http://dx.doi.org/10.7848/ksgpc.2013.31.1.41

대용량 3차원 포인트 클라우드의 탐색을 위한 메모리 효율적인 옥트리의 설계

Design of Memory-Efficient Octree to Query Large 3D Point Cloud

한수희[1)]
Han, Soohee

Abstract

The aim of the present study is to design a memory-efficient octree for querying large 3D point cloud. The aim has been fulfilled by omitting variables for minimum bounding hexahedral (MBH) of each octree node expressed in C++ language and by passing the re-estimated MBH from parent nodes to child nodes. More efficiency has been reported by two-fold processes of generating pseudo and regular trees to declare an array for all anticipated nodes, instead of using new operator to declare each child node. Experiments were conducted by constructing tree structures and querying neighbor points out of real point cloud composed of more than 18 million points. Compared with conventional methods using MBH information defined in each node, the suggested methods have proved themselves, in spite of existing trade-off between speed and memory efficiency, to be more memory-efficient than the comparative ones and to be practical alternatives applicable to large 3D point cloud.

Keywords : LiDAR, 3D Point Cloud, Query, Octree

초 록

본 연구는 대용량 3차원 포인트 클라우드의 탐색을 위한 메모리 효율적인 옥트리를 설계하는 것을 목표로 한다. 이를 위하여 C++ 언어로 구현된 옥트리 노드의 구성요소 중 최소 경계 입방체 좌표 정보 등을 위한 변수를 제거하는 대신, 부모 노드에서 자식 노드 접근시 최소 경계 입방체 좌표를 계산하여 전달하였다. 아울러 자식 노드 등의 생성시마다 new 연산자를 사용하는 대신, 수도 트리와 정식 트리를 생성하는 이중적인 과정을 통해 노드 등을 배열로 미리 선언함으로서 메모리 효율성을 더욱 개선하였다. 1800만개 이상의 포인트로 구성된 실제 포인트 클라우드를 대상으로 트리를 구성하고 인접 포인트를 탐색하는 실험을 수행하였다. 최소 경계 입방체 좌표 정보를 노드에 저장하는 경우와 비교한 결과 메모리 사용량과 탐색 속도의 트레이드오프가 존재하지만 제안한 방식이 비교군보다 메모리 효율적이어서 대용량 포인트 클라우드에 적용할 수 있는 대안임을 확인할 수 있었다.

핵심어 : 라이다, 3차원 포인트 클라우드, 탐색, 옥트리

1. 서 론

3차원 지상 레이저 스캐너로부터 취득한 포인트로 구성된 3차원 포인트 클라우드(point cloud)의 크기는 스캐너의 성능 향상과 스캐닝 대상 영역의 확대에 의하여 급격히 증가하고 있다. 여러 위치에서 취득한 포인트 클라우드간 접합을 수행하거나 대상물의 형상을 추출/가시화하기 위해서는 포인트간 인접성 분석(proximity analysis)이 필수적이며, 이를 방대한 크기의 포인트 클라우드를 대상으로 수행하기 위해서는 효율적인 자료구조 또는 색인(indexing) 방식이 필수적이다. 대용량 3차원 포인트 클라우드의 색인을 위하여 Han 등(2011)은 3차원 R-tree와 옥트리(octree)의 성능을 비교 분석하여 옥트

1) 정회원 · 경일대학교 위성정보공학과 조교수(E-mail:scivile@kiu.ac.kr)

리가 3차원 R-tree에 비하여 포인트 탐색(query) 속도는 월등히 빠르고 메모리 사용량은 다소 많음을 증명하였다. Han 등(2012)은 가상격자(virtual grid)와 해시(hash)를 접목한 해시기반 가상격자(hash-based virtual grid)를 제안하여 가상격자 및 옥트리와 성능 비교를 수행하여 해시기반 가상격자가 탐색 속도 및 메모리 사용량 면에서 절충안임을 언급하였다. 즉, 탐색 속도는 가상격자, 해시기반 가상격자, 옥트리의 순서로 빠르고 메모리 사용량은 옥트리, 해시기반 가상격자의 순서로 적게 기록되었다. 상기 두 연구에 의하면 적절한 탐색 속도를 유지하면서 메모리 효율성에 보다 큰 비중을 둘 경우 옥트리가 합리적인 선택이라 판단되며, 이미 지상 및 항공 레이저 스캐닝 포인트 클라우드의 처리와 관련된 다양한 연구에서 옥트리를 사용하고 있다(Saxena 등, 1995; Woo 등, 2002; Wang 등, 2004; Schnabel 등, 2007; Cho 등, 2008; Marechal, 2009).

