Real-Time Marching Cubes on the Vertex Shader

Frank Goetz†, Theodor Junklewitz and Gitta Domik

Research Group Computer Graphics, Visualization and Image Processing, University of Paderborn, Germany

Abstract

In this paper we propose a new approach for visualizing volumetric datasets by their isosurfaces. For an interactive isosurface reconstruction an optimized version of the well-known marching cubes algorithm is used. We extend the original algorithm by an additional vertex shader program. Contrary to other hardware-accelerated solutions our program is not based on a tetrahedral algorithm and thus the implementation for structured grids is more effective. Furthermore, surfaces of time-varying datasets at distinguished threshold values can be extracted in real-time.

Categories and Subject Descriptors (according to ACM CCS): I.3.3 [Computer Graphics]: Isosurface computation and rendering

1. Introduction

The visualization of volumetric datasets by their isosurfaces (e.g. retrieved from computer tomography (CT), magnet resonance tomography (MRT), or positron emission tomography (PET)) is an interesting and useful method to represent unknown data (Figure 1). This can help to gain insights into the dataset to detect diseases in a very early phase. Until now a reconstruction of such surfaces was not possible in real-time, though a real-time solution is necessary to work interactively. In this case "interactively" means that the threshold value, determining the represented surface, can be changed on the fly.

2. Related Work

Nearly all approaches that try to enhance interactive reconstruction of isosurfaces by using vertex or fragment shader programs are based on a generalization of the marching cubes algorithm [LC87], the so called tetra-cubes algorithm [CSK96]. Instead of calculating up to five surfaces in a single grid cell (cube) they calculate up to one surface in up to six tetrahedrons per grid cell of a structured cartesian grid. An advantage of the tetrahedral-based solutions is that they can easily be adapted to unstructured grids. Currently, this additional feature is not important for us since we are only working with regularly structured grids.

† frank.goetz@uni-paderborn.de

A tetrahedral solution based on vertex shaders is described by Pascucci in [Pas04] and Reck et al. in [RDG*04]. Klein, Stegmair, and Ertl presented an improved and fragment-based version using *ATI's SuperBuffers* in [KSE04]. Other hardware-accelerated volume rendering techniques are not based on polygons: e.g. texture-based volume rendering can be used to display shaded surfaces [WE98, RSEB*00]. For unstructured grids a hardware-accelerated cell-projection technique is available [RKE00]. In the last two years hardware-accelerated ray-casting methods became popular, too. Different approaches in this area can be found in [RGW*03], [WKME03], [KW03], and [RW04].

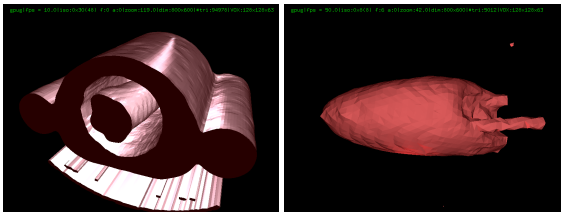

Figure 1: *CT of human phantom & PET of heart ventricle (datasets from Heart & Diabetes Center NRW)*

The main problem of all direct volume rendering methods is that the isosurface is computed on a per-pixel basis without constructing a polygon mesh. Thus, a lot of common polygon-based computer graphics algorithms – like texture

mapping, bump mapping, environment mapping, or even shading – cannot (easily) be used with these approaches.

3. Implementation

Our implementation of the isosurface reconstruction focuses on time-varying structured volumetric datasets. In our application users like scientists or engineers have the possibility to change threshold values on the fly (Figure 2), allowing them to get the resulting image immediately. This is ensured by a regular recalculation of the polygon mesh that is constructed from the underlying dataset. Our current implementation supports only volumetric datasets with values from 0 to 255.

The fastest way to display polygon meshes of static datasets at distinguished threshold values is using appropriate caching methods like *VertexBufferObjects*, *VertexArrays*, or *Display Lists*. However, by using such methods the recalculation of the polygon mesh is very time consuming when the threshold value and/or the dataset are changed. Therefore, to get a better performance when reconstructing the surface, we take advantage of the current graphics hardware. In our special case we need *Shader Model 3.0* to make a 2D texture lookup in the vertex shader.

