doi: 10.32131/jes.9.51

Journal of the Earth Simulator, Volume 9, March 2008, 51–63

Vectorization of Polygon Rendering for Off-line Visualization of a Large Scale Structural Analysis with ADVENTURE System on the Earth Simulator

Hiroshi Kawai[1], Masao Ogino[2]*, Ryuji Shioya[2], and Shinobu Yoshimura[3]

[1] *Faculty of Science and Technology, Keio University, 3-14-1 Hiyoshi, Kohoku-ku, Yokohama 223-8522, Japan*
[2] *Faculty of Engineering, Kyushu University, 744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan*
[3] *Department of Quantum Engineering and Systems Science, The University of Tokyo, 7-3-1 Hongo, Bunkyo-ku, Tokyo 113-8656, Japan*

(Received June 14, 2007; Revised manuscript accepted March 13, 2008)

Abstract Using ADVENTURE system, a 3-D structural analysis of a very large scale problem can be performed on supercomputers such as the Earth Simulator (ES). However, the data size of the analysis results also becomes huge. We developed an off-line visualizer for visualization of a huge scale 3-D structural analysis on ES. The off-line visualizer performs rendering of surface patch triangles to produce image files of deformation plot and stress contour plot. It is vectorized. The vectorization scheme of its polygon rendering using a look-up table is explained in detail, and its performance evaluation on ES is carried out. It is also demonstrated that a 3-D structural analysis over 200 millions DOFs can be visualized efficiently using our off-line visualizer.

Keywords: Visualization, polygon rendering, vectorization, FEM, structural analysis

1. Introduction

A variety of distributed-parallel supercomputers, such as the Earth Simulator (ES), IBM BlueGene/L, Cray X1E, Hitachi SR11000 and SGI Altix, are available for a heavy-load numerical simulation task. Using those computational servers, a simulation user can perform a huge scale 3-D solid structural analysis based on the finite element method, whose degrees of freedom exceed 100 millions [1].

For example, currently, we are planning an earthquake response simulation of a full-scale nuclear power plant. In this simulation, a solid Finite Element (FE) model over 200 million Degrees Of Freedom (DOFs), which represents a whole nuclear pressure vessel, is employed. It contains lots of structural details, such as nozzles, pipes, control rods and other support structures. It takes just a few hours to execute a dynamic analysis job on ES.

However, each analysis task produces gigantic analysis result data files, which may occupy a disk space size of terabytes order. Because of this size issue, it is difficult and time consuming for the simulation user to move those analysis result data files back to his or her workstation for visualization purposes. Although the visualization capability of a workstation or a PC terminal on the client side still remains an issue, the low network speed between the computational server and the client terminal is also a serious issue against the huge scale visualization.

Including ES, there are a couple of computational servers in the world, with an unprecedented level of computational power. Ideally, such an extremely powerful supercomputer should not be dominated by any single organization, and it should be shared among many users nationwide, including researchers at universities, national research centers and research sections in the industries, or, if possible, worldwide also. In this case, those users should not be forced to only visit the supercomputer center directly and stay inside the center building while using the supercomputer, but it is also desirable for them to use the supercomputer remotely from their own laboratories far away from the supercomputer center. This implies that there should be some effiective ways to access the supercomputer over the Internet. Remote visualization should also be considered seriously in this context.

Actually, in the Earth Simulator Center (ESC), there are some graphics workstations. Each of them has a dedi-

* ***Corresponding author***: *Masao Ogino, Faculty of Engineering, Kyushu University, 744 Motooka, Nishi-ku, Fukuoka 819-0395, Japan. E-mail: ogino@mech.kyushu-u.ac.jp*

cated 3-D graphics hardware and OpenGL library, and they are connected to ES through the local area network called ES-LAN. There is also virtual reality equipment in the ESC called BRAVE and its software package called VFIVE [2]. However, to use these graphics workstations and the virtual reality environment, we have to visit the ESC. It is more convenient to perform our visualization tasks over the Internet, using a client PC in our laboratories. Of course, because the Internet is far slower than ES-LAN, it usually takes days or weeks to move our analysis result data files back into our laboratories.

Here in this paper, we assume that the average transmission speed through the Internet is currently about 100 M bps or less, based on the number measured actually between the ESC at Yokohama, located at the east region of Japan, and our laboratories in Kyushu University at Fukuoka, located at the south region of Japan. There are many routers on the communication path. The network is unstable, and some extra efforts are usually required to guarantee the completion of long-term data transfer for more than one day.

In this paper, we explain about our implementation of a server-side visualizer running on ES, one of the most powerful computational servers in the world, for a 3-D solid structural analysis using the finite element method.

After a huge scale FE analysis finishes, the visualization of the analysis result data is performed at the post-processing stage. Two of the most important visualization techniques for a structural analysis are a scalar contour plot and a deformation plot. In case of a 3-D solid problem, a contour plot of stress or strain distribution over the surface of the analysis model is drawn. It may also be combined with a deformation plot, which renders a displacement vector distributed over the model surface. This type of visualization, which belongs to surface rendering in a classical categorization of scientific visualization, is the main target of our research. If the FE analysis is dynamic, animation is also required.

To visualize a surface contour and deformation plot efficiently, it is necessary to obtain the boundary surface information of the 3-D solid FE mesh. It is called the 'surface patch' of the mesh. In case of using tetrahedral solid elements, which are often used to model a complicated 3-D solid structure, the surface patch is composed of triangles. Of all 4 triangular faces of all the tetrahedral elements, only the triangles just on the surface boundary of the mesh are collected and compose the surface patch.

Usually, it is time-consuming to extract the surface patch from a given huge scale structural model. Fortunately, the surface patch has already been extracted from the analysis model before the post-process stage begins, because this information is also required to specify loads and constraints at the pre-process stage.

At this stage, we have mainly two ways to visualize the huge analysis result data. One is to generate on the computational server only intermediate data to represent mainly the geometry information, and then, transfer and visualize them on a remote client PC. The other is to generate a final image data directly on the server.

In case of the former approach, first, triangles are generated from the surface patch and its associated analysis result data on the computational server. Then the generated geometry data are sent over the Internet from the ESC to our university. In our laboratory, the geometry data can be visualized interactively, using the powerful graphics hardware in an ordinary client PC. We can pan, zoom and rotate those triangles.

However, there are still some problems in this former approach. Except for viewing parameters, anytime we need to change any of the visualization parameters, such as contour type and range, vector magnification ratio, as well as selection of physical quantity types and their evaluation schemes related to structural analyses, new geometry data may have to be re-generated at the computational server. Moreover, in case of a dynamic analysis with many time steps, the amount of geometry data transferred through the Internet becomes huge. It is also mentioned as a serious issue in the reference [3]. For example, let us consider about our earthquake simulation of a nuclear pressure vessel model, with 200 M DOFs and 1000 time steps. Before sending the data over the Internet, we first extract only the relevant portion of the result data on the surface boundary of the model as an enriched geometry model, which is dedicated for our visualization purpose. Then, the extracted data are converted into a binary format called ADVENTURE I/O [4], and the data are compressed also. Usually, the size of the enriched geometry data, limited on the model surface only, can be reduced into about one tenth of the size of the whole original analysis result data. Even using those data on the surface patch only, it still takes a few days to transfer them over the slow Internet.