본 연구는 대용량 3차원 포인트 클라우드의 탐색을 위한 옥트리의 구현시 메모리를 효율적으로 사용하는 옥트리를 설계하는 것을 목표로 한다. 이를 위하여, 옥트리를 C++ 언어로 구현시 메모리 사용량에 영향을 미칠 수 있는 옥트리 노드 클래스의 구성 변수와 트리 생성 방식을 분석하고 개선 방안을 제시하였다. 실제 대용량 3차원 포인트 클라우드에 적용하여 옥트리를 구성하는데 소요되는 시간과 메모리의 크기, 인접 포인트 탐색에 소요되는 시간 등을 측정하고 분석하였다.

2. 효율적인 옥트리의 설계

2.1 기본적인 옥트리의 구조와 구현

옥트리는 계층적 트리 구조의 일종으로 상위 계층인 부모 노드(parent node)가 여덟 개의 자식 노드(child node)와 연결된 구조이며 쿼드트리(quadtree)의 3차원 확장이다. 공간적으로는 상위 노드가 관할하는 3차원 공간을 가로, 세로, 높이 방향으로 각각 양분하여 총 8개의 동일한 크기의 공간을 자식 노드가 관할한다(Figure 1). 같은 계층에 속하는 노드는 같은 단계(level) 값을 가지며 자식 노드는 부모 노드에 비해 1이 증가된 단계 값을 갖는다. 트리의 최상부 계측에 속하는 노드는 헤드 노드(head node)라 칭하고 가장 하부 계층에 속하는 노드는 리프 노드(leaf node)라 칭한다. 즉, Figure 1 우측 트리 구조에서 헤드 노드의 단계 값은 1이고 2행 8개의 단계 값은 2이며 3행 16개의 노드는 단계 값은 3이다. 트리의 계층은 목표 단계(destination level) 값만큼 생성하며 목표 단계가 1 증가할 때마다 개개의 리프 노드가 관할하는 공간의 크기는 1/8로 축소된다.

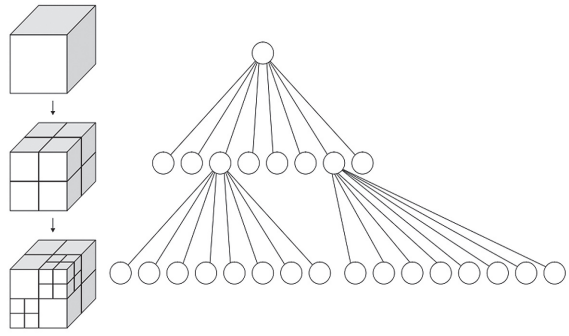

Figure 1. Octree (Wikipedia, 2010)

3차원 포인트 클라우드를 색인화하기 위한 옥트리 생성 과정은 다음과 같다.

① 포인트 클라우드 전체의 최소 경계 입방체(Minimum Bounding Hexahedral, 이 후 MBH)에 해당하는 공간을 헤드 노드에 부여하고 포인트를 헤드 노드에 입력한다.
② 부모 노드(처음엔 헤드 노드)에 포인트가 입력되면 노드의 공간을 x, y, z 방향으로 각각 양분하여 생성된 8개의 하위 공간을 자식 노드에 부여한다.
③ 입력된 포인트의 3차원 좌표 값을 이용하여 포인트가 8개의 자식 노드 중 어느 것에 포함되는지 확인한 후 해당 자식 노드에 입력한다.
④ 과정 ②와 ③을 재귀적으로 반복하여 자식 노드의 단계가 목표 단계에 이를 경우 리프 노드에는 입력된 포인트의 3차원 좌표 또는 포인트가 실제로 저장된 주소, 즉 포인터를 저장한다.
⑤ 포인트 클라우드의 포인트를 하나씩 헤드 노드에 입력하여 ②~④의 과정을 거치대, 포인트가 입력된 노드의 자식 노드가 존재할 경우 과정 ②를 생략한다.

