VILMA Data Structure Design

VRVis Rendering Research Area

Basic Design For Visualizing Linear Edifices

Document Version: 2009-07-21

Document Authors: Robert F. Tobler, Thomas Ortner

Contents

1 Introduction

Currently tunnels are scanned using fixed laser scanners, and a number of digital images are taken, that cover the area of the scans. The change for the future is the automatic orthoscan based on a moving platform.

Both types of scans result in the following data:

- a rectangular array of scan points which we call OPC (ordered point cloud)
- a number of images that overlap this scan area

2 Basic Data Structure

Each OPC is split into one or more patches, each patch containing all the data that is valid within a square section of fixed size (e.g. 256x256) of the OPC.

For each patch the following data is maintained:

- the array of 3D positions of OPC points
- the index array of (non-planar) quads that make up the OPC
- one or more parts of images that are valid for the patch geometry

Note that it is possible, that a patch only contains a sub-set of all possible (256x256) quads, this can be due to invalid OPC data, or due to its location on the edge of a OPC.

For each image part the following data is maintained:

- the actual image part data (RGBA texture)
- the texture coordinates for mapping the image part onto the patch
- a validity texture, that encodes valid and invalid regions of the image part
- patch coordinates for the image part (i.e. the whole patch defines a [0...1], [0...1] patch coordinate system), so that the validity texture is defined in the patch coordinate system

3 Hierarchical Data Organization

In order to merge OPCs in areas of overlapping OPC data, a so-called virtual scan can be performed: the overlapping scans are each intersected with a bundle of rays that simulate a virtual scanner. In regions where more than one scan has been obtained, the virtual scan computes a scan value that interpolates all intersected original scans.

All original and virtual scans are split into patches, whereby each time maintains a link to its original scan. This comprises the finest detail level that can be displayed. In order to provide additional levels of detail four neighboring patches are merged and subsampled by a factor of two in order to compute new, coarser patches. Each patch maintains links to the four more detailed patches it was computed from.

4 Data Structure Specification

4.1 Tunnel

A tunnel contains a number of OPC and a number of (optional) OPC-groups.

An OPC represents the data resulting from an actual or a virtual laser OPC, together with the images that are used to texture the OPC geometry.

An OPC-group groups a number of OPCs into a logical unit, so that they can be simply selected for viewing. A OPC can be part of multiple OPC -groups.

The logical structure of a tunnel is defined using a directory. The name of the directory is the tunnel name, it contains a number of sub-directories representing the OPCs within the tunnel and a number of xml-files defining the OPC -groups within the tunnel.

The directory structure of a tunnel:

```
name-of-tunnel/
    opcs/
        guid-of-opc-1/
        guid-of-opc-2/
          . . .
        guid-of-opc-N/
        name-of-opc-group-1.xml
        name-of-opc-group-2.xml
          . . .
        name-of-opc-group-N.xml
    images/
        guid-of-image-1/
        guid-of-image-2/
          . . .
        guid-of-image-N/
```

An OPC XML file has the following form (note that the version attribute for the Aardvark and OpcGroup Elements is optional, if it is not specified, version=”0” is assumed):

```
<?xml version="1.0" encoding="utf-8"?>
<Aardvark version="0">
  <OpcGroup version="0">
    <TagList>
      <Tag>text-of-tag-1</Tag>
      <Tag>text-of-tag-2</Tag>
       ...
      <Tag>text-of-tag-N</Tag>
    </TagList>
    <EpochVisibility>
          visibility-enumeration (as specified below)
    </EpochVisibility>
    <List>
      <string>guid-of-opc-1</string>
      <string>guid-of-opc-2</string>
       . . .
      <string>guid-of-opc-N</string>
    </List>
  </OpcGroup>
</Aardvark>
```

4.1.1 EpochVisibility

The `EpochVisibility` tag describes the different time dependent behaviors OPCs can have according to the given time interval. Following modes are available:

- **`never`**
  the specified time interval is ignored and OPC is never shown
- **`always`**
  specified time interval is ignored and OPC is always shown
- **`older`**
  all OPCs within the specified interval are shown
- **`last`**
  only the latest OPCs within the specified interval are shown
- **`younger`**
  only OPCs younger than the specified interval are shown

Note: EpochVisibility only makes sense on a group of OPCs or Data in general. Specifying it for each Patch might give unclear meaning to the `'last'` value.