In the latter approach, instead of geometry data, image or animation data are generated directly on the computational server. Only those image data are sent to our laboratories. The size of the data is usually small, an order of M bytes or less. Obviously, the drawback of this approach is that, when the animation data are shown on a client PC, except time step parameters, all the visualization and viewing parameters have already been fixed and we cannot change them interactively.

We think both of two approaches are useful. While waiting for huge amount of extracted geometry data on the model surface coming from the ESC over the Internet,

we can browse through image data generated by our off-line visualizer and monitor the status of the analysis job running at the same time. Once the geometry data have come, then using a PC on the client-side, we can dive into the model to search more detailed information thoroughly in an interactive environment, using walkthrough visualization techniques [5]. In this paper, we focus on a visualization technique on the server-side based on the latter image generation approach.

Originally, the keyword, 'off-line rendering', often used in the computer graphics field, means non-interactive image generation. It is different from 'interactive rendering' in front of the monitor screen of a graphics workstation through mouse and keyboard operations. In some computer graphics studio companies, off-line rendering may be performed using a PC cluster or a supercomputer. Here, we use this keyword as image generation for scientific visualization on a supercomputer.

The concept of off-line visualization is shown in Fig. 1. For example, a user may invoke the off-line visualizer from a remote terminal using the time-sharing system mode of a supercomputer, or the off-line visualizer may be executed as a batch job through the job manager. In either case, the visualizer reads the result data files and performs image rendering on the supercomputer. It produces image files or animation files rather than 3-D interactive graphics on a monitor screen.

Further, to implement this image generation approach actually, we can use either surface rendering or volume rendering.

In case of the volume rendering approach, usually, ray tracing or ray casting are used. There are some research such as Max [6] and GeoFEM/HPC Middleware [7] [8]. An image is generated directly from the surface patch and its associated result data, in a pixel-wise manner.

In case of the surface rendering approach, not only triangle geometry data are generated, but also an image is rendered from those temporary triangles. Rendering of the image from the triangles is performed using a triangle rasterization algorithm in a triangle-wise manner. Both triangle and image generation processes are performed on the computational server.

We have chosen the latter surface rendering approach, because this approach is fully compatible with most of the visualization methods and existing applications used in the structural analysis field. It is relatively easy to port an existing visualization application on ES, by simply switching an implementation of the 3-D graphics portability layer, such as [9], inside the application code.

In summary, the essential task of our off-line visualizer

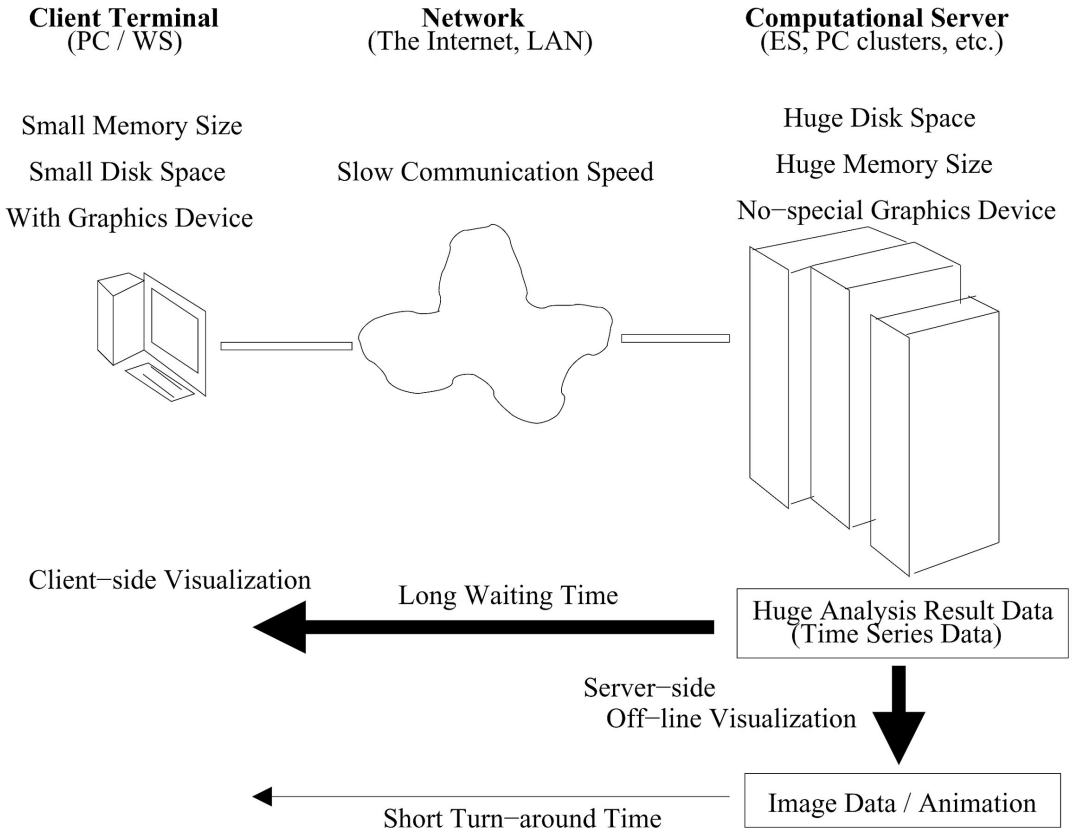

Fig. 1 Off-line visualization on a computational server.

is, to render a huge amount of triangles forming the surface patch, which has already been extracted. This means, we need a polygon renderer running on the supercomputer. A deformation plot can be rendered as a set of deformed triangles of the surface patch reflecting the displacement vector. A scalar contour plot can also be rendered using one dimensional texture mapping.

In this approach, however, we have to prepare a pure software-based polygon renderer running on ES. Most of the supercomputers, including ES, have no dedicated 3-D graphics hardware. It is also possible that no 3-D graphics library is available, or its implementation is not effcient even if available. Furthermore, because ES is a vector-parallel type architecture supercomputer, the polygon renderer should be both vectorized and parallelized.

As for the parallelization of the polygon rendering, there are already several methods available. Molnar [10] classified parallel rendering algorithms into three categories, sort-first, sort-middle and sort-last. The sort-first approach is used mainly for an existing sequential application to parallelize only its rendering part. This approach is usually implemented as the substitution of a 3-D graphics library from the sequential version to the parallel one. Nowadays, the sort-middle approach is used only for the implementation of new graphics hardware, because it requires heavy communication.

