INF219 Technical Report: Conversion of Geological Channels from the Eclipse File Format to the Wavefront File Format

Andreas J. Lind[1]
[1]Institute of Informatics University of Bergen

January 9, 2012

Abstract

When simulating reservoir data in the oil industry the eclipse format is often used to store both the results of and the model for the simulation. This format however is not very well suited for visualization, because among other features it includes redundant information and stores the data in non-regular grids. This has implications with regards to the time and space required to simplify the polygonal representation of each channel (reduce number of faces and vertices) and remove artifacts such as overlapping faces. To solve this problem we have developed a method which simplifies the polygonal representation of the eclipse-data and stores the result as wavefront objects. The theoretical complexity of this algorithm has been found and the real-world difference in running time, storage usage and result has been compared to a simple conversion which does not simplify and does not remove artifacts. The results were good, with up to larger than 90% removal of faces and vertices. As well as a removal of most visual artifacts. We also discuss possible future applications of this work as well as possible improvements and challenges for the method.

I.3.3Computer GraphicsApplications, Miscellaneous

1 Introduction

The goal of oil and gas reservoir simulation is to simulate the exploitation of real world oil and gas reservoirs, without incurring the costs of testing with real world reservoirs. This method of gaining understanding about oil and gas reservoirs and their use is considered to be one of the most important tools for making decisions regarding the development and operations of oil and gas extraction endeavors[Pet11]. One of the most used such simulation tools is the Eclipse system by Geoquest/Schlumberger. Eclipse stores the input for these simulations on the eclipse data-format. This input data contains among other information, data about channels to be simulated. Channels are locations of former rivers which have become subterranean (underground). These areas i.e. rivers and river-deltas often contain large quantities of oil and gas which has seeped in from shales below. The channel data used by eclipse is gained through the use of stochastic methods in geomodelling tools such as IRAP RMS by geologists.

The channels are stored on a format containing a non-regular grid called a corner-point grid and also contains redundant information[Pet11]. This has the effect of making the polygonal representation difficult to simplify and it the files much larger than simple visualization methods require. Other file-formats could therefore be used to improve the performance of the visualization. One such file-format is the wavefront object obj file-format, which will be used in this work. The obj file format store the polygonal representation of the data directly. For direct channel visualization this has advantages with regards to space consumption and preprocessing. Since the channels are represented by parallel-piped cubes which can be

represented directly with triangle- or quad-polygons unlike for instance cylinders, which would have to be approximated through sampling.

The purpose of this work will be to create a method which produces a wavefront object version of the eclipse simulation model. This wavefront object version should remove artifacts in the visualization such as those generated by overlapping faces and reduce the number of vertices and faces used as much as possible. This method will then be compared to a naive conversion of the eclipse data. The naive conversion does no simplification and remove no artifacts. The success of the simplifying conversion can then be measured by the relative reduction of the number of vertices and faces, and the visible removal of artifacts in the conversion, as compared to the naive conversion. We conclude this report by discussing the possible applications of this work and possible future work based on this method.

2 Eclipse Out-Data

A corner-point grid is defined by a three dimensional coordinate system with each point being defined by a position $\begin{bmatrix} u \\ v \\ w \end{bmatrix}$. Where u and v are defined according to a Cartesian grid, but w can give different height values based on the corresponding $\begin{bmatrix} u \\ v \end{bmatrix}$ position. The position of each corner for each cell in the grid is given by grid-lines originating in the points $\begin{bmatrix} u' \\ v' \\ 0 \end{bmatrix}$ and being sampled along the grids w'-axis.

From this a representation of the input the model and the result of a simulation can be described by defining the grid in two matrices. The matrix (x, y, z) represent which channel if any the corresponding cell is part of, and the (x', y', z')-matrix represent the position of each of the corners of the cells from the (x, y, z)-matrix. Where $|x'| = 2 \times |x|$, $|y'| = 2 \times |y|$ and $|z'| = 2 \times |z|$ because every cell has 4 corners. The following show sections how this information is stored, in the eclipse file format is explained in greater detail.

2.1 SPECGRID, Channel and ACTNUM

The tags "SPECGRID", "Channel" and "ACTNUM" are used to define the (x, y, z)-matrix, and are defined as follows. The SPEGCRID tag is followed by a line that describes the "X Y Z"-size of the grid, defining how many cells are in the simulation. The channel tag is the name of the channels system being described in the file. This tag is followed by a list of integers of $|X| \times |Y| \times |Z|$ length, which defines which channel number each cell in the grid belongs to. The "ACTNUM" tag is also defined as a list with one entry for each cell. Each entry is either given as 0 or 1 and describes whether the cell is active and therefore whether it should be part of the simulation. Therefore we have decided that only the should be active cells should be rendered in our visualization.