위에서 언급한 옥트리 생성 과정을 C++ 언어로 구현하기 위한 수도 코드(pseudo code)는 Figure 2와 같다. 여기서 노드는 클래스 CNode로 정의하고 최소 경계 입방체는 클래스 외부에 구조체(struct) MBH로 정의하였다. CNode 내부에는 MBH형 변수와 노드의 단계를 저장할 정수형 변수 mbh, 자식 노드를 참조할 포인터 변수 pChild, 노드가 리프 노드일 경우를 위하여 C 언어 Standard Template Library(STL)의 vector형 변수 ptList를 정의하였다. CNode의 내부 함수로는, 포인트를 입력 받아 재귀적으로 트리를 생성하기 위한 함수 AddPoint, 포인트의 좌표와 최소 경계 입방체의

좌표를 이용하여 자식 노드의 ID를 결정하기 위한 함수 SelectChildUsing, 자식 노드들의 최소 경계 입방체의 좌표를 결정하기 위한 함수CalcMBHUsing을 정의하였다. 포인트 입력을 통한 트리 생성 코드는 Figure 3과 같다. 최초에 헤드 노드인 Head를 생성하고 포인트 클라우드의 포인트를 AddPoint 함수를 통해 하나씩 입력한다. 포인트가 입력된 노드의 단계 값이 목표 단계인 DestLevel 보다 작을 경우 자식 노드를 8개 생성하여 적절한 자식 노드에 포인트를 입력하고, 목표 단계에 이르면 자식 노드, 즉 리프 노드에서 포인트를 저장하고 입력 과정을 종료한다.

```
struct MBH{ // 최소 경계 입방체 좌표 저장을 위한 구조체
    double minX, minY, minZ, maxX, maxY, maxZ // 좌상, 우하 3차원 좌표
}

Class CNode{
  MBH mbh // 최소 경계 입방체 좌표 구조체형 변수
  int level // 현 노드의 단계 저장 변수
  vector * ptList // 리프 노드일 경우 포인트를 저장하기 위한 vector형 포인터
  CNode * pChild // 자식노드들에 대한CNode형 포인터

  AddPoint(Point pt){ // 포인트 입력을 통한 트리 생성 함수
    int n = SelectChildUsing(mbh, pt) // 포인트를 포함하는 자식 노드 ID 결정
    if (level == DestLevel-1) // 현노드가 리프노드의부모 노드인지 판별
      if (pChild[n].ptList == NULL)
        // 리프 노드에 vector 구조체가 선언되어 있는지 판별
        pChild[n].ptList = new vector
          // 리프 노드에 vector 구조체 동적 할당
      pChild[n].ptList.Add(pt) // 리프 노드에 포인트 저장
    else
      if (pChild == NULL) // 현 노드에 자식 노드가 없을 경우 판별
        pChild = new CNode[8] // 8개의자식노드생성
        for i=0 to 7
          pChild[i].mbh = CalcMBHUsing(mbh, i)
            // 자식 노드에 새 MBH 부여
          pChild[i].level = level + 1 // 자식노드에새 단계 값 부여
      pChild[n].AddPoint(pt) // 선택된 자식 노드에 재귀적으로 포인트 입력
  }
}
```

Figure 2. Codes for basic octree node

```
CNode Head // 헤드 노드 생성
Head.mbh = GetGlobalMBH() // 포인트 클라우드 전체 최소 경계 입방체 좌표 입력
Head.level = 0 // 최상위 노드의 단계 값을 0으로 지정
for each Point in PointCloud
  Head.AddPoint(Point) // 포인트 클라우드의 포인트를 하나씩 입력
```

Figure 3. Codes for basic octree generation

Figure 2의 CNode의 크기는 클래스 내에 선언된 변수의 크기에 의해 결정되며, 특히 최소 경계 입방체의 좌표를 구성하는 변수의 형태에 따라 크게 영향을 받는다. 즉, 구조체 MBH의 좌표 변수를 배밀도 실수형(double)으로 구현할 경우 CNode 한 개의 크기는 mbh(6*8 bytes)+level(4 bytes)+ptList(4 bytes)+pChild(4 bytes) = 60 bytes이며, 단밀도 실수형(float)으로 구현할 경우는 mbh(6*4 bytes)+level(4 bytes)+ptList(4 bytes)+pChild(4 bytes) = 36 bytes이다. 자식 노드의 단계가 증가할 때마다 한 개의 부모 노드에 대하여 자식 노드가 8개씩 생성되므로 어느 경우이든 메모리 사용량이 크게 증가함을 예측할 수 있다.