For most of the applications already parallelized based on domain-decomposition, including our case, the sort last approach is suitable. This approach generates an image domain-by-domain, then these domain-wise images are collected into the master process, and they are composed into a final single image. For the surface rendering without transparency, depth buffer information is used for the image composition. In our system, we use a binary-tree communication approach [5].

Therefore, the only remaining issue is, how to vectorize the polygon rendering process for a vector-type processor. There are some approaches, such as RVSLIB [11] [12], PATRAS [13] and MovieMaker [2], to utilize simply an existing implementation of triangle rasterization available for scalar-type processor. However, in these approaches, the rendering process can become a performance bottleneck on a vector-type supercomputer. It is especially true if the rendering task is also executed within the analysis process, or in the same large-scale batch job.

This paper focuses on this important missing part, vectorization of the triangle rasterization, which runs on ES and generates image data of surface scalar contour and deformation plots using the pre-extracted triangle surface patch. The most significant feature of our approach is the usage of a look-up table to accelerate the triangle rasterization. It is necessary not only to vectorize the majority of the rendering code but also to minimize the performance cost at the remaining scalar bottleneck.

Here, our vectorized polygon rendering algorithm using the look-up table is explained briefly. At the first stage, it performs triangle-wise calculation. At the second stage, it performs triangle fragment-wise calculation. And at the third stage, it performs image pixel-wise calculation. There are two kind of object sorting in our algorithm. One is in the middle of the rendering process, at the transition between triangle-wise and fragment-wise calculations. The other sorting is at the last of the process, at the transition between fragment-wise and pixel-wise calculations. The look-up table reduces execution time of the former transition step, which is a scalar section and the main bottleneck of our algorithm. As far as we know, there seems to be no other research activities about vectorization of polygon rendering, although our algorithm has some similar points with those for SIMD machines [14] [15] [16].

One thing we assume is that almost all the triangles rendered at visualization of a huge scale 3-D structural analysis are very small, usually less than 10 by 10 pixels in the screen coordinate space. This assumption is the same as the one used in other research [14] [15]. Based on this assumption, our algorithm uses the look-up table for most of the triangles to perform in-out detection and generation of fragments at the scan conversion stage efficiently.

In this paper, the size of our look-up table is relatively large. It may occupy about tens or hundreds of M bytes. It is understandable that there has been no practical application using this sort of look-up table-based approaches ever before, because, until recently, such a table has been considered prohibitively large. However, now we think it is well acceptable, because the memory capacity per single processor of ES is 2 G bytes.

2. Polygon Rendering Algorithm

Here, we describe our polygon rendering algorithm in detail. Its vectorization on a vector type supercomputer such as ES is also explained.

The main functionalities of our polygon renderer currently supported are as follows.

- pan, zoom and rotate
- orthogonal projection
- lighting and shading (flat shading)
- hidden surface removal using a depth buffer
- triangle polygon fill
- smooth and band contours using 1-D texture mapping
- clipping by arbitrary section planes

2.1 General Rendering Procedure

Here, for convenience, we introduce a general procedure to render triangle polygons briefly. The procedure

shown below is a classical one. As we did in this paper for a vector-type parallel computer, depending on the specific hardware configuration of each system, some modifications may occur.

1. With each triangle, coordinate transformation from the world coordinate system to the screen coordinate system is performed.
2. With each triangle, clipping is performed. Triangles outside the clipping volume are removed.
3. With each triangle, lighting calculation is performed and vertex colors are calculated.
4. With each triangle, scan conversion is performed and the triangle is decomposed into multiple fragments.
5. With each fragment, shading is performed and its color is interpolated among the color of three vertices.
6. With each fragment, depth test is performed for hidden surface removal. If the fragment passes the test, it is written into the corresponding pixel in the screen image.

We further explain about the triangle scan conversion process in more detail. It is shown in Fig. 2.

With a given triangle, which is already transformed into the screen coordinate space, the scan conversion process decomposes the triangle into multiple scan lines horizontally. Then, each scan line is further decomposed into multiple fragments vertically.

Here, these fragments are the pixels composing the triangle. Usually, only a portion of the fragments, which pass depth test using a depth buffer, is reflected into the corresponding pixels in the screen image.

2.2 Assumption

A few assumptions are made for the basic design of our polygon renderer. These assumptions are derived from the typical characteristics of surface patches of our 3-D solid structural analyses on ES, and the usage patterns of our off-line visualizer to render surface contour and deformation plots.

First, there are millions of triangles in the surface patch of a huge scale 3-D structural analysis with hundreds of millions DOFs. If the off-line visualizer is parallelized, each processor manages at least tens of thousands of triangles or more.

Second, most of the triangles rendered on the screen image is very small compared with the image size. When using a 1,000 by 1,000 resolution image, an average pixel size of triangles on the screen coordinate system is less than 10 by 10 pixels. The larger the analysis scale

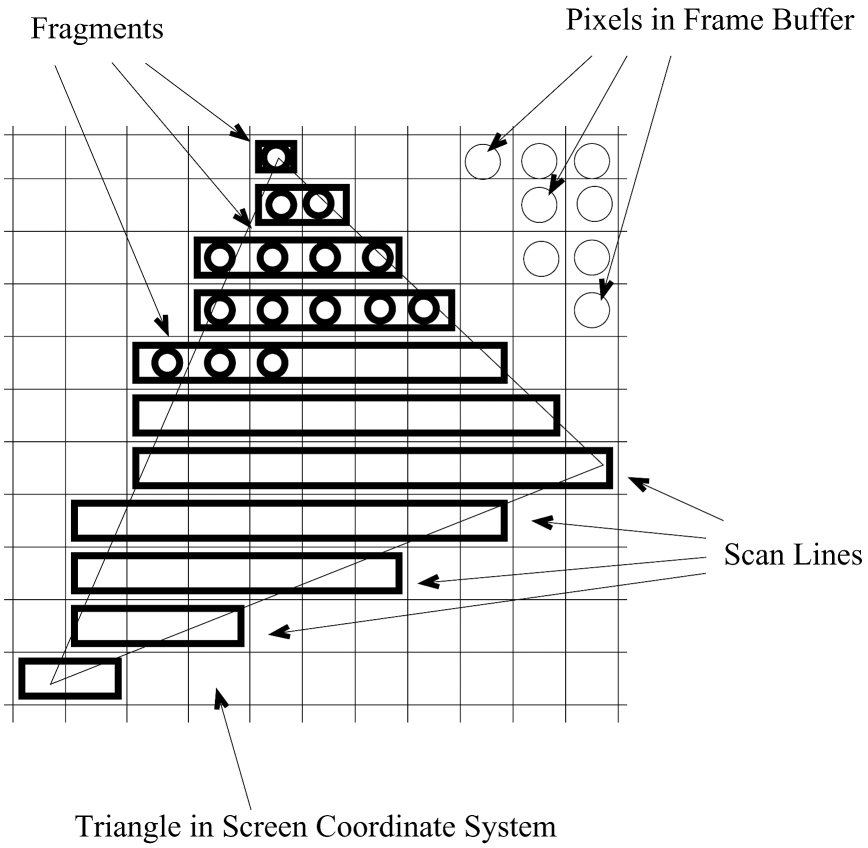

Fig. 2 A scan conversion of a triangle. A triangle is decomposed into multiple scan lines, then each scan line is decomposed into multiple fragments.

becomes, the more the number of triangles in the surface patch is, while the resolution size of the monitor screen on a client side terminal has kept almost constant in recent years, because of technical difficulties of LCD monitor production. Therefore, the average triangle pixel size will become smaller and smaller.

