JUMP Unified Mapping Platform

Technical Report

Prepared for:

Ministry Of Sustainable Resource Management

Base Mapping and Geomatic Services Branch

DRAFT

Prepared by:

VIVID SOLUTIONS

Document Change Control

REVISION NUMBER	DATE OF ISSUE	AUTHOR(S)	BRIEF DESCRIPTION OF CHANGE
Draft	4 march 2003	Martin Davis	Document Created

Document Sign-Off

The undersigned have read and agree with the content of this document.

Martin Davis
VIVID Solutions Inc.

Mark Sondheim
Ministry of Sustainable Resource Mapping

Table of Contents

1. INTRODUCTION

The JUMP Unified Mapping Platform is a GUI and API for performing spatial data processing. It has been developed to support the development of the Java Conflation Suite, but is generally useful in its own right. This document details the design approaches and algorithms used to build JUMP.

1.1 RELATED DOCUMENTS

- JUMP User Guide
- JUMP Developer Guide
- JUMP Installation Guide
- Java Topology Suite (JTS) documentation (http://www.vividsolutions.com/jts/jtshome.htm)

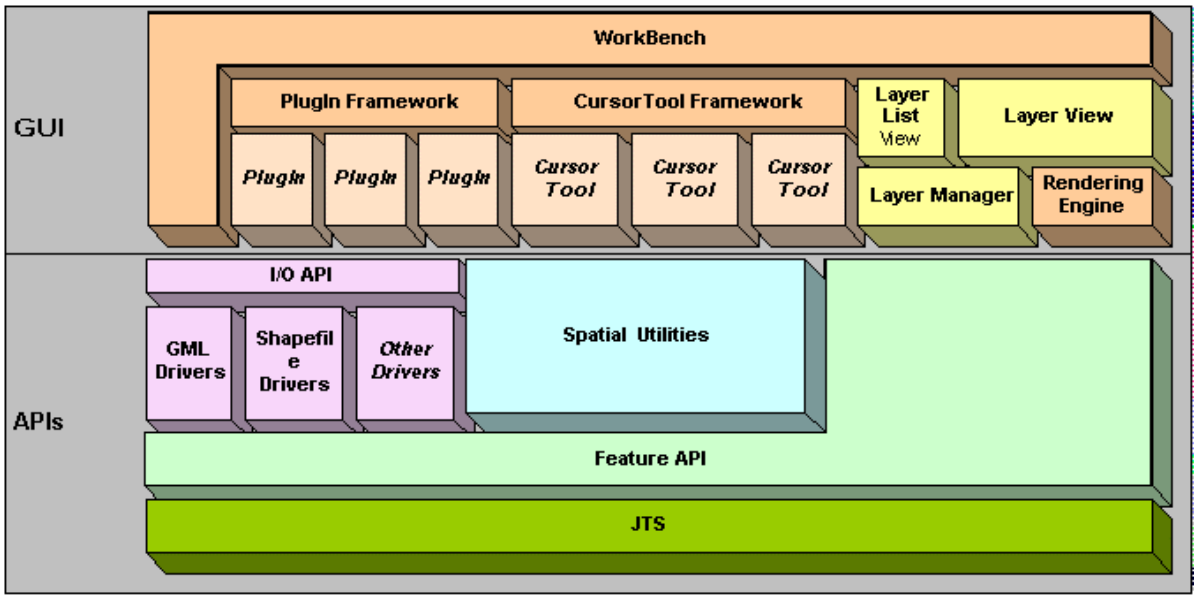
2. JUMP SYSTEM ARCHITECTURE

The JUMP codebase is organized into two major packages: the JUMP API, and the JUMP Workbench.

JUMP API	A set of APIs that support spatial data I/O, features and collections, and fundamental spatial algorithms such as validation and spatial access methods. These APIs make use of JTS to provide their underlying geometric model.
JUMP Workbench	The JUMP Workbench is an interactive GUI, which provides a framework for executing spatial functions and visualizing the results. It provides many tools for creating, viewing, and manipulating spatial data.

Each of these packages is architected to be as extensible, modular, and reusable as possible. They rely heavily on the flexibility and dynamic extensibility of the Java platform to accomplish this. They incorporate modern object-oriented design patterns and programming techniques.

The diagram below shows the major components of the architecture. Each of these components is further described below.



The JUMP System Architecture

2.1 API LAYER

The API layer is modular and easily reused as standalone APIs. Wherever possible, the APIs are split into interfaces and implementations, to allow easy extensibility. For example, the I/O API provide interfaces and support classes for Readers and Writers, which allows other drivers to be developed and used in the same framework.

2.1.1 JTS

JUMP relies on the Java Topology Suite (JTS) to provide its geometry model and basic geometric operations. JTS is not considered to be part of JUMP, but is used by almost all JUMP components.

2.1.2 I/O API

The I/O API contains a simple framework to support interfacing to I/O drivers. I/O drivers typical consist of a Reader and/or Writer for a particular spatial data format. Currently GML (in several different profiles), Well-Known Text and ESRI Shapefiles are supported. The I/O API is designed to provide a general framework for I/O classes and to be easily extensible.

2.1.3 Feature API

The Feature API contains classes to model spatial Features. It also contains classes to provide various kinds of FeatureCollections (including both simple lists and spatially indexed)

2.1.4 Algorithm API

JUMP provides implementations of some common spatial algorithms, such as overlay and polygonization.

