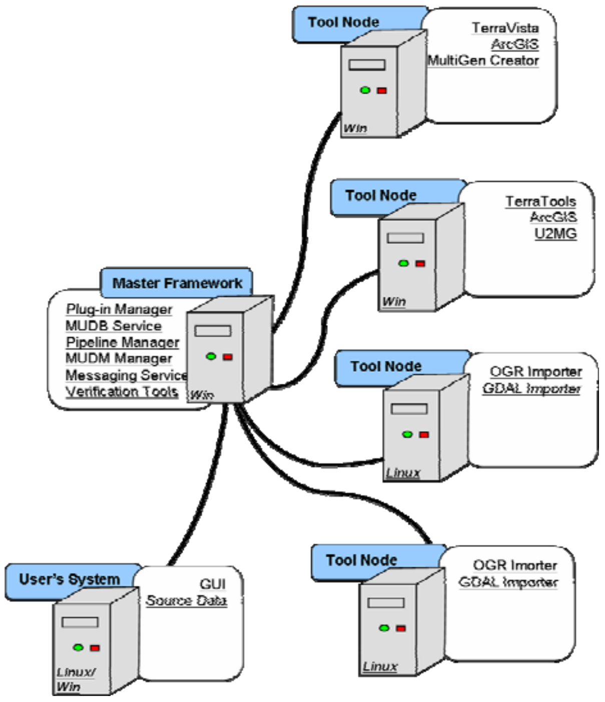

Figure 5. Terrain database processing products from various COTS/GOTS developers integrated within a distributed plugin architecture.

Distributing processing among several heterogeneous workstations can increase performance (Vinoski, 1997), and also allows us to take advantage of tools running on different operating systems, as shown in Figure 5. This is critical to completing the terrain generation within the 96 hour timeframe.

We explored several remote procedure call implementations for reuse as the information distribution mechanism, including SOAP (Ehnebuske, 2000), XML-RPC (Winer, 2000), and CORBA (OMG, 99). The need to support C++ clients ruled out use of the language-specific Java RMI (Sun, 1998). SOAP and XML-RPC use XML as their transport mechanism. Their underlying text-based communication results in greater network traffic and slow translation. CORBA's binary object transfer provides better performance. Unfortunately, it requires more development effort. In the end, the speed gain using CORBA outweighed the shorter implementation effort of SOAP and XML-RPC.

We designed the CORBA plugin interfaces to provide two-way communication. The framework houses servants for the data components and plugin management objects. The plugin SDK encapsulates servants for the plugins themselves.

Implementing the CORBA layer required several parts. We wrote IDLs for the various plugin interfaces and data component structures. The basic data components require representation in the CORBA IDL so the framework can distribute them via CORBA interfaces. A remote plugin manager CORBA IDL object encapsulates remote plugin access to the framework. This CORBA-based plugin manager object allows remote GOTS/COTS plugins to register and interact with the centralized framework. To expose remote plugin capabilities through the distributed CORBA mechanism, we recreated each identified plugin interface with a CORBA IDL object. This results in a 1-to-1 mapping from the original java plugin interface to a CORBA object that the framework can use in distributed processing.

The CORBA interfaces evolved throughout development. As we identified a new need for framework information, such as direct access to the MUDM, we implemented the IDL structures and the servants to expose those objects over CORBA.

We monitored several concerns from the start. The framework needed to optimize performance, memory management, data transfer, and tool integration flexibility.

A functional source data importer allowed stress testing the framework implementation using larger amounts of data. Our first noticeable problem occurred in the CORBA data component servant implementation. Our inexperience with CORBA resulted in inefficient memory usage. Rather than building CORBA-defined structures, we exchanged feature attribution through servants. With millions of objects flowing through the system, each with upwards of 10 attributes, these servants saturated the memory footprint. We refactored the CORBA implementation to translate structures and only spawn CORBA servants when necessary. In addition, we added a local cache of data component information to minimize CORBA traffic. These two modifications resulted in an increase in performance while significantly reducing memory usage and network traffic.

With the plugin interfaces in place, the architecture expanded to include inspection tools and metrics collection interfaces. We implemented these interfaces in pure java and run them as local plugins. We implemented a prototype plugin manager to keep track of all plugins and make their services available to the rest of the system.