2.3 General Strategy

Based on the assumptions described above, our polygon rendering algorithm deal with only small triangles whose pixel size fits with the look-up table. Here, we call these triangles as 'table-fit' triangles.

Therefore, input triangles are divided into two groups, 'table-fit' or not. The criterion for this classification is the pixel size of each triangle. Usually, the size less than 10 or 20 pixels are classified as 'table-fit'. In case of our usage patterns, most of the input triangles in a surface patch are classified into the 'table-fit' group. The rest of the input triangles are processed using a conventional triangle scan conversion algorithm.

In vectorization, the vector loop length, which is the loop length of the inner most loop, is either the number of triangles or the number of fragments generated from those triangles. Usually, they are tens of thousands or more.

2.4 Triangle Coordinate System

Here, we introduce a keyword, a 'triangle coordinate system', for convenience of the discussions below.

At the coordinate transformation stage, a triangle defined in the world coordinate system is transformed finally into the screen coordinate system. The screen space is a 2-D XY integer value coordinate system.

Further, we transform the triangle from the screen coordinate system into the 'triangle coordinate system' of its own. The triangle coordinate system of a given triangle has the same scale and orientation as the screen coordinate system and its origin point is at one of the three vertices of the triangle. It is shown in Fig. 3.

Suppose there are three vertices in the triangle, v0, v1 and v2, respectively. In the triangle coordinate system of this triangle, the origin point is the position of vertex v0. Thus, the coordinate value at vertex v0 is (0, 0) in the triangle coordinate system.

The coordinate values of other two vertices, v1 and v2, are (tx1, ty1) and (tx2, ty2) in the triangle coordinate system, respectively. Each of these four coordinate component values, tx1, ty1, tx2, ty2, is an integer value and it may be positive, zero or negative. If the pixel size of the triangle is small, the component value fits within the cer-

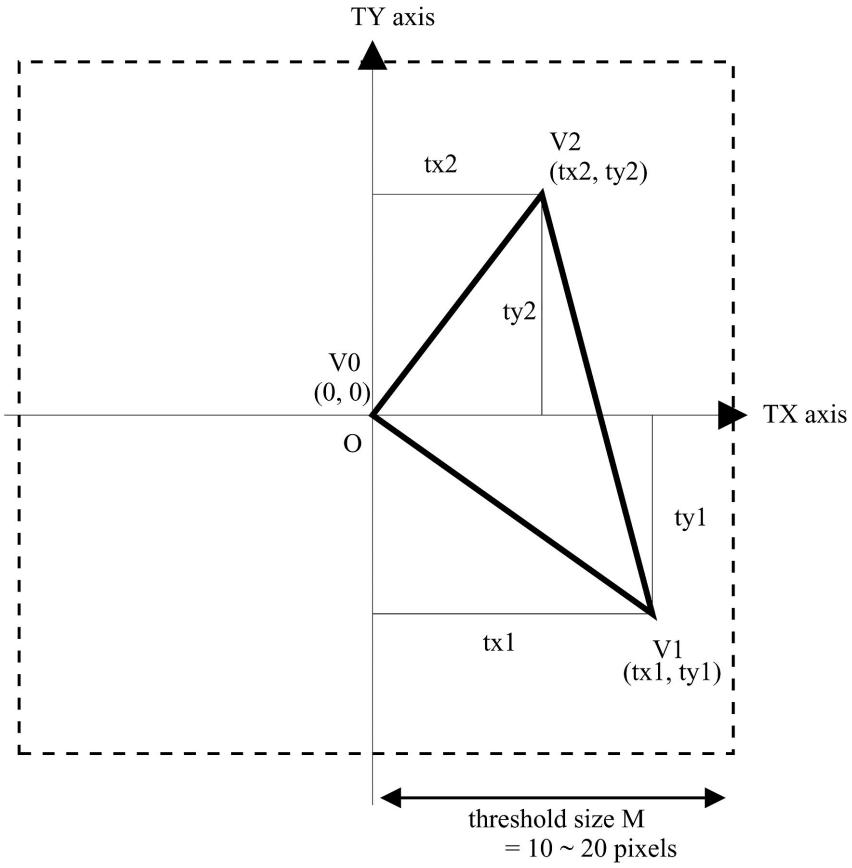

Fig. 3 Triangle coordinate system, which is a special system for assigning a triangle defined in the screen space.

tain range, for example, between –10 to 10, or between -15 to 15, or, at most, between –20 to 20 for most of the cases.

Any allowable combination of four coordinate component values, tx1, ty1, tx2, ty2, specifies the fill pattern of the corresponding triangle uniquely. The set of four integer values completely decides its scan conversion. Therefore, if we prepare all the fill patterns defined by all the combinations of these four values as a look-up table on memory, it can accelerate the scan conversion of a triangle of an arbitrary shape.

The number of all the possible combinations of four integer values is the maximum triangle pixel size powered by 4. It occupies a very large memory space. This threshold pixel size is limited by the allowable memory size of a user process of the off-line visualizer running on ES. Typically, we choose from 10 to 20 as the threshold pixel size. It is sufficient for most of our 3-D structural analysis cases.

As a more concrete representation of the look-up table, we store the beginning and ending TX indices of all the scan lines for each pattern.

Here is an example of a triangle, $tx1 = 10$, $ty1 = 4$, $tx2 = 4$, $ty2 = 10$, shown in Fig. 4.

The look-up table is organized as TX coordinates at the left and right end point of each scan line for each pattern.

In Fig. 4, TX values of the first scan line are from 4 to 4, those of the 2nd scan line are from 4 to 5, those of the 3rd scan line are from 3 to 6, and so on. If the threshold pixel size value is 10, the look-up table becomes the one listed as Table 1.

The memory size required to store a look-up table is proportional to the threshold pixel size powered by 5. A TX coordinate value can be stored as a signed integer of one byte. For example, if the threshold value is 10, 15, and 20, it occupies 6.4 M bytes, 49 M bytes and 205 M bytes, respectively.