2.2 COORD and ZCORN

The (x', y', z')-matrix has been decomposed into the COORD- and ZCORN-tags. Each line after the COORD tag are on the form "$u\ v\ w\ u'\ v'\ w'$", where $\begin{bmatrix} u \\ v \\ w \end{bmatrix}$ defines the starting point of the grid-line and $\begin{bmatrix} u' \\ v' \\ w' \end{bmatrix}$ defines the end point of the grid-line. The ZCORN-tag defines the height for every corner-point for every cell in the grid. It does so by first defining the heights for every cell along the x-axis the top north-facing corner-points. This is done defining the western corner first, then the eastern corner. The same is then done for the south facing corner-points. The process North-South and West-East for the top of the cell on the x-axis is then repeated along the entire y-axis. When this is done the same thing is done for the bottom and this continues along z-axis until the z-values of all the corner-points are defined. This method of storage is perhaps better illustrated by these cells in

Fig. 1 taken from ResSimNotes[Pet11]. This corresponds to the corner-point notation shown in Fig. 2 with, T,NW = C1, T,NE = C2, T,SW = C3, T,SE = C4, B,NW = C5, B,NE = C6, B,SW = C7 and B,SE = C8.

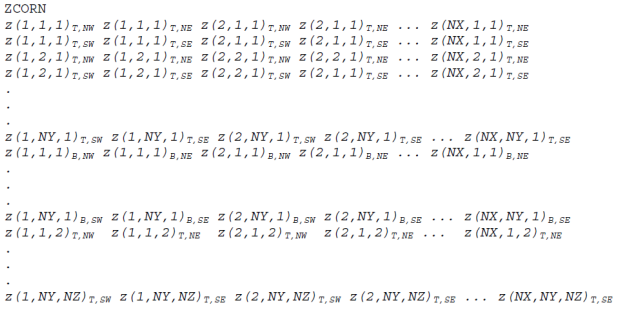

Figure 1: An explanatory figure of ZCORN from the ResSimNotes text[Pet11]. Each line contain the two x-axis neighbors in the cell (see Fig. 2). It goes through the x-axis first then iterates along the y-axis then lastly the z-axis.

3 Naive Conversion

Out of the two conversion methods discussed in this work naive conversion is definitely the simplest one. We iterate through each cell in (x, y, z). If a cell is part of a channel c then simply take the corresponding corner-points from (x', y', z'), define each side of cube as two triangles and then add it to c's wavefront object. An pseudo-code version of this process can be found in Alg. 1. Since the algorithm simply has to run through the matrices it has a running time bounded by $O(x \times y \times z)$. The naive conversion does however result in a large number of vertices and faces being generated. Also there may be overlap between different polygons in the wavefront object model, which corresponds to an overlap between faces in the simulation model. This overlap often results in rendering artifacts as seen in Fig. 3.

Algorithm 1 Naive_Conversion(eclipse_data d)

Load_Data is assumed to be known from Sec. 2.
($chan\_matrix$, $corner\_point\_matrix$, num_chans) ← Load_Data(d)
$wavefronts \leftarrow []$
for $channel$ in num_chans **do**
  $wavefronts$.append([])
end for
Create_Cube returns a list of triangles defining the cube in position x,y,z. See Alg. 2.
for x, y, $z <$ $chan\_matrix$.range() **do**
  $c \leftarrow chan\_matrix[x][y][z]$
  if $c > -1$ **then**
    $wavefronts[c]$.append(Create_Cube($corner\_point\_matrix$, x,y,z)
  end if
end for
Calculates the normal of triangle and writes each channel as a wavefront. How to do this is assumed to be already known by the reader.
for $c <$ num_chans **do**
  Write_Obj($wavefronts[c]$, c)
end for

Algorithm 2 Create_Cube(3D_Matrix d, x, y, z)

Find the cornerpoints. Each points location is shown in Fig. 2.
$c1 \leftarrow d[x \times 2][y \times 2][z \times 2]$
$c2 \leftarrow d[x \times 2 + 1][y \times 2][z \times 2]$
$c3 \leftarrow d[x \times 2][y \times 2 + 1][z \times 2]$
$c4 \leftarrow d[x \times 2 + 1][y \times 2 + 1][z \times 2]$
$c5 \leftarrow d[x \times 2][y \times 2][z \times 2 + 1]$
$c6 \leftarrow d[x \times 2 + 1][y \times 2][z \times 2 + 1]$
$c7 \leftarrow d[x \times 2][y \times 2 + 1][z \times 2 + 1]$
$c8 \leftarrow d[x \times 2 + 1][y \times 2 + 1][z \times 2 + 1]$
Create the polygons.
$ret \leftarrow []$
ret.append($c1$,$c3$,$c5$)
ret.append($c3$,$c5$,$c7$)
ret.append($c1$,$c3$,$c2$)
ret.append($c3$,$c2$,$c4$)
ret.append($c2$,$c4$,$c6$)
ret.append($c4$,$c6$,$c8$)
ret.append($c5$,$c7$,$c6$)
ret.append($c7$,$c6$,$c8$)
ret.append($c1$,$c2$,$c5$)
ret.append($c2$,$c5$,$c6$)
ret.append($c3$,$c4$,$c7$)
ret.append($c4$,$c7$,$c8$)
return ret

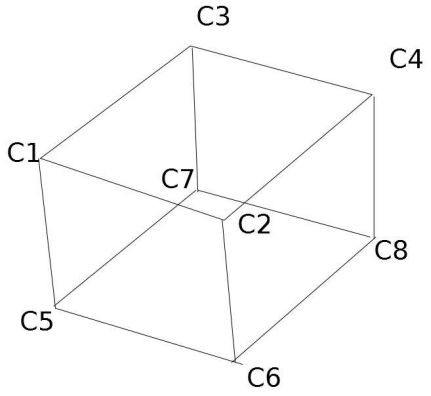

Figure 2: A cell in the corner-point grid seen from the front. The marks c1 through c8 denotes the different corner-points on the cube.