C++ Plugin SDK

Since C++ is the most commonly used language for terrain generation tools, we needed a C++ plugin interface with which they could integrate. We started with the goal of making the C++ Plugin SDK very easy to use and understand. We wanted to hide all CORBA complications away from users of the SDK. To accomplish this, we wrapped each CORBA interface and structure with a local implementation as shown in Figure 6. Under the covers, the CORBA interfaces hook directly to abstract methods, which allow plugin processing to enter third party plugin code.

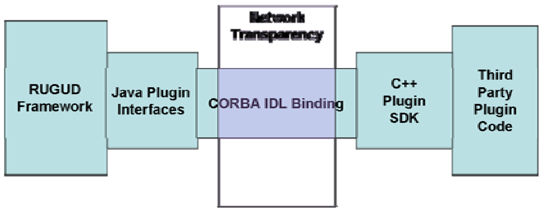

Figure 6. The C++ Plugin SDK hides the complexity of the CORBA layer and distributed programming from third party plugin developers.

The initial C++ plugin SDK contained several deficiencies. The original design neglected to provide the ability to configure plugins with various parameters. We implemented a generic set of plugin parameter types and exposed an API in the C++ plugin SDK for third-party plugins. This API allows plugins to describe their parameters with a name, description, parameter type, and default value. With this interface update, plugins could expose their parameters to users of the system using a common interface.

We implemented several test tool plugins on top of the C++ plugin SDK. They print out debugging information about data objects to help debug processing problems and verify the integrity of the data translations in the importer tools. The test tools accept RUGUD data objects and print the object representation information to a plain text file. We compared this file to output from the original source data to determine whether any data loss occurred.

Metadata

A good terrain generation capability should collect and maintain metadata throughout the process. Metadata provides valuable information about the origin, content, and history of the data. It allows others to quickly and easily understand details of the data stored within the MUDB. Otherwise clients would need to interrogate the data, expending valuable resources just to determine if the dataset contains the desired content for their purpose.

We designed a very flexible key-value metadata implementation to associate with each in-memory data component. We wanted to support any string metadata key/value pair. Our longer-term design uses an XML driven set of mandatory metadata requirements, with validation of those requirements built into the system. This design remains in place, but the implementation simply supports open-ended metadata values.

MUDM Mapper

The framework normalizes imported data to the MUDM. This mapping involves translating feature and attribute labels and converting units of measurement to the metric system.

Mapping source data to a consistent internal representation is a difficult task. Some data sources, such as shape and OpenFlight, can represent data using radically different labels and arrangements. This led us to think about a way to facilitate creating and using mappings from different sources to the internal MUDM representation.

We designed and implemented a MUDM mapper. Mappings are contained in an XML-based mapping file. We developed a configurable mapping library that uses a mapping file to control the normalization of incoming source data. The beauty of this data-driven approach is that the library interfaces to the MUDM mapper should never change. As the MUDM, the MUDM mapper, and the XML structure evolve, the importer plugin code reaps the benefits of the updated mapping infrastructure with a simple re-link to the new library.

We have implemented simple one-to-one mappings with unit/scale conversions. In the future, we can fully flesh out the MUDM mapper. The one-to-one mappings can handle a large percentage of the datasets in the world. Additionally, we implemented the ability to capture metrics about missing or incomplete mapping files, so that as users of the framework encounter more diverse sets of source data with various attribution schemes, they have the information needed to quickly adapt the mapping files to handle that new source data.

Pipeline Processing

The processing engine manages data flow and execution during the terrain generation process. To address project goals, the design took desired capabilities into account, including parallel processing, easy addition of future capabilities, and flexible configuration.

We began thinking of each data processing capability as a separable component. We defined an XML-based specification for components to define their specific requirements and capabilities. The engine became completely open-ended with respect to processing flow.

A user can compose *pipelines* by sequencing capabilities together in a logical fashion as demonstrated in Figure 7. This approach allows for different terrain generation processes, which may prove useful for generating different output formats. The pipeline approach supports using the right tool for the right job.