Table 1 A Look-up table to define a triangle in a triangle coordinate system.

TY	start TX	end TX
10	4	4
9	4	5
8	3	6
7	3	7
6	2	8
5	2	9
4	2	10
3	1	8
2	1	6
1	1	3
0	0	1
-1	Null	Null
-2	Null	Null
.		
.		
.		
-10	Null	Null

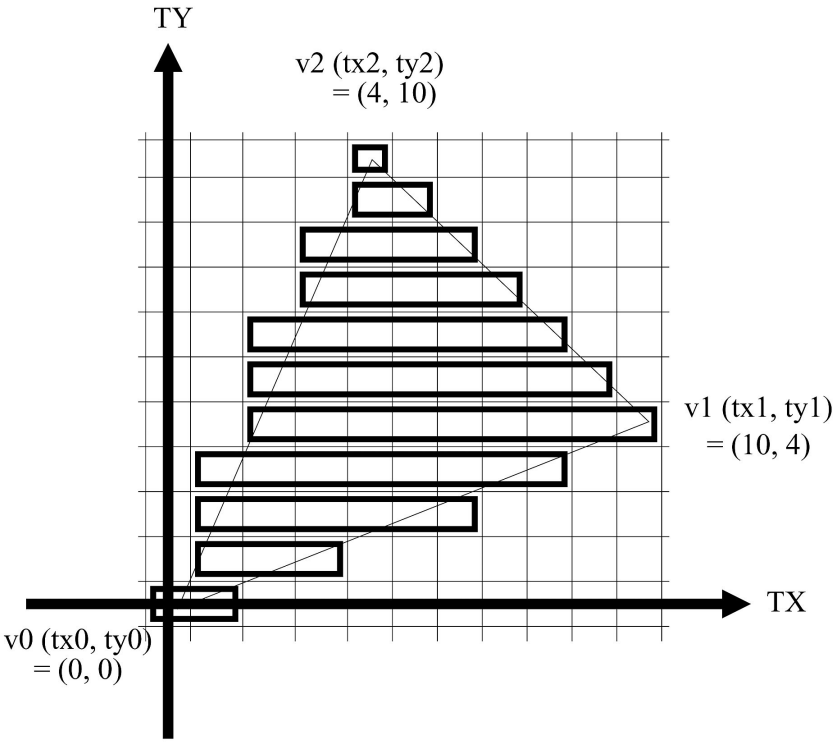

Fig. 4 Scan conversion of a triangle using a look-up table.

Here is a sample implementation code in C language of the definition of the look-up table, shown below.

```
#define MAX_TRI_SIZE 10

signed char ScanLineMinTx

[MAX_TRI_SIZE * 2][MAX_TRI_SIZE * 2]
[MAX_TRI_SIZE * 2][MAX_TRI_SIZE * 2]
[MAX_TRI_SIZE * 2];

/* -1 : no scan line */
/* or 0 ... (MAX_TRI_SIZE - 1) (include) */

signed char ScanLineMaxTx
[MAX_TRI_SIZE * 2][MAX_TRI_SIZE * 2]
[MAX_TRI_SIZE * 2][MAX_TRI_SIZE * 2]
[MAX_TRI_SIZE * 2];

/* -1 : no scan line */
/* or 0 ... (MAX_TRI_SIZE-1) (include) */
```

A constant parameter, MAX TRI SIZE, which is the threshold size, is specified as 10. There are two arrays of signed byte type, ScanLineMinTx and ScanLineMaxTx. The former 4 array indices represent tx1, ty1, tx2, ty2, respectively. The last 5th array index represent the TY value of each scan line. Each array occupies 3.2 M bytes. Note that all the TX values stored in these arrays and the TY value of each scan line are incremented by MAX TRI SIZE, so that a valid value becomes non-negative. If the value is –1, this scan line does not have to be filled.

2.5 Triangle-wise Calculation

The former part of our polygon rendering is triangle-wise calculation including coordinate transformation and lighting. It is performed for all the input triangles.

coordinate transformation and lighting For each vertex of each triangle, coordinate transformation and lighting are performed. Flat shading is used for the lighting calculation.

gradient for depth value and scalar interpolation For each triangle, the gradient vector of depth component in the viewing coordinate system, namely, Z value, is evaluated. If a contour plot is required, the gradient vectorof scalar value distribution over the triangle is also calculated.

pixel size and range in the screen coordinate system For each triangle, the pixel size and the range in the screen coordinate system are calculated. Triangle coordinates of vertices v1 and v2, and a set of four values, tx1, ty1, tx2, ty2, are also calculated.

classification of table-fit triangle If the pixel size of the triangle is less than the threshold value, it is classified as 'table-fit'.

As a result of these procedures above, only table-fit triangles are collected. Almost all of these procedures can be vectorized. The vector loop length is the number of the input triangles. It is long enough. Of all the computational load of polygon rendering, typically the half is on this side.

2.6 Scan Conversion

The latter part of our polygon rendering is the scan conversion stage. Here, we only explain how to deal with small triangles marked as 'table-fit', because other big triangles are handled using a conventional algorithm. At this stage, fragments are generated from each of the 'table-fit' triangles, and they are written down into image data.

fragment generation For each triangle, fragments are generated.

With the set of four coordinate component values, tx1, ty1, tx2, ty2, of the triangle, the corresponding section of the look-up table is referred. This section contains all the scan lines of the triangle fill pattern. Each scan line is represented as the TX coordinate component of the starting and the terminating end points. Using them, only fragments passing in-out detection are generated efficiently.

Fragments generated from multiple triangles are stored into the fragment pool of this vectorization session. Usually, tens of thousands fragments are stored at one session to keep the vector loop length long enough and save the workspace memory as minimum. Each fragment data store the source triangle ID where it originated, and its TX and TY coordinates in the triangle coordinate system of the source triangle.

This step is a scalar procedure. It is difficult to vectorize this step because a new fragment has to be checked one by one and added into data arrays. Owing to the look-up table, the number of arithmetic operations is minimized.

fragment calculation For each fragment, RGB color components and a depth value are calculated using the precalculated gradient values of the source triangle of the fragment.

On the color calculation, one dimensional texture mapping is used for producing a smooth contour or a band contour fill pattern.

Fragment-wise clipping check is also performed. Each fragment is tested against the range of image resolution as well as user-specified additional clip planes of arbitrary orientations.

This step can be fully vectorized. The vector loop length is the number of generated fragments in the fragment

pool of this vectorization session. It is long enough.

fragment write For each fragment, the corresponding pixel in the image data is identified. Then, the depth value of the fragment and the one of the pixel, which is stored in the depth buffer, are compared. If it passes the depth test, the fragment color is written down into the image data at the pixel location.

This step is a scalar procedure. It is difficult to vectorize this step because of write dependency on the depth buffer.

