4D WebViewer prototype

# Introduction

This document describes the BPA2 4D web viewer prototype. The aim for this project is to do server-side 3D rendering of basin models, and display the changes in basin geometry over time (hence ‘4D’). The user will interact with the rendering through a web browser.

# Overview

The main goals for this project are

* Test the rendering performance of Open Inventor MesViz XLM with large data
* Evaluate the Open Inventor RemoteViz solution for client-server rendering

The requirements for this prototype are

* Draw a mesh (skin)
* Switch between meshes from different time steps (snapshots)
* Map properties on the mesh, and switching between properties
* Draw I, J slices and be able to move them through them mesh interactively
* Build a client / server setup using RemoteViz to check the web connection

An important assumption that we allow ourselves to make is that all data fits into available RAM. Given that the development / test machines have 64GB available, this should still allow for reasonably large datasets.

# Data

The data to be displayed is organized as a 3D hexahedral grid. Grid spacing is regular in *x* and *y*, where *z* can be arbitrary.



Figure 1 Grid structure

A data set defines 2 separate grid resolutions: a low resolution grid on which formations are defined; and a high-resolution grid for reservoirs. The entire data set consists of a number of ‘snapshots’, instances of both low and high resolution meshes for a certain point in geological time.

# SDKs

The project is built using the Open Inventor toolkit by FEI. In particular, the components MeshViz XLM and RemoteViz are used.

## MeshViz XLM

MeshViz XLM is a library that allows us to visualize large meshes in a variety of ways. It has support for line, surface and volume meshes, obviously we’re mostly interested in the latter. Meshes are visualized by *extracting* certain features from the data, e.g. we can extract and render a single slice from a volume mesh.

The MeshViz extraction and data mapping algorithms work with *interfaces*, that developers can implement in order to provide MeshViz with access to their own data structures. This means that only one copy of the data needs to reside in memory, which is quite convenient when working with data sets that can easily take up dozens of GB of space.

The main types of volume meshes are

* **Unstructured meshes**: The simplest and most general type of mesh. Tetrahedral meshes use this type, but topology can be completely arbitrary. Each volume cell can define its own facets, and specify the triangulation of those facets.
* **Structured meshes**: Using a 3D tetrahedral grid topology. The geometry can be regular, rectilinear or curvilinear.
* **Unstructured IJK meshes**: This type of mesh assumes a hexahedral IJK grid, but each cell can independently define its vertex indices, which allows for representation of faults and other discontinuities.

At first sight, the appropriate choice for this project would be a structured mesh. However, the available types of structured mesh don’t allow for arbitrary *z* coordinates. This leaves us with the third option, unstructured IJK meshes. This type of interface is a bit too general for our purposes, which has some performance implications (see Skin extraction).

Rendering of mesh data in Open Inventor happens by adding several nodes to a scenegraph: a data node that stores the complete topology and geometry information (one of the mesh types mentioned above), and one or more ‘rendering nodes’, which extract certain features from the data set, and render them in their own way. Some available render nodes are:

* Mesh skin – consisting of all the faces that belong only to one volume cell
* Axis-aligned slices
* Arbitrary cross sections
* Isosurfaces (based on property values)

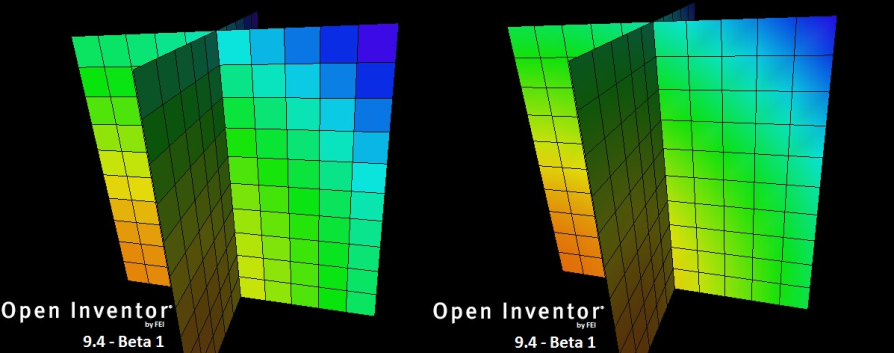


Figure 2 Per-cell vs per-node property mapping

Scalar and vector properties can be mapped onto the mesh on a per-cell or per-node basis. Colors can be assigned using pre-defined or custom colormaps. When neighbouring cells share a grid vertex, they also share the property value on that vertex. If this is unwanted, e.g. because of discontinuous properties on formation boundaries, the vertex needs to be duplicated.

