CONF-9507192--1
ANL/DIS/CP--87196

A Smalltalk-based Extension to Traditional Geographic Information Systems

Peter A. Korp
Gordon R. Lurie
John H. Christiansen

Advanced Computer Applications Group
Argonne National Laboratory
{lurie, korp, jhc}@dis.anl.gov

Abstract

The Dynamic Environmental Effects Model©(DEEM[1]), under development at Argonne National Laboratory, is a fully object-based modeling software system that supports distributed, dynamic representation of the interlinked processes and behavior of the earth's surface and near-surface environment, at variable scales of resolution and aggregation. Many of these real world objects are not stored in a format conducive to efficient GIS usage. Their dynamic nature, complexity and number of possible DEEM entity classes precluded efficient integration with traditional GIS technologies due to the loosely coupled nature of their data representations. To address these shortcomings, an intelligent object-oriented GIS engine (OOGIS) was developed. This engine provides not only a spatially optimized object representation, but also direct linkage to the underlying object, its data and behaviors.

[1]DEEM Has Been Copyrighted by Argonne National Laboratory. All Rights Reserved. Argonne is operated for the Department of Energy by the University of Chicago. Work supported under a military interdepartmental purchase request from the U.S. Department of Defense, Joint Chiefs/J-8, through U.S. Department of Energy contract W-31-109-Eng-38.

Introduction

The Dynamic Environmental Effects Model©(DEEM), under development at Argonne National Laboratory, is a fully object-based modeling software system which supports distributed, dynamic representation of the interlinked processes and behavior of the earth's surface and near-surface environment, at variable scales of resolution and aggregation. DEEM has been under development since late 1992. It began as a response to a Department of Energy (DOE) need for a means to realistically visualize the impacts of environmental remediation measures at various DOE sites -- before, during, after, and long after remediation. In addition to photo-realistic 3D color visualization, this effort required modeling of interrelated environmental effects, such as linked climatological, hydrological, and ecological (forest succession) modeling for reforestation at DOE sites. The DEEM concept was subsequently adopted in mid-1993 for prototype terrain reasoning and synthetic terrain generation applications, under sponsorship of the Joint Chiefs of Staff J-8 Directorate. Shortly thereafter, the U.S. Air Force selected DEEM as the software framework for multi-disciplinary environmental modeling in support of theater-level mesoscale weather forecasting. Many of these real world objects used by DEEM are not stored in a format conducive to efficient GIS usage. Their dynamic nature, complexity and number of possible DEEM entity classes precluded efficient integration with traditional GIS technologies due to the loosely coupled nature of their data representations. To address these shortcomings, an intelligent object-oriented GIS engine (OOGIS) was developed. This engine provides not only a spatially optimized object representation, but also direct linkage to the underlying object, its data and behaviors.

Members of the OOGIS development team had previously worked on a multitude of cross-platform GIS and object oriented projects over the past seven years. Lessons learned over that period contributed greatly to help us determine how key architectural issues should be addressed. The need for the GIS engine to be a drop-in module or framework was key because of the many different sponsors and projects it would need to support. The ability to represent data in a context-sensitive fashion, an integrated robust query protocol, cross-platform support and an optimized storage mechanism for geo-spatial data were all key concerns the development team focused on.

The initial stages of the OOGIS development were done in C++ on a UNIX workstation. It quickly became apparent that development of the system in C++ would require the creation of object meta-data facilities, run-time dynamic dispatching of methods, run time type information and other features commonly available in Smalltalk. After a review by the development and management teams it was decided that the OOGIS system would become the Smalltalk proof-of-concept for DEEM. ParcPlaces' VisualWorks was chosen as the development environment because of the strong cross-platform support and mature nature of the product. After 3 weeks of development on the OOGIS system, it was quite evident that development times were significantly reduced over the C++ version. This was true even though it was our group's first Smalltalk-based project.

The programming and development paradigms that Smalltalk encouraged allowed the development team to experiment with new designs and ideas. Because there was no need to wait for endless compile and link cycles, new approaches could be tried and discarded without a significant amount of wasted effort. The large class library present in VisualWorks and the ease of integration with legacy C code allowed the developers to focus on key infrastructure issues rather than mundane re-coding of common tasks.

Object Protocol

Before work on the Smalltalk version of OOGIS began, it was decided that a mechanism was needed for objects and their classes to convey information about functionality that they implement, the attributes that they wish to make public, and their method signatures. The use of a common set of protocols was chosen as that mechanism. Objects of any class were free to implement any of a given set of protocols with the restriction that all methods of a given protocol must be implemented for that object to conform to the protocol. Objects are free to inherit and conform to any number of protocols. This allows us to declare an interface to an object while concealing its class and to capture similarities among related classes that not hierarchically related. These features are useful in building a distributed object framework and more generally for the implementation of proxies. By freeing developers from having to subclass off of a common superclass, yet still being able to strongly check protocol conformance dynamically, it is now practical to plug existing non-related classes into the OOGIS framework.

This protocol based, dynamic, object oriented approach mitigates some problems that have plagued traditional GIS systems:

1. The relational model of the world, that even some so-called object oriented GISs suffer from. Losing information about an objects structure by forcing objects into BLOBS. OOGIS lets you view the data in a relational manner if you so choose, but doesn't force you to map conceptual and real-world objects into and out of a relational model.
2. Difficulty in integration with existing external models. OOGIS provides a simple driver-based framework to simplify the process of model and data integration and ingestion.
3. The proprietary "scripting" languages used by traditional systems. There has been a lack of integration between the scripting language, query language and traditional languages used in external models and applications. Many times the scripting language is simply that, lacking the expressiveness of a full-blown programming language. OOGIS circumvents these issues by using Smalltalk for all three purposes. A Smalltalk API is available to drive the OOGIS framework. By using common Smalltalk container protocols, the spatial query language is implemented as an extension to Smalltalk.

Protocols can vary in complexity. A simple protocol may consist of a single method, such as the Spatial Indexing protocol. This protocol allows any object to be spatially indexed by responding to a method which signifies its extent. Complex protocols, such as the Query protocol, define methods that themselves will dynamically define another protocol, or meta-protocol. In practice there are several ways for an object to conform to a protocol. It can implement all the methods of the protocol itself. However, an object doesn't really have to implement every method in the protocol. It can selectively override methods of an inherited protocol. It can implement a subset of methods if a delegate object exists that implements the missing methods, or a default object for that class of objects exists that implements the missing methods of the protocol. Note that these objects can themselves contain cascaded references to delegates. Or an object can use some combination of these methods.

OOGIS relies on a small set of protocols for some of its advanced features:

- A dynamic visualization protocol, allows objects to dynamically alter their visual representation based on some external factors. This is discussed in more detail later.

- A query protocol, discussed below.

- A spatial indexing protocol, allows arbitrary objects to be stored for efficient retrieval via spatial queries.

- An object editing protocol, allowing objects to define meta-data describing how to build a GUI inspector for that object. The meta-data defines a hierarchical structure which supports the creation of more complex and appropriate types of inspectors.