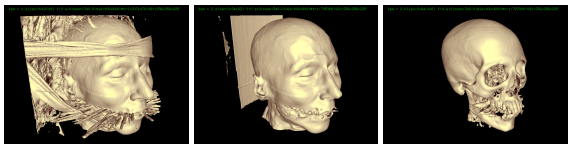

Figure 2: *Skull displayed with three different threshold values (dataset from* volvis.org*)*

3.1. Marching Cubes

A lot of marching cubes implementations pass each grid cell in turn to the algorithm until a transition, the surface, is found. In contrast we iterate over all cells of the dataset (algorithms that follow the isosurface across the dataset are not suited for a hardware-accelerated implementation). Our implementation is adapted from Paul Bourke's article: "Polygonising a scalar field" [BBC97]. The value of each vertex is compared against the threshold to label it as 'inside' or 'outside'. After this, the list determining if vertices are inside/outside the volume is used to determine which edges and where the surface intersects. The algorithm is well suited for parallel processing, because each grid cell can be calculated independently. The location of where the surface intersects the given edge is found by linearly interpolating between the isovalues of the vertices at either end of the edge. With 256 different combinations of vertices up to five triangles can be created. Since we are using a *marching cubes code* lookup table, which describes the 256 possible intersections of the triangles with the 12 edges of a cube, it is not necessary to reduce the problem to only 15 cases, like other solutions do. Due to the determination of vertices that lie inside/outside the volume also grid cells that consist of no surfaces can directly be rejected without sending them to the GPU. This reduces the bus transfer to the graphics board enormously, particularly if the isosurface consists of only a few triangles. Furthermore, by using the *marching cubes code* we know the necessary number of triangle-points within a grid cell. These numbers are stored in an additional table. The eight scalar values of the vertices and the index of the current point, which describes one of the 256 possible cases for having a polygon mesh inside a grid cell, are stored in three integer values. The current grid coordinate (position of the current cell in the grid) is stored in another integer value. Now, one triangle-point on the vertex shader can be calculated with one *glVertex4i*.

In an earlier implementation the dataset was stored in a 3D texture. Since the 3D texture lookups in the vertex shader are emulated in software and thereby are very time-consuming, we save a lot of GPU time with our latest solution. Use of 2D textures for volumetric 3d datasets is not recommended either, because the accuracy of floating points would be insufficient to index large datasets.

3.2. Vertex Shader

At this point, the vertex shader which distinguishes this solution from other implementations comes into play. The data that was sent with *glVertex4i* is unpacked, e.g. the current grid coordinates are extracted into temporary variables. The same is done with the scalar values and the index of the current point. After that, the necessary *marching cubes code* is calculated again. The *marching cubes code* will not be sent in a free vertex attribute to the vertex shader because this is inefficient and costs more GPU time. Here the *Shader Model 3.0* is necessary because we store the *marching cubes code* table in a 2D texture. In a *for* loop, iterating the scalar values of all 12 edges, the exact intersections for the triangle-points are determined and stored. Only the current point and two additional points for the surface-normal calculation are necessary. Subsequently, a 2D texture lookup is accomplished to assign the earlier found intersection-point to the current triangle-point. The 2D texture lookup uses, as indices, only the *marching cubes code* and the computed index of the current point as parameter. At this point the calculation is nearly finished, if no further surface-normal computation is necessary. Only a matrix multiplication with the ModelView-Projection matrix, which is stored as a uniform parameter, has to be calculated.

For the surface-normal calculation two more edges have to be looked up in the 2D texture (*marching cubes code* table) in correct order. The normal vector at each triangle-point has to point into the correct direction. Finally, the surface-normals can be used to calculate the diffuse and specular lighting as shown in Figure 3.