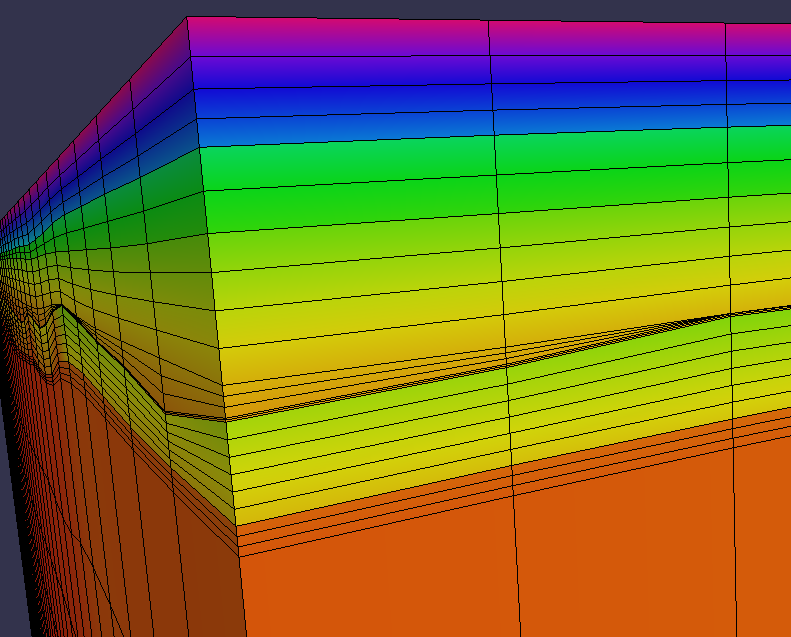


Figure 3 Mapping of discontinuous properties

MeshViz does not come with a canned solution for mesh decimation. When the mesh data becomes too big to fit into available memory, a custom solution is necessary to display a lower resolution mesh. Fortunately, because of the regular nature of our data, it is not too hard to generate a lower resolution version of it.

## RemoteViz

The 4D prototype is built using a beta version of the RemoteViz library. This library provides both server-side and client-side components that enable developers to build remote rendering solutions with a browser-based user interface.

The server-side components allow us to build a rendering application with all the functionality that Open Inventor provides, but instead of rendering to the screen, it renders to an offscreen buffer, which is encoded to a JPEG image and sent to the client.

The client side of the toolkit is based on HTML5, and a javascript library is provided to handle the user interaction. All communication between server and client happens over a websocket connection: mouse and keyboard events are sent from client to server, rendered images from server to client, along with custom string-based messages defined by the developer.

A typical setup would look like Figure 4.

Figure RemoteViz server-side setup

Web server (IIS, Apache, NGINX etc.)

Client

Render server

HTTP(S)

websocket

Data

# Project structure

The 4 main parts of the project are

* Desktop application – Primarily used to develop the core rendering features with MeshViz XLM, make debugging easier and faster, and get an accurate sense of the actual performance of the rendering.
* Server application – Based on the server-side components of MeshViz XLM.
* Visualization dll – Shared rendering code that is used in both the desktop and server applications. Builds a complete Open Inventor scenegraph for the provided basin model.
* Html web client – Heavily based on one of the RemoteViz sample applications, the web client implements a subset of the features of the desktop application, and Is mainly used to get an idea of network performance, image quality and used bandwidth.

Data is loaded using the Basin Modeling *DataAccess* library.

# Scenegraph

The MeshViz interface that we’re using for our data is [MiVolumeMeshHexahedronIjk](http://oivdoc93.vsg3d.com/content/c-api?apiurl=http://oivdoc93.vsg3d.com/APIS/RefManCpp/class_mi_volume_mesh_hexahedron_ijk.html), which belongs to the class of *unstructured ijk* meshes. The main function of this interface is to provide access to the corresponding topology and geometry classes.