4.2 OPC - Patchhierarchy

An OPC consists of a number of **patches** and a number of **images**

Each **patch** contains the data that is valid within a square section of the OPC.

Each **image** contains the image pyramid for a specific source image.

The logical structure of a OPC is defined using a directory. The name of the directory is the OPC name, it contains a directory named “patches” that contains all patches as subdirectories, and a directory called “images” that contains all images as subdirectories.

Thus the directory structure of a OPC looks like this:

```
guid-of-opc/
    patches/
        patchhierarchy.xml
        guid-of-patch-1/
        guid-of-patch-2/
          . . .
        guid-of-patch-N/
```

The file **`patchhierarchy.xml`** defines the hierarchy of patches by specifying the sub patches for each patch in the patches directory. Here is the structure of the **`patchhierarchy.xml`** file:

```
<?xml version="1.0" encoding="utf-8"?>
<Aardvark version="0">
  <PatchHierarchy version="0">
    <AcquisitionDate>
         time-stamp (as specified below)
    </AcquisitionDate>
    <TagList>
      <Tag>text-of-tag-1</Tag>
      <Tag>text-of-tag-2</Tag>
       ...
      <Tag>text-of-tag-N</Tag>
    </TagList>
    <RootPatch>name-of-root-patch</RootPatch>
    <SubPatchMap>
      <item>
        <key>guid-of-patch-1</key>
        <val>
          <sw>guid-of-south-west-sub-patch-of-patch-1</sw>
          <se>guid-of-south-east-sub-patch-of-patch-1</se>
          <nw>guid-of-north-west-sub-patch-of-patch-1</nw>
          <ne>guid-of-north-east-sub-patch-of-patch-1</ne>
        </val>
      </item>
      <item>
        <key>guid-of-patch-2</key>
        <val>
          <sw>guid-of-south-west-sub-patch-of-patch-2</sw>
          <se>guid-of-south-east-sub-patch-of-patch-2</se>
          <nw>guid-of-north-west-sub-patch-of-patch-2</nw>
          <ne>guid-of-north-east-sub-patch-of-patch-2</ne>
        </val>
      </item>
       . . .
    </SubPatchMap>
  </PatchHierarchy>
</Aardvark>
```

Note: not all of the four sub-patches of a patch need to be present.

4.2.1 TimeStamp

To parse a TimeStamp string correctly it must match a specific pattern.

YYYY-MM-DDTHH:mm:ss.fffffffff

2009-07-22T12:27:01.133713371

Y...	Year
M...	Month
D...	Day
T...	separation character 'T'
h...	Hour
m...	Minute
s...	Second
f...	Fraction of a second

Note: Since DateTime.Parse(string timeStamp) is used every other valid TimeStamp format can be used (i.e. a timestamp without 'T')

4.3 Patch

A patch contains the geometry within a square subsection of the OPC. A single file called **`patch.xml`** defines the structure of the patch, and may reference an arbitrary number of files with extensions **`.aara`** , **`.tiff`**, or **`.jpg`**, that contain the actual data. The referenced files define 1-, 2-, or 3-dimensional arrays of vectors or colors.

```
guid-of-patch/
    patch.xml
    guid-of-index-array-file.aara
    guid-of-positions-array-file.aara
    guid-of-color-array-file.aara
    guid-of-texture-1-weights.jpg
    guid-of-texture-2-weights.tiff
```