4 Simplifying Conversion

Due to problems with artifacts and the size of the resulting polygonal representation from the naive conversion a more sophisticated conversion method is needed. This method must remove overlapping faces and reduce the number of vertices and polygons needed to represent the model. Redundant data can be removed by looking at a bounding box for the cells connected through the z-axis (column-wise). Since each cell is shaped as a cube and are defined along the grid-lines described in Sec. 2, each connected component along the z-axis can be stored as a bounding box without any loss of data. The removal of overlapping faces can be accomplished by processing the data in such a way that the neighbors for each cell are found first. The faces that overlap with any neighboring bounding box are then removed or

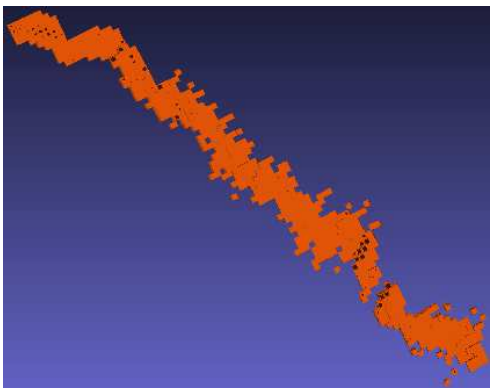

Figure 3: The naive conversion takes each cube in the corner-point grid, which is part of the channel and writes it directly to the wavefront object without simplifying or removing overlapping faces. As can be seen by the black spots in the channel, this results in artifacts being introduced into the rendering.

resized. We describe out algorithm for accomplishing this in the following sections.

4.1 The algorithm

The main algorithm can be found in Alg. 3. First the eclipse grid data is loaded into the variables *chan_matrix*, *corner_point_matrix* and *num_chans*. Where *chan_matrix* corresponds to the $(x,\ y,\ z)$-matrix from Sec. 2 and *corner_point_matrix* to the $(x',\ y',\ z')$-matrix from the same sections. *num_chans* is the number of channels the grid is divided into. Lists are then assigned to hold the bounding boxes for every channel in the *channels* variable. Each cell is then extracted from *chan_matrix* and *corner_point_matrix* and stored in the three dimensional array *cells*. The bounding boxes are constructed by iterating through *cells* and their neighbors found by searching through each bounding box's 4 neighboring parallel pipes for overlap. The algorithm then goes through every neighbor of every box removing and resizing the overlapping faces. Lastly the modified bounding boxes are written to the wavefront object-files. The result of this process can be seen in Fig. 4, which demonstrates the removal of the visual artifacts from the naive conversion in Fig. 3.

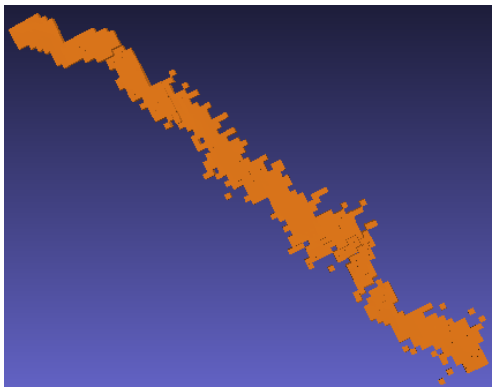

Figure 4: The simplifying conversion converts each group of cubes in a channel connected by the z-axis into a single bounding box. The neighbors in the x-y direction of this bounding box are then found and the overlapping areas of the bounding boxes are removed. This results in the overlapping faces artifacts from Fig. 3 being removed and a significant drop in the number of faces and vertices. Note: There are still some black dots present in this figure. However these are due to aliasing which is a consequence of the rendering technique, not overlapping faces.

4.2 The Cell Class

The Cell class described in Alg. 4, is used to store the information about each individual cell. This information is taken primarily from *chan_matrix* and *corner_point_matrix*. The private variable *chan* corresponds to the channel number from *chan_matrix*. c1 through c8 contains the cells corner-points from *corner_point_matrix*, with the variables are ordered as shown in Fig. 2. The cell also contains a *visited* boolean used when creating the bounding boxes as seen in Alg. 3 and Alg. 5.

