한국교통대학교 융·복합기술연구소 논문집, Vol. 10, No. 1, pp. 7-12 (2020)
https://doi.org/10.22710/JICT.2020.10.1.007

ISSN : 2233-8667

효율적인 폴리곤 곡면 재건 알고리즘

박 상 근*

한국교통대학교 기계공학

An Efficient Polygonal Surface Reconstruction

Sangkun Park*

Department of Mechanical Engineering, Korea National University of Transportation,
Daehak-ro 50, Chungju-si, Chungbuk 380-702, Korea
(Received 2020.08.25 / Accepted 2020.10.30)

Abstract : We describe a efficient surface reconstruction method that reconstructs a 3D manifold polygonal mesh approximately passing through a set of 3D oriented points. Our algorithm includes 3D convex hull, octree data structure, signed distance function (SDF), and marching cubes. The 3D convex hull provides us with a fast computation of SDF, octree structure allows us to compute a minimal distance for SDF, and marching cubes lead to iso-surface generation with SDF. Our approach gives us flexibility in the choice of the resolution of the reconstructed surface, and it also enables to use on low-level PCs with minimal peak memory usage. Experimenting with publicly available scan data shows that we can reconstruct a polygonal mesh from point cloud of sizes varying from 10,000 ~ 1,000,000 in about 1~60 seconds.

Key words : polygonal surface, 3D convex hull, signed distance function, marching cubes, surface reconstruction

1. 서 론

3차원 측정점 샘플로부터 3차원 폴리곡 곡면을 재건하는 알고리즘은 전산 기하학, 컴퓨터 그래픽스, 역설계 공학 등의 분야에서 꾸준히 연구되어 왔다. 관련 알고리즘은 3차원 스캐너 장치에 의해 측정된 점 데이터를 피팅하고, 곡면의 구멍을 채우고, 기존에 생성된 곡면을 리메싱하는 등 그 사용 영역이 적지 않다[1]. 그러나 기존에 발표된 폴리곡 곡면 재건 기법을 살펴보면 구현 알고리즘이 매우 복잡하며 상당한 계산 시간을 필요로 한다[1].

관련 연구를 살펴보면 크게 두 가지 방식의 알고리즘으로 나눌 수 있다. 하나는 기하학적 계산에 의한 재건 방법이고, 다른 하나는 음함수(implicit) 방식을 이용한 재건 방법이다.

기하학적 방법으로서 가장 널리 알려진 방법 중에 하나가 PowerCrust[2]이다. 이 방법은 이론적으로 완성된 방법으로서 점 데이터에 밀착된 곡면을 생성한다. 그러나 점 데이터의 밀도 분포가 균일하지 않을 경우에 부적합하다는 평가를 받고 있다.

음함수 방식으로서 Poisson[3], Hoppe[4], MPU[5]가 널리 알려져 있으며, VTK, PCL, MeshLab 등의 라이브러리에 구현되어 현재 가장 많이 사용되고 있다. 이들은 모두 오픈 소스이며 누구나 연구용 등으로 자유롭게 사용할 수 있다. 음함수 방식은 장점으로 노이즈 데이터, 점 데이터의 비균일 분포, 나아가 구멍 등에 크게 영향을 받지 않는다. 그러나 단점으로 점 데이터 외에 점 데이터 위치에서의 법선 벡터가 요구된다.

본 연구 알고리즘은 음함수 접근 방식을 사용하여 곡면을 재건한다. 임의의 한 점과 점 데이터 사이의 최소 거리를 계산하고 그 점이 재건할 곡면의 내부에 존

*Corresponding author, E-mail: skpark@ut.ac.kr.

재하면 음수를, 외부에 존재하면 양수를, 경계면 상에 존재하면 0을 부여하는 방식으로 부호화 거리 함수 SDF (signed distance function)을 생성한다. 그리고 마칭 큐브(marching cubes) 알고리즘을 사용하여 생성된 SDF로부터 폴리곤 곡면을 생성한다. 본 연구도 기존 연구와 유사하게 단점으로서 법선 벡터가 요구된다. 그러나 장점으로서 노이즈, 비균일 점 데이터, 구멍 등에 영향을 받지 않는다.