2.2 WORKBENCH LAYER

The JUMP Workbench is an interactive application that that provides a User Interface to the functionality in the JUMP APIs. Its primary purpose is to provide a convenient User Interface to carry out spatial processing tasks and visualize the results.

Many of the components used in the Workbench could be useful in other applications. Wherever possible Workbench GUI components have been designed to be modular and loosely coupled, to offer easy reusability in independent applications.

The JUMP Workbench takes full advantage of the dynamic class loading capabilities of Java to provide simple and complete extensibility. The Workbench supplies a framework and API allowing GUI-based extensions to be added seamlessly. The Workbench Extensibility framework allows adding the following components:

2.2.1 PlugIn Framework

The PlugIn framework allows new functionality to be easily incorporated into the Workbench. PlugIns are dynamically loaded classes that are provided with access to most of the internals of the Workbench. They can hook themselves into the menu system, and manipulate and create Workbench objects as needed. PlugIns can run either in separate threads or within the main Workbench thread. PlugIns may receive input from the user using dialogs and by the selection state of other UI elements (such as Layers in the LayerList). They can produce text output on the Output window as well as altering the state of Layers in LayerViews. Various utilities are provided to allow easy construction of new PlugIns (such as a Dialog Builder).

Most of the standard functions provided with JUMP are themselves implemented as PlugIns.

2.2.2 CursorTool Framework

The CursorTool Framework is similar to the PlugIn Framework in that it allows easily extending the Workbench UI with new cursor-based tools. The distinction between PlugIns and CursorTools is that CursorTools interact with the user only through the cursor. Typically they define a mode of the Workbench, in which the user indicates information or manipulates features using the cursor.

2.2.3 UI Components

UI components are Swing-based classes, which have been modularised for easy use in other applications.

2.2.3.1 Layer Manager

The Layer Manager contains a list of layers. It supports events indicating when layers are add, removed or changed.

2.2.3.2 Layer View

The Layer View is a graphical view of the geometry data within a layer. It supports different kinds of symbolization via the Rendering Engine.

2.2.3.3 Layer List View

The Layer List View is an UI that provides a view of the Layers in a Layer Manager. It has two main functions: provide a visual element for the layer to supports user interaction with the layer (e.g. via a pop-up menu); and provide a legend for an accompanying Layer View. It depicts the current symbolization of the layer. It allows layer display to be toggled on/off. It allows layers to be categorized into

2.2.3.4 Rendering Engine

The Rendering Engine produces the depiction of geometric data in the LayerView. It provides most of the usual rendering settings such as line and fill colour, width, and alpha blending. It can render vertices using selectable decorators. It relies on the rendering abilities of the Java2D API.

3. JUMP WORKBENCH

The JUMP Workbench is an interactive GUI which provides a framework for executing spatial functions and visualizing the results. The Workbench provides many tools for creating, viewing, and manipulating spatial data. The Workbench is a modern multi-window GUI built on Java Swing, which offers portability to any platform running Java. It takes full advantage of the multi-threading and exception handling capabilities of Java to provide a performant and robust environment.