The topology for the mesh is defined by an implementation of the [MiHexahedronTopologyExplicitIjk](http://oivdoc93.vsg3d.com/APIS/RefManCpp/class_mi_hexahedron_topology_explicit_ijk.html) interface. This interface has methods to retrieve the number of cells in I, J and K direction, for each cell get the 8 coordinate indices of the vertices that define that cell, and request whether a certain cell is ‘dead’ (undefined) or not. The geometry for the mesh is stored in a 1D array of coordinates, and accessed via an implementation of [MiGeometryI](http://oivdoc93.vsg3d.com/APIS/RefManCpp/class_mi_geometry_i.html).

Figure Scene graph structure

Timestep switch

Snapshot 1

MoScalarSetI

MoMesh

Rendermode switch

MoMeshSkin

MoMeshPlaneSlice

Snapshot 2

MoScalarSetI

Snapshot 3

## Time roaming

Working on the assumption that all data fits in memory, stepping through time is accomplished by generating meshes for all snapshots, and only displaying the one belonging to the current timestep. In Open Inventor this can be done making the meshes child nodes of an *SoSwitch* node, which allows us to select which of the children is currently visible. Moving through time is then a matter of changing the index on the switch node.

## Render mode switch

The same mechanism is used when switching between render modes (i.e. skin, slices, cross section). As mentioned before, MeshViz works by adding a data node ([MoMesh](http://oivdoc93.vsg3d.com/content/c-api?apiurl=http://oivdoc93.vsg3d.com/APIS/RefManCpp/class_mo_mesh.html)) to the scene graph containing the core data, and adding one or more rendering nodes ([MoMeshSkin](http://oivdoc93.vsg3d.com/APIS/RefManCpp/class_mo_mesh_skin.html), [MoMeshPlaneSlice](http://oivdoc93.vsg3d.com/APIS/RefManCpp/class_mo_mesh_plane_slice.html), etc.) that extract certain features from that data in order to render them. By putting these render nodes under a switch node, selecting a different render mode is a matter of changing the index on the switch.

## Properties

Scalar properties are mapped onto the mesh by adding a [MoScalarSetI](http://oivdoc93.vsg3d.com/APIS/RefManCpp/class_mo_scalar_set_i.html) node to the scene graph containing the property data. Colors can be assigned by adding one of the subclasses of [MoColorMapping](http://oivdoc93.vsg3d.com/APIS/RefManCpp/class_mo_color_mapping.html).

# Server

The main components of the render server are

* **Service** – the central component that does all connection handling
* **ServiceListener** – interface for handling new connections and render areas
* **RenderArea** – takes care of rendering the scene graph
* **RenderAreaListener** – interface for responding to client-side events, such as mouse and keyboard events, resize events and custom messages

Figure RemoteViz components

Service

ServiceListener

RenderArea

RenderAreaListener

Device

# Web client

The client-side part of RemoteViz is based on HTML5, and comes with a Javascript library to handle communication with the server. Developers are free to design the page however they want, as long as it contains a *canvas* element where RemoteViz can do its part.

The rendered images arrive at the client encoded as JPEGs. The necessary bandwidth can be controlled using several settings:

* **Interactive vs high-quality mode** – when the user is interacting with the scene (rotating, dragging etc.) RemoteViz requests new images using interactive quality settings. As soon as interaction is finished, a high quality image is requested.
* **JPEG compression rate** – the JPEG quality can be set separately for interactive and high quality settings
* **Scaling** – Interactive quality images can optionally use a scaling factor, where the image is rendered at a lower resolution, and stretched to full size in the browser (although this did not seem to work in the beta version)
* **FPS limit** – a limit can be placed on the number of frames per second that are rendered

During discussions the requirement of ‘picking’ or ‘hit-testing’ came up. This is something that Open Inventor ususally supports quite well. Since all mouse events are already sent to the server, we can use the full range of OIV picking support on the server, and send any results to the client using the custom messaging facility.