본 연구 알고리즘에 관한 자세한 설명은 2장에 있으며, 3장은 수치 실험 및 성능에 관해 기술하며, 마지막으로 4장에 결론이 있다.

2. 재건 알고리즘

3차원 점 데이터(입력)로부터 폴리곤 곡면(출력)을 재건하는 본 연구 알고리즘의 전체 과정은 Fig. 1과 같으며 다음의 순서에 의해 진행된다. 참고로 본 연구에서 개발한 Javascript 코드는 부록 A와 같다.

○ **전체 진행 과정**

1) 격자 구조 생성
2) 볼록 집합(3D convex hull) 생성
3) 옥트리(octree) 생성
4) 부호화 거리 함수(SDF) 생성
5) 마칭 큐브에 의한 곡면 생성

○ **격자 구조**

일정한 간격의 격자 구조를 생성하기 위해 3차원 점 데이터로부터 최소 한계 박스(AABB)를 구하고, 다시 1.2배 정도 확대시켜 폴리곤 곡면이 존재할 한계영역을 구한다.

그리고 이 한계영역을 x축 방향, y축 방향, z축 방향의 해상도 값으로 나누어 각 축의 방향 별로 격자 셀의 크기를 구한다. 여기서 해상도 값은 사용자의 입력 값으로 격자 셀의 개수를 의미한다.

위에서 구한 격자 셀의 크기는 격자점의 위치를 계산하기 위해 사용되며, 격자점의 위치는 인덱스만으로 계산한다. 결국 본 연구에서 격자 구조는 메모리 사용을 통한 격자점의 저장 방식이 아닌 격자 점 셀의 크기와 인덱스를 통하여 계산하는 방식으로 메모리 사용량의 최소화하였다.

○ **3차원 볼록 집합**

3차원 점 데이터로부터 3차원 볼록 집합[6]을 생성한다. 볼록 집합이 필요한 이유는 계산 시간을 줄이기 위함이다. 즉 임의의 한 점이 볼록 집합의 외부에 존재하는지를 쉽게 확인할 수 있다. 그 과정을 간략히 살펴보면 다음과 같다.

1) 3차원 점 데이터와 검사 점을 각 축 방향으로 투영시킨다.

2) 3차원 점 데이터의 투영 점에서 2차원 볼록 집합을 구한다.

3) 투영된 검사 점이 2차원 볼록 집합의 내부인지 외부인지 확인한다.

4) 3개 투영면에서의 결과가 모두 외부이면 검사 점은 3차원 볼록 집합의 외부에 존재하게 된다.

○ **옥트리**

위에서 계산된 한계 영역과 입력된 옥트리 깊이에 의해 옥트리[7]를 초기화한다. 초기화된 옥트리에 3차원 데이터 점을 삽입하여 옥트리를 완성한다. 옥트리가 필요한 이유는 임의의 한 점에서 입력점까지의 최소 거리를 신속히 계산하기 위함이다.

○ **부호화 거리**

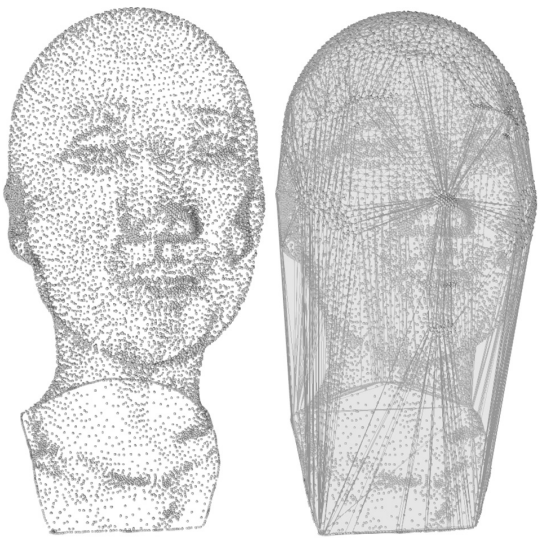
인덱스에 의해 계산된 격자점 x에서 측정점 데이터까지의 부호화 거리를 식 (1)과 같이 계산한다.

$$d = (x - p_i) \cdot n_i \tag{1}$$

여기서 d는 부호화 거리이며, p_i는 최소 거리에 해당하는 점이고, n_i는 p_i에 대응하는 법선 벡터이다. 모든 격자점에서 d가 계산되면 이로부터 선형 보간을 통해 임의의 위치에서의 부호화 거리를 계산할 수 있다.