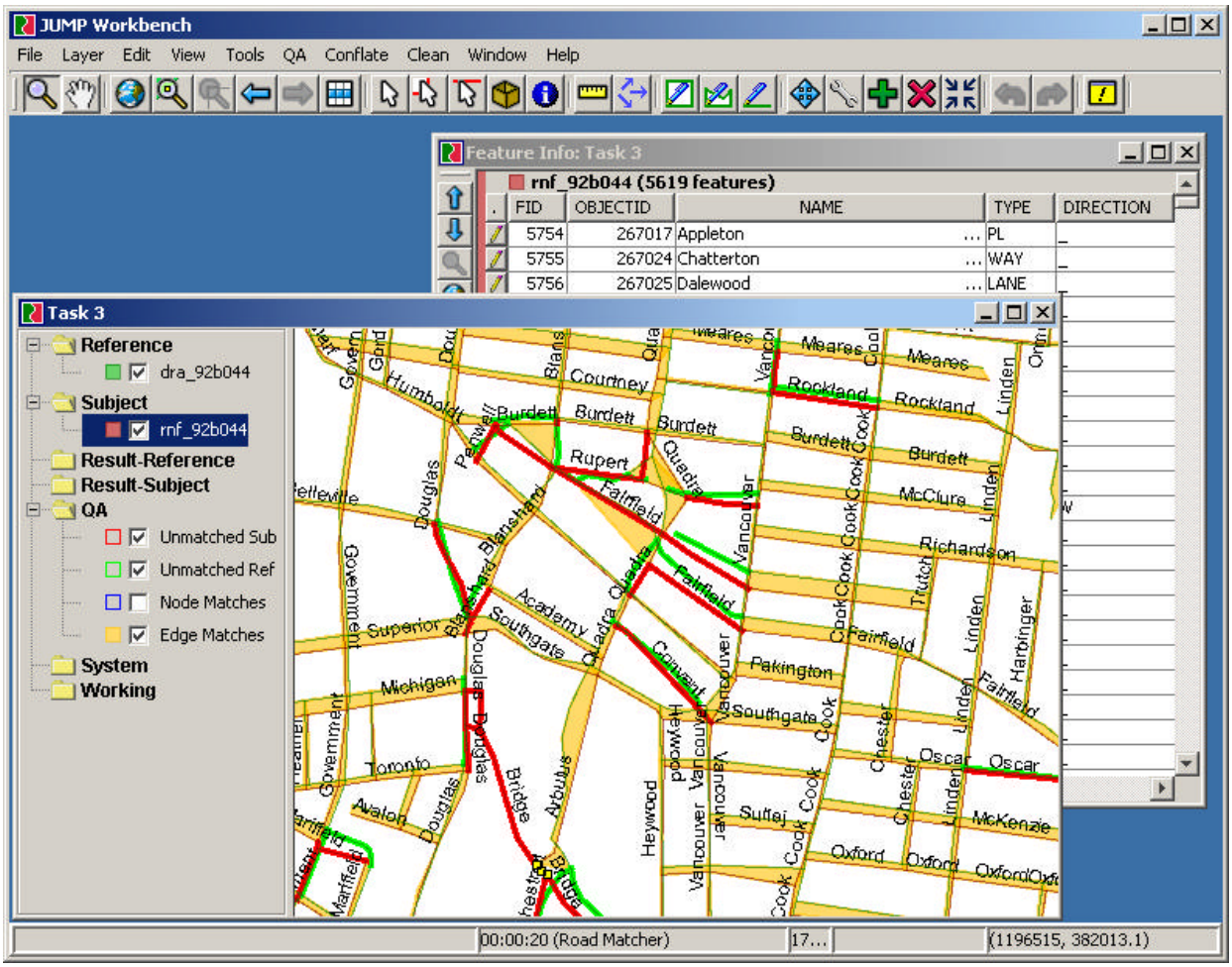


Figure 1 - The JUMP Workbench

The JUMP Workbench is further documented in the JUMP User Guide and the JUMP Developer Guide

4. GEOMETRY & FEATURE MODEL

JUMP utilizes the Java Topology Suite (JTS) API to provide it's underlying geometry model and basic geometric operations. JTS implements the OGC Simple Feature Specification, which provides a full set of 2-D linear geometric objects together with spatial predicates and boolean operations based on them.