# Results

The final prototype has a web interface where the user can

* choose between formations and reservoirs
* choose a property to be mapped onto the mesh
* choose between skin and slice rendering
* move sliders to choose which I,J slices to render
* move a slider to step through time
* set the quality settings for interactive and high-quality mode
* rotate, scale and translate the meshes using the mouse

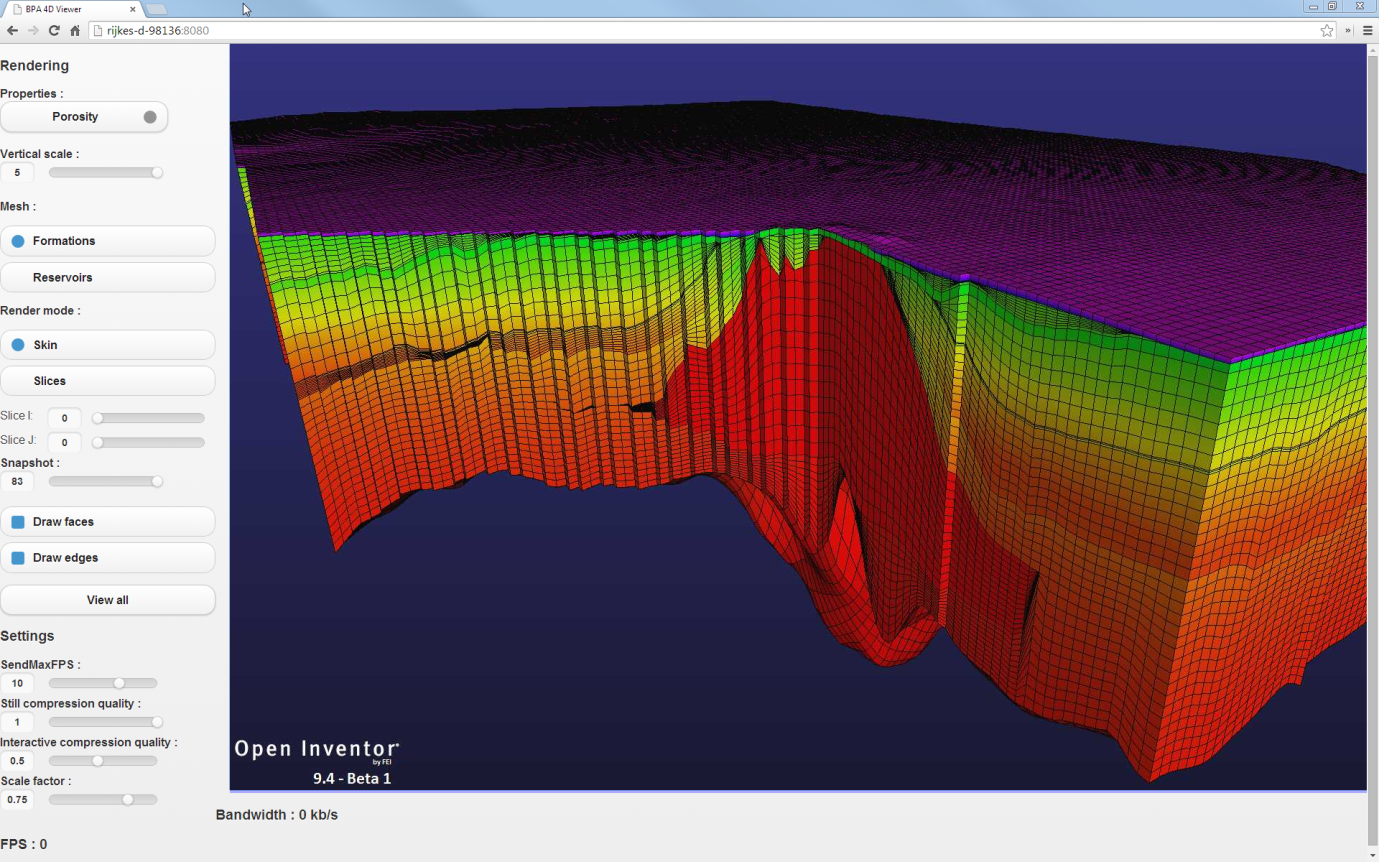


Figure 7 The final web interface of the prototype

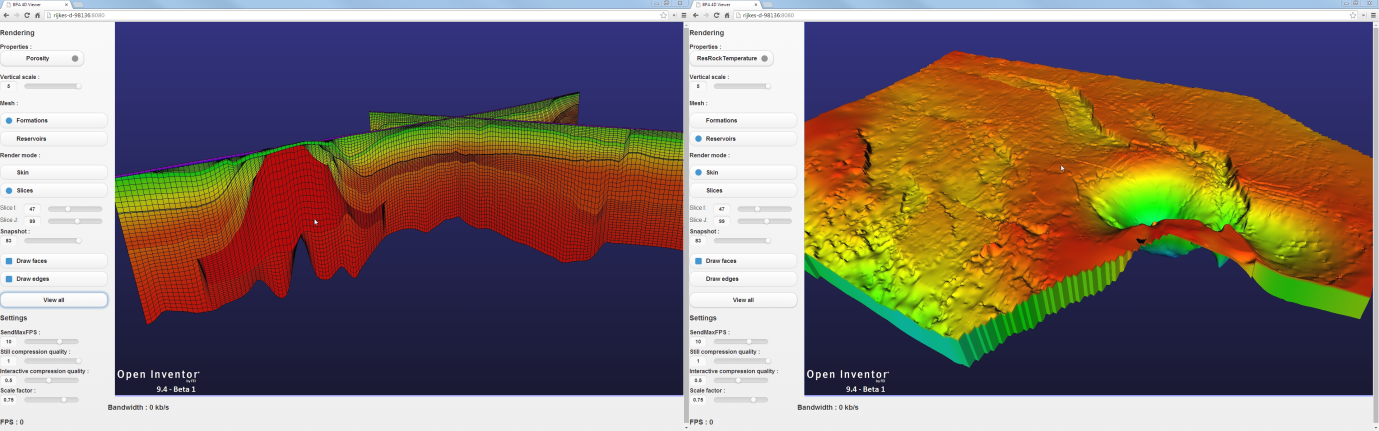


Figure 8 Slices (left) and reservoir (right)

## Performance

This section presents some numbers on the performance of the 4D prototype, as measured on a system with the following specs:

* CPU: Intel Xeon E5620
* Memory: 64GB
* GPU: 2x Nvidia Quadro 5000
* OS: Windows 7 (64-bit)

The data set for the test has the following properties:

* Nr. of time steps: 84
* Low-res grid: 132x187 (formations)
* High-res grid: 525x748 (reservoirs)
* Size on disk: approx. 36GB

The number of cells in the K direction varies per timestep, but has a maximum of 93 for the formations mesh, and 3 for the reservoir mesh. This makes for a total number of cells of 2.3M and 1.2M respectively. Some of the measurements were taken while simulating a larger mesh, by subdividing this one along the I and J axes.

A note on the time to load the data: disabling the *connectMigrations()* function in the DataAccess library decreased loading time from 2:30 to 0:20 minutes.

### Skin extraction

In order to view our data, some kind of renderable mesh needs to be extracted, e.g. the outer skin, slices, iso-surfaces etc. The extraction process takes a volume mesh (e.g. hexahedral, tetrahedral) and generates a surface mesh (e.g. triangles) from it.

Preparing the surface mesh for rendering requires a couple of extra steps from Open Inventor, such as generating the vertex buffers for the triangle mesh and the gridline mesh, and generating normal vectors.