This self-describing object technology was designed for the original C++ OOGIS. While the authors were aware of similar efforts in this area it was still a pleasant surprise to find that the actual process by which the inspectors are built is similar to the way a UIBuilder builds windows in VisualWorks. In fact the entire self-describing object hierarchy is conceptually very similar to SubCanvasses in VisualWorks.

Query Facilities

The OOGIS query mechanism depends heavily on meta-protocols. These meta-protocols extend the expressive power of the query engine beyond that of relational calculus. It is once again important to note that these meta-protocols are dynamic in nature, derived from the base query protocol, allowing run-time changes to the query engine.

By using common container protocols like, "Select:, Collect:, etc.,,", the query engine is implemented as an extension of Smalltalk. As stated before this approach has several advantages. The most noticeable is that no special handling is needed to code a spatial vs. a traditional Smalltalk query. As a side-effect, this extends nicely into the object database domain. Performance optimizations are possible with an OODB if that OODB engine is capable of running Smalltalk code on the server side. Since the spatial query is Smalltalk code, the server can determine exactly which objects are necessary to send to the front end. Another benefit of this approach is that queries can be dynamically compiled by Smalltalk into executable methods. There is no longer a need to write code covering every possible query combination.

Allowing the query-protocol-derived meta-protocols to provide source code for inclusion in the query execution is essential in explaining the flexibility of OOGIS. This means that queries can implement not only typical predicate-based operations, but can also invoke methods on arbitrary objects for query result calculations. Since operations like drawing are essentially queries into the spatial database, objects can inject code containing arbitrary method invocations, calculations and dynamic attribute computations. These snippets of code can perform actions such as executing external applications, which is particularly important in the modeling and simulation communities. The "standard" parts of the query protocol define typical spatial operators such as: within, intersect, buffer type operations, etc... A default implementation of these operators is defined by the GeoObject class in the cases where "smarter" objects are not required or do not implement the protocol.

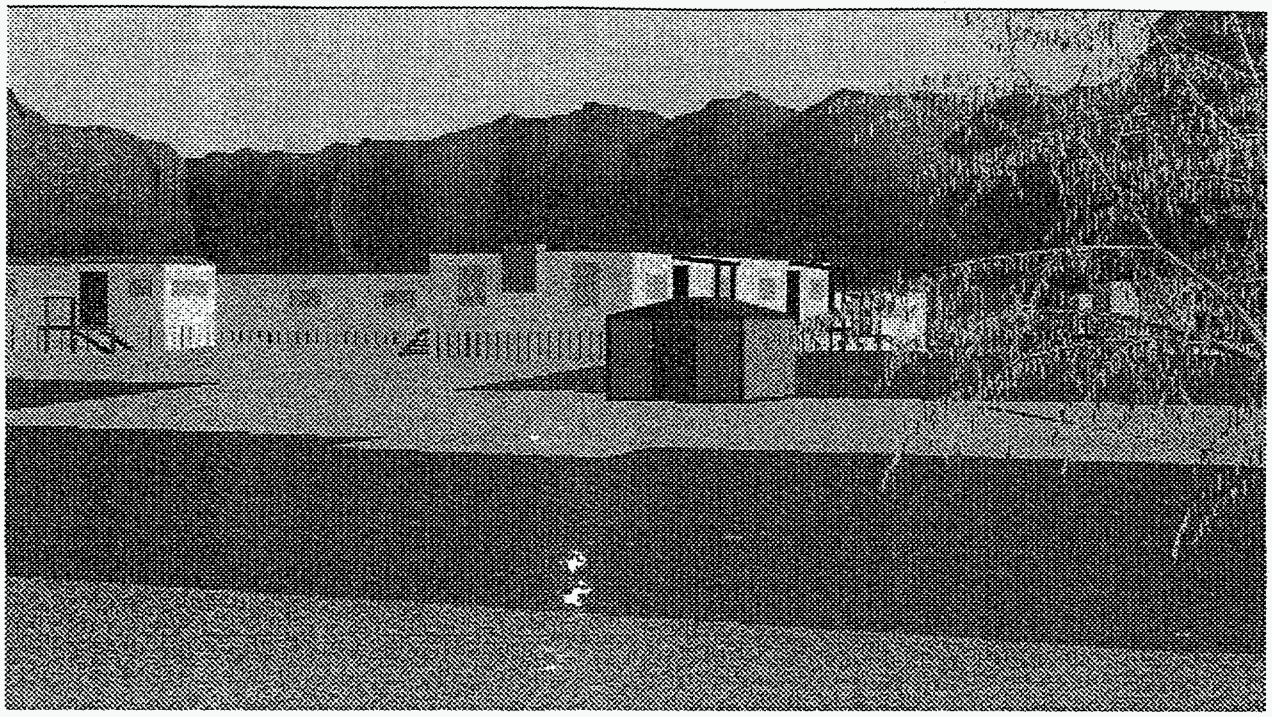
With all the dynamic behaviors occurring in a spatial access it would seem logical that performance would be less than optimal. In actuality the move to the dynamic protocols had the effect of speeding up some query operations by an order of magnitude while hardly impacting the performance of simple queries such as drawing. This is not to indicate that we have found the perfect balance of dynamicism and ultimate performance. We continue in researching optimized access methods for large complex spatial objects.

GeoDB framework

At the heart of the OOGIS engine lies a framework that encompasses spatial data representations, indexing, and context-sensitive visualization information. The foundation of the spatial data representation is a lightweight object, GeoObject, whose main purpose is to provide storage for the optimized spatial geometry and can optionally provide a delegate to access the "intelligent" underlying objects. If a delegate exists it can optionally provide context sensitive spatial geometry without requiring storage in the GeoObject, but at a performance penalty. For example, depending on a viewer's relative "distance" from a forest object, the forest could provide its spatial geometry as a rough mesh, a detailed geometry including individual trees and finally as trees with branches and leaves. The performance penalty incurred is an extra method lookup and execution. The execution of this method could involve something as simple as a table lookup or something as complex as running a model or reading external ArcInfo files. Though protocols can many times obviate the need for certain types of subclassing, the GeoObject class can be, and is, subclassed to provide additional features such as rasterization or gridding of data.

The spatial indexing framework is quadtree-based and provides storage and access for objects that respond to the spatial indexing protocol. An adaptive data-driven pruning algorithm, with similarities to Peano-Hilbert, is incorporated in the quadtree. Each layer of the tree is conceptually independently indexed, limiting the depth of the trees. Thus, whether there are 10 or 10 million objects in the tree the depth of the tree is at a level appropriate for that layer. In reality, only one index tree exists, with each layer's tree overlaid on the next. This approach provides memory-efficient storage, rapid access and effective search pruning. In addition to these methods, balancing algorithms have been developed that distribute objects within nodes of the tree to optimize locality of reference.