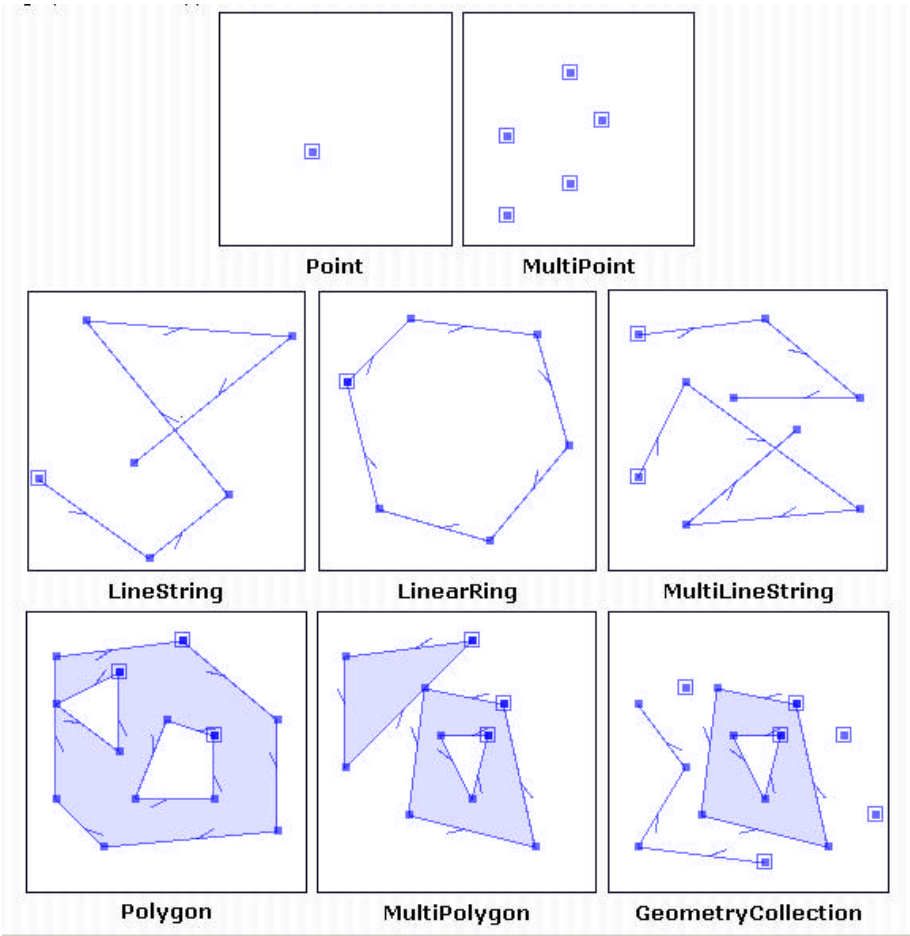


Figure 2 - JTS provides a full set of linear Geometry types

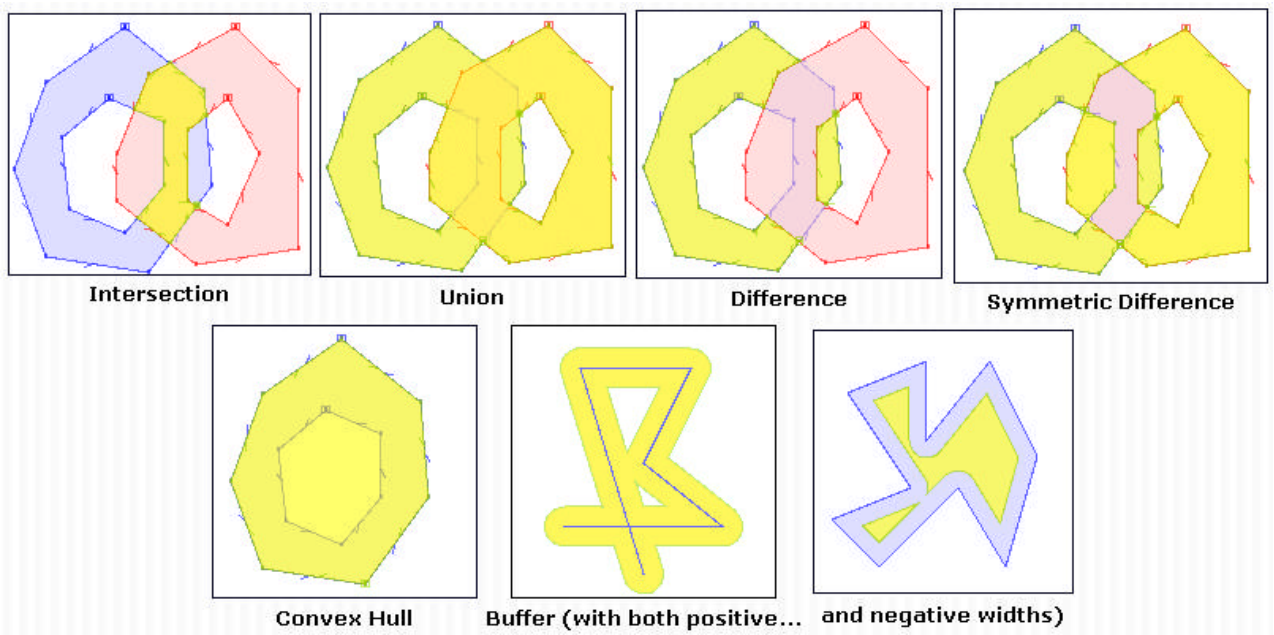


Figure 3 - JTS provides a full set of boolean operations

5. GEOMETRY CREATION & EDITING TOOLS

JUMP provides a full set of geometry creation and editing tools. Geometries may be displayed and edited either visually or in an ASCII text form. The following capabilities are provided:

- Creation of points / lines / polygons (with holes)
- Moving & Deleting points / lines / holes / components of Multi-geometries
- Inserting and Deleting vertices
- Combining and exploding of geometries to and from Multi-geometries
- Snap to vertex / line / grid
- User-controllable size and display of grid

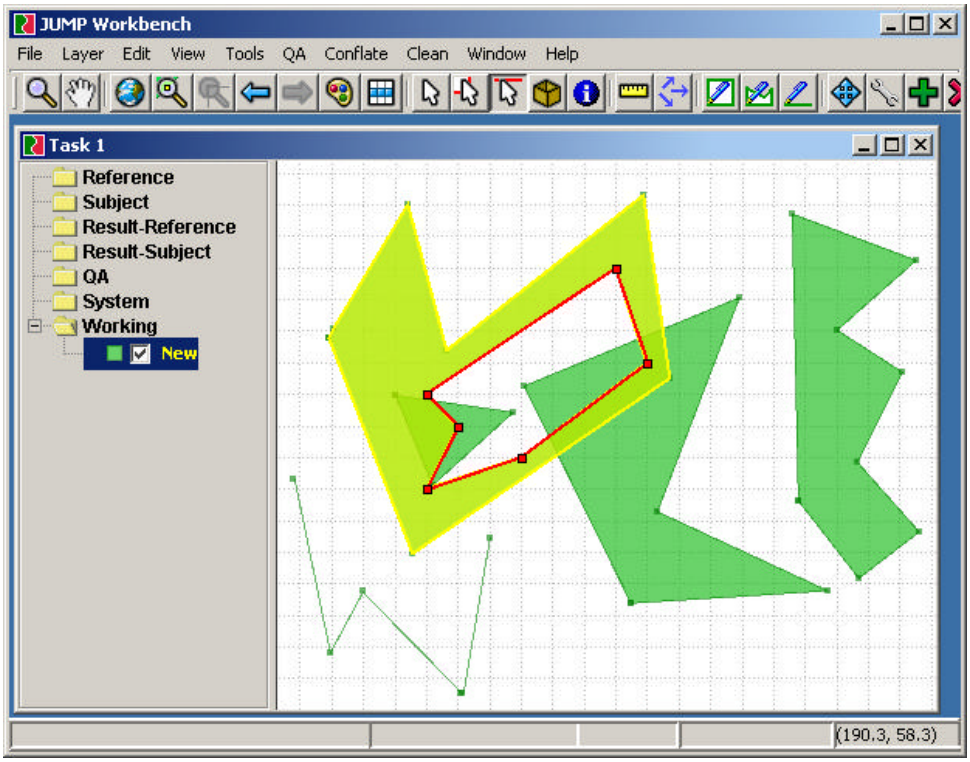


Figure 4 - Editing A Polygon Hole in JUMP

6. VISUALIZATION

The JUMP Workbench provides a rich set of tools to control the rendering of spatial data.