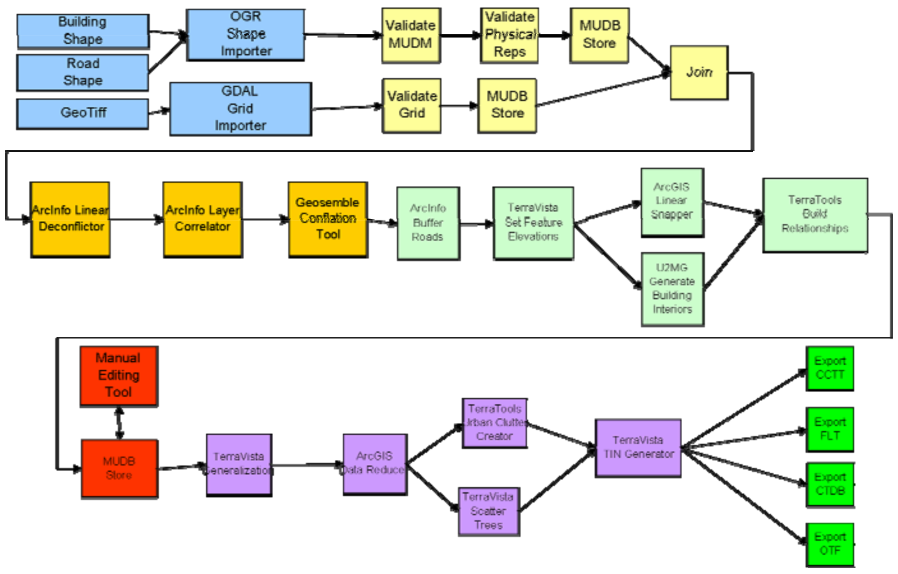

Figure 7. Pipelines compose various processing capabilities into a cohesive processing unit.

Pipelines can start by importing data from a native data source via an importer plugin. Alternatively, pipelines can start by specifying an MUDB load operation with a query for which types of data should be loaded for pipeline processing. In other words, pipelines can use new or previously processed data sources transparently.

The pipeline approach supports parallel processing through simultaneous execution of multiple components. The processing engine splits processing into multiple parallel threads and combines them back together as they complete. We concerned ourselves with the problem of safely sharing data across parallel threads of execution. The initial design used a fine-grained locking mechanism. This placed a burden on plugin developers to lock and unlock data when editing. We decided we needed a mechanism to filter data to components while guaranteeing a unique dataset for each thread.

Data Filtering

Populating a high-resolution urban terrain database requires an enormous quantity of data. In a composable framework, this data flows from component to component and, possibly, workstation to workstation. We recognized the need for data filtering to reduce the amount of data transferred between nodes.

Data filtering occurs on many levels. Pipeline components may filter data to extract only the features and attributes they need to perform their task. Remote tool plugins also needed to be able to provide filtering information, so we implemented interfaces to specify filters in the C++ plugin SDK. Terrain generation operators may add their own level of data filtration to pipeline processing components to further customize the pipeline process. Each stage potentially reduces the amount of data transferred.

Data filters also assist parallel processing. Using filters reduces the set of required information for a component. Since the component doesn't need all data for an area, another component can work on the same area provided the data requirements do not overlap. Using the XML-based configuration files, the validation module can check for data requirement conflicts between parallel components during pipeline construction.

GUI

A graphical user interface (GUI) aids construction of urban database generation pipelines. Without it, pipeline construction would require manual editing of XML-based configuration files. We performed some research and found the Eclipse Rich Client Platform (Lam, 2005), a mature open source user interface builder. It had all of the capabilities we required and allowed us to develop a graphical front-end to the framework with minor effort.

The GUI puts power and usability in the hands of the user. It provides drag and drop pipeline construction using components. Split and join components present a means to parallelize execution into multiple threads. Newly developed components appear in the component list and become available for immediate use.

As shown in Figure 8, the GUI incorporates several additional features. A source data scanner can analyze directories and present discovered source data for import. A metrics window presents real-time statistics such as memory usage, number of threads executing, number of features imported by type, and execution time of each component. An information window displays metadata for source data products, components, and MUDM elements when selected. A fault window reports verification and validation errors. A logger window provides several levels of system information, such as warnings and debug statements.