2.2 메모리 효율적인 옥트리의 구현

메모리 효율적인 옥트리를 구현하기 위하여 노드(CNode)에 정의된 변수의 종류를 최소화할 필요가 있으므로 Figure 4와 같이 최소 경계 입방체 좌표와 단계 저장 변수를 제거하였다. 이 경우 CNode의 크기는 ptList(4 bytes)+pChild(4 bytes) = 8 bytes로서 2.1 단원에서 정의한 CNode의 크기인 60 bytes 또는 36 bytes 보다는 대폭 축소된 크기이다. 대신 Figure 4처럼 포인트 입력 함수 AddPoint를 변형하여 각 노드는 자신의 최소 경계 입방체의 좌표 mbh와 단계 값 level을 부모 노드

로부터 전달 받아 갱신 후 자식 노드에게 전달한다. 헤드 노드에서 리프 노드에 이르는 최소 경계 입방체 좌표의 전달 과정에서 발생할 수 있는 지체를 최소화하기 위하여 mbh는 포인터 형식으로 선언하여 좌표 값을 갱신하고 포인터만을 전달한다. 따라서 Figure 5의 옥트리 생성 코드에는 매 포인트 입력 시 mbh의 좌표 값을 포인트 클라우드 전체의 최소 경계 입방체 좌표로 환원해주는 코드가 추가된다. 반면 Figure 2에서의 자식 노드에 새로운 최소 경계 입방체 좌표와 단계 값을 지정해 주는 코드는 Figure 4에서 제거한다.

```
Class CNode{
    vector * ptList
    CNode * pChild

    AddPoint(Point pt, MBH *mbh, int level){
        int n = SelectChildUsing(mbh, pt)
        if (level == DestLevel-1)
            ...
        else
            if (pChild == NULL)
                pChild = new CNode[8]
            mbh = CalcMBHUsing(mbh, i)
            pChild[n].AddPoint(pt, mbh, level+1)
                // 선택된 자식 노드에 포인트, 갱신된 mbh와
                   단계 값을 재귀적으로 입력
    }
}
```

Figure 4. Codes for MBH-free octree node

```
CNode Head
for each Point in PointCloud
    MBH mbh = GetGlobalMBH()
        // 포인트 클라우드 전체 최소 경계 입방체 좌표 환원
    Head.AddPoint(Point, &mbh, 0)
        // 포인트 클라우드의 포인트를 하나씩 입력
```

Figure 5. Codes for MBH-free octree generation

2.3 배열을 이용한 메모리 효율성 개선

Figure 2 또는 Figure 4의 코드에서 자식 노드를 생성하고 리프노드에서 vector 구조체를 생성하는 과정에서 메모리 동적 할당을 위한 new 연산자를 사용한다. 메모리 동적 할당 또는 동적 메모리 할당은 컴퓨터 프로그래밍에서 실행 시간 동안 사용할 메모리 공간을 할당하는 것으로 프로그램이 실행되기 전 컴파일 단계에서 미리 프로그램이 사용할 메모리 크기를 계산하여 변수의 배치가 이루어지는 스택 등의 정적 메모리 할당과 대조적이다(Wikipedia, 2012). 그러나 작은 크기의 메모리를 반복적으로 동적 할당하는 과정에서 메모리가 추가적으로 소요됨을 실험을 통해 확인할 수 있었다. 따라서 실제 트리 생성에 앞서 수도 트리(pseudo tree)를 생성하여 모든 부모 또는 자식 노드의 수와 리프 노드의 수를 산정하여 배열로 선언한 후 정식 트리를 생성하는 방법을 구현한다. 즉, Figure 6의 AddPoint_Counting 함수는 Figure 4의 AddPoint 함수와 마찬가지로 트리 구조를 생성하기는 하지만 실제로 리프 노드에 포인트를 추가하지는 않고 부모 또는 자식 노드의 수 nNode와 리프 노드의 수 nLeaf를 산정한다. Figure 6의 AddPoint 함수는 자식 노드와 리프 노드를 동적 할당 하는 대신 미리 선언한 노드 배열과 리프 노드 배열에 매핑(mapping)하는 구조를 가진다. 이를 위하여 Figure 7에서는 AddPoint_Counting 함수를 통해 수도 트리를 생성하여 노드의 수를 산정한 후 수도 트리를 메모리에서 제거한다. 다음으로 노드와 포인트 저장 구조체 전체를 각각 배열 nodeArray와 ptListArray로 선언하고 AddPoint 함수를 통해 정식 트리를 생성한다. 이러한 이중 과정을 통해 트리 생성 시간은 다소 증가될 수 있지만 메모리 효율성을 높일 수 있다.

```
int nNode, nLeaf // 노드와 리프 노드의 개수 산정을 위
                    한 전역 변수
int nCounter1, nCounter2 // 노드와 리프 노드의 매핑을
                            위한 전역 변수
Class CNode{
    vector * ptList
    CNode * pChild

    AddPoint_Counting(Point pt, MBH *mbh, int level) {
        // 수도 트리 생성을 통해 nNode와 nLeaf 산정
        int n = SelectChildUsing(mbh, pt)
        if (level == DestLevel-1 && pChild[n].ptList ==
        NULL)
            nLeaf = nLeaf +1
        else
            if (pChild == NULL)
                pChild = new CNode[8]
```

```
        mbh = CalcMBHUsing(mbh, i)
        nNode = nNode + 8
        pChild[n].AddPoint(pt, mbh, level+1)

    AddPoint(Point pt, MBH *mbh, int level) {
      int n = SelectChildUsing(mbh, pt)
      if (level == DestLevel-1)
        if (pChild[n].ptList == NULL)
          pChild[n].ptList = &ptListArray[nCounter1 += 1]
            // 리프 노드의 vector 구조체를 배열에 매핑
        pChild[n].ptList.Add(pt)
      else
        if (pChild == NULL)
          pChild = &nodeArray[nCounter2 += 8]
            // 8개의 자식 노드를 배열에 매핑
        mbh = CalcMBHUsing(mbh, i)
        pChild[n].AddPoint(pt, mbh, level+1)
    }
}
```