6.1 STYLING

Many different kinds of styling can be applied to displayed geometry. These include:

- Fill colour
- Line colour and width
- Layer Transparency
- Vertex display with variable sizing
- Geometry Decorations (including arrowheads and other line terminators)

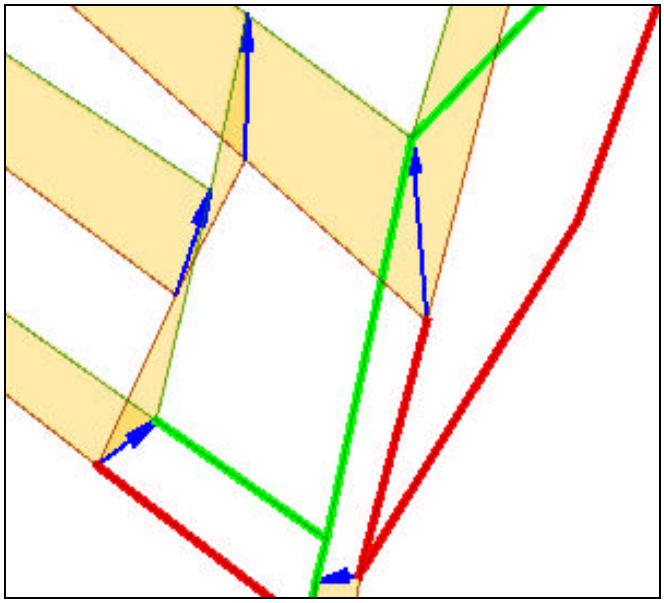


Figure 5 - An example of styling

6.2 LABELLING

The JUMP Workbench supports dynamic labelling capabilities, including

- Labeling from an attribute, with ability to specify colour, font, size, and vertical alignment
- Label rotation and size can be driven from an attribute
- Labels can scale with view or be fixed-size
- Labels center on visible portion of geometry
- Overlapping labels can be automatically hidden

Figure 6 - An example of line labelling

7. WEB MAP SERVER CLIENT

In order to provide access to external spatial databases, JUMP provides a client for the OGC Web Map Server (WMS) Standard. As with all JUMP components, this client consists of both a programmatic API (which can be used as an independent component) and a GUI interface to allow the Workbench to display WMS Layers.

The WMS API and GUI has the following features:

- A WMS Layer consists of an image which displays one or more of the data layers provided by a WMS server
- The set of layers and their ordering can be edited
- Images from multiple servers can be displayed simultaneously
- The alpha blend value can be user-controlled for each WMS layer
- Currently images can only be displayed in coordinate systems supported by the WMS server, but the potential exists to reproject the image in JUMP itself.

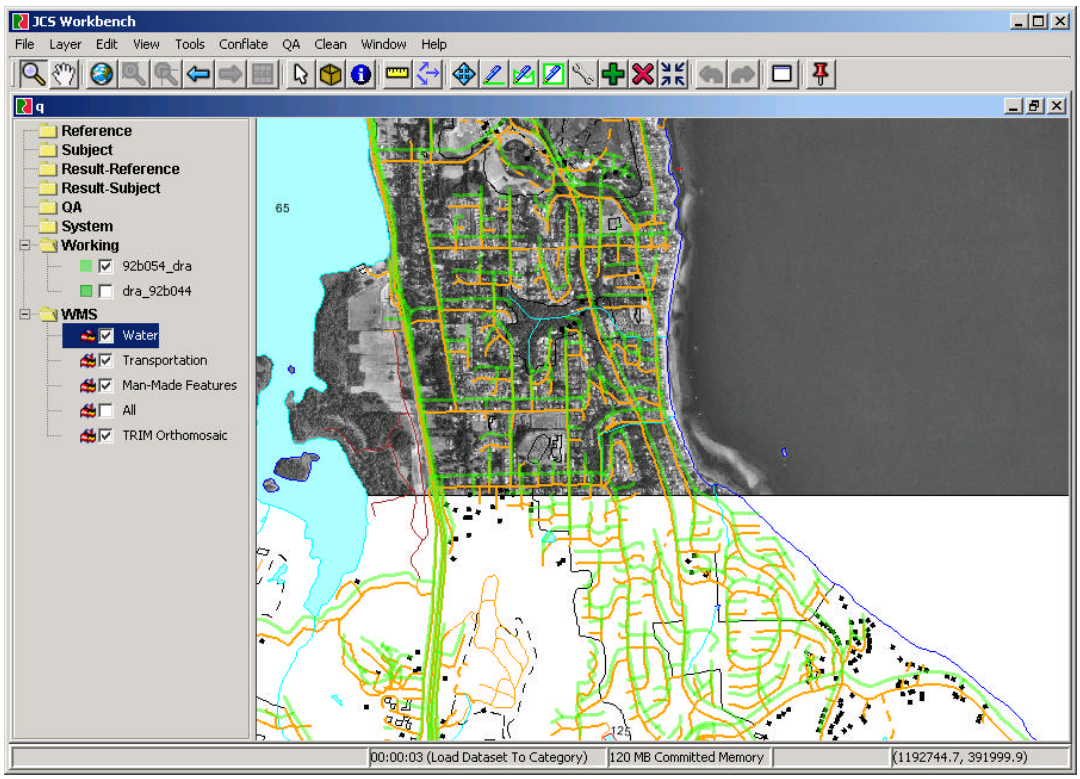


Figure 7 - A JUMP screenshot showing both vector and WMS image layers

8. QUALITY ASSURANCE TOOLS

A prerequisite for most spatial processing is that it operate on geometrically and topologically valid data. In addition, it is often useful to know if a dataset contains topological or geometric errors. There are few tools currently available to perform these kinds of checks rigorously. For this reason the JUMP API implements a variety of basic

topological and metric checks. The JUMP Workbench also provides visualization tools to help identify the location of these errors precisely.