○ **마칭 큐브**

앞에서 정의한 해상도, 한계영역을 가지고 마칭 큐브[8] 알고리즘을 수행하여 폴리곤 곡면을 생성한다. 빠른 계산 속도를 위해 GPU상에서 마칭 큐브를 수행할 수 있으나 본 연구 테스트 결과 CPU상에서도 빠른 속도를 보이므로 CPU기반 마칭 큐브 알고리즘을 사용하였다.



(a) 3D point cloud (b) convex hull

(c) octree (d) reconstructed mesh

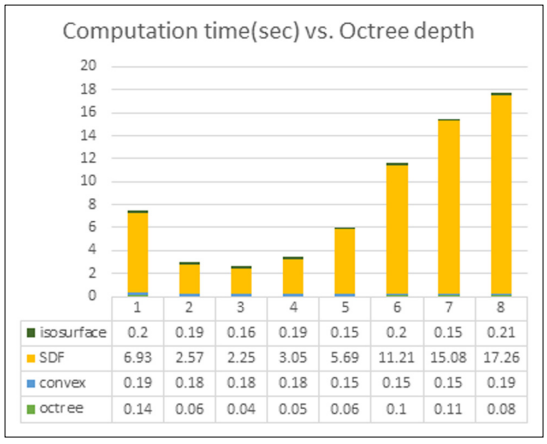
Fig. 1 Proposed procedure for surface reconstruction

3. 수치 실험

본 연구에서 제시한 곡면 재건 알고리즘의 입력 파라미터는 옥트리 깊이와 해상도이다. 옥트리 깊이별 계산 시간과 해상도별 최대 순간 메모리 사용량에 관한 고찰을 통해 본 연구 알고리즘의 가능성을 확인하고자 한다.

○ 옥트리 깊이와 계산 시간

옥트리 깊이가 깊을수록 계산 시간이 많이 소요되는 것은 아니다. Fig. 2에서 가로 축은 옥트리 깊이를, 세로 축은 계산 시간을 초단위로 나타낸 것이며, 대상 모델은 Fig. 1의 샘플모델이다. Fig. 2에서 보는 바와 같이 옥트리 깊이가 증가할 때 계산 시간은 감소하다가 증가한다. 옥트리 깊이가 3일 때 최소 시간이 소요되는 것으로 나타났다. 다른 테스트 모델의 경우에도 마찬가지이다. 깊이가 너무 작으면 옥트리의 옥탄트(octant) 내에 속한 점의 개수가 많아 비교적 큰 SDF 계산 시간이 소요되며, 반대로 깊이가 너무 크면 옥탄트의 개수가 많아져 역시 SDF 계산에 많은 시간이 소요된다. 따라서 최소의 계산 시간에 해당하는 옥탄트 깊이가 존재하며 수치적으로 3임을 확인하였다. 추후 연구로서 이론적 고찰이 필요하다.



	1	2	3	4	5	6	7	8
isosurface	0.2	0.19	0.16	0.19	0.15	0.2	0.15	0.21
SDF	6.93	2.57	2.25	3.05	5.69	11.21	15.08	17.26
convex	0.19	0.18	0.18	0.18	0.15	0.15	0.15	0.19
octree	0.14	0.06	0.04	0.05	0.06	0.1	0.11	0.08

Fig. 2 Computation time required for reconstruction by octree depth

한편 각 과정별 계산 시간을 살펴보면 SDF 생성 시간이 매우 큼을 확인할 수 있다. 이는 SDF 과정에서 상당 규모의 곱셈 연산이 진행되기 때문이다. SDF과정은 SPMD(single program, multiple data)가 가능하므로 빠른 계산을 위해 GPU상에서의 병렬 계산을 고려해 볼 필요가 있다.