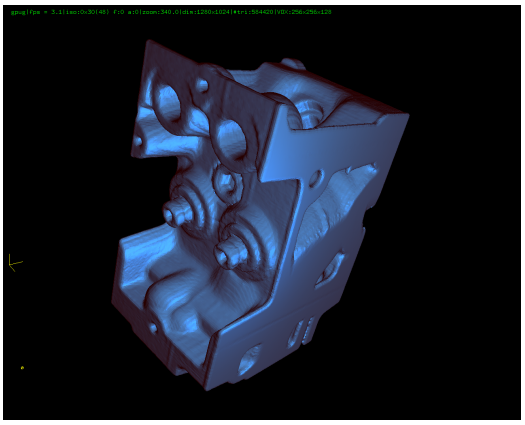

Figure 3: *Engine block with diffuse and specular lighting (dataset from* volvis.org*)*

3.3. Time-Varying Datasets

Since we are not using 3D textures, 2D textures, or *VertexArrays* and since we send the parameters of each grid cell to the vertex shader separately, it is very easy to extend our implementation to the use of time-varying datasets. These datasets are previously loaded into the main memory. They can be loaded continuously from the hard drive, if not enough main memory is available.

4. Results

For our tests three different system configurations were used. A graphics board comparison was accomplished on an Intel Pentium IV system with two different NVIDIA GeForce 6 boards (Table 1). Furthermore, an optimized CPU version of our algorithm was compared to the GPU version on another Intel Pentium IV 2.0 GHz system. It is pointed out that the clock speed of the Intel CPU (2000 MHz) is nearly six times higher than the clock speed of the NVIDIA GPU (350 Mhz), but the NVIDIA Geforce 6800GT board has six vertex shader units (Table 2).

Table 1: *Comparison between GeForce 6600GT & GeForce 6800GT in frames per second (both on Pentium IV 2.8 GHz)*

Dataset	Size	Triangles	GPU	GPU
skull	256x256x225	829978	1.4	2.4
engine	256x256x110	584420	1.9	3.6
neghip	064x064x064	31456	28.0	61.6
phantom	128x128x063	94978	10.0	18.6
heart	128x128x063	3982	49.2	50.0

Table 1 shows that the performance of the NVIDIA GeForce 6800GT is in some cases twice as fast as the NVIDIA GeForce 6600GT because the NVIDIA GeForce 6600GT has only three instead of six vertex units.

Table 2: *Comparison between CPU and GPU version in frames per second (both on Pentium IV 2.0 GHz)*

Dataset	Size	Triangles	CPU	GPU
skull	256x256x225	829978	1.1	1.9
engine	256x256x110	584420	1.8	3.1
neghip	064x064x064	31456	49.7	56.8
phantom	128x128x063	94978	14.8	16.7
heart	128x128x063	3982	43.3	45.7

Table 2 shows that the GPU version is faster than the CPU version. Especially with large datasets enormous differences are reached. The skull and the engine are nearly one third faster on the GPU than on the CPU. By using small datasets the differences between GPU and CPU version are not worth mentioning.

5. Conclusion

Currently, we integrate our marching cubes algorithm into OpenSG (an open source and platform independent scene graph system). This makes our implementation available on various platforms like Linux, IRIX, Windows, or Mac OS X. While reconstructing isosurfaces instead of using other volume rendering techniques a lot of common polygon-based computer graphics methods like texture mapping and (Phong) shading can be used with our approach. One benefit of our algorithm is that no preprocessing step (e.g. conversion of the dataset) is necessary. The application would even work if only a continuous data stream from an external device is provided. Our approach automatically improves with an increasing amount of parallel shaders in next generation GPUs (XBox360 game console from Microsoft has 48 parallel shaders capable of operating on data for both pixels and vertices). Finally we should mention, that especially on slow systems our approach is an alternative to updating the current computer, by only buying an off-the-shelf graphics board.

References

[BBC97] BOURKE P., BLOYD C. G., CHANDRASHEKARA R.: Polygonising a scalar field from http://astronomy.swin.edu.au/~pbourke.

[CSK96] CARNEIRO B. P., SILVA C., KAUFMAN A.: Tetra-cubes: An algorithm to generate 3d isosurfaces based upon tetrahedra. In *Proc. SIGGRAPH '96* (1996), pp. 205–210.