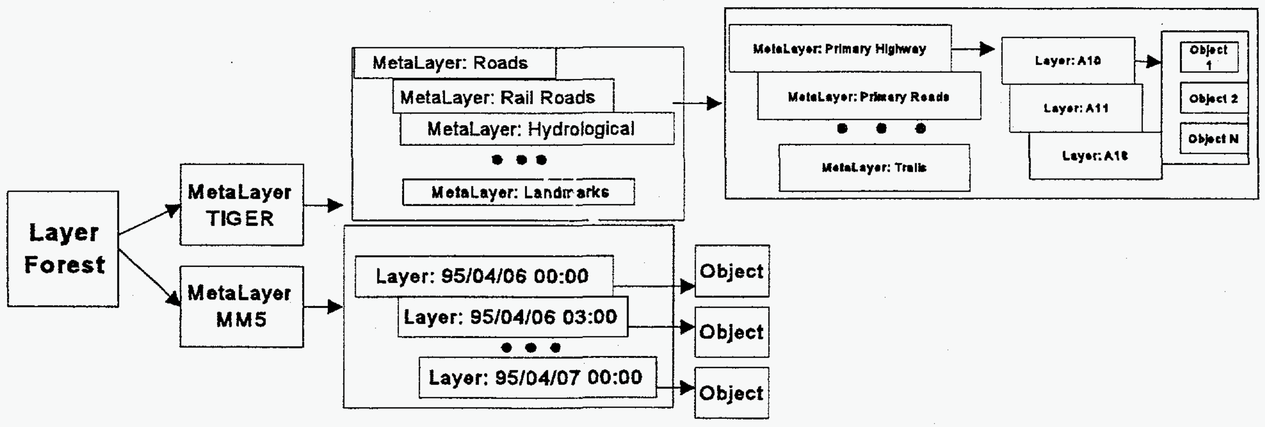
Spatial data is commonly related in a hierarchical, not just a lay in nature. A context-sensitive layer management framework provides for the creation and management of a forest of hierarchies (Diagram 1) that models natural relationships between the data. This framework also provides context-sensitive visualization information, which specifies how to visualize object geometry. This differs from the GeoObject context-sensitive information in that this is a built-in framework that requires no programming for the manipulation of representations. For example, extending the forest example used before, the layer manager can display the forest as a point, then an icon and finally whatever representation provided by the underlying GeoObject or its delegate. As before this representation can vary to be anything ranging from a simple mesh to complex polygons modeling the tree and leaves.



DEEM rendering showing various levels of object accuracy

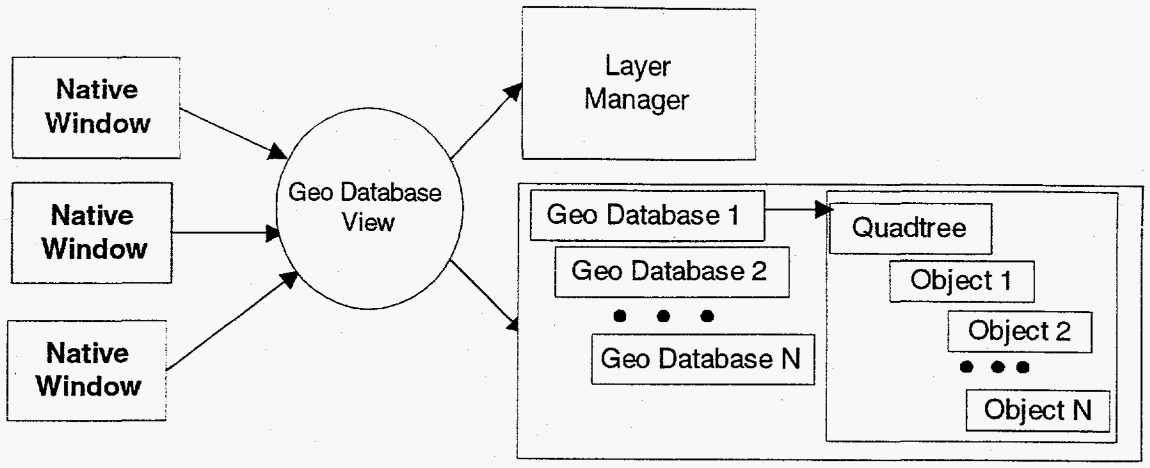
An object's representations can be thought of as a discrete sampling along a continuum of representations. The context in which an object is asked to visualize itself describes a point in that continuum. That point maps to some predefined set of visualization parameters that represent the object to a desired level of accuracy. As the context changes, the point moves, the information needed to represent that object changes, and a new set of parameters is retrieved.

Diagram 1



Layer management and inheritance hierarchy

Diagram 2



Geo DataBase Framework Hierarchy

All of the above are encompassed with a GeoDBView (Diagram 2). This object is a platform and window independent abstraction of the current view on a given geographical database. A given application's native window system window type is directly related to a GeoDBView. An application can have multiple native windows open that are all related to a single GeoDBView. While viewing the same data, each window can independently control the areas and visualizations occurring within it. This then is a realization of the plug-in capabilities sought in the design of OOGIS.

Data Access Layers

A driving requirement in the design of DEEM and the OOGIS engine was the need to encapsulate and ingest a variety of modeling and legacy data. This dictated that no one set of access methodologies could be used to efficiently ingest or access all possible data sets. An extensible framework was therefore created that utilized three different methods for the encapsulation of such data. This framework includes the following components:

- A full translation type, where the legacy data and its spatial aspects are captured and converted into a Smalltalk object schema. Benefits: Access speeds. Drawbacks: memory overhead, if native data keeps changing translations need to be rerun.

- A hybrid type, where only certain aspects of the legacy data are captured into the Smalltalk schema. Commonly this is the spatial aspects and possibly static or commonly accessed attributes. The remainder of the data resides in the native format and is accessed via a driver(see below). Benefits: Reduced memory overhead, fast access to spatial and other captured aspects. Drawbacks: Slow access to non-captured aspects. Difficulty in writing interactive driver in terms of performance and conceptual design.

- An external type, here only a reference to the external data source, is captured within the Smalltalk schema. Access to all aspects is via a driver. This approach is useful when accessing enormous amounts of base data types. This is especially true of the types where little or nothing is gained by objectizing them, such as satellite imagery. Encapsulation of model data is almost always done in this manner since most models were not written with Smalltalk in mind. Benefits: Significant reduction in storage requirements. Drawbacks: Slow access. Difficulty in driver design as in the hybrid layer.

So far we have discussed the need to write drivers for external access. These drivers can be likened to serial line drivers. Drivers are usually implemented as, but not limited to, a Smalltalk and C interface. The C interface allows calling any language supported on the target platform, which allows the reuse of access and modeling software. This greatly simplifies the process of writing a driver.

Drivers can perform a range of translations, from full translation(lossless) to various degrees of partial translation(with losses). The choice of translation modes is dependent on the specific needs of an implementation. The level of translation performed can be determined dynamically based on a given execution context. In the previous forest example where detail is needed to the leaf level, the driver would run a botanically correct tree growing model to generate the spatial geometry required for the GeoObject representation.