Figure Mesh extraction pipeline

Volume mesh

OpenGL mesh

Surface mesh

Both the extraction of the surface mesh, and the preparation of a renderable OpenGL mesh take considerable time.

The extraction process can be done in several different ways:

* implicitly by Open Inventor when specifying a rendering node for the data
* explicitly as a pre-processing step, adding only the extracted surface mesh to the scene graph
* explicitly using a custom extractor

As mentioned in section ‘MeshViz XLM’, the unstructured IJK mesh type is a bit too general for our purposes. This has performance implications that quickly become painful when extracting the skin surface from larger meshes. A custom extraction algorithm was written which provided a 10x speedup over the standard extractor, by using detailed knowledge of the data structures that is not available to MeshViz.

## Measurements

This section presents the actual performance measurements.

### Pre-extracting the skin mesh

The skin rendering measurements in XXX also contain values for rendering pre-extracted meshes. This requires a preprocessing step that also takes time, the measurements of which are presented here. The ‘OIV extractor’ is an instance of the [MiSkinExtractUnstructuredIjk](http://oivdoc93.vsg3d.com/content/c-api?apiurl=http://oivdoc93.vsg3d.com/APIS/RefManCpp/class_mi_skin_extract_unstructured_ijk.html) class, which uses a parallelized extraction algorithm. The custom extractor algorithm itself is not parallelized, but the data is still processed in a parallel way by doing multiple extractions simultaneously.

|  |  |  |
| --- | --- | --- |
| **Subdivision** | **OIV extractor** | **Custom extractor** |
| 1 | 66 | 5 |
| 2 | 264 | 22 |

Table 1 Time in seconds to pre-extract all skin meshes

A subdivision factor of 2 means twice as many cells in both the I and J direction, increasing the total number of cells by a factor of 4.