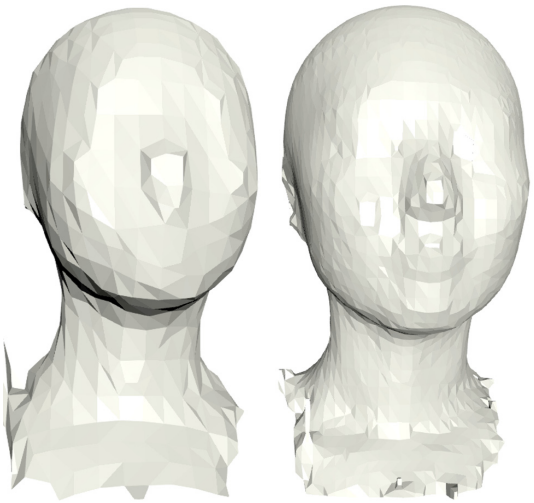
○ **해상도와 메모리 사용량**

본 연구에서 측정한 메모리 사용량은 최대 순간 메모리이다. 이것 역시 SDF과정에서 발생하며 사용자가 입력한 해상도에 따라 다음과 같이 계산할 수 있다.

$$Peak\ memory\,[Mbytes] = (r_x \times r_y \times r_z \times 4) / 1{,}048{,}576 \quad (2)$$

여기서 r_x, r_y, r_z는 각 축 방향의 해상도이고, 4는 실수형 메모리 bytes이다.

한편 Fig. 3에서 보는 바와 같이 해상도가 클수록 재건된 곡면은 점차적으로 3차원 측정점 데이터에 가까워지고 있음을 확인할 수 있다. 재건 곡면이 측정점 데이터에 어느 정도 가까운지를 나타내 주는 평가 지표에 관한 추가 연구가 필요하다.



(a) resolution = 16 (b) resolution = 32

(c) resolution = 64 (d) resolution = 128

Fig. 3 Resolution of the Face model with the resolution: (a) 16, (b) 32, (c) 64, and (d) 128

4. 결 론

본 연구에서는 3차원 측정점 데이터로부터 폴리곤 곡면을 재건하는 알고리즘을 서술하였다. 본 연구 알고리즘은 음함수 형태의 SDF를 생성하고 이로부터 마칭 큐브에 의해 곡면을 재건하는 방식이다. 비교적 간결하며 직관적이다. 또한 SDF 생성 시간을 줄이기 위해 3차원 볼록 집합과 옥트리를 생성하였다. 이 볼록 집합과 옥트리는 격자점과 측정점 데이터 사이의 부호화 최소 거리를 계산하는데 사용된다. 또한 격자점을 저장하지 않고 필요시 계산을 통해 그 위치를 계산함으로써 최대 순간 메모리 사용량을 최소화하였다. 다음은 본 연구 수치 실험을 통해 얻은 본 연구 알고리즘의 특징이다.

○ **다중 해상도**

입력 파라미터인 해상도 값의 조정을 통해 원하는 해상도의 폴리곤 곡면을 재건할 수 있다.

○ **낮은 메모리 사용량**

기존 연구[1)]에 비해 최대 순간 메모리 사용량이 비교적 낮다. 이는 낮은 성능의 PC에서도 본 연구 알고리즘에 의해 폴리곤 곡면이 재건될 수 있음을 의미하며 알고리즘 활용성 측면에서 기존 연구에 비해 우월하다고 말할 수 있다.

○ **알고리즘 강건성**

본 연구 알고리즘은 매우 간결하며, 예를 들어 점 데이터의 분할, 국부적 기저함수의 사용 등이 없으며, 또한 복잡한 경우의 수가 없어 곡면 재건 시 에러의 발생 가능성이 낮다.

○ **노이즈 및 비균일 데이터의 처리**

본 연구 방식은 노이즈 데이터 혹은 비균일 데이터 분포에 영향을 받지 않으며, 구멍이 있는 경우에도 구멍을 채워 곡면을 재건한다.

향후 연구 과제[9)]로서 앞에서도 언급한 최적의 옥트리 깊이 찾기, GPU상에서의 SDF 생성, 재건 곡면의 정밀도 평가지표 개발 등이 필요하다.

Acknowledgement

이 논문은 2020년 한국교통대학교 지원을 받아 수행하였음.