Performance

Traditional Smalltalk wisdom, with regards to performance, is to not worry about it until you find that your applications is too slow. The sheer number of on-line spatial objects that we were required to support made it prudent for us to think about this beforehand. Having our previous generation C based engine on hand for comparison made it practical to determine if our designs would carry over to Smalltalk largely intact.

It had been decided early in the project that some sort of object database would ultimately be used for secondary storage. However, because of the multitude of uses and sponsors that this framework would have, it was critical that we not be tied to any one particular OODB implementation. Additionally, certain objects are not conducive to efficient database storage, be it object oriented or relational. A more practical approach is to have an object reference in the database but use a customized driver object to access attribute and spatial data in the external object. As part of the development of the C-based predecessor of OOGIS, a customized object store had been created. This object store, which included a three-level caching scheme, provided average access times about half that of direct memory accesses. Because this object store was intended to support CD-ROM-based applications it was optimized for largely static data. This custom object store was also several times more efficient storage-wise than the commercial OODBMS we had been using at the time. Unfortunately, it is not practical to write custom object stores for every possible sponsor.

A common feature found in object programming is localization of memory/object references. This is especially so for a spatial index and spatially related objects. Studies conducted by NeXT Inc., for their own Objective-C object system, has shown that the decrease in performance caused by dynamic binding can be more than offset by the improvement in speed of so-called zone allocation. The idea behind zone allocation is that the programmer can ask the system to put an object about to be created into the same memory zone as a given object. For example, by allocating a GeoObject and the instance variable objects of that object in the same zone, the programmer is expecting that references to a GeoObject will be followed by references to the object's instance variable objects. This can be far more efficient than the possibly near-random pointer walking that can occur in typical C and C++ programs that don't use some sort of pool allocation techniques.

Most of the previous performance issues had been anticipated in the course of our development. Some less obvious, but no less serious, performance issues quickly cropped up as the number of objects to be drawn grew. It turns out that ParcPlace VisualWorks 2.0 has no provision for using features like backing store on UNIX workstations. While this probably provides an extra level of portability for ParcPlace, it provides lots of performance headaches for us. An action as simple as raising a window could cause upwards of 7 repaint events to be sent. Normally this would not have been too much of a problem since a typical solution is to send a list of dirty areas into a view's repaint method. Alas, though, somewhere in the VisualWorks window and controller hierarchy there exists a list of damaged areas; they are not passed on and there is no way to collapse the multiple exposure events. We could have modified the system classes but deemed that untenable as too many classes would be modified and future compatibility could be difficult to maintain. Our solution was to find the proper place in the GraphicsDevice and DisplaySurface objects where the X Display, graphics context and window values were located. We then created a C-callable routine to turn on backing store for the window. Not an elegant solution, not portable to non-UNIX based workstations, but very necessary for interactive performance and more agreeable than the alternative of system class hacking.

Other fallouts of the portability issue come in the obtuse 24 bit color support, support for reasonable XOR drawing, dashed lines, full alpha channel, and stippled drawing support. Having ported the earlier C-based version of OOGIS to NeXTSTEP it was disappointing to see the rudimentary or slow dithering and rudimentary transparency support. The poor dither performance severely limits the color selections available for drawing. Solutions to these color issues are still being addressed, though most of the suggested answers currently point to external calls to non-portable C routines that twiddle with X directly.

The typical complaint against Smalltalk is that it is too slow. For reasons of efficient porting and speed we decided to create Smalltalk wrappers and keep quite a bit of code on the C side. It was therefore necessary to do a case study of exactly what impact calling C routines had on performance. It would also help us determine when in fact it would be more efficient to stay in Smalltalk. The case study for this became the point-in-polygon algorithm. C executed the call 6 times faster than Smalltalk, but the overhead of calling C and returning, with no processing took longer than doing it in Smalltalk. This did not include the conversion of Smalltalk data to C data types. In a typical case 27% of time was calling C, 61% time was converting data and 12% was spent in doing point-in-polygon calculations. The calling percentage continues to decrease and the conversion goes up as the number of points increases. Thus for this example it never becomes practical to call C. The solution is a standard way of interpreting simple Smalltalk objects in C without having to convert; since Float and Int are stored the same way in Smalltalk as C this should not be to difficult. We just need a fast way to get to our C routine and then determine if the number is a more complex type requiring conversion. This idea may be abhorrent to Smalltalk purists, but it really comes down to a matter of practicality.

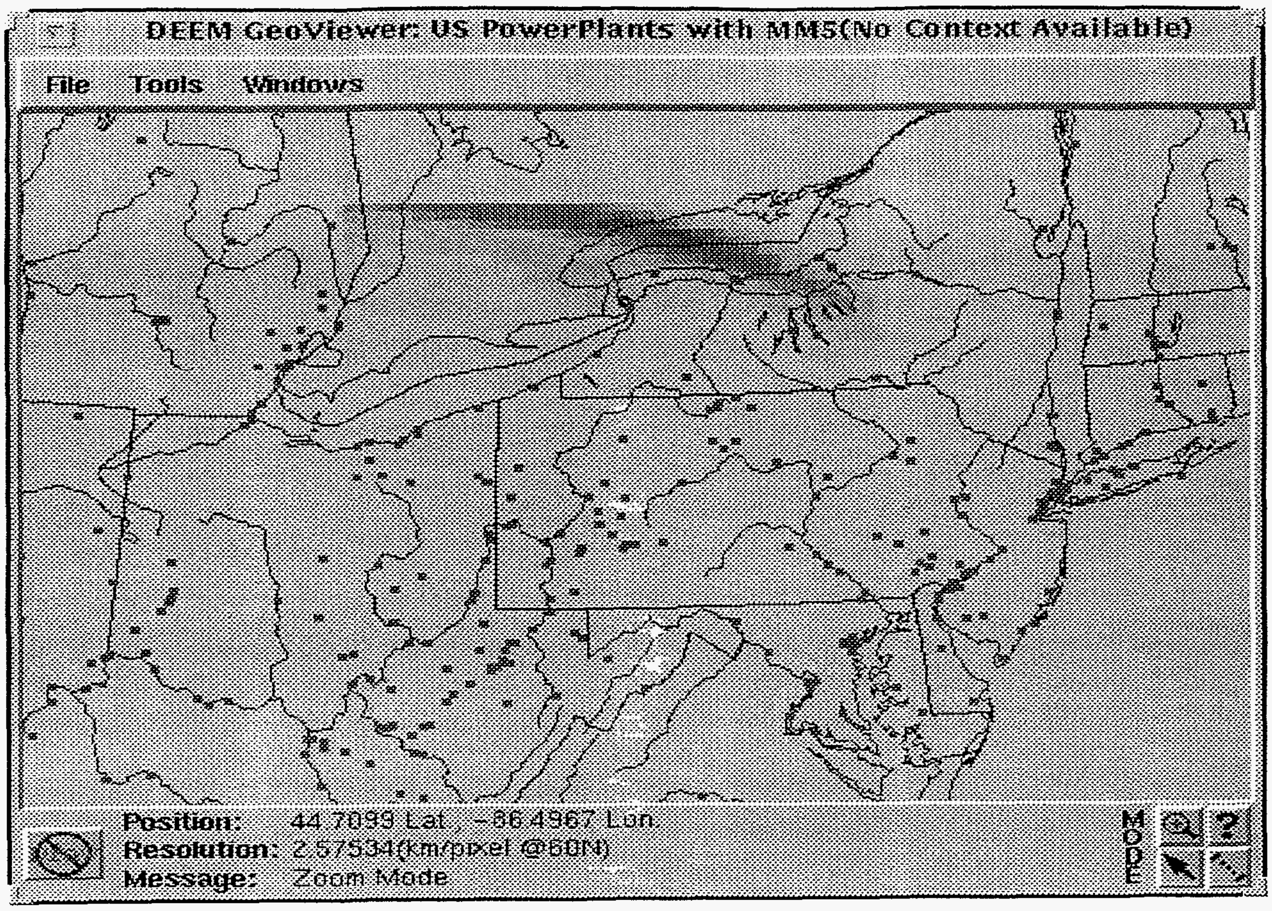
Moving towards the other end of the spectrum, considerable performance improvements could be realized, in both speed of execution and space savings, with some relatively simple optimizations. Removing duplication of GeoPoints, our coordinate pairs, can reduce storage requirements of the total database by 25-50%. This is a direct result of most spatial vector data being represented by chains, which share common nodes. Even separate chains and objects commonly share nodes. Similarly, names of objects are often repeated. Keeping these references to unique string objects results in another 2-5% reduction in DB size. This technique can be extended for any attribute data stored in the database. Obviously using these techniques means that care must be taken so as to not modify the underlying objects, as other objects might still be referencing them. Another performance boost can be achieved as a side effect of having only one instance of a particular GeoPoint. In this case an association can be established with the projected value of that GeoPoint. As long as the context of the screen does not change, the projected value of the GeoPoints does not need to be recalculated which results in an enormous improvement in redraw speed. Another simple, though not at first obvious, optimization that provided a large performance boost are things like using whileTrue: in place of do: to loop over collections of objects. This change allows the compiler to perform additional optimizations on the block. Finally, a method we were implementing which dealt in color value calculations was taking an inordinate amount of the redraw time. Simplifying the method by removing temporary references and cascading method calls resulted in the removal of only two op-codes from the virtual machine code. However, the resultant code performed an order of magnitude faster.