4.3 Creating Bounding Boxes

A bounding box is a non-regular cube which consists of six sides, with each side defined by four points, as

Algorithm 3 Simplifying_Conversion(eclipse_data *d*)

Load_Data is assumed to be known from Sec. 2.
(*chan_matrix*, *corner_point_matrix*, *num_chans*) ← Load_Data(*d*)
channels ← []
for $c = 0$, $c <$ *num_chans*, $c++$ **do**
  channels.append([])
end for
x_size ← *chan_matrix*.range()[0]
y_size ← *chan_matrix*.range()[1]
z_size ← *chan_matrix*.range()[2]
cells ← Cell[*x_size*][*y_size*][*z_size*]
for *x*, *y*, *z* < *chan_matrix*.range() **do**
  The Cell class is defined in Alg. 4.
  cells[*x*][*y*][*z*] ← Cell(*chan_matrix*, *corner_point_matrix*, *x*, *y*, *z*)
end for
bounding_boxes ← []
bb_neighbors ← []
for *x*, *y*, *z* < *chan_matrix*.range() **do**
  c ← *chan_matrix*[*x*][*y*][*z*]
  if $c > -1$ and $\neg$*cells*[*x*][*y*][*z*].*visited* **then**
    See Alg. 5 for Create_Box.
    (*bounding_box*, *neighbors*) ← Create_Box(*cells*, *x*, *y*, *z*)
    bounding_boxes.append(*bounding_box*)
    bb_neighbors.append(*neighbors*)
    channels[*c*].append(len(*bounding_boxes*) -1)
  end if
end for
for *i* < len(*bounding_boxes*) **do**
  for *j* < len(*bb_neighbors*[*i*]) **do**
    for *i'* < len(*bounding_boxes*[*i*]) **do**
      for *j'* < len(*bounding_boxes*[*bb_neighbors*[*i*][*j*]]) **do**
        See Alg. 10.
        (*x*, *y*) ← Remove_Conflict(*bounding_boxes*[*c*], *bounding_boxes*[*bb_neighbors*[*i*][*j*]][*j'*])
        bounding_boxes[*i*][*i'*] ← *x*
        bounding_boxes[*bb_neighbors*[*i*][*j*]][*j'*] ← *y*
      end for
    end for
  end for
end for
for c < len(channels) **do**
  Write_Obj(*bounding_boxes*, *channels*[*c*], *c*)
end for

Algorithm 4 CLASS: Cell

public:
Cell Cell(int[][][] *chan_matrix*, double[][][] *corner_point_matrix*, *x*, *y*, *z*){
visited ← FALSE
chan ← *chan_matrix*[*x*][*y*][*z*]
*c*1 ← corner_point_matrix[$2 \times x$][$2 \times y$][$2 \times z$]
*c*2 ← corner_point_matrix[$2 \times x + 1$][$2 \times y$][$2 \times z$]
.
.
.
*c*8 ← corner_point_matrix[$2 \times x + 1$][$2 \times y + 1$][$2 \times z + 1$] }
boolean visited

private:
int *chan*

There is a need to store the corner-points shown in Fig. 2, in a such a way that they are easy to look up.
int[3] c1
int[3] c2
int[3] c3
int[3] c4
int[3] c5
int[3] c6
int[3] c7
int[3] c8

seen in Fig. 6. When creating each bounding box both the bounding box and its neighbors are defined. To create the bounding box we begin by defining a queue q and adding a seed cell to this queue. We get the seed cell by iterating through the cells in Sec. 4. We also store the starting cells channel in the *chan* variable and set the variables min_z and max_z to the starting cells z value. A breadth first search (BFS) is then conducted to find the lowest and highest cell in the bounding box. The bounding box is then defined by letting the top be the top cell's c1, c2, c3 and c4 and the bottom be the bottom cell's c5, c6, c7 and c8 (see Fig. 2). To avoid defining the box more than once we set every cell found in the BFS to visited. Any cell set to visited will not be iterated through in Sec. 4.1. Also for every cell visited in the search find this cells neighbors in the 4 neighboring columns and add these to a list of neighboring cells. The pseudo-code for the creation of bounding boxes is given in Alg. 5.

4.4 Finding Neighbors

The 4-neighborhood of any given cell is defined as the cells incident to it in the columns (z-axis) to its left, right, front and back. We are not interested in its neighbors on the z-axis because they have already been found the last section. To find a cells 4-neighborhood requires the use of a search algorithm for ordered lists. This is because although the z-axis is on an irregular grid it is still sorted according to height. By using a binary search this can be accomplished in O(log n) time-complexity. This binary search finds a seed in the neighboring column from which a BFS in the z-axis can be run to find all the possible neighbors of the cell in that column. The neighbors that have the same channel as the cell are then concatenated and returned. The overall algorithm for this is shown in Alg. 6.

4.5 Binary Search and BFS

The binary search is conducted by comparing neighboring corner-points from Fig. 2 based on which column-position (*pos*) it is being compared against. The algorithm is shown in Alg. 7 and is assumed

Algorithm 5 Create_Box(array[][][] $cells$, x, y, z)