8.1 BASIC GEOMETRY CHECKS

Topology Validation	Validates that the geometry for a feature is a valid OGC Geometry
Repeated Points	Tests whether a geometry contains repeated points (which are allowed in OGC-valid geometries)
Polygon Orientation	Tests whether the rings of a polygon are oriented with the shell CW and the holes CCW.
LineStrings Simple	Tests whether linestrings are simple

8.2 GEOMETRY METRIC CHECKS

Segment Length	Reports geometries which contain a segment of less than a specified length
Minimum Angle Size	Reports geometries which contain an angle of less than a specified size
Polygon Area	Reports geometries which have less than a specified area

8.3 GEOMETRY TYPE CHECKS

All of the JTS Geometry types can be checked for their presence in a dataset. This can be useful to assure that a dataset contains only geometries of a given type.

9. WARPING

9.1 AFFINE TRANSFORM WARPING

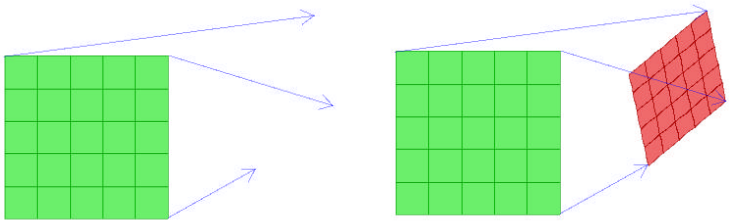
The Affine Transform warp (see Figure 9-1) allows you to perform a deformation of a dataset via translations, rotations, scales, flips, and shears.

In the general case three vectors are specified. The initial and final vertices of the vectors define two triangles. The affine transform that deforms the initial triangle into the final triangle can be computed from the warp vectors. The general affine transform includes scaling, translation, rotation, flips, and shears. The affine transform is applied pointwise to the geometries in the dataset.

The warp will fail if the vertices of either triangle are collinear.

If two vectors are specified, the warp will create triangles by rotating each line by 90˚. This results in an affine transform, which includes scaling, translation, and rotation.

If a single vector is specified, the warp will create triangles by picking points at x + 10 and y + 10. This results in an affine transform, which includes only a translation.

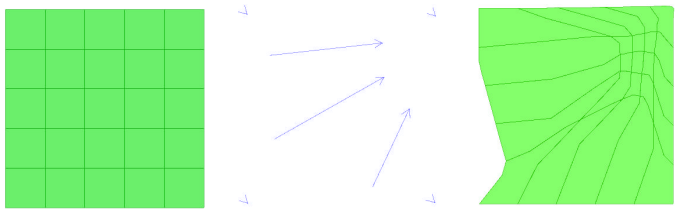


(a) (b)

Figure 9-1 – Affine-transform warping. [a] The original dataset (green) and the three warping vectors. [b] The warped dataset (red).

9.2 RUBBER-SHEET WARPING

The Rubber Sheet warp (or more precisely the Bilinear Interpolated Triangulation warp) is a simple technique for deforming a dataset using a set of control point pairs. Two triangulations are created, one using the source control points and one using the destination control points. Each source triangle / destination triangle pair implicitly defines a linear transform that carries the each point of the source triangle to the corresponding point of the destination triangle. The dataset is transformed pointwise using the transform associated with the source triangle in which each data point falls.



(a) (b) (c)

Figure 9-2 – Rubber-sheet warping. [a] Original dataset. [b] Warping vectors. [c] Warped dataset.

Full details on this algorithm are given in [Whi85] and [Saa85].

10. SPATIAL DATA I/O

JUMP supports input and output of a variety of spatial data formats. Spatial data format I/O is implemented as drivers. It provides an abstract framework for loading, initializing and executing drivers which process the spatial data formats. A driver is either a Reader or Writer of a given format. Drivers may take various parameters to specify where to find the data and how to read it (for instance, to specify the pathname of a template for interpreting a GML schema).

10.1 WELL-KNOWN TEXT

Well-Known Text (WKT) is supported for input and output. The OGC specifies the WKT syntax for a single geometry. JUMP extends this to allow files of WKT, which may contain multiple WKT items. WKT does not support specifying attributes.

10.2 GML

Like XML, GML is not a single format but a meta-format that supports many different kinds of spatial models. Essentially, it provides arbitrarily nested feature collections, with features containing arbitrary spatial and attribute information. In contrast, the JUMP spatial model is more restricted. It supports single-level feature collections, with features that contain a single spatial attribute and multiple scalar attributes. The JUMP GML drivers are not intended to provide model-to-model transformation. The GML support in JUMP is primarily focussed on providing the ability to input datasets which fit the JUMP Feature model. This is a common model for spatial datasets; so many GML datasets will fit this model.

10.2.1 Input Templates

What templates can specify

10.2.2 Output Templates

10.2.3 JCS GML

GML file including template and data
Example

10.2.4 FME GML

Template created from file, with datatypes

10.3 ESRI SHAPEFILE

ESRI Shapefiles are supported for both input and output. The Shapefile geometry and feature model maps directly to the JUMP Feature model. All types of Geometry and attributes are supported.