The majority of the computational cost is spent on the fragment calculation step. Other two step, fragment generation and fragment write, cannot be vectorized because of data dependency. The look-up table is used to accelerate the former step.

Here is a sample implementation code in C language of the fragment generation, shown below. In this code section, arrays ScanLineMinTx and ScanLineMaxTx of the look-up table are used.

```
/* TC : triangle coordinate */

/* coordinate of vertex1 and vertex2 in TC */
int VecTri_v1Tc[2][MAX_TRIS];
int VecTri_v2Tc[2][MAX_TRIS];
/* box range in TC */
int VecTri_minTc[2][MAX_TRIS];
int VecTri_maxTc[2][MAX_TRIS];

/* the number of fragments */
/* in the fragment pool */
int NFragments;

/* triId : mapping from the fragment */
/* to source triangle */
int Fragment_triId[MAX_FRAGMENTS];

/* tx, ty : coordinate of the fragment in TC */
int Fragment_tx[MAX_FRAGMENTS];
intFragment_ty[MAX_FRAGMENTS];

void MakeFragment (void)
{
int iTri;

  /* NON-VECTORIZABLE LOOP */
  NFragments = 0;
  for (iTri = 0; iTri < NVecTris; iTri++) {
    int tx1 = VecTri_v1Tc[0][iTri];
    int ty1 = VecTri_v1Tc[1][iTri];

    int tx2 = VecTri_v2Tc[0][iTri];
    int ty2 = VecTri_v2Tc[1][iTri];

    int minTy = VecTri_minTc[1][iTri];
    int maxTy = VecTri_maxTc[1][iTri];

    int itx, ity;

    for (ity = minTy; ity <= maxTy; ity++) {
      int scanLineMinTx;
      int scanLineMaxTx;

      scanLineMinTx = ScanLineMinTx
        [tx1][ty1][tx2][ty2][ity];
      if (scanLineMinTx == -1) {
        continue;
      }

      scanLineMaxTx = ScanLineMaxTx
        [tx1][ty1][tx2][ty2][ity];

      assert(0 <= scanLineMinTx);
      assert(scanLineMinTx <= scanLineMaxTx);
      assert(scanLineMaxTx < MAX_TRI_SIZE * 2);

#pragma vdir novector
      for (itx = scanLineMinTx;
           itx<=scanLineMaxTx;
           itx++) {
        assert(NFragments< MAX_FRAGMENTS);
        Fragment_triId[NFragments]= iTri;
        Fragment_tx[NFragments] = itx;
        Fragment_ty[NFragments] = ity;
        NFragments++;
      }
    }
  }
}
```

Array VecTri_v1Tc stores the triangle coordinate (tx1, ty1) at vertex v0, and array VecTri_v1Tc stores (tx2, ty2)at vertex v1. Arrays VecTri_minTc and VecTri_maxTc are the box range of each triangle in its triangle coordinate system. The values in these arrays have already been filled in the previous steps.

Because the fragment generation step is a scalar process, the length of the inner loop in function MakeFragment is too short to be vectorized. All the fragments generated are stored into new arrays,

Fragment_triId, Fragment_tx and Fragment_ty. They represent the source triangle ID, TX and TY coordinates ofthe fragment, respectively.

Here is a sample code section to evaluate the Z depth value at each fragment.

```
void CalculateFragmentDepth (void)
{
  int iFragment;

  /* VECTORIZABLE LOOP */
  /* containing indirect array reference */
  /* and read access */
  for (iFragment = 0;
       iFragment < NFragments;
       iFragment++) {
    int triId = Fragment_triId[iFragment];

    int tx = Fragment_tx[iFragment];
    int ty = Fragment_ty[iFragment];

    int dx = tx - MAX_TRI_SIZE;
    int dy = ty - MAX_TRI_SIZE;

    /* z value interpolation */

    float vz0 = VecTri_v0Dc[2][triId];

    float za = VecTri_za[triId];
    float zb = VecTri_zb[triId];

    float z = vz0 + za * dx + zb * dy;

    Fragment_zDc[iFragment] = z;
  }
}
```

Array VecTri_v0Dc stores the x, y and z component values at vertex v0 in the screen coordinate system. Arrays VecTri_za and VecTri_za are the gradient coefficients to specify the depth value distribution over the source triangle. The Z depth value of each fragment is stored in array Fragment_zDc.

All the fragment-wise loops, such as the code section in function CalculateFragmentDepth, are vectorized. If any data about the source triangle are required, the information is accessed using indirect index reference.

3. Performance Evaluation

For the implementation of our off-line visualization system, we used an open-source CAE system, ADVENTURE [4]. AutoGL library in ADVENTURE Auto module in ADVENTURE system is used for the implementation base of the polygon renderer [9].

Using this off-line visualizer, a performance evaluation is carried out on ES. Here, the vectorization performance of the polygon renderer running on a single processor of ES is shown.

As a benchmark problem, we prepared an artificial problem to render a scalar contour plot on semi-cylindricalsurface sheets. The model consists of 8 semi-cylindrical sheets placed in depth order and each of the sheets is composed of a 250 by 250 grid. In total, the surface patch is composed of 1 million triangles. The distribution of a scalar quantity over the semi-cylindrical sheets is artificially made so that the magnitude of the scalar value is proportional to the distance from the center of each sheet. Fig. 5 shows the benchmark model. In this case, the resolution of the imageis 1,000 by 1,000. An average pixel size of the triangles is about 5 by 5 pixels.

To render this image data on ES using a single Arithmetic Processor (AP), it takes 0.82 seconds to render 1 million triangles. Rendering performance of 1.22 million triangles per seconds is obtained. Acceleration ratio by vectorizationis about 4.2 times.

Here, as an example of the problem with a large number of triangles, a Pantheon model was performed shown in Fig. 6. Though detail of the model is described in the section 4, since the Pantheon model consists of about 4.1 million solid tetrahedron elements, total number of triangles amounts to 16.6 million when rendering in four faces of all elements. In rendering such large number of polygons model, our system achieved rendering performance of 2.33 million triangles per seconds, and acceleration ratio by vectorization of 4.9 times.

Next, we further investigated the performance bottleneck of our vectorized code using ftrace utility offered by the ESC. Table 2 and 3 show performance analysis of each rendering step using ES ftrace utility.

