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 OIV rendering performance with large data
* Evaluate OIV RemoteViz

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.



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.

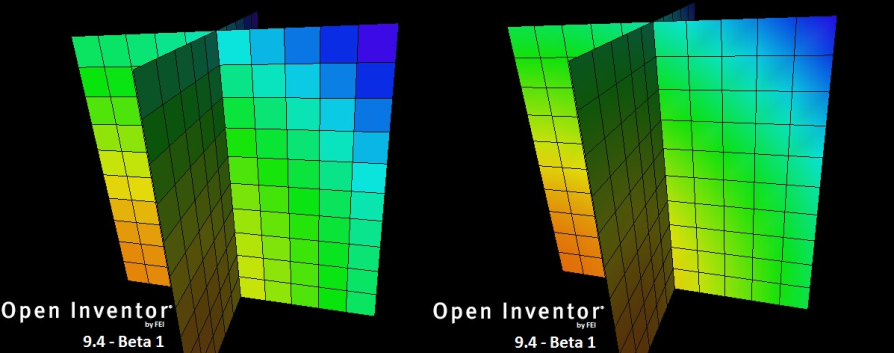
The main types of volume meshes are

* **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.

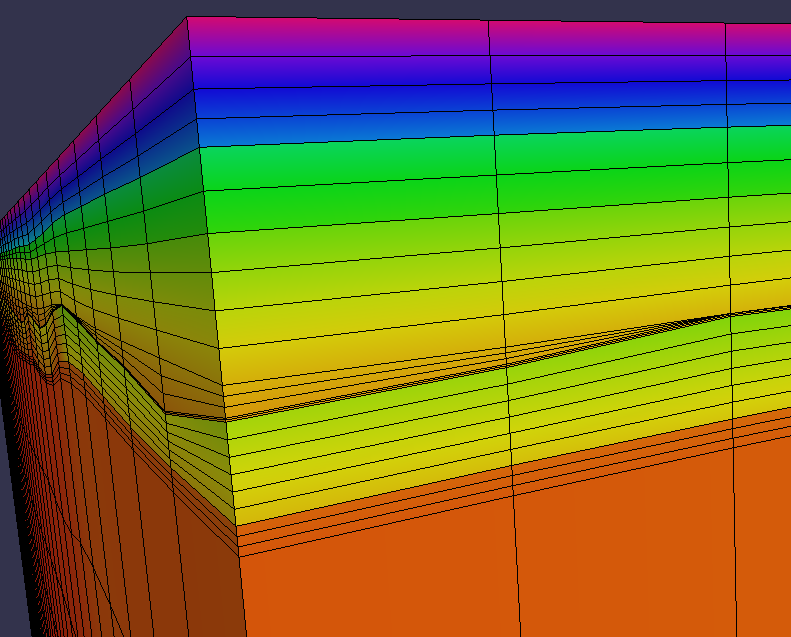
At first sight, the appropriate choice for this project would be a structured mesh. However, the available types of structured mesh don’t allow for arbitrary *z* coordinates. This leaves us with the third option, unstructured IJK meshes. This type of interface is a bit too general for our purposes, which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

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)



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.



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 x

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).

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

Service

ServiceListener

RenderArea

RenderAreaListener

Device

Client and server can also exchange custom, string-based messages.

# 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

Lots of screenshots

# Performance

* Test machine specs
* Loading time
* Skin extraction
* Roaming through time w/ mesh skin
* Custom extraction algorithm
* Switch property (local vs global)
* Bandwidth (lo/hi quality, image size, max fps)

# Issues

Run as service?

Load balancing

# Conclusion

# TODO