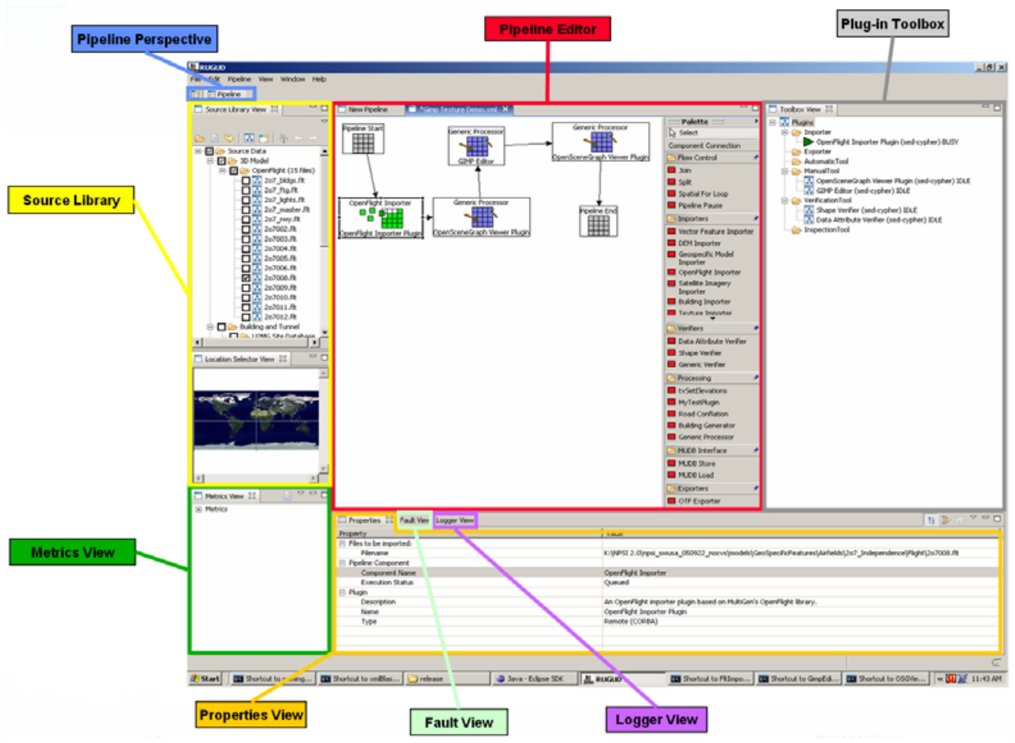

Figure 8. The RUGUD user interface presents information in a compact, intuitive manner.

Messaging

Our initial GUI design and implementation was in the same process space as our framework processing objects and threads. This tight coupling caused certain GUI operations to impinge upon the framework's ability to accomplish the data processing tasks. It also left us with only the single GUI interface to interact with the system. It also forced us to run the master framework processing on the same workstation as the desktop components of the system. In addition, we have a requirement for a web-based interface to the framework system, which the current mechanism was not adequate to handle.

We analyzed several messaging frameworks for this separation of the framework from the GUI, including reusing CORBA or implementing SOAP interfaces. We had encountered a message-oriented middleware called xmlBlaster (Ruff, 2000) during our early research for the distributed plugin architecture. Since it was a cross-platform, multi-language solution as shown in Figure 9, we decided to prototype a messaging architecture using xmlBlaster.

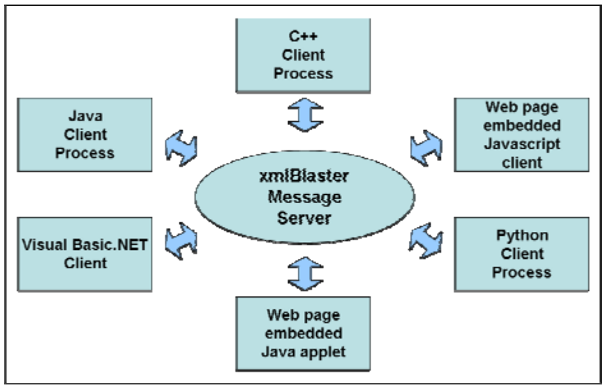
C++ Client Process
Java Client Process
Web page embedded Javascript client
xmlBlaster Message Server
Visual Basic.NET Client
Python Client Process
Web page embedded Java applet

Figure 9. xmlBlaster provides a cross-platform, multi-language distributed messaging middleware.