Figure 6. Codes for array-based octree node

```
CNode Head
for each Point in PointCloud
    MBH mbh = GetGlobalMBH()
    Head.AddPoint_Counting(point, &mbh, 0)

Head.Clear() // 수도 트리 제거
nodeArray = new CNode[nNode]
ptListArray = new Vector[nLeaf]

for each Point in PointCloud
    MBH mbh = GetGlobalMBH()
    Head.AddPoint(Point, &mbh, 0)
```

Figure 7. Codes for array-based octree generation

3. 실험 및 고찰

2.1 단원에서 설명한 기본적인 옥트리 생성 방식 중 최소 경계 입방체 좌표 변수로 단밀도 실수형(float)을 사용한 방식(이후 M1_Float 방식)과 배밀도 실수형(double)을 사용한 방식(이 후 M2_Double 방식), 2.2 단원에서 제시한 최소 경계 입방체 변수를 노드에서 제거한 방식(이 후 M3_MBHfree 방식), 2.3 단원에서 제시한 배열을 이용한 방식(이 후 M4_Array 방식) 등 네 가지 방식을 실제 터널 내부에서 취득한 포인트 클라우드에 적용하여 성능을 평가하였다. 성능 평가는 총 다섯 목표 단계(7, 8, 9, 10, 11)에 대한 트리 생성 시간, 메모리 사용량, 탐색 시간을 측정하고 분석함으로써 수행하였다. 탐색은 Figure 8처럼 임의의 포인트로부터 주어진 반경 안에 존재하는 포인트의 수를 산정하는 방식을 사용하였으며(Han 등, 2012) 검색 반경은 15㎝로 설정하였다. 실험에 사용한 자료의 제원과 처리 환경은 각각 Table 1과 Table 2와 같다.

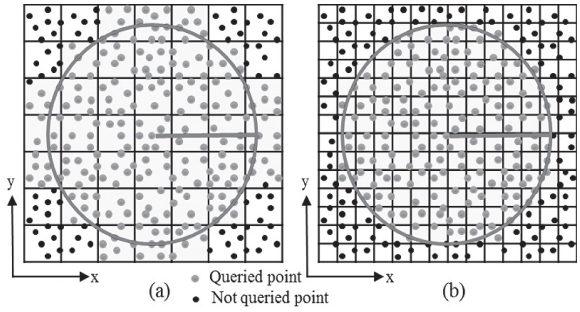

Figure 8. Query method expressed in 2D: (a) query in level *n*, (b) query in level *n+1* (Han et al., 2012)

Table 1. Specification of data

Terrestrial Laser Scanner	Scan Station 2, Leica Geosystems
Scanned Object	A Tunnel in Sejong-Si, South Korea (Length: 55.82m, Curvature: 252.50m, Width:10.52m)
Point Cloud	Number of points: 18,376,726 points Point Density: Approx. 0.25 points/1㎠