[KSE04] KLEIN T., STEGMAIER S., ERTL T.: Hardware-accelerated reconstruction of polygonal isosurface representations on unstructured grids. In *Proc. Pacific Graphics 2004* (2004), pp. 186–195.

[KW03] KRUEGER J., WESTERMANN R.: Acceleration techniques for gpu-based volume rendering. In *Proc. IEEE Visualization '03* (2003).

[LC87] LORENSEN W. E., CLINE H. E.: Marching cubes: A high resolution 3d surface construction algorithm. *Computer Graphics 21*, 4 (July 1987), 163–169.

[Pas04] PASCUCCI V.: Isosurface computation made simple: Hardware acceleration, adaptive refinement and tetrahedral stripping. In *Proc. Eurographics/IEEE TVCG Symposium on Visualization '04* (2004), pp. 293–300.

[RDG*04] RECK F., DACHSBACHER C., GROSSO R., GREINER G., STAMMINGER M.: Realtime isosurface extraction with graphics hardware. In *Proc. Eurogrpahics 2004 Short Presentations* (2004), pp. 33–36.

[RGW*03] ROETTGER S., GUTHE S., WEISKOPF D., ERTL T., STRASSER W.: Smart hardware-accelerated volume rendering. In *Proc. VisSym 2003* (2003), pp. 231–238.

[RKE00] ROETTGER S., KRAUS M., ERTL T.: Hardware-accelerated volume and isosurface rendering based on cell-projection. In *Proc. IEEE Visualization '00* (2000), pp. 109–116.

[RSEB*00] REZK-SALAMA C., ENGEL K., BRAUER M., GREINER G., ERTL T.: Interactive volume rendering on standard pc graphics hardware using multi-textures and multi-stage rasterization. In *Proc. Graphics Hardware '00* (2000).

[RW04] REIS G., WAGNER D.: Hardware accelerated, interactive, time-dynamic volume rendering. In *Proc. VIIP'04* (2004), pp. 602–607.

[WE98] WESTERMANN R., ERTL T.: Efficiently using graphics hardware in volume rendering applications. In *Proc. SIGGRAPH '98* (1998), pp. 169–177.

[WKME03] WEILER M., KRAUS M., MERZ M., ERTL T.: Hardware-based ray casting for tetrahedral meshes. In *Proc. IEEE Visualization '03* (2003), pp. 333–340.