The file **`patch.xml`** has the following structure.

```
<?xml version="1.0" encoding="utf-8"?>
<Aardvark version="0">
  <Patch version="0">
    <GeometryType>Type</GeometryType>
```

```
<TagList>
  <Tag>text-of-tag-1</Tag>
  <Tag>text-of-tag-2</Tag>
   ...
  <Tag>text-of-tag-N</Tag>
</TagList>
<Local2Global>
    [[ 1.0, 0.0, 0.0, 396.0 ], [ 0.0, 1.0, 0.0, 400.0 ],
     [ 0.0, 0.0, 1.0, 112.0 ], [ 0.0, 0.0, 0.0, 1.0 ]]
</Local2Global>
<GlobalBoundingBox>
   [[ xmin, ymin, zmin ], [ xmax, ymax, zmax ]]
</GlobalBoundingBox>
<Indices>guid-of-index-array-array-file.aara</Indices>
<Positions>guid-of-positions-array-file.aara</Positions>
<Colors>guid-of-colors-array-file.aara</Positions>
<Normals>guid-of-colors-array-file.aara</Normals>
<Coordinates> // aara
  <PatchCoordinates>guid-of-patch-coords-file.aara
                    </PatchCoordinates>
  <DiffuseColor1Coordinates>guid-of-coords-1-array-file.aara
                            </DiffuseColor1Coordinates>
  <DiffuseColor2Coordinates>guid-of-coords-2-array-file.aara
                            </DiffuseColor2Coordinates>
  <DiffuseColor3Coordinates>guid-of-coords-3-array-file.aara
                            </DiffuseColor3Coordinates>
</Coordinates>
<Textures>
  <!-- weights 0-255, everything as images
       tiff 32 bit rgba, tiff 8bit gray, oder png 32 bit rgba,

       8 bit gray, 16 bit quantized normales
  -->
  <DiffuseColor1Texture>guid-of-image-1/tile.jpg
                        </DiffuseColor1Texture>
  <DiffuseColor1Weights>guid-of-texture-1-weights.aara
                        </DiffuseColor1Weights>
  <DiffuseColor2Texture>guid-of-image-2/tile.png
                        </DiffuseColor2Texture>
  <DiffuseColor2Weights>guid-of-texture-2-weights.aara
                        </DiffuseColor2Weights>
  <DiffuseColor3Texture>guid-of-image-3/tile.tiff
                        </DiffuseColor3Texture>
  <DiffuseColor3Weights>guid-of-texture-3-weights.aara
                        </DiffuseColor3Weights>
</Textures>
<Cameras>
```

```
      <DiffuseColor1Camera>
        [[ 1.0, 0.0, 0.0, 0.0 ], [ 0.0, 1.0, 0.0, 0.0 ],
         [ 0.0, 0.0, 1.0, 0.0 ], [ 0.0, 0.0, 0.0, 1.0 ]]
      </DiffuseColor1Camera>
      <DiffuseColor2Camera>
        [[ 1.0, 0.0, 0.0, 0.0 ], [ 0.0, 1.0, 0.0, 0.0 ],
         [ 0.0, 0.0, 1.0, 0.0 ], [ 0.0, 0.0, 0.0, 1.0 ]]
      </DiffuseColor2Camera>
    </Cameras>
  </Patch>
</Aardvark>
```

The Xml Elements within the patch file define the various parts of a piece of geometry. Some of these Elements are required, a number of them are optional. Here is a breakdown of the elements, their use and the necessary requirements they have to fulfill:

- **Aardvark, Patch**: the version attribute is optional, if it is not specified version=”0” is assumed
- **GeometryType**: one of the following strings: **PointList**, **LineList**, **TriangleList, QuadList**. If it is not specified, **TriangleList** is assumed.
- **Positions**: if the Positions file contains a 2D-array of points, this implicitly defines a grid of quads. If the Positions file contains a 1D-array of points, the Indices array must be specified.
- **Normals** (optional): a normal vector array of type V3f or V3d (1D or 2D, exactly as the positions array) that specifies the normal vector at each position.
- **Offsets** (optional): an offset value area of type float or double that specifies an additional offset point at each position which is calculated by adding the normal vector times the offset value to the original position.
- **Indices** (optional): a 1D-array of indices, which is only useful if GeometryType = LineList ,TriangleList or Quadlist. In case of a LineList, each 2 indices specify the endpoints of a line, in case of a TriangleList, each 3 indices, specify the vertices of a triangle, in case of QuadList, each 4 indices, specify the vertices of a quad. If the Positions array is two dimensional this is not needed, since the indexing is implicit.
- **Colors**: an optional 1D-array of type C3b, C3f, C4b, or C4f that specifies the color at each position.
- **Local2Global**: a single 4x4 matrix that specifies the origin and transformation of all the data of this patch with respect to the global coordinate system. In order to compute the global position of all vertices of the patch, its local coordinates (the ones stored in the Positions array) can be multiplied with this matrix.

