

Rendering Optimization

This summary of several ideas that have been floating around the commiity for a while. If I touch on your favorite idea and mangle to totally mess up the explanation please jump in and fix and comment.

Definitions

- Throughput, Number of bits/features/transactions per unit of time
- Performance Time, per bit/feature/transaction (inverse of throughput)
- Latency, Wait time for a response
- Capacity, Number of entities the system can support in a given configuration at a fixed level of performance
- Scalability, Ability to increase capacity by adding hardware
- Reliability, Length of time the system can run without failing
- Response Time, Total perceived time it takes to process a request

Subjective Performance

Response Time is arguably the most important measure with respect to rendering.

DJB: but **Response Time** is difficult to define. Udig probably wants to put the first feature on the screen quickly. Geoserver WMS wants to get the image to the user as quickly as possible.

Response time is a subjective quality - and subjective experience is hackable:

- Do Something quickly, by responding at all the percieved processing time is reduced.
- Adjust percieved response time by rendering longer lines first.
`order by length(the_geom) DESC`
User's "feel" that the render is basically done, and largely ignore it filling in a bunch of "small" stuff near the end
- Many fast, studies show that a a progress that has many small steps each of which has a fast progress bar, appears to go faster then providing one process bar that slowly works though each of the steps. I seem to recall people percived the difference to be as high as 30%.

Objective Performance

- Capacticty, the big concern is not making use of a system that limitis the number of features that can be rendered to the size of memory
- Latency, we would like a minimal wait for response to start being displayed on screen. This is mostly bound by the turnaround time of the DataStore prociding the features
- Throughput (or Performance Time) indicates the efficiency of of drawing process, although factors such as the speed in parsing GML data make actually provide the bottleneck
- Scalability (throwing hardware at the problem) is desirable but not the end all and be all of solutions. Using an OpenGL based renderer is wonderful for rendering graphical content, but you may force your users into the purchase of a new video card.

Index

The use of an index (specifically a spatial index) is an optmization we can do that will actually help. That said it may not allways help. For our shapefile datastore, for small shapefiels, the entire file is likely to be memory mapped (and an index is likely to get in the way).

Links on Spatial Indexes

- <http://www.cs.ucr.edu/~mariah/spatialindex/>

There are three index techniques that I have heard of:

- STR-Tree - JTS implementation. Need to provide all the features before use, get a nice balanced tree. Seems to be the best when you can use it. So fastest for use, poor choice in the face of edits.
- R-Tree (Region Tree) - I think [Shapefile Index Support](#) was going this way.
- Quad-Tree - subdivide and conquer, my impression was that this one wanted to know the total bbox first (something we can get from a shapefile), and then splits into 4 (ish) boxes when the content grows too big for one node in the tree.

The abilities I am looking for out of the index system are the following:

- spatial query the usual point of a spatial index
- ability to short circuit the rendering process based on index information.

The short circuit idea is my own (well thanks to David Blasby too) and I am not sure how sane it is. If the features in an index node are all less than one pixel at the current resolution then:

- one feature needs to be displayed for line data like roads
- no feature needs to be displayed for polygon information like lakes

My best case scenario for indexing is the following:

- Quad Tree implemented for Shapefile, so each node has a distinct spatial location
- Ability to perform a spatial query for the area on the screen.
- Ability to "recognize" when a the bounding box for a node is less than a pixel in size. If so renderer can request one (and only one) feature, for rendering as a single pixel. Feature returned should be reproducible (something like the **first** feature) across runs.
- Ability for the index to be memory mapped into memory. I would prefer not taking on the added complexity of an index that is cached into memory manually.
- based on mapserver .qnx files? There is C source code to port, and a working executable to generate the correct index files.
- The volume of feature data required should remain fairly constant at various levels of zoom. As we zoom in the spatial query provides us with less features. As we zoom out the short-circuit reduces the number of features we need to render.

Note: that these index requirements, and the joining requirements start to make this whole thing look like a limited spatial database. Admittedly it can talk to more data sources (but we should look into working with others on these issues).