We quickly defined several message types, including event/subscribe, request/response, and command/result message pairs. Because xmlBlaster is very flexible regarding the content of the messages it propagates, we simply serialized/de-serialized our java-based message classes into the xmlBlaster message payload. This seemed the quickest way to prototype the distributed messaging system. The downside of this design decision is that the messages are often far larger than the actual content due to the overhead of java serialization information. The upside is that we are able to define new messages and implement them in a matter of minutes due to the generic serialization of messages into the stream.

With the distributed messaging system in place, we separated the GUI from the framework and defined all messages required for the GUI to interact with the framework. This approach supports running the GUI from a remote workstation and attaching to any running framework. In addition, it enables rehosting the GUI on a different platform to support, for example, handheld player units.

Conclusion

We have presented the basis for an open, adaptable framework that supports cooperation among COTS and GOTS terrain generation tools. This framework is in place and capable of proof-of-concept operation. We have demonstrated a simple urban terrain generation composed of several COTS and GOTS tools in cooperation. This demonstration illustrates the ability to use the right tools for the right job in an easily configurable system.

The framework follows the modular architecture design. Well-defined interfaces encapsulate each component from other components. Modification or replacement of components does not affect other parts of the system. This approach provides an evolutionary development path for future expansion.

The modular architecture supports monitoring and observation utilities. Our implementation supports metrics collection, data verification, revision tracking, and data preview. Developers can extend or add new types of capabilities to provide additional insight into the generation process as it executes.

This effort shows a reduction in urban terrain generation cost and schedule is achievable. The key is that source data is processed and stored once, at its highest resolution and fidelity. Newly collected source data augments or replaces existing data. Data filtering provides the means for reusing datasets to generate multiple databases at different fidelities. This approach supports generation of a wide variety of correlated end-use products including visual and SAF runtime databases, paper maps, and interchange formats like SEDRIS.

An open, accessible, low- or no-cost urban terrain generation capability remains our long-term objective. We continue to improve the underlying infrastructure to support advanced capabilities while extending the breadth of urban database generation resources.

ACKNOWLEDGEMENTS

The authors would like to thank RDECOM and PEO STRI for their continued support of the RUGUD effort.

REFERENCES

Mann, J., Pigora, M., Kraus, M. (2004). A High Resolution Urban and Underground Model Generator. Simulation Interoperability Workshop, Spring 04 Proceedings.

Miller, D., Janett, A., Nakanishi, M. (2002). An Environmental Data Model for the OneSAF Objective System. (02-SIW-082), Proceedings of the Fall Simulation Interoperability Workshop, September 2002.

Birkel, Paul A. (1999). SEDRIS Data Coding Standard. (99S-SIW-011), Proceedings of the Spring Simulation Interoperability Workshop, March 1999.

S. Vinoski (1997). CORBA: Integrating diverse applications within distributed heterogeneous environments. IEEE Communications Magazine, 14(2), February 1997.

OMG (1999). CORBA overview, Object Management Group, http://www.omg.org/corba/, 1999.

Marcel Ruff (2000). White Paper xmlBlaster : Message Oriented Middleware (MOM), http://xmlblaster.org/xmlBlaster/doc/whitepaper/whitepaper.html, 2000.

Lam, T., Gotz, A. (2005). Leveraging the Eclipse Ecosystem For The Scientific Community, Proceedings of the 2005 ICALEPCS, Geneva, 2005.

Box, D., Ehnebuske, D., Kakivaya, G., Layman, A., Mendelsohn, N., Nielsen, H. F., Thatte, S., and Winer, D. (2000). Simple Object Access Protocol (SOAP) 1.1. World Wide Web Consortium (W3C) note 08 May 2000. Available at http://www.w3.org/TR/SOAP/.

Dave Winer (2000). XML-RPC, 2000. http://www.xmlrpc.com/spec.

Sun Microsystems (1998). Java RMI specification, October 1998.

ANSI (1992). SQL Standard, 1992. X3.135-1992.