$q \leftarrow [[x, y, z]]$
$chan \leftarrow cells[x, y, z].chan$
$min\_z \leftarrow z$
$max\_z \leftarrow z$
$neighbors) \leftarrow []$
while $q \neq []$ **do**
  $x', y', z' \leftarrow q[0]$
  $q \leftarrow q[1:]$
  $cells[x'][y'][z']$.visited $\leftarrow$ True
  if $z' < min\_z$ **then**
    $min\_z \leftarrow z'$
  end if
  if $z' > max\_z$ **then**
    $max\_z \leftarrow z'$
  end if
  if $\neg cells[x'][y'][z' - 1]$.visited and $chan = cells[x', y', z' - 1].chan$ **then**
    q.append($[x', y', z' - 1]$)
  end if
  if $\neg cells[x'][y'][z' + 1]$.visited and $chan = cells[x', y', z' + 1].chan$ **then**
    q.append($[x', y', z' + 1]$)
  end if
  Find this cells neighbors.
  $neighbors$.concatenate(Find_Neighbors($cells$, x, y, z))
end while
Define the bounding box based on the corner-points from Fig. 2.
$bounding\_box \leftarrow$ [[$cells[x][y][max\_z]$.c1], [$cells[x][y][max\_z]$.c2], [$cells[x][y][max\_z]$.c3], [$cells[x][y][max\_z]$.c4], [$cells[x][y][min\_z]$.c5], [$cells[x][y][min\_z]$.c6], [$cells[x][y][min\_z]$.c7], [$cells[x][y][min\_z]$.c8]]
return ($bounding\_box$, $neighbors$)

Algorithm 6 Find_Neighbors(Cell[][][] $cells$, x, y, z)

Since the grid is irregular on the z-axis it is necessary to use a binary search for a neighbor in each of the neighboring columns.

For the back-side column.
$back \leftarrow$ Binary_Search($cells$, 'B', $x-1$, y, z)
For the front-side column.
$front \leftarrow$ Binary_Search($cells$, 'F', $x+1$, y, z)
For the left-side column.
$left \leftarrow$ Binary_Search($cells$, 'L', x, $y-1$, z)
For the right-side column.
$right \leftarrow$ Binary_Search($cells$, 'R', x, $y+1$, z)

Since the z-axis is irregular there may be more than a single neighbor. Therefore use the neighbors found so far as starting points for a Breadth First Search (BFS) through the z-axis

For the back-side column.
$back \leftarrow$ BFS($cells$, 'B', $x-1$, y, $back[0][2]$)
For the front-side column.
$front \leftarrow$ BFS($cells$, 'F', $x+1$, y, $front[0][2]$)
For the left-side column.
$left \leftarrow$ BFS($cells$, 'L', x, $y-1$, $left[0][2]$)
For the right-side column.
$right \leftarrow$ BFS($cells$, 'R', x, $y+1$, $right[0][2]$)

$ret \leftarrow back$.concatenate($front$)
$ret \leftarrow ret$.concatenate($left$).concatenate($right$)
return ret

to be self explanatory. For the breadth first search we begin by identifying top and bottom corner-points from the original cell which we will use to compare the height values of the neighbors (see Alg. 8). A BFS is then run from the seed position found in Alg. 7, and it is stopped once the current cell in the search is outside the boundaries of the cell whose neighbors are being identified. All neighbors which have the same *chan* value as the original cell are added to the final neighborhood. The other cells are simply discarded. This last part of the BFS is shown in Alg. 9.

4.6 Removing Overlapping Faces

Finally we are ready to remove or resize the overlapping faces. This is done by a function Remove_Conflicts which takes two faces overlapping faces fa and fb as argument. We begin by getting the z-value of every point in fa and fb calling them $az1$, ..., $az4$ and $bz1$, ..., $bz4$ based on face and point position from Fig. 6. Which points that are located inside the other face is then determined by a simple inside outside test based on the z-values. We can do this because the faces are located on a grid defined by grid-lines, therefore any overlap of two neighboring faces on the z-axis automatically infers a overlap on the x-y axes as well. We let any overlap result in the point inside the other face being moved to a suitable endpoint of the other face. If all the points of one or both of the faces are completely inside the other then simply remove one or both of them. The algorithm describing this in greater detail can be seen in Alg. 10 and the results of the inside outside tests can be seen in Fig. 4.6. This operation is done for all neighboring faces found in Alg. 6 as described in Alg. 3, and the resulting simplified faces are used to replace the old bounding box faces. Once this is done the simplified bounding box model can be simply be translated to triangle-polygons in a similar manner as the one described for the cell cubes in Alg. 2 and written to files as wavefront objects.