An added advantage of the custom extractor is that all processing happens in the background, meaning that the user can already interact with the available meshes while the rest is still being extracted.

### Roaming through time

**Target: 10fps**

The performance of moving through time while displaying the outer skin of the mesh depends on a number of factors:

* The resolution of the meshes
* Rendering of grid lines (edges)
* Availability of pre-extracted meshes
* Cache effects

In order to simulate larger datasets, a subdivision along the I and J axes is used. This had to be limited to a factor of 3 (for formations) or even 2 (for reservoirs) to still be able to fit all data in memory. Since lots of processing needs to happen to first display a mesh, the numbers for a second run are included to show any caching effects.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **Type** | **Subdivision** | **Edges** | **Pre-extracted** | **Time (first run)** | | **Time (second run)** | |
| **Avg** | **Max** | **Avg** | **Max** |
| Formations | 1 | yes | Yes | 0.27 | 0.39 | 0.01 | 0.05 |
| Formations | 1 | no | Yes | 0.20 | 0.29 | 0.005 | 0.03 |
| Formations | 3 | yes | Yes | 3.3 | 9.5 | 0.4 | 4.5 |
| Formations | 3 | no | Yes | 1.6 | 4.9 | 0.16 | 1.0 |
| Formations | 1 | yes | No | 0.26 | 0.38 | 0.01 | 0.04 |
| Formations | 1 | no | No | 0.21 | 0.30 | 0.005 | 0.02 |
| Formations | 3 | yes | No | 3.1 | 9.3 | 0.27 | 4.5 |
| Formations | 3 | no | No | 1.6 | 4.2 | 0.19 | 2.2 |
| Reservoirs | 1 | yes | Yes | 1.8 | 9.2 | 0.3 | 5.5 |
| Reservoirs | 1 | no | Yes | 0.4 | 1.5 | 0.15 | 0.4 |
| Reservoirs | 2 | yes | Yes | 4.3 | 14.3 | - | - |
| Reservoirs | 2 | no | Yes | 2.7 | 13.9 | 0.6 | 5.3 |
| Reservoirs | 1 | yes | No | 2.2 | 9.9 | 0.3 | 2.6 |
| Reservoirs | 1 | no | No | 0.9 | 2.4 | 0.16 | 0.3 |
| Reservoirs | 2 | yes | No | 8.9 | 41.5 | - | - |
| Reservoirs | 2 | no | No | 4.5 | 11.7 | 0.3 | 1.0 |

Table 2 Times (in seconds) for moving through time with mesh skins

In some cases the measurement could not finish because of extreme memory pressure on the system. Hence the empty cells in the table.

Some conclusions that can be drawn from these numbers:

* The target of 10fps cannot be met in a significant number of cases
* Pre-extracting the skin mesh only helps performance for the larger meshes, but at the same time causes problems because of the massive amount of memory it needs.

Moving through time while rendering I and J slices is a different story, see Table 3. This proved to be sufficiently fast for all cases tested.

|  |  |  |  |
| --- | --- | --- | --- |
| **Type** | **Subdivision** | **Avg time (1st run)** | **Avg time (2nd run)** |
| Formations | 1 | .003 | .001 |
| Formations | 3 | .004 | .002 |
| Reservoirs | 1 | .002 | .001 |
| Reservoirs | 3 | .002 | .001 |

Table 3 Times (in seconds) for moving through time with I,J slices

### Switch property

**Target: 500 ms**

Switching to a different property and re-rendering the scene was measured by automatically moving through a list of properties, and computing the average time per frame. When subsequently rendering the mesh, a color map is applied based on the minimum and maximum values of the property. These values can be local or global, i.e. they can be the min / max property values for the *current* mesh, or for *all* meshes in the dataset. Of course, scanning the property values for all time steps is a fair bit more expensive than for just a single mesh.

* + Local property change: .09 s
  + Global property change: 3.6 s

Alternatively, color maps can be applied based on certain fixed value ranges, or the data format could be adjusted to include fast lookups of the minimum and maximum property values.