Performance will continue to be an issue. While OOGIS in Smalltalk performance is very good, we have seen an order of magnitude drop in performance over previous C-based versions. It seems that there is no perfect balance of size, execution time and capabilities. Efforts to raise performance will focus on bottlenecks in the Smalltalk code. Analysis of these bottlenecks will determine what remedy to attempt. Do we try to make the Smalltalk code faster, do we redesign something to eliminate the offending bottleneck or is migration to C feasible? As an example in which we are effectively "cycle-counting" Smalltalk code is in the query subsystem. Some queries are called upwards of 3600 times to generate a single result. Each query executes in only a few milliseconds, but the sheer number of queries makes optimizing them a priority. The mere thought of improving query performance by migrating pieces of code to C is enough to make us delirious. (Imagine re-implementing the dynamic query facilities in a language like C).

Examples

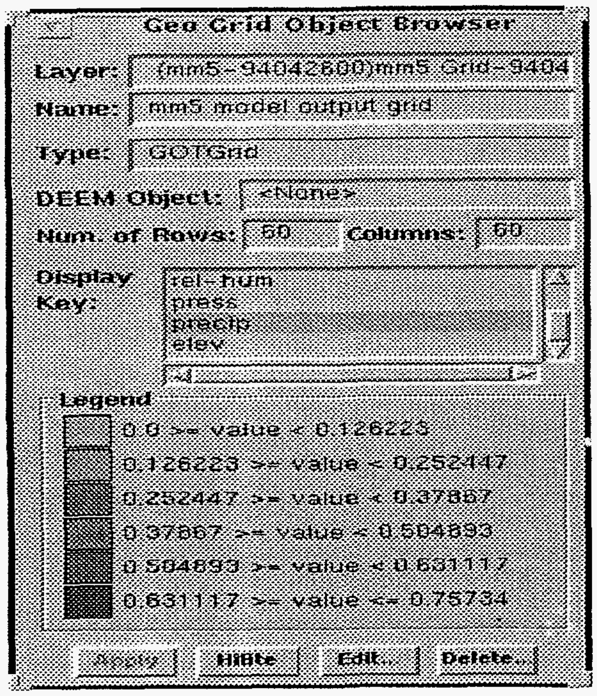
Northeast US and Southern Canada with MM5:

This example illustrates OOGIS operating as a framework within DEEM. The DEEM simulation manager controls the execution of various models, including the Mesoscale Meteorological Model (MM5). When MM5 has finished updating the state of atmosphere for a given timestep, an external event is created. This event is captured by DEEM which proceeds to update the OOGIS database to reflect the current state of the atmospheric simulation. The update may take the form of


DEEM GeoViewer: US PowerPlants with MM5(No Context Available)
File
Tools
Windows
MODE

Position: 44.7099 Lat , -86.4967 Lon
Resolution: 2.57534(km/pixel @60N)
Message: Zoom Mode

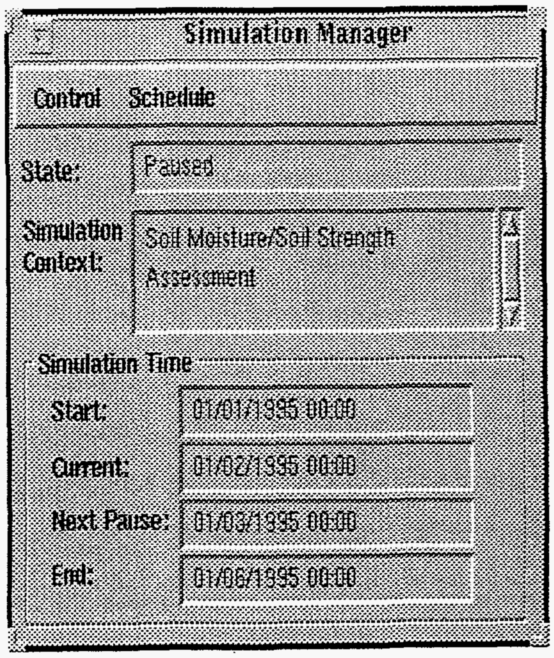
Figure 1: MM5 Output precipitation data overlayed on US/Canada Map



Layer: (mm5-94042800)mm5 Grid-9404
Name: mm5 model output grid
Type: GOTGrid
DEEM Object: <None>
Num. of Rows: 60 Columns: 60
Display Key: rel-hum, press, precip, elev