Table 2. Processing environments

CPU	Intel Core i5-2300 @2.80GHz
RAM	3 GB
OS	Windows 7 Professional 64bit
Code Development	Visual C++ win 32 (Visual Studio 2010)

Table 3과 Table 4는 각각 18,376,726개의 포인트로부터 7부터 11까지의 다섯 가지 목표 단계별로 옥트리를 생성하는

데 소요된 시간과 메모리의 크기이다. M1_Float 방식과 M2_Double 방식의 생성 시간이 유사하고 M3_MBHfree 방식이 약간의 차이로 뒤를 따른다. M4_Array 방식의 경우 이중적인 트리 생성 과정을 거치기 때문에 M3_MBHfree 방식보다 약 1.5~1.6배의 시간을 소요함을 확인할 수 있다. 트리 생성 시 사용되는 메모리의 크기는 M2_Double 방식이 가장 크며 목표 단계 11에서는 시스템 용량 초과로 인해 옥트리를 생성할 수 없었다. M1_Float 방식을 기준으로 다른 방식이 사용한 메모리의 크기는, M2_Double 방식의 경우 같은 목표 단계에서 1.03배(7단계)~1.47배(10단계), M3_MBHfree 방식의 경우 0.97배(7단계)~0.44배(11단계), M4_Array 방식의 경우 0.97배(7단계)~0.36배(11단계)이다.

Table 3. Time lapse for octree generation (sec)

Dest Level	M1_ Float	M2_ Double	M3_ MBHfree	M4_ Array
7	1.83	1.75	2.25	3.68
8	2.17	2.21	2.73	4.42
9	2.85	2.96	3.51	5.54
10	4.15	4.29	4.84	7.16
11	5.96	N/A	6.43	9.80

Table 4. Memory usage for octree generation (MB)

Dest Level	M1_ Float	M2_ Double	M3_ MBHfree	M4_ Array
7	95	98	92	92
8	107	120	96	94
9	167	215	121	112
10	404	594	219	190
11	1252	N/A	546	445

Table 5는 앞에서 생성한 옥트리에서 367,535(전체 포인트 클라우드의 1/50)개의 포인트를 중심으로 반경 15㎝내에 존재하는 포인트를 탐색하는데 소요된 시간이다. 네 가지 방식 모두 목표 단계가 증가함에 따라 탐색 시간이 감소하여 목표 단계 9에서 가장 작은 탐색 시간을 기록하였다. 이 후, 목표 단계가 증가함에 따라 탐색 시간이 다시 증가하는 양상을 나타내는 데, 그 이유는 다음과 같이 설명할 수 있다. 즉, Figure 8(a)에서 목표 단계가 1단계 증가하면 Figure 8(b)와 같이 리프 노드의 각 축별 크기가 절반으로 줄어들어 검색 영역의 형태가 원과 더욱 유사해진다. 이로 인해 검색 영역이 축소되어 검색 대상 포인트의 수가 감소하므로 탐색 시간이 단축된다. 이와 같은 현상은 목표 단계 7에서 목표 단계 9까지 영향력을 미친다. 그러나 목표 단계가 그 이상으로 증가하면 검색 영역의 축소 효과는 줄어드는 반면 헤드 노드로부터 리프 노드까지의 경로는 더욱 길어져 리프 노드까지의 접근 시간이 증가하므로 전반적인 탐색 시간은 증가하게 된다(Han 등, 2012). 각 목표 단계에서의 포인트 탐색 속도는 포인트 포인트의 수와 포인트 분포의 균질성에 영향을 받는다. 즉, 본 실험보다 많은 포인트의 포인트 클라우드에 적용하면 개별 리프 노드당 포인트 수도 증가하여 목표 단계를 높여 검색 영역을 축소시키는 것이 유리하며 반대의 경우엔 목표 단계를 낮추어 경로를 짧게 하는 것이 유리하다. 그러나 포인트가 일부 리프 노드에 집중되는 경우엔 목표 단계를 더욱 높여야 하는 상황이 발생할 수 있다. 결국 최적의 목표 단계를 이론적으로 결정하기는 어려우며 이와 같은 실험을 통해 산출해야 한다.