### Space roaming (I/J slices)

**Target: 10fps**

Moving I,J slices through a mesh is quite fast, even when using a subdivision of 3 (for a total increase of the number of cells by a factor 9). The fact that rendering is faster for the reservoir meshes is probably due to the significantly higher number of cells in the K-direction for formation meshes.

|  |  |  |
| --- | --- | --- |
| **Type** | **Subdivision** | **Avg time** |
| Formations | 1 | .03 |
| Formations | 3 | .09 |
| Reservoirs | 1 | .006 |
| Reservoirs | 3 | .015 |

Table 4 Times (in seconds) for moving I,J slices through the mesh

### Camera movement

**Target: 10fps+**

Manipulating the view (rotating, scaling, translating) for a single mesh is very fast, regardless of the render mode used (skin, slices, cross sections, etc.). Disabling V-Sync on the graphics card (to free us from the 60Hz frame rate of the monitor) yielded frame rates of over 400 fps, which shows that the GPU has no problem whatsoever with the data presented to it. The main bottleneck is in extracting the geometry that is to be rendered from the dataset.

### Bandwidth

As mentioned before, RemoteViz works with separate settings for *interactive* mode, and *high*-*quality* mode. When the user is manipulating the view (rotating, scaling, etc.), the interactive mode is active. As soon as the user stops interacting, a high quality render is performed. This allows for a reduction in bandwidth usage, while still giving visual feedback. The parameters that can be adjusted for interactive mode are

* the maximum frames per second (FPS)
* the JPEG quality
* the image scale

Note that downscaling the image did not seem to work in the beta version.

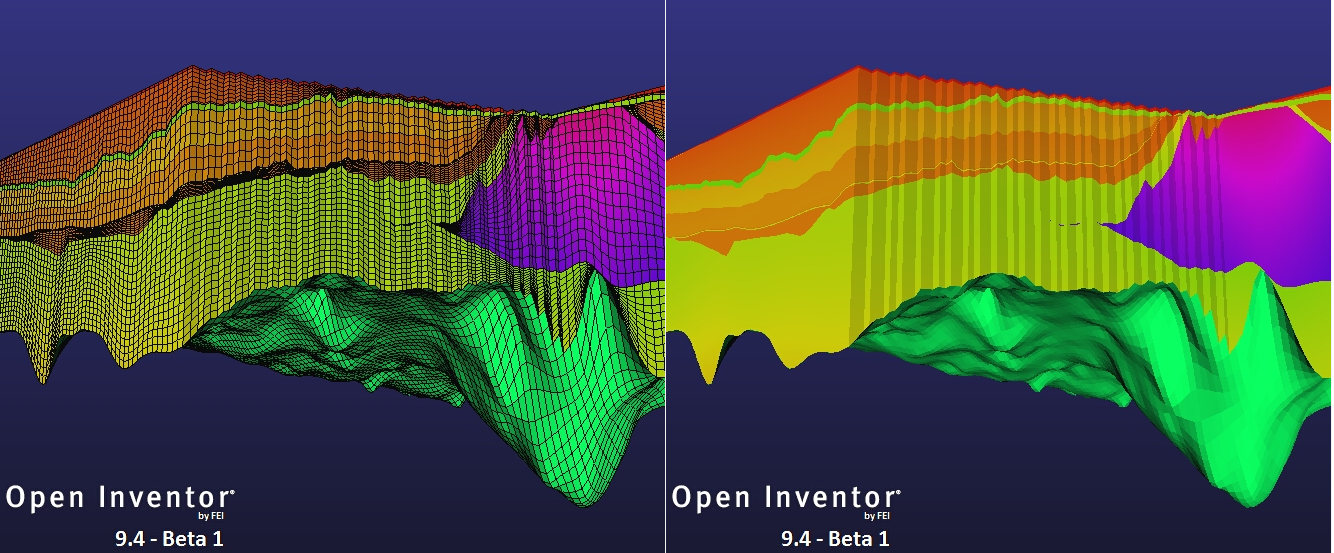


Figure 10 Edges vs no edges

Two obvious factors that influence the bandwidth used when interacting are of course the max FPS, and the image resolution. However, the actual image content also plays a large role. When the viewport contains more empty space, the resulting image will compress better. Also, enabling the rendering of edges to see the grid introduces so much high-frequency content that the compressed image size can easily go up by a factor of 3! It is therefore quite hard to come up with reliable bandwidth measurements.