Appendix

```
//** CPU SOURCE CODE OF THE OPENGL DISPLAY LOOP **//
void display(void)
{
  int pos, eFlag, m0, m1, m2, mc_code = 0;
  static unsigned char mc[8];
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  triCount = 0;
  setcamera();
  cgGLEnableProfile(cgVertexProfile);
  glBegin(GL_TRIANGLES);
// scan the complete volume cell by cell (t = time step)
  for (int z=0; z< (VOX_Z - 1); z++)
  {for (int y=0; y < (VOX_Y - 1); y++)
   {for (int x=0; x < (VOX_X - 1); x++)
    {mc[0]=vol[t][ z ][ y ][ x ];mc[1]=vol[t][ z ][ y ][x+1];
     mc[2]=vol[t][ z ][y+1][x+1];mc[3]=vol[t][ z ][y+1][ x ];
     mc[4]=vol[t][z+1][ y ][ x ];mc[5]=vol[t][z+1][ y ][x+1];
     mc[6]=vol[t][z+1][y+1][x+1];mc[7]=vol[t][z+1][y+1][ x ];
     mc_code = 0;
   // calculate the marching cubes code
     for(int vCube = 0; vCube < 8; vCube++)
     {if(mc[vCube] <= (iISO)) mc_code |= 1«vCube;}
   // which edges are intersected ?
     eFlag = eFlags[mc_code];
   // if no edge is intersected
     if(eFlag == 0) continue_with_next_grid_cell;
   // else pack data for the vertex shader
     m0=mc[0]|(mc[1]«8)|(mc[2]«16);
     m1=mc[3]|(mc[4]«8)|(mc[5]«16);
     m2=mc[6]|(mc[7]«8); pos=x|(y«8)|(z«16);
     for (int w=0; w<countTriTab[mc_code]; w++)
     {glVertex4i(pos, m0, m1, m2|(w«16));}
     triCount+= (w/3); }
    }
  }
  glEnd();
  cgGLDisableProfile(cgVertexProfile);
  glutSwapBuffers();
}
```

```
//** CG SOURCE CODE OF THE VERTEX SHADER PROGRAM **//
#define NUMBER_OF_EDGES 12
void main(
  float4 in_Pos:Pos, out float4 hpos:Pos, out float4 col:Col,
  uniform float4x4 modelViewP, uniform float4x4 imodelViewP,
  uniform float3 lightPos,     uniform float3 lightCol,
  uniform float3 eyePos,       uniform float shininess,
  uniform float3 gridOffset[8],uniform float3 edgeDirect[12],
  uniform float iso,           sampler2D trisEdgeTab)
{
  int powerOfTwoArray[12]={1,  2,   4,   8,   16,   32,
                           64, 128, 256, 512, 1024, 2048};
  int mapEdgeToVert[12][2] = { {0,1}, {1,2}, {2,3}, {3,0},
                               {4,5}, {5,6}, {6,7}, {7,4},
                               {0,4}, {1,5}, {2,6}, {3,7}};
  float3 gridPos; float voxVal[8]; float3 VertPos[12];
// get the current grid cell at x,y,z
  gridPos.x = floor(fmod(in_Pos.x,         0x100));
  gridPos.y = floor(fmod(in_Pos.x/0x100,   0x100));
  gridPos.z = floor(fmod(in_Pos.x/0x10000, 0x100));
// get the 8 voxel values of the current grid cell
  voxVal[0] = floor(fmod(in_Pos.y,         0x100));
  voxVal[1] = floor(fmod(in_Pos.y/0x100,   0x100));
  voxVal[2] = floor(fmod(in_Pos.y/0x10000, 0x100));
  voxVal[3] = floor(fmod(in_Pos.z,         0x100));
  voxVal[4] = floor(fmod(in_Pos.z/0x100,   0x100));
  voxVal[5] = floor(fmod(in_Pos.z/0x10000, 0x100));
  voxVal[6] = floor(fmod(in_Pos.w,         0x100));
  voxVal[7] = floor(fmod(in_Pos.w/0x100,   0x100));
// which point from 0 to 15 should be calculated ?
  float triVert=floor(fmod(in_Pos.w/0x10000,0x100));
// calculate marching cubes code
  int mc_code = 0;
  for(int i = 0; i < 8; i++)
   if(voxVal[i] <= iso) mc_code += powerOfTwoArray[i];
  for(int i = 0; i < NUMBER_OF_EDGES; i++) {
   float voxVal1 = voxVal[mapEdgeToVert[i][1]];
   float voxVal0 = voxVal[mapEdgeToVert[i][0]];
   float diff = voxVal1 - voxVal0;
   float mid  =(iso - voxVal0);
   VertPos[i] = gridPos.xyz +
   gridOffset[mapEdgeToVert[i][0]]+mid*edgeDirect[i]/diff;
  }
  int triFirstEdgeOffset=(floor(triVert/3.0)*3);
  int triEdges[3];
  for (int i=0; i<3; i++) triEdges[i]=fmod(triVert+i,3);
// get 3 triangle edges from 2D texture
  int edge[3];
  for (int i=0; i<3; i++) edge[i] = (tex2D(trisEdgeTab,
    float2((triFirstEdgeOffset + triEdges[i])
    * (1.0/16.0), mc_code * (1.0/256.0))).r) * 255.0;
  float3 triVertPos0,triVertPos1,triVertPos2;
  for(int i=0; i<NUMBER_OF_EDGES;i++){if(edge[0]==i)
    triVertPos0 = float3(VertPos[i]);}
  hpos = mul(modelViewP, float4(triVertPos0, 1));
// calc normal ... (code here)
// calc diffuse lightning ... (code here)
// calc specular lightning ... (code here)
}
```