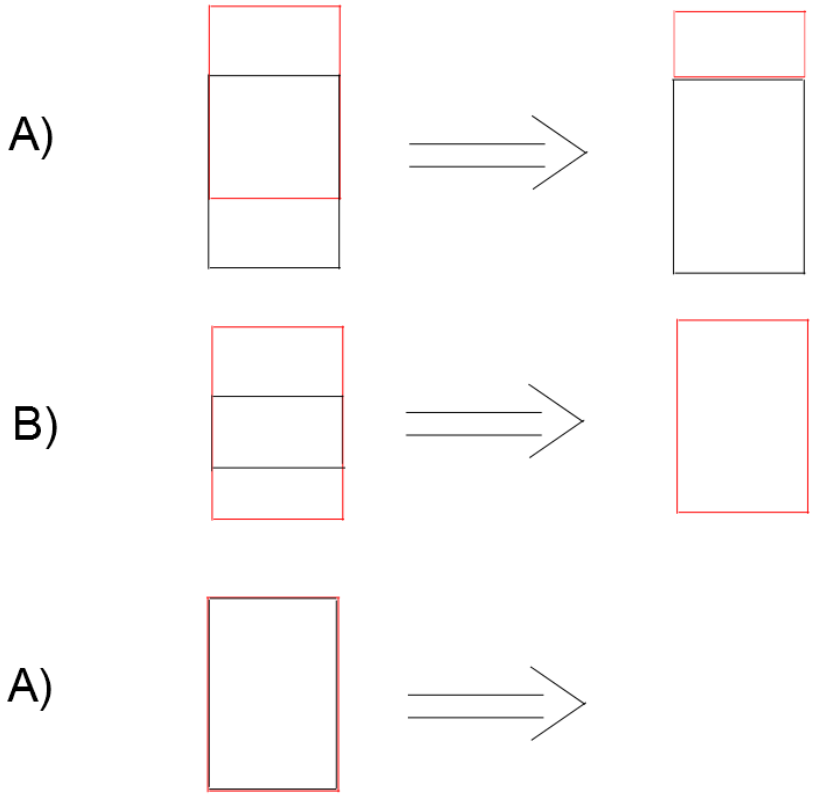

Figure 5: The figure represents the removal/resizing operation from Alg. 10. With the red polygon being fa and the black polygon being fb. In A) the black polygon only partially overlap with the red polygon so the the red polygon being fa is simply resized to remove the overlap. In B) the black polygon is contained entirely inside the red one and is there fore removed. In C) both polygons are contained entirely inside each other. Therefore the both polygons are inside the channel and the view of them obscured, so both can 7 safely be removed.

5 Implementation

Both the naive conversion and the simplifying conversion are implemented in the Python 2.7 language. Only standard libraries have been used and no other dependencies are required. Both scripts should work on any platform with a Python 2-7 implementation. However so far it has only been tested on the Windows XP Professional with Service Pack 3 and Fedora Linux 14. The testing was done on the University of Bergen Norway's computer system. The file system was accessed over the universities local network using the Samba protocol. This has a negative performance effect on IO compared to using a local BUS connected hard-disk. For the testing in Sec. 6 a machine with Windows XP was used, it had 4GB of physical memory and a 2.3 Ghz GenuineIntel CPU.

Algorithm 7 Binary_Search(Cell[][][] *cells*, *pos*, *x*, *y*, *z*)

```
p_z ← NONE
c_z ← NONE
if pos = 'B' then
  p_z ← cells[x + 1][y][z].c1
end if
if pos = 'F' then
  p_z ← cells[x − 1][y][z].c3
end if
if pos = 'L' then
  p_z ← cells[x][y + 1][z].c1
end if
if pos = 'R' then
  p_z ← cells[x][y − 1][z].c2
end if

max ← cells.range()[2]
min ← 0
q ← []
while q = [] do
  last_z ← z
  z ← (min+max)/2
  if last_z = z then
    q.append([-1, [x, y, z]])
  end if
  if pos = 'B' then
    c_z ← cells[x][y][z].c3
  end if
  if pos = 'F' then
    c_z ← cells[x][y][z].c1
  end if
  if pos = 'L' then
    c_z ← cells[x][y][z].c2
  end if
  if pos = 'R' then
    c_z ← cells[x][y][z].c1
  end if

  if p_z < c_z then
    max ← max − 1
  end if
  if p_z > c_z then
    min ← min + 1
  end if
end while
return q
```

Algorithm 8 BFS(Cell[][][] *cells*, *pos*, *x*, *y*, *z*), Part 1.

```
p_zt ← NONE
p_zb ← NONE
p_chan ← NONE
if pos = 'B' then
  p_zt ← cells[x + 1][y][z].c1
  p_zb ← cells[x + 1][y][z].c5
  p_chan ← cells[x + 1][y][z].chan
end if
if pos = 'F' then
  p_zt ← cells[x − 1][y][z].c3
  p_zb ← cells[x − 1][y][z].c7
  p_chan ← cells[x − 1][y][z].chan
end if
if pos = 'L' then
  p_zt ← cells[x][y + 1][z].c1
  p_zb ← cells[x][y + 1][z].c5
  p_chan ← cells[x][y + 1][z].chan
end if
if pos = 'R' then
  p_zt ← cells[x][y − 1][z].c2
  p_zb ← cells[x][y − 1][z].c6
  p_chan ← cells[x][y − 1][z].chan
end if
while q ≠ [] do
  last_z′ ← q[0][0]
  x′, y′, z′ ← q[0][1]
  q ← q[1:]
  if cells[x′, y′, z′].chan = p_chan then
    neighbors.append([x′, y′, z′])
  end if
  c_zt ← NONE
  c_zb ← NONE
  if z′ − 1 ≠)last_z′ and z′ − 1 > 0 then
    if pos = 'B' then
      c_zt ← cells[x′][y′][z′ − 1].c3
      c_zb ← cells[x′][y′][z′ − 1].c7
    end if
    if pos = 'F' then
      c_zt ← cells[x′][y′][z′ − 1].c1
      c_zb ← cells[x′][y′][z′ − 1].c5
    end if
    if pos = 'L' then
      c_zt ← cells[x′][y′][z′ − 1].c2
      c_zb ← cells[x′][y′][z′ − 1].c6
    end if
    if pos = 'R' then
      c_zt ← cells[x′][y′][z′ − 1].c1
      c_zb ← cells[x′][y′][z′ − 1].c5
    end if
  end if
  continue with Alg. 9 here.
end while
return neighbors
```

Algorithm 9 BFS(Cell[][][] *cells*, *pos*, x, y, z), Part 2.