11. SPATIAL ACCESS METHODS

Spatial Access Methods (also commonly known as spatial indexes) are ways of efficiently accessing subsets of spatial datasets based on spatial queries. For instance, a typical spatial query is to return all features from a dataset which intersect a given range rectangle. Without a spatial access method, answering this query necessitates scanning every item in the dataset. If a spatial access method is used, only a potentially small subset of the data items need be scanned.

JUMP algorithms often use spatial queries to find candidate sets of features for such things as finding all polygons which overlap or all linestrings which match. In order to make these queries efficient it uses spatial access methods for the datasets being queried. In the course of the JUMP project several different spatial access methods have been implemented. The characteristics and performance of these methods are discussed below.

There is a large number of Spatial Access Methods that have been researched and documented. Choosing a spatial access method that is optimal for a given task involves weighing many different criteria. These include:

Storage Technology	Most spatial access methods can be implemented as in-memory only or utilizing secondary storage such as disk. Implementations, which use secondary storage, can store much larger volumes of data, but are significantly more complex to implement.
Dynamic VS Static Data	Most spatial access methods are designed to support dynamic datasets that can change. Dynamic indexes support both insertion and deletion of data items and attempt to provide good query performance over any sequence of operations. Some situations involve static datasets that do not change. Static indexes do not support deletion or insertion, but can offer better performance than dynamic ones.
Implementation Complexity	Some spatial access methods require complex data structures and algorithms. These may require a substantial effort to implement, and just as importantly to validate for correctness.

The JCS project has investigated a number of relatively straightforward spatial access methods. The emphasis has been on implementations that support in-memory datasets only.

In most cases spatial queries in conflation algorithms are carried out on static (unchanging) datasets. This allows the use of static spatial access methods such as the STR-Packed R-Tree. Static spatial access methods offer superior performance to more general indexes such as Quadtrees. JUMP uses STR-Packed R-Trees as its default spatial index.

Spatial Access Methods can be categorized into two broad groups: Space-driven structures and Data-driven structures.

Space-Driven Structures have a structure that is determined by the coordinate space that the data is embedded in, rather than the actual coordinates of the data itself. The size and performance of the index is generally independent of the order of insertion of the data.

Data-Driven Structures have a structure that is determined by the coordinates of the data that is inserted into them. Usually the order of insertion of the data affects the size and performance of the index. This offers the opportunity of optimizing the index for static situations by changing the order of insertion of data. This is known as packing the index.

11.1 SPACE-DRIVEN STRUCTURES

11.1.1 Quadtree

A Quadtree is an index based on a division of space into a hierarchy of quadrants. A given quadrant may contain up to 4 child quadrants. Items are stored in the smallest quadrant that completely contains the item. Querying the Quadtree involves returning all items that are contained in any quadrants, which intersect the query range rectangle.

An important feature of the Quadtree implementation in JUMP is that it does not require the index to be parameterised by the total extent of the dataset to be indexed. This makes it fully dynamic.

11.1.2 BinTree

A BinTree (Binary Tree) index is a 1-dimensional analogue of the Quadtree index. It has exactly the same properties as the Quadtree. It is useful for performing efficient stabbing line queries.

11.1.3 QB-Tree

A QBTree (Quad-BinTree) is a Quadtree in which each quad node contains two BinTrees that index the items in the node along the x and y axes. In theory the QBTree offers better query performance for datasets where quads contain many items. With the kinds of datasets used in the JCS project we have not observed a great deal of difference in performance between the QBTree and the QuadTree.

11.2 DATA-DRIVEN STRUCTURES

11.2.1 STR-Packed R-Tree

The STR-packed R-tree is an R-tree created using the Sort-Tile-Recursive (STR) algorithm [Rig02]. It is a spatial access method for static data; it makes no provision for inserting or deleting items. Compared to R-trees, STR-packed R-trees have far less overlap between nodes. This tree is simple to implement and maximizes space utilization; that is, all leaves are filled to capacity.

A STR-packed R-tree is constructed from the ground up, beginning with the items, and then creating the level-0 nodes (the leaves), then the level-1 nodes, and so on until the top-level node is created (the root). The STR algorithm is as follows:

- Let N be the number of items and M be the order of the tree (or node capacity). Then the number of leaves P is $\lceil N/M \rceil$.
- Compute the midpoints of the envelopes of the items.
- Order the items by the x-values of the midpoints, and group them into $\lceil \sqrt{P} \rceil$ vertical slices.

- For each slice, order the items by the y-values of the midpoints, and group them into runs of size M.
- For each run, create a new node.
- The level-0 nodes are created. Recursively create higher-level nodes using the same algorithm.

Figure 11-1 below shows the nodes of an STR-packed R-tree. The dataset used for testing was a set of approximately 15,000 city lot polygons from Victoria, B.C.