- **GlobalBoundingBox**: the axis-aligned bounding box of the geometry of this patch in the global coordinate system. This can be used to decide which data to load before actually loading the data arrays. This is optional, however if it is not available, initial decisions on what to display can take significantly longer.
- **Textures**, **Coordinates**, **Cameras**: these three structures specify all valid textures for the patch. Matching field names are used to define the association between texture, texture-coordinates, and texture projection camera:
  - **DiffuseColor*N*Texture**: the field name of texture *N* in the Textures structure
  - **DiffuseColo**r***N*Weights**: the field name of a weights texture in the Textures structure that defines the weights of texture *N* across the patch (optional, if omitted weight=1 is assumed for all pixels that are covered by the texture)
  - **DiffuseColor*N*Coordinates** (optional): the field name of the coordinates array for texture *N* in the Coordinates structure. This is only necessary if there is no camera specified for the corresponding texture.
  - **DiffuseColor*N*Camera** (optional): the field name of the projection camera for texture *N* in the Cameras structure. This is only necessary if no Coordinates are specified.
  - additional field names for various other uses can be defined as necessary
- **Coordinates**: contains the coordinates arrays for all textures, and an additional patch coordinates array.
  - **PatchCoordinates** (optional): the key of the coordinates array that defines the patch coordinates within u = [0, 1], v = [0, 1] for each vertex of the patch. If this is not present, it can be automatically generated, if the Positions-Array is a 2-D array.
- **Textures**: for small textures that are only used by one tile, the image file can directly reside within the patch directory. If it is used by more than one patch, the image file needs to reside in the image subdirectory of the opc.
  If the image texture is larger than a predefined threshold (see section **Image** below), a specific image tile within an image pyramid is referenced, by a two part image name: the image name specifying the image pyramid and the tile name within this pyramid separated by a slash. Note that the associated projection camera should be specified for the exact tile that is used, and must therefore define an off-axis projection in this case.
- **Cameras**: contains one camera projection matrix for each texture that is projected onto the quad mesh. Each matrix is defined like the DirextX 9 camera projection matrix, only we use the transposed version (i.e. rows and columns are swapped with respect to DirectX 9). For a single texture either a camera or a coordinate array must be specified, but not both [future extension, currently Coordinates are preferred].

4.3.1 Camera Specification

`CameraMatrix` consists of a 3x3 rotation matrix and a 3 component translation vector. The last row expands the matrix to a homogenous 4x4 matrix. Every component is of type **Double.**

$$\begin{bmatrix} \boldsymbol{RightX} & \boldsymbol{RightY} & \boldsymbol{RightZ} & \boldsymbol{Tx} \\ \boldsymbol{UpX} & \boldsymbol{UpY} & \boldsymbol{UpZ} & \boldsymbol{Ty} \\ \boldsymbol{LookAtX} & \boldsymbol{LookAtY} & \boldsymbol{LookAtZ} & \boldsymbol{Tz} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & \mathbf{1} \end{bmatrix}$$

Right, Up and **LookAt** define the camera's orientation, whereas **T** defines its position in world space. The three orientation vectors describe a coordinate system their own referred to as camera space.

```
<CameraMatrix>
      [[ R11, R12, R13, Tx ],
       [ R21, R22, R23, Ty ],
       [ R31, R32, R33, Tz ],
       [   0,   0,   0,  1 ]]
</CameraMatrix>
```