Continues inside while-loop
if $c\_zt \neq$ NONE and $c\_zt < p\_zt$ and $c\_zb < p\_zb$ **then**
  Do nothing.
else
  q.append([z',[x', y', $z'-1$]])
end if
$c\_zt \leftarrow$ NONE
$c\_zb \leftarrow$ NONE
if $z'+1 \neq$)$last\_z'$ and $z' <$ *cells*.range()[2] **then**
  if *pos* = 'B' **then**
    $c\_zt \leftarrow cells[x'][y'][z'+1]$.c3
    $c\_zb \leftarrow cells[x'][y'][z'+1]$.c7
  end if
  if *pos* = 'F' **then**
    $c\_zt \leftarrow cells[x'][y'][z'+1]$.c1
    $c\_zb \leftarrow cells[x'][y'][z'+1]$.c5
  end if
  if *pos* = 'L' **then**
    $c\_zt \leftarrow cells[x'][y'][z'+1]$.c2
    $c\_zb \leftarrow cells[x'][y'][z'+1]$.c6
  end if
  if *pos* = 'R' **then**
    $c\_zt \leftarrow cells[x'][y'][z'+1]$.c1
    $c\_zb \leftarrow cells[x'][y'][z'+1]$.c5
  end if
end if
if $c\_zt \neq$ NONE and $c\_zt > p\_zt$ and $c\_zb > p\_zb$ **then**
  Do nothing.
else
  q.append([z',[x', y', $z'+1$]])
end if

Algorithm 10 Remove_Conflicts(array[4] fa, array[4] fb)

Faces are stored left to right and top to bottom in the array [C1, C2, C3, C4] as shown in Fig. 6.
Get the height values for comparison.
$az1 \leftarrow fa[0][2]$
$az2 \leftarrow fa[1][2]$
$az3 \leftarrow fa[2][2]$
$az4 \leftarrow fa[3][2]$

$bz1 \leftarrow fb[0][2]$
$bz2 \leftarrow fb[1][2]$
$bz3 \leftarrow fb[2][2]$
$bz4 \leftarrow fb[3][2]$
Check which points are inside the other polygon.
$inside\_a \leftarrow$ [FALSE, FALSE, FALSE, FALSE]
$inside\_b \leftarrow$ [FALSE, FALSE, FALSE, FALSE]
if $az1 >= bz1$ and $az1 <= bz3$ **then**
  $inside\_a[0] \leftarrow$ TRUE
  $fa[0] \leftarrow fb[2]$
end if
if $az2 >= bz2$ and $az2 <= bz4$ **then**
  $inside\_a[1] \leftarrow$ TRUE
  $fa[1] \leftarrow fb[3]$
end if
if $az3 <= bz3$ and $az3 >= bz1$ **then**
  $inside\_a[2] \leftarrow$ TRUE
  $fa[2] \leftarrow fb[0]$
end if
if $az4 <= bz4$ and $az4 >= bz2$ **then**
  $inside\_a[0] \leftarrow$ TRUE
  $fa[3] \leftarrow fb[1]$
end if

if $bz1 >= az1$ and $bz1 <= az3$ **then**
  $inside\_b[0] \leftarrow$ TRUE
  $fb[0] \leftarrow fa[2]$
end if
if $bz2 >= az2$ and $bz2 <= az4$ **then**
  $inside\_b[1] \leftarrow$ TRUE
  $fb[1] \leftarrow fa[3]$
end if
if $bz3 <= az3$ and $bz3 >= az1$ **then**
  $inside\_b[2] \leftarrow$ TRUE
  $fb[2] \leftarrow fa[0]$
end if
if $bz4 <= az4$ and $bz4 >= az2$ **then**
  $inside\_b[0] \leftarrow$ TRUE
  $fb[3] \leftarrow fa[1]$
end if
Continue to Alg. 11.

Algorithm 11 Remove_Conflicts(array[4] fa, array[4] fb)

Continued from Alg. 10.

If there one or both of the polygons are entirely inside the other, then remove it completely.
if $inside\_a$[0] and $inside\_a$[1] and $inside\_a$[2] and $inside\_a$[3], **then**
  $fa \leftarrow$ NONE
end if
if $inside\_b$[0] and $inside\_b$[1] and $inside\_b$[2] and $inside\_b$[3], **then**
  $fb \leftarrow$ NONE
end if

return (fa, fb)

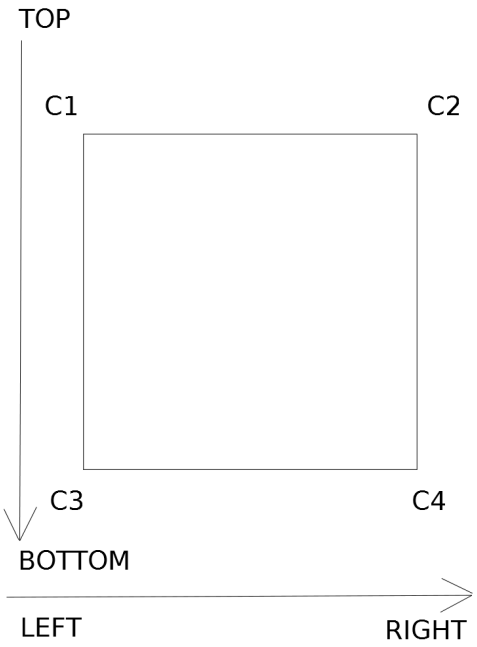

Figure 6: A face of the bounding box in Sec. 4 represented by the array [c1, c2, c3, c4]. The markings c1, c2, c3 and c4 corresponds to the top leftmost, top rightmost, bottom leftmost and bottom rightmost respectively, as can be seen in the figure.