References

1) M. Kazhdan, M. Bolitho, and H. Hoppe, “Poisson surface reconstruction”, Symposium on Geometry Processing, pp. 61-70, 2016.
2) https://web.cs.ucdavis.edu/~amenta/powercrust.html
3) M. Kazhdan, and H. Hoppe, “Screened Poisson surface reconstruction”, ACM Trans. Graphics, 32(3), 2013.
4) H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle, “Surface reconstruction from unorganized point”, Computer Graphics, 26, pp.71‑78, 1992.
5) Y. Ohtake, A. Belyaev, M. Alexa, G. Turk, and H. Seidel, “Multi-level partition of unity implicits”, TOG, pp.463‑470, 2003.
6) https://github.com/mauriciopoppe/quickhull3d
7) https://github.com/vanruesc/sparse-octree
8) W. Lorensen and H. Cline, “Marching cubes: A high resolution 3d surface reconstruction algorithm”, SIGGRAPH, pp.163‑169, 1987.
9) S. Park, “A Polygonal Surface Reconstruction from Oriented Point Set”, Korean J. of Comput. Des. Eng., In press, 2020.

부록 A.

```
function createIsosurfaceFromPoints(
        positions, normals, resolution, octreeDepth )
// positions = surface positions
// normals = surface normals (unit vector)
// resolution = no of grid cells along each axis
// octreeDepth = max octree depth
{
  var resolution = resolution || 64;
  var octreeDepth = octreeDepth || 0;

  // resolution
  var rx, ry, rz;
  if( typeof resolution === 'number' )
  {
    rx = ry = rz = resolution;
  }
  else
  {
    rx = resolution[ 0 ];
    ry = resolution[ 1 ];
    rz = resolution[ 2 ];
  }
  var rmax = Math.max( rx, ry, rz );

  // bbox
  var size = new THREE.Vector3();
  var bbox = new THREE.Box3();
  bbox.setFromArray( positions );
  bbox.getSize( size );
  var padding = Math.min(size.x, size.y, size.z) * 0.1;
  bbox.expandByScalar( padding );

  // bounds
  var x0, x1, y0, y1, z0, z1;
  x0 = bbox.min.x; x1 = bbox.max.x;
  y0 = bbox.min.y; y1 = bbox.max.y;
  z0 = bbox.min.z; z1 = bbox.max.z;
  var bounds = [ [x0,y0,z0], [x1,y1,z1] ];

  // vsize, dmax
  var dx = (x1 - x0) / (rx - 1);
  var dy = (y1 - y0) / (ry - 1);
  var dz = (z1 - z0) / (rz - 1);
  var vsize = Math.sqrt( dx*dx + dy*dy + dz*dz );
  var dmax = vsize * 2; // max signed distance

  // octree
  var octree = new SPARSEOCTREE.PointOctree(
      bbox.min, bbox.max, 0.0, 0, octreeDepth );
  var points = [];
  for( var i = 0, l = positions.length; i < l; i += 3 )
  {
    var pi = { pi: i };

    var p = new THREE.Vector3();
    p.fromArray( positions, i );
```

```
    octree.insert( p, pi );

    points.push( p );
  }
  // convex
  var convexGeometry
        = new THREE.ConvexBufferGeometry( points );
  var convexInOut
        = new JAMIE.ConvexInOut( convexGeometry );

  var item, p, pi, nx, ny, nz, d;
  var volumeData = [];
  var q = new THREE.Vector3();

  // volume data (grid)
  for( var k = 0, z = z0; k < rz; k++, z += dz )
  for( var j = 0, y = y0; j < ry; j++, y += dy )
  for( var i = 0, x = x0; i < rx; i++, x += dx )
  {
    q.set( x, y, z );

    // octree: near surface
    item = octree.findNearestPoint( q, vsize );

    if( item === null )
    {
      // convex: outside
      var outside = convexInOut.isPointOutside( q );
      if( outside )
      {
        volumeData.push( dmax ); continue;
      }

      // convex: inside
      // octree: away from the surface
      item = octree.findNearestPoint(
            q, vsize * rmax/4 );
      if( item === null )
      {
        item = octree.findNearestPoint( q );
      }
    }

    p = item.point;
    pi = item.data.pi;
    nx = normals[ pi ],
    ny = normals[ pi + 1 ],
    nz = normals[ pi + 2 ];
    d = (q.x - p.x) * nx + (q.y - p.y) * ny + (q.z - p.z) *
nz;

    volumeData.push( d );
  }

  // isosurface
  var geometry = new JAMIE.IsosurfaceGeometry(
            [rx,ry,rz], bounds, volumeData );

  return geometry;
}
```