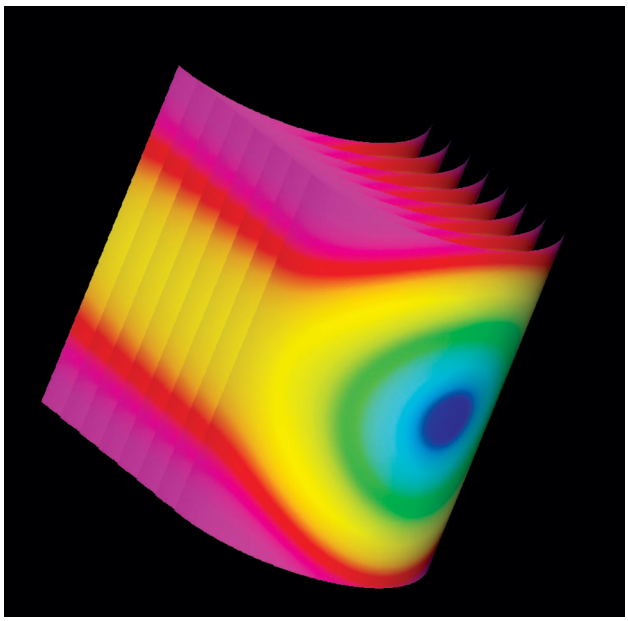

Fig. 5 Scalar contour plots on semi-cylindrical sheets as a benchmark problem, which has 1 million triangles.

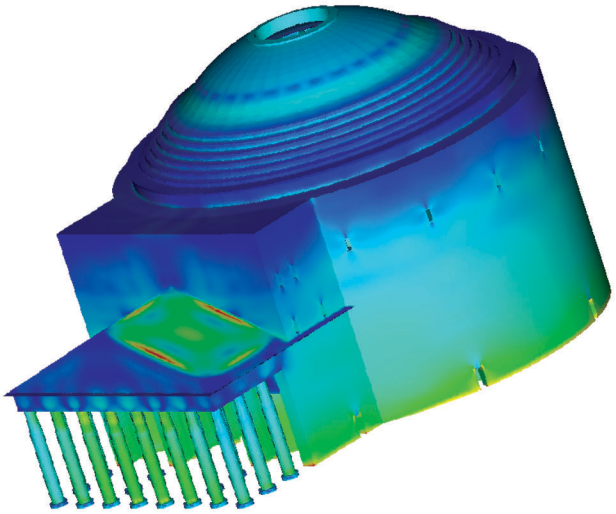

Fig. 6 Deformation and stress contour plots of a Pantheon model with 18 million DOFs.

Table 2 Performance analysis of rendering steps. In this table, Vec. means whether a step can be vectorized, and vec-off/on show runtime performances with vectorization off/on.

Rendering step	Vec.	Time vec-off (sec.)	Time vec-on (sec.)	Accel. ratio
transform and light	Y	0.982	0.043	24
clip triangle	Y	0.011	0.003	4
depth gradient	Y	0.121	0.010	12
scalar gradient	Y	0.172	0.015	12
range and TC	Y	0.197	0.008	26
collect table-fit	Y	0.314	0.053	13
generate fragment	N	0.289	0.288	1
interpolate z	Y	0.427	0.036	12
interpolate color	Y	0.617	0.068	9
write fragment	N	0.295	0.296	1

Table 3 Performance analysis of rendering a Pantheon model with 16.6 million triangles.

Rendering step	Vec.	Time vec-off (sec.)	Time vec-on (sec.)	Accel. ratio
transform and light	Y	10.712	0.663	17
clip triangle	Y	0.187	0.076	3
depth gradient	Y	2.154	0.171	13
scalar gradient	Y	2.901	0.216	14
range and TC	Y	2.615	0.149	18
collect table-fit	Y	4.100	0.798	9
generate fragment	N	2.693	2.687	1
interpolate z	Y	3.338	0.259	13
interpolate color	Y	4.661	0.581	8
write fragment	N	1.513	1.511	1

There are mainly ten rendering steps in our algorithm. All the steps before generate fragment step are triangle-wise vectorized calculations, and two steps, interpolate z and interpolate color, are fragment-wise vectorized calculations. However, fragment generation and fragment write steps, which are in the scan conversion stage, are not vectorized.

When run with vectorization off, no vectorized steps occupy 17 and 12 percents of total execution time in 1 million triangles and 16.6 million triangles model, respectively. With vectorization on, vectorized steps are accelerated about 10 times or more, while they remain the same. As a result, they are shown up as the bottleneck. The fragment generation step, which is accelerated by our look-up table approach, occupies about 71 or 59 percents of the bottleneck steps.

We also prepared a scalar-tuned version of the off-line visualizer, which is mainly intended for PC, WS and PC clusters. And, in rendering a benchmark model, the vectorized version is 4.3 times faster than the scalar-tuned version, when they are executed on ES. One reason is that the ES processor runs at 500MHz and its scalar performance is relatively slow compared with its vector performance. This means, it is still better to prepare a vectorized version optimized for ES or other vector machines than to bring a scalar version running on other scalar-processor platform and keep using it on ES.

4. Visualization Examples

Here, some visualization examples of huge scale 3-D structural analyses are demonstrated.

First, Pantheon in Roma, Italy is analyzed. The total DOFs in FE model is 18 million. The surface patch of the Pantheon model consists of 0.65 million triangles. A gravity force is applied as a body force. The result of elasto-static analysis is visualized using 8 APs of ES. Runtime performance of visualization with generating 100 image data from various angles is about 218 seconds. Fig. 6 and Fig. 7 are deformation and equivalent stress contour plots of a Pantheon model.

The next example is an earthquake response simulation of a full-scale nuclear power plant. It is a Boiled Water Reactor (BWR). The FE model contains 200 million DOFs. The surface patch consists of 16.7 million triangles. A horizontal acceleration force is applied as an earthquake load. The visualization with 100 images of elasto-dynamic analysis results is successfully performed in 7.5 minutes using 32 APs of ES. Fig. 8 and Fig. 9 are deformation and equivalent stress contour plots of a BWR model. In Fig. 9, the amount of deformation is magnified by 5,000 times.

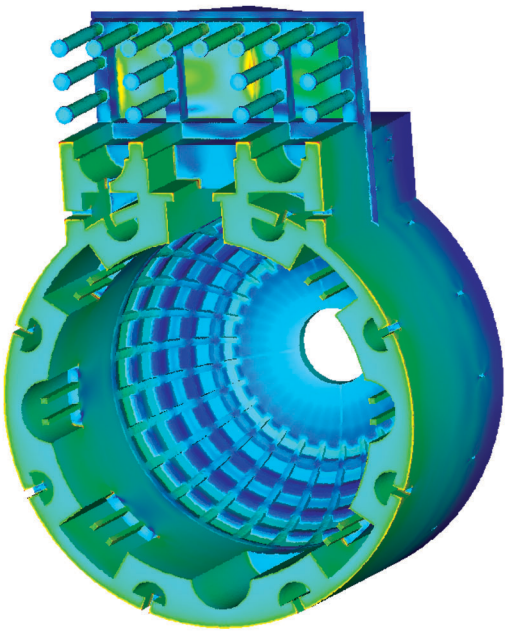

Fig. 7 Bottom view of a Pantheon model with deformed configuration and stress distribution.