한편, 각 방식별 탐색 시간을 비교하면, 목표 단계 7에서 8까지는 M1_Float, M2_Double 방식이 M3_MBHfree, M4_Array 방식과 비교하여 유사하거나 약한 작게 측정되었으며 메모리 사용량 면에서는 반대의 경향이 나타났다. 최적 목표 단계 9에서는 M3_MBHfree 방식의 속도가 M1_Float 방식의 89%, M2_Double 방식의 84% 수준이며, M4_Array 방식의 속도는 M1_Float, M2_Double 방식의 94%, 89% 수준으로 나타났다. 반면 메모리 사용량은 M3_MBHfree 방식이 M1_Float, M2_Double 방식의 72%, 56% 수준, M4_Array 방식이 M1_Float, M2_Double 방식의 67%, 52% 수준인 것으로 나타났다. 목표 단계 10 이상에서는 그 격차가 더욱 벌어졌으나 메모리 사용량의 격차가 속도의 격차보다 크게 벌어짐을 확인할 수 있다. 속도 격차는 M3_MBHfree, M4_Array 방식의 경우 부모 노드에서 자식 노드 접근 시 자식 노드의 최소 경계 입면체의 좌표 정보를 갱신해 주어야 하는 특성에 기인한 것으로, 목표 단계의 증가와 함께 리프 노드까지의 경로가 길어지는 영향을 더욱 많이 받는 것으로 해석된다. 그러나 네 가지 방식 모두 time complexity가 O(n)으로서 동일한 차수에 해당하므로 근본적인 시간 효율성의 차이는 크지 않다고 판단된다.

Table 5. Time lapse for querying neighbor points (sec)

Dest Level	M1_ Float	M2_ Double	M3_ MBHfree	M4_ Array
7	84.21	77.39	85.29	77.77
8	57.45	53.13	59.06	54.30
9	45.04	42.60	50.62	47.91
10	90.96	89.92	117.94	116.07
11	386.43	N/A	539.04	537.63

Table 6은 각 목표 단계에서 각 방법에 의해 검색된 인근 포인트의 개수이다. 특별히 M1_Float 방식의 경우 M2_Double, M3_MBHfree, M4_Array 방식과는 달리 목표 단계에 따라 차이가 발생함을 확인할 수 있다. 이는 M1_Float 방식에서 사용된 자식 노드의 최소 경계 입면체 좌표의 변수형이 단밀도 실수형(float)으로서, 배밀도 실수형(double)만큼의 정밀도를 갖추지 못한 것에 기인한 것으로 해석된다.

Table 6. Number of queried neighbor points

Dest Level	M1_ Float	M2_ Double	M3_ MBHfree	M4_ Array
7	1331927743	1331926997	1331926997	1331926997
8	1331910444	1331926997	1331926997	1331926997
9	1331922670	1331926997	1331926997	1331926997
10	1331983145	1331926997	1331926997	1331926997
11	1331975906	N/A	1331926997	1331926997

결과적으로, 탐색 속도와 메모리 사용량 간에는 트레이드오프(trade off)가 명백히 존재함을 확인할 수 있다. 트리를 생성하는데 소요되는 시간은 모든 목표 단계에서 M1_Float, M2_Double 방식이 M3_MBHfree, M4_Array 방식보다 적게 측정되었지만 그 차이는 수 초 이내이며 트리 생성이 실제 어플리케이션에서 빈번히 발생하는 과정이 아니라는 점을 감안하면 방식 선택 기준으로서 큰 영향은 미칠 수 없다고 판단된다. 인접 포인트 탐색 실험에서 본 연구에서 사용한 포인트 클라우드에 대한 최적 목표 단계인 9단계까지는 네 가지 방식이 수 초 이내의 차이를 기록한 가운데 M1_Float, M2_Double 방식이 약간의 우세를 보였다. 10단계 이상에서는 그 격차가 심화되어 충분한 메모리가 확보된다면 M1_Float, M2_Double 방식이 유리할 것으로 판단된다. 그러나 목표 단계 7을 제외하면 M3_MBHfree 방식이 M1_Float, M2_Double 방식과 비교하여 상당한 메모리 효율 개선 효과가 있고 M4_Array 방식은 추가적인 개선 효과가 있으며 목표 단계가 증가할수록 그 격차는 심화되었다. 궁극적으로, 초대형 포인트 클라우드에 적용할 경우 10단계 이상의 목표 단계에서 최적의 성능이 나올 가능성이 있으며 이러한 경우 M1_Float, M2_Double 방식은 메모리의 제약으로 인해 트리를 생성하지 못할 가능성이 높다. 아울러, M1_Float의 경우 목표 단계에 따라 탐색된 포인트의 수가 달라지는 결과가 도출되었으므로 고도의 정확도를 요구하는 어플리케이션에는 적합하지 않을 것으로 판단된다. 따라서 본 연구에서 제안한 M3_MBHfree 방식 또는 M4_Array 방식은 메모리가 제약된 전산 환경에서 대용량 3차원 포인트 클라우드에 적용할 수 있는 현실적인 대안으로 사료된다.