Legend
- ▨ 0.0 >= value < 0.126223
- ▨ 0.126223 >= value < 0.252447
- ▨ 0.252447 >= value < 0.37867
- ▨ 0.37867 >= value < 0.504893
- ▨ 0.504893 >= value < 0.631117
- ▨ 0.631117 >= value <= 0.75734

Figure 2: MM5 Grid object brower



State: Paused
Simulation Context: Soil Moisture/Soil Strength Assessment

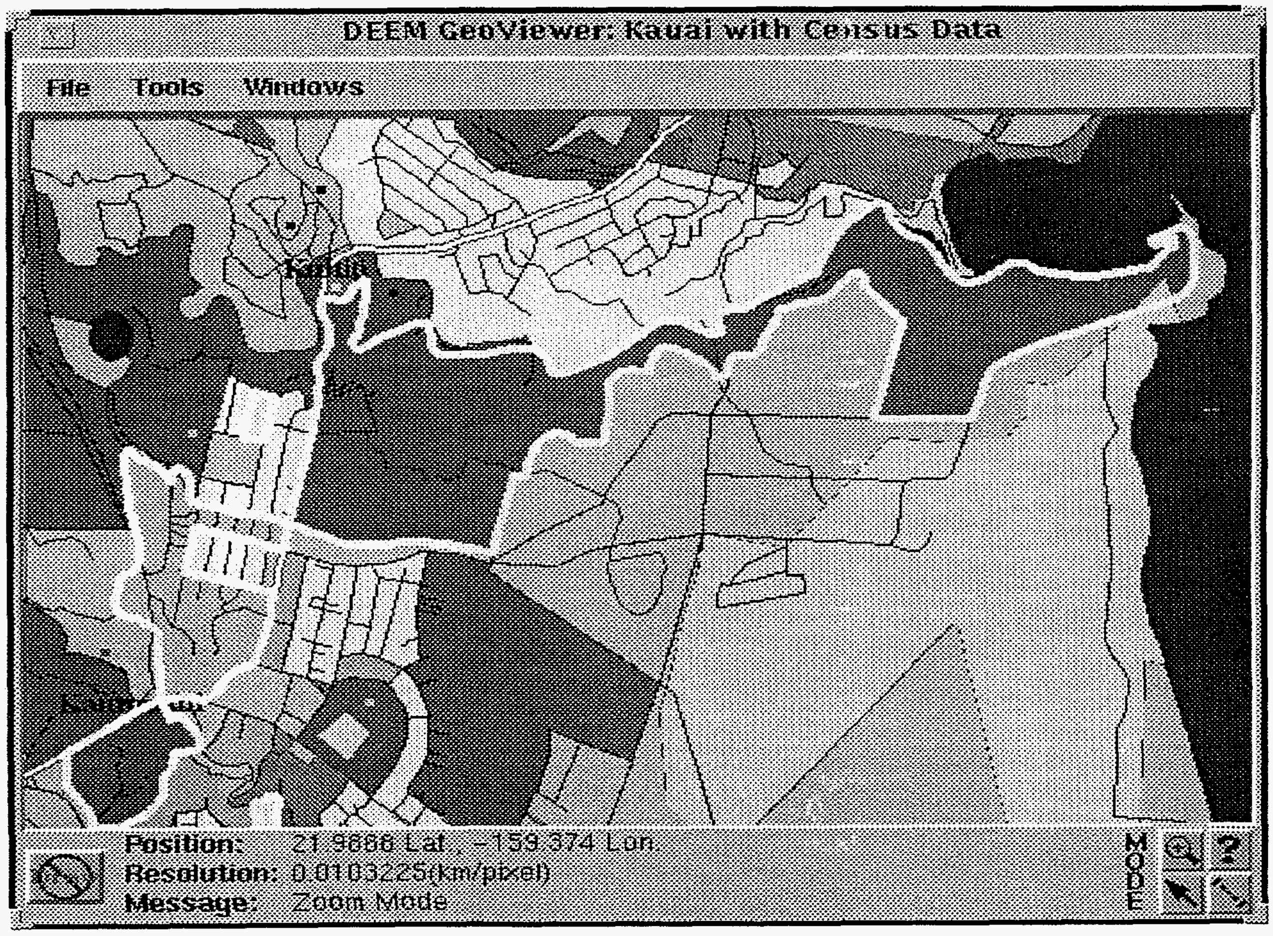
Simulation Time
Start: 01/01/1995 00:00
Current: 01/02/1995 00:00
Next Pause: 01/03/1995 00:00
End: 01/06/1995 00:00

Figure 3: DEEM simulation manager controlling OOGIS

adding objects or modifying current ones. Here DEEM has added a rasterized output grid from an MM5 model run. Grids can contain any number of attributes for each grid cell. Here the precipitation attribute has been selected for display and is color coded by rainfall. During this pause in the simulation the user can query other objects based on the current state of the atmosphere or historical states. The red squares are electrical power generation stations.

Kauai, Hawaii:

This example illustrates the use of various data sources in a seamless manner and the use of protocols to extend the built-in query and drawing facilities. We have ingested the United States Census Bureau's TIGER/Line database and census demographic information for the island of Kauai in Hawaii (figure 4). A thematic map shows census blocks color codedby population density with TIGER data and the current query result on it.



Position: 21.9888 Lat., -159.374 Lon.
Resolution: 0.0103225(km/pixel)
Message: Zoom Mode