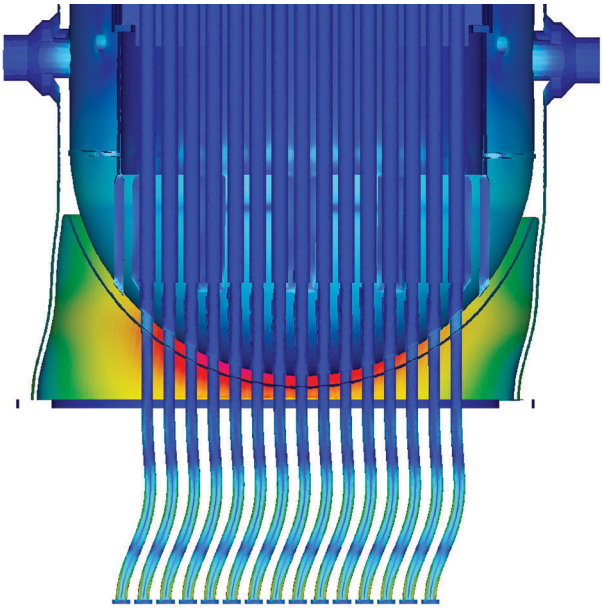

Fig. 9 Cross section around a skirt portion of a BWR model. The amount of deformation in this figure is magnified by 5,000 times.

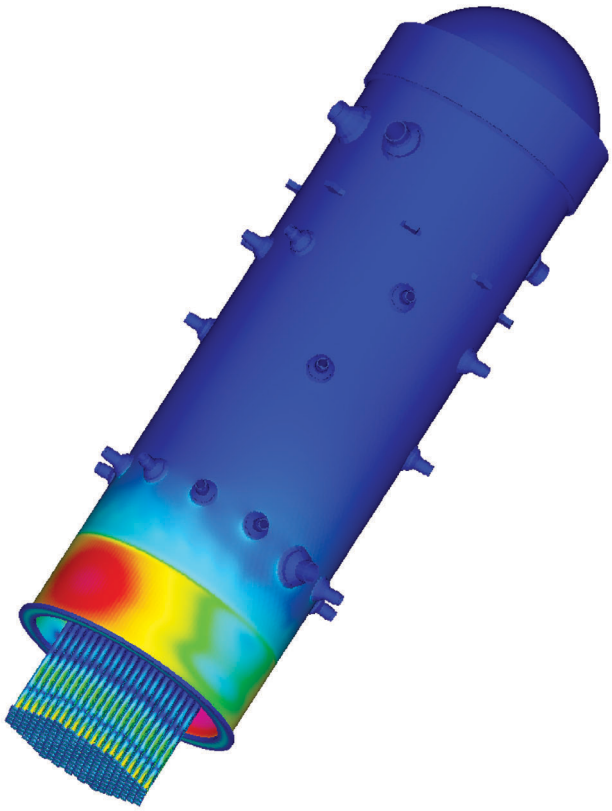

Fig. 8 Deformation and stress contour plots of a BWR model with 200 million DOFs.

5. Conclusions

An off-line visualizer is developed for visualization of a huge scale 3-D structural analysis on ES. The off-line visualize performs rendering of surface patch triangles to produce image files of deformation plot and stress contour plot. It is vectorized and parallelized. The vectorization scheme of its polygon rendering using a look-up table is explained in detail, and its performance evaluation on ES is carried out. Rendering performance of 1.22 million triangles per seconds is obtained per a single processor of ES. Acceleration ratio by vectorization is about 4.2 times. And it is 4.3 times faster than the corresponding implementation specially tuned for a scalar-type processor. It is also demonstrated that a 3-D structural analysis over 200 millions DOFs can be visualized efficiently using our off-line visualizer.

6. Acknowledgement

One of the authors (Hiroshi Kawai) was financially supported by The 21st Century COE Program, 'System Design: Paradigm Shift from Intelligence to Life', of the Ministry of Education, Science and Culture of Japan.

(This article is reviewed by Dr. Horst D. Simon.)

References

[1] M. Ogino, R. Shioya, H. Kawai, and S. Yoshimura, Seismic Response Analysis of Nuclear Pressure Vessel Model with ADVENTURE System on the Earth

Simulator, *Journal of The Earth Simulator*, vol.**2**, pp.41–54, 2005.

[2] F. Araki, H. Uehara, N. Ohno, and S. Kawahara, Visualization of Large-scale Data Generated by the Earth Simulator, *Journal of The Earth Simulator*, vol.**6**, pp.25–34, 2006.

[3] J. S. Painter, H. -P. Bunge, and Y. Livnat, Case Study: Mantle Convection Visualization on the Cray T3D, *IEEE Visualization 96*, p.409, 1996.

[4] http://adventure.q.t.u-tokyo.ac.jp/

[5] H. Kawai, M. Ogino, R. Shioya and S. Yoshimura, Parallel Visualization of Huge Scale 3-D Structural Analysis Using Server-side Rendering and Walkthrough, *APCOM'07*, Dec 3–6, 2007, Kyoto, Japan.

[6] N. L. Max, Vectorized Procedural Models for Natural Terrain: Wave and Islands in the Sunset, *ACM SIGGRAPH*, vol.**15**, no.3, pp.317–324, 1981.

[7] http://geofem.tokyo.rist.or.jp/

[8] Y. Okuda and K. Nakajima, *Parallel Finite Element Analysis [I] Cluster Computing*, Baifukan, Japan, 2004 (in Japanese).

[9] H. Kawai, ADVENTURE AutoGL: A Handy Graphics and GUI Library for Researchers and Developers of Numerical Simulations, *CMES*, vol.**11**, no.3, pp.111–120, 2006.

[10] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, A Sorting Classification of Parallel Rendering, *IEEE CG&A*, vol.**14**, no.4, pp.23–32, 1994.

[11] http://www.sw.nec.co.jp/APSOFT/SX/rvslib/

[12] S. Doi, T. Takei, and H. Matsumoto, NEC Supercomputer Visualization Efforts, *IEEE CG&A*, pp.10–13, 2001.

[13] Y. Suzuki, K. Sai, F. Araki, H. Uehara, and H. Haginoya, Development of a Visualization Scheme for Large-scale Data on the Earth Simulator, *ES Annual Report, April 2002–March 2003*, pp.139–144, 2003.

[14] F. A. Ortega, C. D. Hansen and J.P. Ahrens, Fast Data Parallel Polygon Rendering, *SC*, pp.709–718, 1993.

[15] C. D. Hansen, M. Krogh, and W. White, Massively Parallel Visualization: Parallel Rendering, *Proceedings of SIAM Parallel Computation Conference*, pp.790–795, 1995.

[16] S. Whitman, A Load Balanced SIMD Polygon Renderer, *ACM SIGGRAPH*, pp.63–106, 1995.