4. 결 론

본 연구는 대용량 3차원 포인트 클라우드의 탐색을 위한 메모리 효율적인 옥트리를 설계하는 것을 목표로 하였다. 이를 위하여 C++ 언어로 구현된 옥트리 노드의 구성요소 중 최소 경계 입방체 좌표 변수와 단계 저장 변수를 제거하는 대신, 부모 노드에서 자식 노드 접근 시 최소 경계 입방체 좌표를 계산하여 전달하는 방식을 채택하였다. 아울러 자식 노드 생성 및 리프 노드의 포인트 저장용 벡터 구조체 생성시마다 new 연산자를 사용하는 대신, 수도 트리와 정식 트리를 생성하는 이중적인 과정을 통해 노드 및 벡터 구조체를 배열로 미리 선언함으로서 메모리 효율성을 더욱 개선하였다. 실제 포인트 클라우드를 대상으로 인접 포인트 탐색을 수행한 결과, 제안한 방식이 트리 생성 및 탐색 속도 면에서 비교군에 근접하거나 다소 떨어지나 메모리 효율성 면에서는 우수하여 대용량 포인트 클라우드에 적용할 수 있는 대안임을 확인할 수 있었다. 향후, 본 연구에서 제안한 옥트리 방식과 병렬처리를 접목한 초대용량 3차원 포인트 클라우드의 처리에 관한 연구를 수행하고자 한다.

감사의 글

본 연구는 경일대학교 일반연구비(과제번호:2012A-070)의 지원을 받아 수행한 연구임

References

Cho, H., Cho, W., Park, J. and Song, N. (2008), 3D building modeling using aerial LiDAR data, *Korean Journal of Remote Sensing*, Vol. 24, pp. 141–152.

Han, S., Lee, S., Kim, S. P., Kim, C., Heo, J. and Lee, H. (2011), A Comparison of 3D R-tree and octree to index large point clouds from a 3D terrestrial laser lcanner, *Korean Journal of Geomatics*, Vol. 29, No. 1, pp. 531–537.

Han, S., Kim, S., Jung, J. H., Kim, C., Yu, K. and Heo, J. (2012), Development of a hashing-based data structure for the fast retrieval of 3D terrestrial laser scanned data,

Computers & Geosciences, Vol. 39, pp. 1–10.

Marechal, L. (2009), Advances in octree-based all-hexahedral mesh generation: handling sharp features, 18th International Meshing Roundtable, Salt Lake City, UT, USA, pp. 65–84.

Saxena, M., Finnigan, P. M., Graichen, C. M., Hathaway, A. F. and Parthasarathy, V. N. (1995), Octree-based automatic mesh generation for non-manifold domains, *Engineering with Computers*, Vol. 11, pp. 1–14.

Schnabel, R., Wahl, R., and Klein, R. (2007), Efficient RANSAC for point-cloud shape detection, *Computer Graphics Forum*, Vol. 26, pp. 214–226.

Wang, M., Tseng and Y.-H. (2004), Lidar data segmentation and classification based on octree structure, XXth ISPRS Congress, ISPRS, Istanbul, Turkey.

Woo, H. ,Kang, E., Wang, S. and Lee, K. H. (2002), A new segmentation method for point cloud data. *International Journal of Machine Tools and Manufacture*, Vol. 42, pp. 167–178.

Wikipedia (2010), Schematic drawing of an octree, a data structure of computer science, Wikimedia Foundation, Inc., URL: http://en.wikipedia.org/wiki/Octree (last date accessed: 20 January 2013).

Wikipedia (2012), C dynamic memory allocation, Wikimedia Foundation, Inc., URL: http://en.wikipedia.org/wiki/C_dynamic_memory_allocation (last date accessed: 20 January 2013).

(접수일 2013. 01. 20, 심사일 2013. 01. 23, 심사완료일 2013. 02. 13)