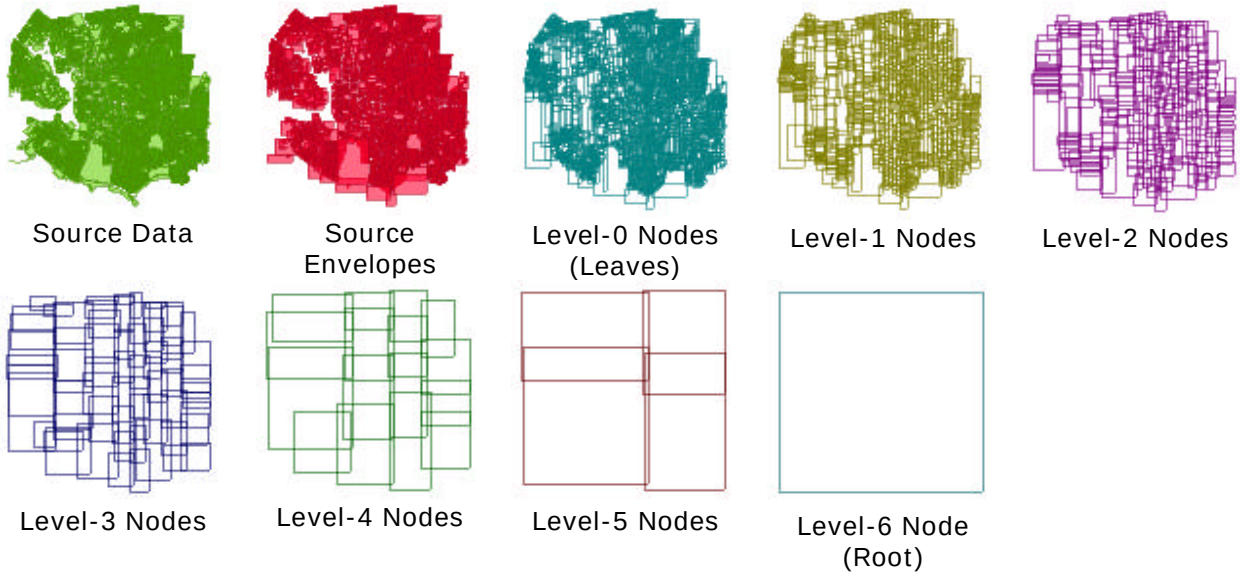


Figure 11-1 – Features of Victoria, their envelopes, and the various levels of the STR-packed R-tree [M=4]. Note that there is little overlap between the nodes in each level.

Figure 11-2 below shows the load times and query times of STR-packed R-trees with various values of the node capacity, M.

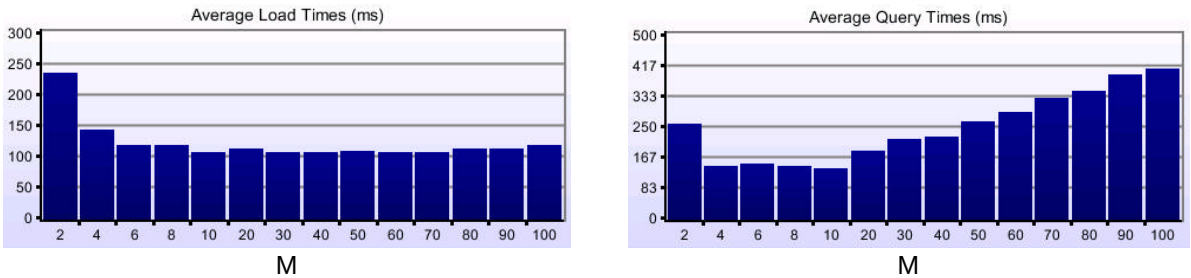


Figure 11-2 – Average load times and query times for STR-packed R-trees with M ranging from 2 to 100. Data: features of Victoria.

The load time is the time to build the tree. The query time is the total time to run queries for the envelope of each item of the original dataset. Times were averaged over 3 runs.

Load time does not appear to change above M=8. Query time is minimized at around M=10; higher node capacities increase the query time but save space.

11.3 PERFORMANCE COMPARISON

Figure 11-3 below compares the load times and query times of the Quadtree, the QB-tree, and the STR-packed R-tree. (The dataset is the same one used above)

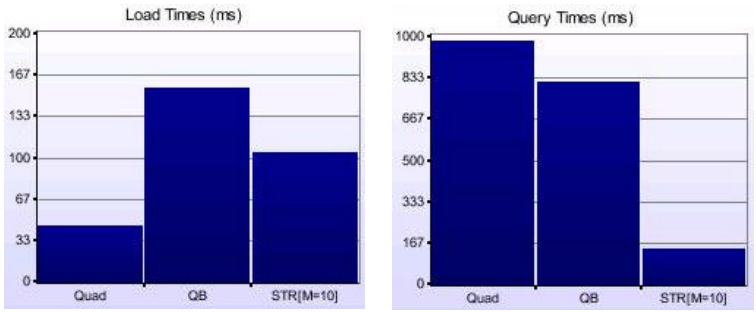


Figure 11-3 – Average load times and query times for the Quadtree, the QB-tree and the STR-packed R-tree with M=10 (Victoria city lot polygon dataset). Times were averaged over 10 runs.

Note that the query time is significantly reduced for the STR tree. The QB-tree query time is 83% that of the quad tree, while the STR-packed R-tree query time is 15% that of the Quadtree.

12. REFERENCES

[Rig02] Philippe Rigaux, Michel Scholl, and Agnès Voisard. Spatial Databases. Morgan Kaufmann, 2002.

[Saa85] Alan Saalfeld. A fast rubber-sheeting transformation using simplical coordinates. The American Cartographer, 12(2):169-173, October 1985.

[Whi85] Marvin S. White, Jr. and Patricia Griffin. Piecewise linear rubber-sheet map transformation. The American Cartographer, 12(2):123-31, October 1985.