Figure 4: TIGER/Line and demographic information for Kauai, Hawaii

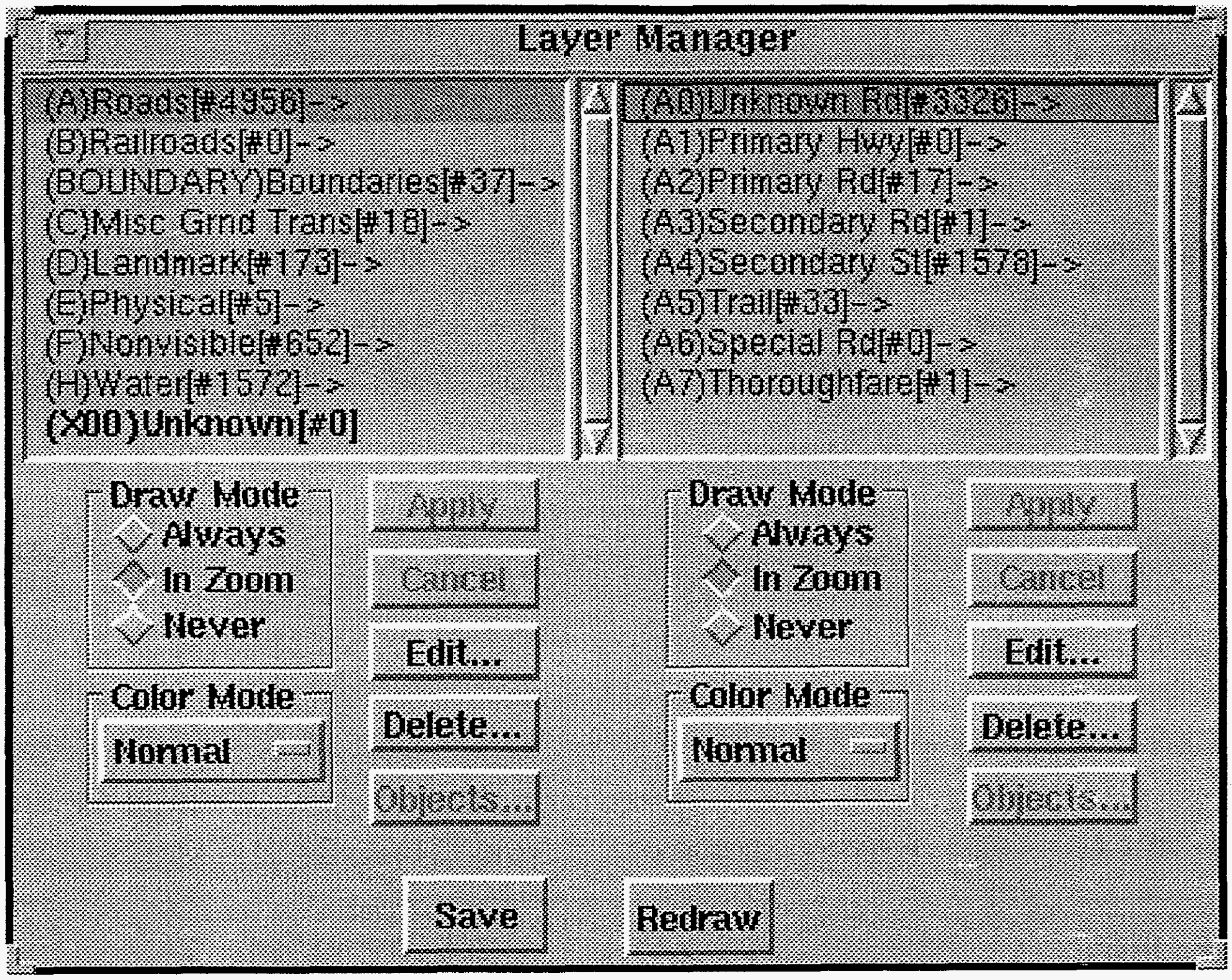


Figure 5: Layer forest hierarchy for TIGER data

The TIGER data has no delegate and is therefore a "dumb" object responding only to a few predefined protocols and built-in attributes. The census data is driven by a "smart" object that understands both the query and drawing protocols. The query protocol has added attributes to the list of those that may be queried against. Some of the attributes are static and others dynamic. In this example the dynamic attribute of population density was used to calculate a polygon fill color.

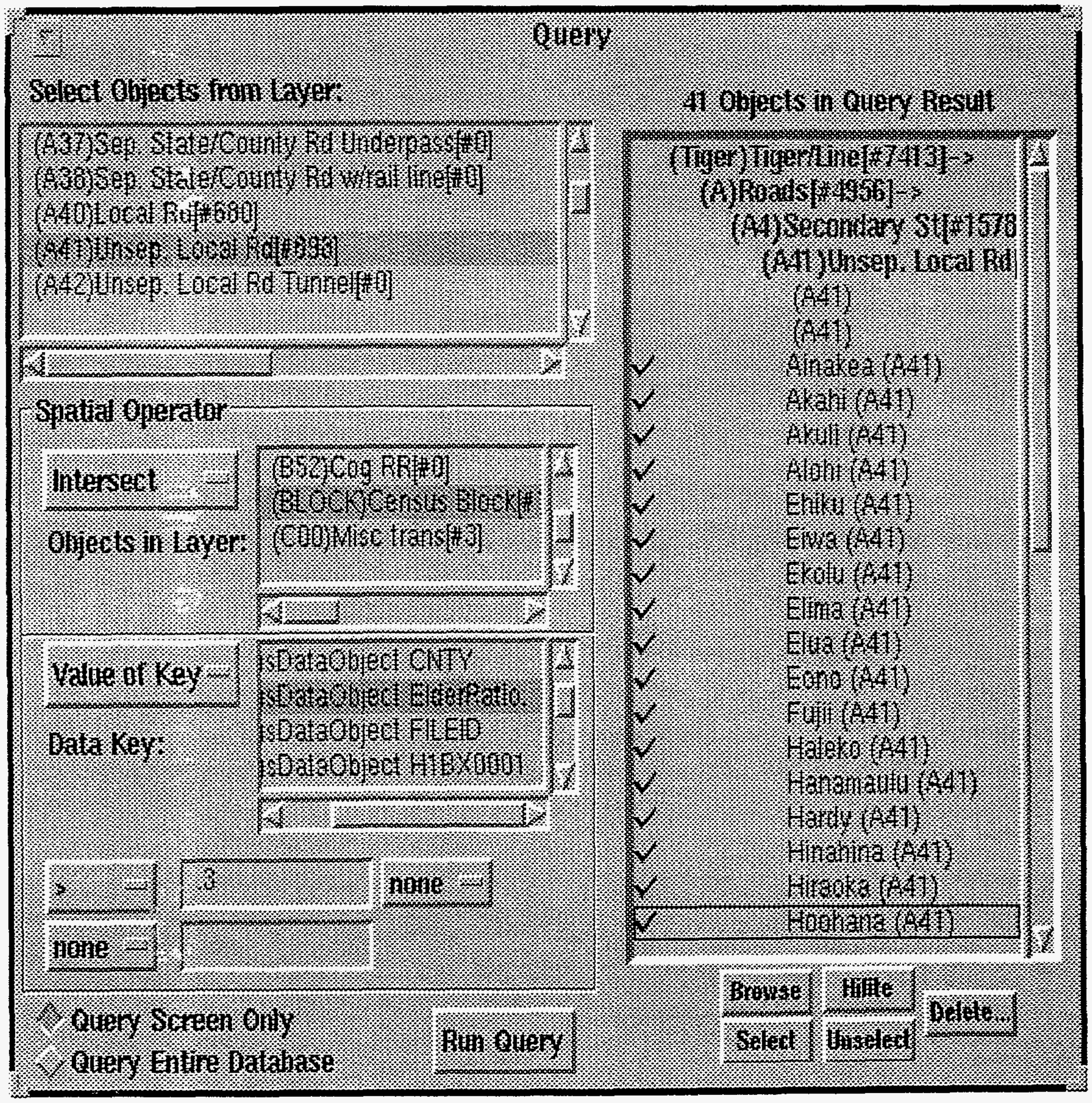
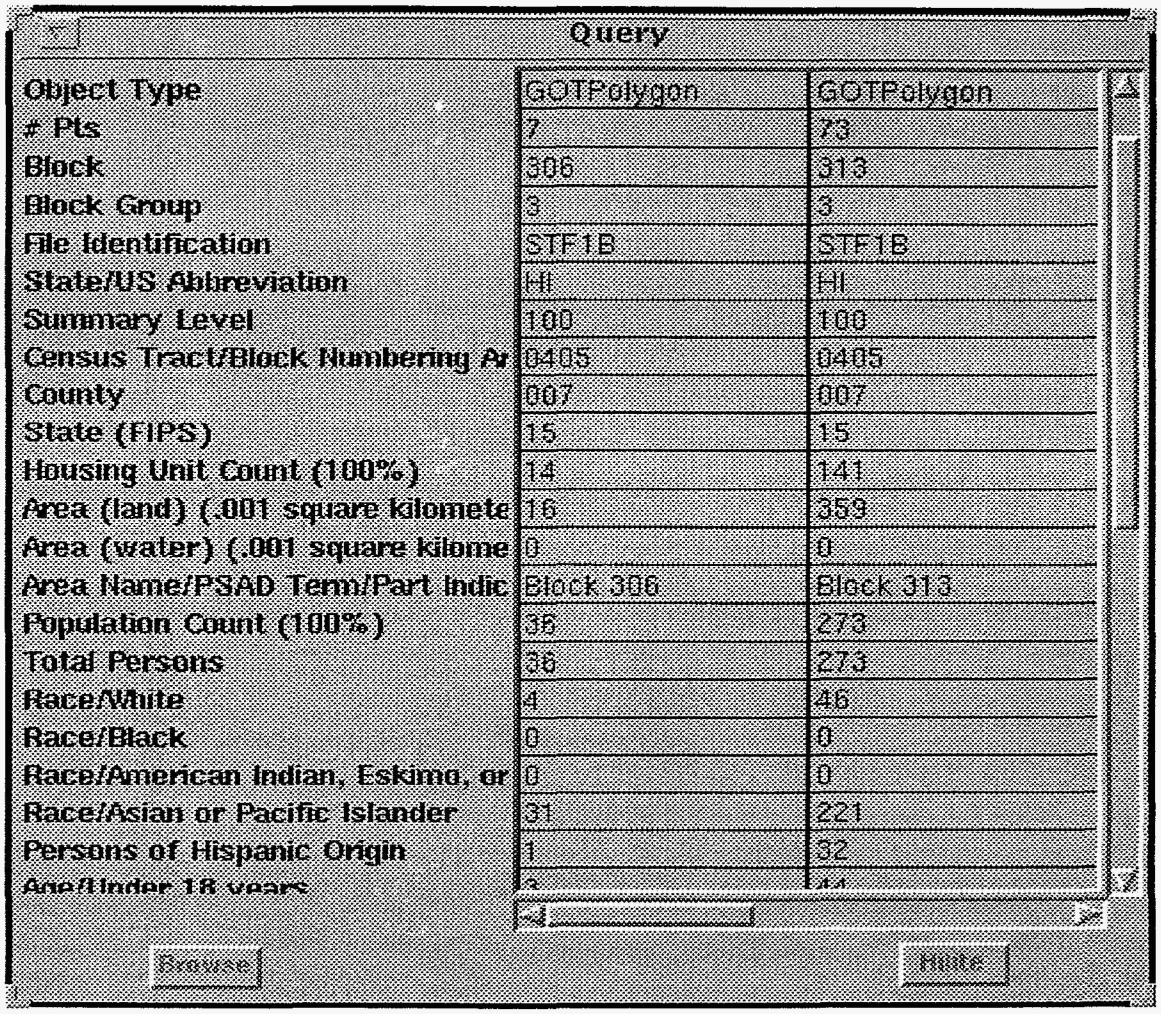