4.3.2 Offsets

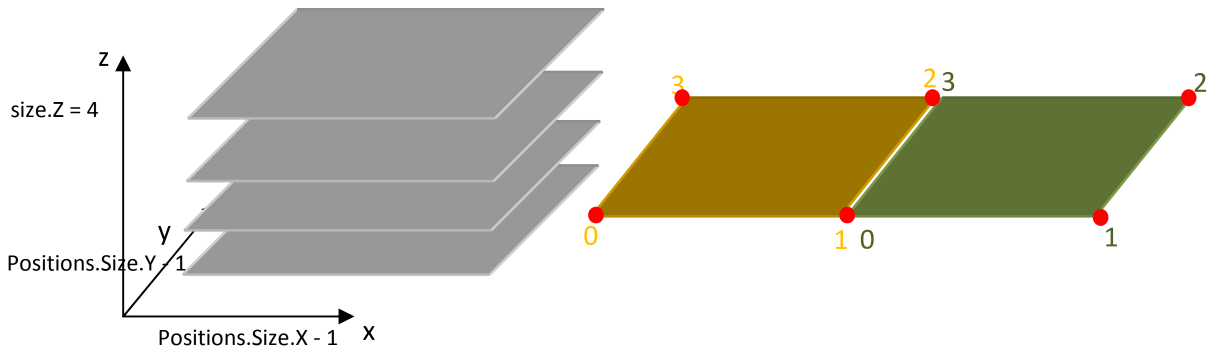
Offsets are scalar values that are used to align faces along the normal directions of their vertices. Consequently one vertex can have 1 to 4 offset values associated with it. The Offset data array may be specified as described as a 3-dimensional .aara data array.

```
Offset.Size.X = Positions.Size.X - 1;

Offset.Size.Y = Positions.Size.Y - 1;

Offset.Size.Z = 4;
```

Offset.Size.XY is smaller than the position array by one since the information is per face.
Size.Z indicates that 4 layers exist over the vertex data specifying 1 value for each corner of a face in a counter clockwise fashion.



4.4 Image

If an image is larger than 2^k (k could be defined as 10 or 11 depending on the available hardware) in one of its dimensions it is not directly located within the patch directory, but is stored as an image pyramid in the **images**-subdirectory of the opc directory. An image pyramid consists of a number of tiles each of which have a maximal size of 2^k in each dimension.

The directory structure of such an image is the following:

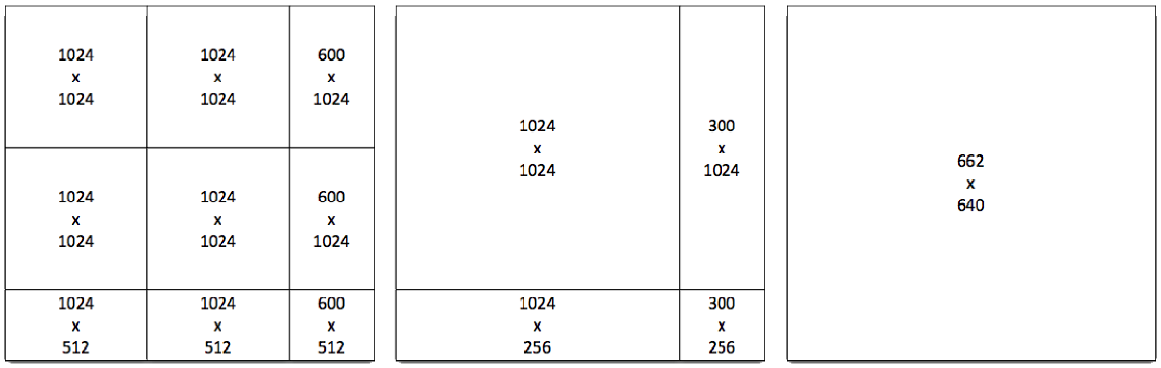
```
guid-of-image/
    imagepyramid.xml
    guid-of-image-tile-1.jpg
    guid-of-image-tile-2.jpg
    . . .
    guid-of-image-tile-N.jpg
    guid-of-mask-of-image-tile-for-patch-1.tiff
    guid-of-mask-of-image-tile-for-patch-2.jpg
    . . .
```

