Slice based pipeline protocol

Jan-Willem Buurlage

1 Introduction

This document describes a communication protocol for a distributed tomographic reconstruction pipeline in which parameters used in the reconstruction can be changed in real-time, taking effect on the running reconstruction immediately.

The protocol is based on the reconstruction of individual slices, typically orthoplanes, and is useful for situations where the projection data is to big to reconstruct completely in real-time. The slices are shown together in a 3D interface, and get updated when e.g.:

- new (projection) data is available
- more iterations for iterative solvers have been applied
- higher resolution reconstructions are available
- ...

The position and orientation of the *active slices* can be changed, and this is communicated back to the reconstruction cluster, which for future updates will then reconstruct the new slices.

2 Pipeline overview

Initially we will focus on the following 'pipeline toplogy'. In a typical setting, the different stages of the pipeline are as follows:

- data source $\rightarrow$ preprocessing $\rightarrow$ reconstruction $\leftrightarrow$ controller $\leftrightarrow$ visualizer

We will ignore the left part of this pipeline for now, and assume that in the 'reconstruction' the proper data is available (or streamed in at real time). We will define the *two-way communciation* between the controller, the visualizer and the reconstructor. Their roles are roughly the following:

- The `reconstructor` manages the incoming (preprocessed) projection data, and registers itself with the controller to create a **scene**. Associated to a scene are the **active slices** and their *orientations*. Note that a reconstructor can actually be a cluster of computers, but they are all

associated to a single *scene* (dataset or real-time measurement). There can be multiple reconstructors supporting the protocol (based on e.g. odl, ASTRA or JW's sandbox)

- The `controller` manages the list of scenes, and corresponding slices, and communicates with the reconstructor(s) about changes in preferred resolution, orientation and so on. There are two reasons that this controller exists; 1) it separates the visualizer from the reconstructor which simplifies the implementation of a visualizer, and 2) it easily allows for the existence of multiple scenes which is useful for running reconstructions with different parameters or when using the pipeline as a plotting utility. I expect we will need a single implementation of such an entity.
- The job of the `visualizer` is to visualize any given scene. It can also communicate using a simple protocol with the controller for changes in (reconstruction) parameters. This is mostly because it can provide a convienient user interface for changing these parameters. We plan on having two visualizers, one based on a Slicer3D plugin for end-users, and one custom made one useful for experimenting and algorithm developers but not targeting end-users.

The reason for looking at (ortho)slice-based reconstruction is largely because of data management. To prevent botlenecks, we do not want to send large parts of data around, nor do we want to reconstruct large volumes. This is why the controller *never* gets to see the raw data, which lives only on the reconstruction node(s).

2.1 Packages

The communication between nodes happens in a using standardized *network packets* that contain data, commands, or parameters. For the precise content of specific packets see Section 6. We will describe the communication between the `reconstructor` and the `controller` first, roughly in chronological order, and ignoring possible sequencing issues at first.

- When a new data-set comes in, or a new scan is started, the reconstructor registers itself with the controller by sending a `MakeScene` packet. This contains:
  - a *name* for the scene
  - the *dimension of the data* (3D or 2D)
  - the *volume geometry* of the object.

  It gets a reply from the controller containing the `scene_id`, which it can use as a reference when sending packages in the future.
- Assuming the dimension is 3D, the controller sends the reconstructor one or multiple `SetSlice` packets. These packets contain:
  - a newly generated slice id
  - the *orientation* of the slice

- Upon receiving these packets the reconstructor starts reconstructing these slices. When data is available, it sends a `SliceData` packet to the controller containing:
  - the *slice identifier*
  - the (bitmap) data for the slice.
- When the user (or an algorithm) changes the slice it wants to see, an `UpdateSlice` packet is sent to the reconstructor. This contains:
  - the old slice id, which gets replaced
  - a newly generated slice id
  - the orientation of the new slice

These packages are enough to implement the initial goal, reconstruction of arbitrary initial slices, the orientation and position of which are controlled in real-time.

2.2 Orientation

We need a convention for representing the orientation of a slice. The orientation is inside *volume space* and is completely independent from the number of pixels (i.e. the 'size' of an individual pixel is implied by the bounding square of a slice). We represent the orientation as 9 real numbers $(a, b, \ldots, i)$ so that:

$$\begin{pmatrix} a & d & g \\ b & e & h \\ c & f & i \\ 0 & 0 & 1 \end{pmatrix} \begin{pmatrix} x_s \\ y_s \\ 1 \end{pmatrix} = \begin{pmatrix} x_w \\ y_w \\ z_w \\ 1 \end{pmatrix} \tag{1}$$

Where the vector $\vec{x}_s = (x_s, y_s)$ lives inside a slice (i.e. the *normalized* pixel coordinates of a slice, in the interval $[0, 1]$), and where $\vec{x}_w = (x_w, y_w, z_w)$ lives inside the volume geometry at the correct place. The pixel coordinates of a slice have the following convention:

$$\begin{pmatrix} (0,m) & \ldots & (n,m) \\ \vdots & \ddots & \vdots \\ (0,0) & \ldots & (n,0) \end{pmatrix} \tag{2}$$

i.e. we start counting from the bottom-left and use a standard cartesian xy convention.

Using this convention, the vector $\vec{b} = (g, h, i)$ is the base point of the slice in world space (i.e. the world coordinates of the bottom left point of a slice). $\vec{x} = (a, b, c)$ is the direction in world space corresponding to the x direction of the slice, and $\vec{y} = (d, e, f)$ corresponds to the y direction.

3 Extensions

Possible extensions for the future include:

- Low resolution 3d reconstruction that is shown in low opacity, or as a preview when changing the position or orientation of the slices.
- Changing the resolution at which the reconstructor reconstructs (down-sampling)

4 Implementation details

We currently plan to use *ZeroMQ* because of its flexibility with respect to programming languages and communication strategies, and its speed. An alternative could be *protocol buffers* which have been developed by Google.

5 Software examples

- Make a packet:

```
// C++
auto packet = MakeScenePacket("Test scene #1");

# Python
packet = MakeScenePacket("Test scene #1");
```

- Send a packet:

```
// C++
packet.send();

# python
packet.send();
```

- ...

6 Appendix: Packet definitions

Orientations are given as:

```
real[9] orientation = {a, b, c, d, e, f, g, h, i};
```

compare with the matrix given above.

```
// Usage: register a scene with the controller
// Reply: a `scene_id`
packet MakeScene:
    string name
    int dimension
    real[dimension] volume_geometry
    [[reply]] int scene_id

// Usage: set the slices to reconstruct,
// Note: published by server
packet SetSlice:
    int scene_id
    int slice_id
    real[9] orientation;

packet RemoveSlice:
    int scene_id
    int slice_id

// Usage: update the data for a slice
packet SliceData:
    int scene_id
    int slice_id
    int[2] slice_size
    unsigned char[pixels] data // pixels = product(slice_size)

// Usage: update the volume data
packet VolumeData:
    int scene_id
    int[3] volume_size
    unsigned char[voxels] data // voxels = product(volume_size)

// Usage: update the volume data
packet GeometrySpecification:
    int scene_id
    bool parallel
    int projections

// Usage: update the volume data
packet ProjectionData:
    int scene_id
    int projection_id
```

```
real[3] source_position
real[9] detector_orientation
int[2] detector_pixels
unsigned char[pixels] data // pixels = product(detector_pixels)
```