Nevertheless, Table 5 presents some numbers anyway. These measurements were taken with the maximum framerate set to 5 fps, and will scale linearly with the framerate. The ‘quality’ column shows the JPEG quality factor that was used.

|  |  |  |  |
| --- | --- | --- | --- |
| **Resolution** | **Quality** | **No edges** | **Edges** |
| 1024 x 768 | 0.2 | 100 – 200 kB/s | .2 – 1.0 MB/s |
|  | 0.5 | 150 – 300 kB/s | .3 – 1.5 MB/s |
|  | 0.8 | 230 – 600 kB/s | .5 – 2.3 MB/s |
| 1920 x 1080 | 0.2 | 250 – 550 kB/s | .4 – 1.5 MB/s |
|  | 0.5 | 350 – 650 kB/s | .6 – 2.0 MB/s |
|  | 0.8 | 450 – 900 kB/s | .9 – 3.0 MB/s |
| 2560 x 1440 | 0.2 | 400 – 650 kB/s | .7 – 2.2 MB/s |
|  | 0.5 | 500 – 800 kB/s | 1.2 – 4.0 MB/s |
|  | 0.8 | 650 – 1100 kB/s | 1.5 – 6.0 MB/s |

Table 5 Bandwidth measurements for different resolutions and quality settings

Subjectively, a quality factor of 0.8 represents good quality, where 0.5 is still acceptable, and 0.2 is the point where it really starts to drop off. Note that, as with the compression ratio, the actual image content has an effect on the perceived quality as well. Images with many hard lines will suffer more from JPEG compression artefacts than smoother images.

# Latency

In order to test latency in a real-world scenario, the server was installed on an Amazon Web Services instance containing a GPU. The instance type used is ‘*g2.2xlarge’*, which has a Nvidia Grid K520 gpu on board (apparently similar to a GeForce 680). The server machine is located in a European datacenter (presumably Ireland), and tests were carried out on Shell Rijswijks network.

The image size used in this test is 1620x840. Because this test focuses on latency, only the faster rendering modes were used (slices, and skin rendering without the grid lines). Measurements were done in javascript, by recording the timestamps when changing the ‘timestep’ slider, and receiving the resulting image.

The average latency hovered around 150ms. When interacting with the model (rotating), a sustained framerate of 13 fps was observed.

On a subjective level, the experience was quite acceptable.

# Issues

Several issues came up during development of the prototype, which are discussed in this section.

## Run as service

The tests described in this document were carried out by starting the server process manually, using some form of remote log-in. While sufficient for performance testing purposes, this is of course not ideal in a production environment. The render server can be compiled as a Windows Service, but testing revealed a problem known as ‘Session 0 Isolation’, which basically means that services don’t have access to the graphics hardware. FEI is aware of the problem, but so far does not have an answer.

## Keeping all data in memory

The prototype was built with the assumption that all data fits in memory. The size of the data that was used, while being the largest dataset available during testing, is still quite modest compared to the sizes mentioned in the requirements document (20 million cells for a large model, increasing to 750 million in the next 5 years). Already cases were encountered where even 64GB was not enough (see Table 2).

## Images vs video

RemoteViz sends the rendered results to the client as JPEG encoded images, instead of using a video stream. The bandwidth requirements for this are shown in Table 5. Since a video encoding scheme can take advantage of coherence between subsequent frames, better compression rates are to be expected when encoding the rendered results as a video stream.

## Loading time

The time needed to load the test data was fairly long: 2.5 minutes. Running the application through a profiler showed that most of that time was spent in a function called *connectMigrations()*, which performs lots of linear searches through the data in order to connect different components. Since this functionality was not needed for rendering, it was disabled; reducing the load time to 20 seconds. This is most likely not acceptable in the final version of the software.

# Conclusions

Regarding rendering performance, most of the requirements look like they can be met by MeshViz XLM. One exception is time-roaming while rendering the skin of the mesh; in a lot of cases the required 10 fps could not be reached. This was even with all data residing in main memory, which is not realistic in practice.

RemoteViz looks to be working reasonably well; despite the fact that we’re working with an early beta version. The main concern here is the bandwidth requirement.

As far as functionality goes, the combination of MeshViz XLM and RemoteViz is quite flexible. Anything that can be built as a regular Open Inventor application can be converted to a server-based solution. The same flexibility is available on the client side; a lightweight javascript library provides lots of freedom to build the UI, without any major constraints.