The actual hierarchy of tiles is defined in the **imagepyramid.xml** file.

```
<?xml version="1.0" encoding="utf-8"?>
<Aardvark version="0">
  <ImagePyramid version="0">
    <TagList>
      <Tag>text-of-tag-1</Tag>
      <Tag>text-of-tag-2</Tag>
       ...
      <Tag>text-of-tag-N</Tag>
    </TagList>
    <RootTile>guid-of-root-tile</RootTile>
    <SubTileMap>
      <item>
        <key>guid-of-tile-1</key>
        <val>
          <sw>guid-of-south-west-sub-tile-of-tile-1</sw>
          <se>guid-of-south-east-sub-tile-of-tile-1</se>
          <nw>guid-of-north-west-sub-tile-of-tile-1</nw>
          <ne>guid-of-north-east-sub-tile-of-tile-1</ne>
        </val>
      </item>
      <item>
        <key>guid-of-tile-2</key>
        <val>
          <sw>guid-of-south-west-sub-tile-of-tile-2</sw>
          <se>guid-of-south-east-sub-tile-of-tile-2</se>
          <nw>guid-of-north-west-sub-tile-of-tile-2</nw>
          <ne>guid-of-north-east-sub-tile-of-tile-2</ne>
        </val>
      </item>
       . . .
    </SubTileMap>
    <PatchMaskMap>
     <item>
        <key>guid-of-tile-1</key>
        <val>
          <item>
            <key>guid-of-patch-1</key>
            <val>guid-of-mask-of-image-tile-for-patch-1.tiff</sw>
          </item>
          <item>
            <key>guid-of-patch-2</key>
            <val>guid-of-mask-of-image-tile-for-patch-2.jpg</sw>
          </item>
        </val>
      </item>
    </PatchMaskMap>
</ImagePyramid>
</Aardvark>
```

Note, that not all of the four sub-patches of a patch need to be present.

Here is an example of the tiles in a 3-level image pyramid with k=10, and thus the max pixel size 1024. Note, that tiles that share a common border need to have the exact same pixel values for that shared border (i.e. the border column/row is stored in both tiles). As a consequence, **the scaling between two levels of the image pyramid is not exactly a factor of two**, since the overall width/height of two neighboring tiles is less due to the repeated pixels.



4.5 Aardvark Array File (extension: .aara)

The Aardvark array file consists of 4 parts, stored in succession: the **Type**, the **Dimension**, the **Size**, and the **Data**. All primitive data types are stored in little-endian (Intel) format.

Type: a string specifying the data type, the string is stored by the string length as byte followed by the characters of the typename in UTF-8. Possible types are: **int**, **long**, **float**, **double**, **V2f**, **V3f**, **V2d**, **V3d**, **C3b**, **C3f**, **C4b**, **C4f**. These data types are exactly defined as in C#.

Dimension: the dimension of the data array, stored as a single byte. Possible values: **1**, **2**, **3**.

Size: the size of the data array, stored as 32-bit integer, separate in each dimension. As an example, if Dimension = 3, the size consists of 3 integers. The sizes are stored in the order as they appear in a C# declaration, e.g. for a V3f[10,20,30], the sizes 10, 20, 30 are stored in that order.