Note: the short circuit idea would need to reverse reproject a pixel(at each corner) back to the world coordinates.

Feature Cache

The idea of caching Feature information is currently in vogue:

- j2d renderer (Martin) makes use of a cache information derived from a feature (point lists?), and enough information to style and paint them.
- lite renderer - currently does not cache
- lite_renderer2 - but Andrea has been asking martin a lot of questions so I kind of expect lite_renderer 2 to go

in this direction

- GO-1 - makes use of a FeatureCanvas as a bridge between Features and Graphic (where graphic has the geometry and style information)

The various implementations have different tradeoffs:

- most use float rather than double (for space saving)
- some do great reprojection (turning straight lines) into curves based on pixel size

Well as you can see above nobody actually caches features they all cache a simplified geometry, paint and style information.

Raster Cache

The focus here is on minimizing the number of requests, and optimizing pan and zoom operations.

1. Request feature information for the current screen
 - a. Render on to the current raster
 - b. Render on to the zoomed-out raster (the content will only over the center of the zoomed-out raster)
 - c. Render on to the zoomed-in raster (only content from the center of the current raster will be used)
2. Request information for "tiles" surrounding the current raster
 - a. Render this information to the surround raster
 - b. Render this information to the zoomed-out raster - the content will now completely render the zoomed out raster

Originally I thought we could favour what requests are need based on the current mode (pan or zoom) but it seems that the same requests need to be made regardless.

- Pan, makes use of raster blit operations to copy information from "surround" raster to the screen
- Zoom, replace the current raster with the zoomed-in raster.

Note: This idea makes more sense with the fixed stepping size usually associated with zoom-in and zoom-out. BBox based zoom is unlikely to be helped by these ideas.

Note: Some features (like polygons) are almost scale in-dependent and requesting a larger zoomed-in raster (and then rescaling it) may work. Line work however does not respond well to scaling, although the idea may warrent attention to provide a form of immidaiate feed back.

Note: For features that are simply styled (say all to the same color), one could get away with storing 2 bit depth rasters and applying the color at rendering time using the graphics2d.drawImage (or equiv JAI operations).

Postgis Hack

There are many simple hacks we can do when asking for spatial data from postgis.

- use wkb - thanks to those who set this up!
- sort by length - improves "response time" experience

But now for something completly different. In working with an OpenGL based renderer years ago one thing we ended updoing was defining wrapper functions that simplified the data sent depending on the current zoom level.

These ideas are mostly applicable to line work, but the ideas are sound. Consider a function that provides a real world "length" assocaited with a the diagonal of a screen pixel.

The easiest implementaiton is to walk along the line, or linear ring, and sample about the distance every pixel occurs. For many real world features this is sufficient. The resulting geometry is only good for display but the

reduction in size is worthwhile.

Another idea would be to provide a modification of wkb that worked with shorts rather than doubles. I know this sounds like a bad idea, but spatial data is often very location dependent (especially with respect to a single geometry). By providing the geometry as a series of short values we can realize a huge reduction in data size. The short values would need to be considered as a scaled value within the bounds provided by the geometry's bounding box, or offsets from the first point.

If we cast our mind back to the earlier Feature Caching conversation, the above reduced Geometry starts to look similar to the Graphic construct from GO-1 or the point lists from the j2d renderer.

If we also included the projected bounding box of the screen we could ask the database to perform clipping for us. I kind of doubt the database would be kind enough to let us hack the internal quad trees (so we could pull off an index based rendering short circuit). So cutting down the size of the transmitted geometry is probably the best we could do.

Summary

All these optimizations have their place, and none of them are ready to be applied. We will need to come up with some hard numbers to see which, if any of these ideas are worth implementing.

Even then there will be run time tradeoffs.

- almost any of these schemes will get in the way (slowdown and complicate) a memory mapped shapefile
- caching Features may be applicable for a data size of 1000 features, caching Graphic (simplified Geometry + styling) may be applicable to to 10,000. Using an index with the short-circuit may be worthwhile over 10,000. Caching Rasters may be worth while for a WMS.