6 Comparison of Results

We ran both the naive and the simplifying conversions on a single dataset with a $96 \times 128 \times 70$ grid and 80 channels. We then compared the results of the two approaches. A clear reduction could be observed however it was definitely best with the larger channels. For instance for channel 37 seen in Fig. 6. The naive conversion results in 182168 vertices and 273252 faces, whereas the simplifying conversion results in 11640 vertices and 5820 faces. This corresponds to a 96% and 98% reduction respectively. We see a consistent reduction of around 90% for the larger channels, which is clearly a good result. The improvement for smaller channels is not as large. For instance for channel 36 we see naive conversion result of 488 vertices and 732 faces and 140 vertices and 70 faces for simplifying conversion. This gives a reduction of 71% of the vertices and 90% of the faces, which is still a good result. The time complexity of the naive and simplifying conversions are $O(x \times y \times z)$ and $O(x \times y \times z \times log(z))$ respectively. When the scripts were tested in practice however both finished in ca. 3 minutes. This could be at least partially due to the fact that the tests where run with file access over the Samba protocol which makes I/O significantly more time consuming. As we have seen the naive conversion produces much larger results than the simplifying conversion.

7 Conclusion

We have in this report presented a method for converting eclipse models of oil and gas simulation data to a polygonal based wavefront object model for the purpose of visualization. This method uses simplification techniques to remove unnecessary and redundant data and removes overlapping faces. This constitutes a marked improvement over a naive conversion also described in this work. Both with regards to the number of faces and vertices needed and a visible removal of graphical artifacts (overlapping faces based). These comparative reductions are for some

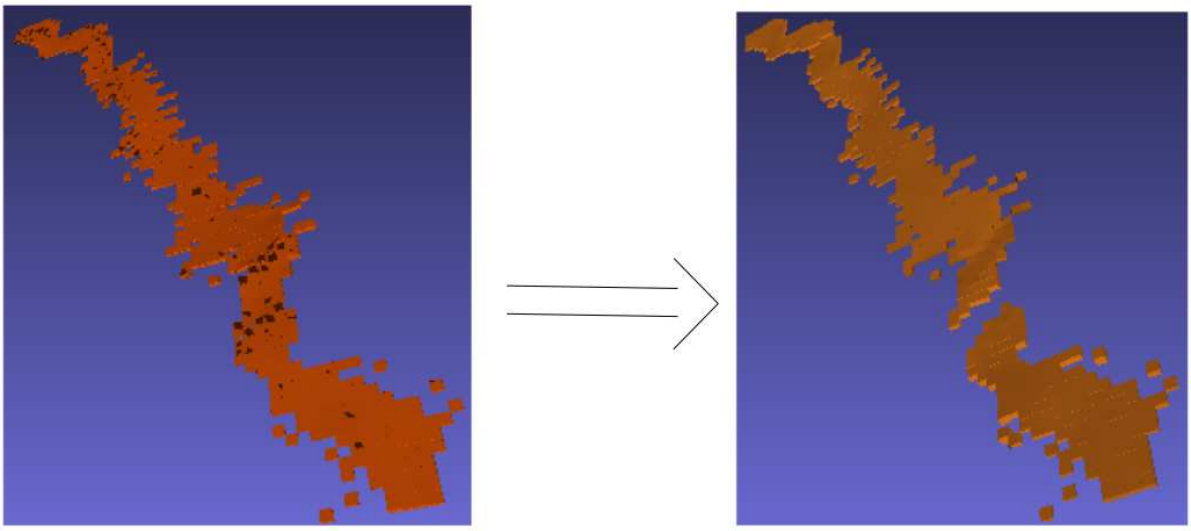

Figure 7: The simplification of the channel seen with naive conversion on the left and simplifying conversion on the right. As can be seen the black spots which are artifacts generated by overlapping faces are not present in the simplified version. For this particular channel the reduction there was 98% in the number of polygons (triangles) compared to the naive version and a 94% reduction in the number of faces.

of the more complex/large dataset in the order of an over 90% reduction of the number of vertices and polygons needed. This is accomplished in practice without a large increase in the resources/time needed to carry out the operation and an increase of time complexity from $O(x \times y \times z)$ to $O(x \times y \times z \times log(z))$.

Special thanks to Endre M. Lidal, not only for supervising this project, but also for providing the dataset for testing the method. As well as the naive conversion program.

References

[Pet11] Pettersen O.: ResSimNotes, http://www.uib.no/People/fciop/Downloads/MAT354/ 2011.