Data: the data elements (**int**s, **long**s, **double**s, **double**s, **V2f**s, **V3f**s, **V2d**s, **V3d**s, **C3b**s, **C3f**s, **C4b**s, **C4f**s) stored in binary format (IEEE for floats and doubles). The storage order of multiple dimensions is the same as in the C# language: in the example of a V3f[2,3,4], 2 planes of 3 lines of 4 V3fs are stored in the file without any separators. The graphic data types are defined as follows:

> **V2f**: a 2D vector or point of two floats (x, y)
>
> **V3f**: a 3D vector or point of three floats (x, y, z)
>
> **V2d**: a 2D vector or point of two doubles (x, y)
>
> **V3d**: a 3D vector or point of three doubles (x, y, z)
>
> **C3b**: a color of 3 bytes (r, g, b)
>
> **C3f**: a color of 3 floats (r, g, b)
>
> **C4b**: a color of 4 bytes (r, g, b, a)
>
> **C4f**: a color of 4 floats (r, g, b, a)

A short example code for writing an **.aara** file in C#:

```
public static void SimpleAaraExample()
{
    var a3 = new V3f[2, 3, 4];
    for (int k = 0; k < 2; k++)
        for (int j = 0; j < 3; j++)
            for (int i = 0; i < 4; i++)
                a3[k, j, i] = new V3f(i, j, k);

    var dir = WorkDir.FindDir("Scratch");
    var fileName = "TestV3f.aara";
    var file = File.Open(Path.Combine(dir, fileName),
                           FileMode.Create);

    using (var writer = new BinaryWriter(file, Encoding.UTF8))
    {
        string typeName = "V3f";
        byte dimensions = (byte)3;
        int[] size = { 2, 3, 4 };
        writer.Write(typeName);
               // Implicity writes the length
               // of the string as byte before
               // the actual string characters as
               // bytes
               // THERE IS NO 0-BYTE AT THE END!

        writer.Write(dimensions);
        writer.Write(size[0]);
        writer.Write(size[1]);
        writer.Write(size[2]);

        for (int k = 0; k < size[0]; k++)
            for (int j = 0; j < size[1]; j++)
                for (int i = 0; i < size[2]; i++)
                {
                    writer.Write(a3[k, j, i].X);
                    writer.Write(a3[k, j, i].Y);
                    writer.Write(a3[k, j, i].Z);
                }

         writer.Flush();
    }
    file.Dispose();
}
```

5 Rendering

A hierarchy of patches represents all levels of detail for a particular real or virtual opc, the renderer loads the patches from coarsest to finest and displays the most detailed data (limited by the screen resolution) that is available at any one time. When quick movements in the geometry are performed, the patches of the detail level that are closest in resolution to the screen are loaded as needed. For perspective rendering, these patches might be of different absolute resolution.

6 Borders Between Arbitrary OPCs

In order to provide clean transitions between arbitrary OPCs, it is possible to introduce clipping planes, and split all the geometry of a OPC (all the patches) at a clipping plane. This clipping has to be performed in advance, and by clipping different OPCs on the same clipping plane, and then performing post processing (i.e. merging) of the intersection line, a smooth transition of the two OPCs can be pre-computed.

7 Bandwidth Estimate

In order to maintain an interactive desktop experience, we estimate that the data for at least 1024 by a 1024 image pixels have to be loaded per second. If we assume that the geometric data is available at a coarser resolution of 256x256, we can make the following estimate for the data that needs to be loaded each second.

Geometry:

- 256x256 geometric points, 3 * 4 bytes each, for a total of about 0.75 MByte
- 255x255 quads , 6 * 4 bytes indices each, for a total of about 1.5 MByte

Image part:

- 1024x1024 image texture, 4 bytes each, for a total of about 4 MByte
- 256x256 texture coordinates, 2 * 4 bytes each, for a total of about 0.5 MByte
- 255x255 validity texture (per quad), 1 byte each, for a total of about 0.0625 MByte
- 256x256 validity texture coordinates, 2 * 4 bytes each for a total of about 0.5 MByte

Total patch data: 1x Geometry 2x Image part (estimate):

- 2.25 MByte Geometry + 2 * 5.0625 MByte Image parts = 12.375 MByte

Based on current hard-disk speeds, 3 to 6 patches of such a size can be loaded per second, so even current hardware should be able to handle the bandwidth necessary for interactive movement through the tunnel data.

END OF DOCUMENT