Figure 6: Query tool allows formulation of complex queries

A query was run to find local streets where the population consists of over 30% elderly. This involved querying data from both the TIGER and demographic database. The elderly ratio (ElderRatio) is a dynamic attribute. The query tool (Figure 6) depicts how this query was composed and its results displayed with the objects layer hierarchy preserved. The results may also be displayed in a object table browser (Figure 7), similar to a spreadsheet but extended to display object attributes via the query protocol. Here we also introduce the Layer Manager. It allows the user to navigate the layer hierarchy and quickly set commonly changed attributes.



Object Type	GOTPolygon	GOTPolygon
# Pts	7	73
Block	306	313
Block Group	3	3
File Identification	STF1B	STF1B
State/US Abbreviation	HI	HI
Summary Level	100	100
Census Tract/Block Numbering Ar	0405	0405
County	007	007
State (FIPS)	15	15
Housing Unit Count (100%)	14	141
Area (land) (.001 square kilomete	16	359
Area (water) (.001 square kilome	0	0
Area Name/PSAD Term/Part Indic	Block 306	Block 313
Population Count (100%)	36	273
Total Persons	36	273
Race/White	4	46
Race/Black	0	0
Race/American Indian, Eskimo, or	0	0
Race/Asian or Pacific Islander	31	221
Persons of Hispanic Origin	1	32
Age/Under 18 years	[illegible]	[illegible]

Figure 7: Query result shown in the object table browser

Conclusion

Immediate plans revolve around usability issues related to the current OOGIS GUI. Based on early feedback from users, efforts need to be focused on: query generation, layer management, model interaction dialogs and object inspectors. Interfaces for these tools need to be made more approachable and intuitive. Development of user tools which are at a higher-level of abstraction will allow us to tailor the software to a particular sponsors needs and culture.

In addition to making things easier for the users, we intend to create reusable higher level interfaces to the OOGIS internals. This will help to shield novice programmers from the minutiae that would be involved in dealing with the low-level OOGIS engine. Areas that will be addressed include:

- A simplified interface to the underlying query engine.
- Abstracting the process of insertion and deletion of objects into OOGIS.
- The creation and management of layers and meta-layers.
- A mechanism for mapping user interactions to OOGIS functionality.
- Abstraction and integration of multiple object stores.

Finally, we are also currently applying the OOGIS technology to a dynamic distributed World Wide Web (WWW) based HotJava applet(DOOGIS). This will allow users of HotJava to view multiple distributed OOGIS databases, and hopefully soon OpenGIS compliant data sources, as coming from a single coherent source. This technology allows state information to be kept in the browser and allows classes and objects to be dynamically downloaded. This provides users an Internet based delivery system for an interface to the OOGIS engine.

References

Samet, H. Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS, New York: Addison-Wesley, 1990.

Faust, N. "Supporting Mechanisms, Cycles and Processes", 1995 GIS World Sourcebook, GIS World Inc., 1995, p.290-292.

Hodgson, M.E. "Unified Landscape", 1995 GIS World Sourcebook, GIS World Inc., 1995, p.293-296

Goldberg, A. And D. Robson. Smalltalk-80 The Language, New York: Addison-Wesley, 1989.

Meyer, B. Object Oriented Software Construction, New York: Prentice Hall, 1988.

Cox, B. J. Object Oriented Programming: An Evolutionary Approach, New York: Addison-Wesley, 1986.

Gall, G., C. Kristian, M. Bradley and J. Rawlings, "Oracle7 MultiDimmension: Advances in Relational Database Technology for Spatial Data Management", Oracle Corp. Whitepaper, March 1995.

Meyer, B. Object Success: A Managers Guide, New York: Prentice Hall, 1995, p. 91.