Journal of the KIMST, Vol. 23, No. 6, pp. 542-553, 2020
ISSN 1598-9127(Print) · 2636-0640(Online)
DOI https://doi.org/10.9766/KIMST.2020.23.6.542

Research Paper 센서·신호처리 부문

Low Level GPU에서 Point Cloud를 이용한 Level of detail 생성에 대한 연구

감정원[1] · 구본우[2] · 진교홍[*,1]

[1] 창원대학교 전자공학과
[2] ㈜심네트 M&S 1본부

Point Cloud Data Driven Level of detail Generation in Low Level GPU Devices

JungWon Kam[1] · BonWoo Gu[2] · KyoHong Jin[*,1]

[1] *The 1st Research and Department of Electronic Engineering, Changwon National University, Korea*
[2] *Department of M&S 1, SIMNET Coperation, Korea,*

(Received 27 August 2020 / Revised 28 October 2020 / Accepted 13 November 2020)

Abstract

Virtual world and simulation need large scale map rendering. However, rendering too many vertices is a computationally complex and time-consuming process. Some game development companies have developed 3D LOD objects for high-speed rendering based on distance between camera and 3D object. Terrain physics simulation researchers need a way to recognize the original object shape from 3D LOD objects. In this paper, we proposed simply automatic LOD framework using point cloud data (PCD). This PCD was created using a 6-direct orthographic ray. Various experiments are performed to validate the effectiveness of the proposed method. We hope the proposed automatic LOD generation framework can play an important role in game development and terrain physic simulation.

Key Words : Level of Detail(세부 수준), Point Cloud Data(정점 집합 데이터), Volumetric Rendering(가상 렌더링)

1. 서 론

오늘날, 게임과 시뮬레이션 분야에서는 대규모 지형 모델의 고품질 렌더링을 요구한다. 국방 분야에서도 중대, 대대 규모의 병력과 장비 모의 시뮬레이터를 위해 대규모 지형 및 각각의 개체 모델의 고품질 렌더링을 요구한다. 최신 게임 엔진에서는 사용자 뷰 포트(User Viewport) 별 쿼드 트리 렌더링으로 3D 모델의 LOD(Level Of Detail)을 지원한다. 또는, 3D 모델의 버텍스를 수학적 계산으로 축소하여 자동 LOD 생성을 지원한다. 하지만, 3D 모델의 버텍스 수를 수학적 계

* Corresponding author, E-mail: khjin@changwon.ac.kr

산으로 줄이게 되면 원본 3D 모델 대비 심한 왜곡이 발생하게 된다. 게임 분야에서는 그래픽 디자이너가 고품질 LOD 3D 모델을 따로 만들어 사용하지만, 시뮬레이터 분야에서는 정밀한 고도 값, 환경요소 등이 필요하므로 쉽게 LOD 3D 모델을 만들 수 없다.

본 논문에서는 6 방향의 직교 광선 투영을 이용하여 생성한 PCD(Point Cloud Data)를 사용하여 원본 3D 모델과 유사한 자동 LOD 3D 모델 생성 방법을 제안한다.

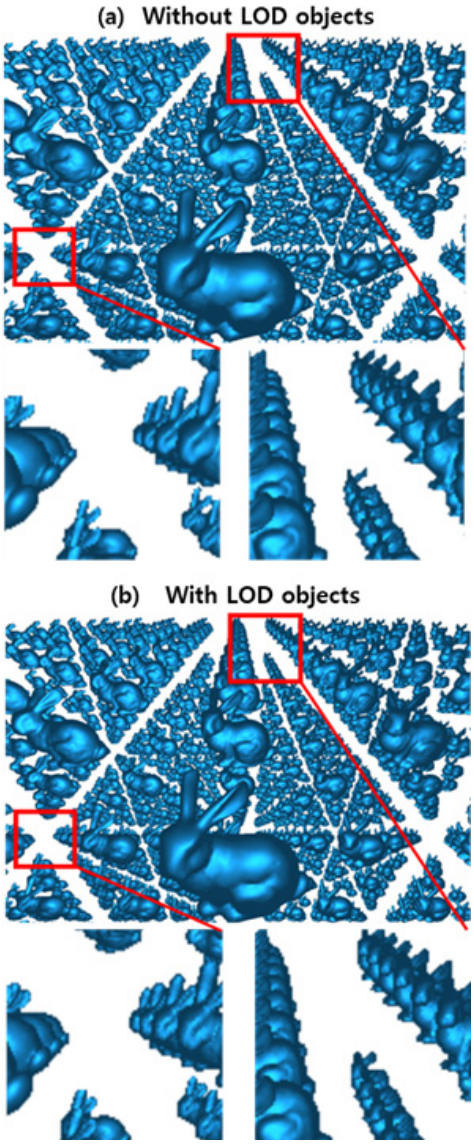

Fig. 1. PCD LOD rendering results

Fig. 1은 본 논문에서 제안된 자동 LOD 3D 모델 생성 결과를 보여준다. Fig. 1(a)는 LOD 없이 원본 Stanford Bunny 오브젝트를 1,331개 렌더링한 결과이다. Fig. 1(b)는 본 논문에서 제안한 LOD 알고리즘으로 5단계의 LOD 오브젝트를 생성한 후 적용한 결과이다. Fig. 1(a)와 Fig. 1(b)의 렌더링 결과는 거의 차이 없는 것을 볼 수 있다. 하지만, 본 논문에서 제안한 LOD는 PCD 기반이기 때문에 Fig. 1(b)에서 Stanford Bunny의 귀부분이 Fig. 1(a)에 비해 조금 뭉개진 것을 확인할 수 있다. 본 논문에서 제안한 LOD는 멀리 있는 오브젝트를 렌더링하기 위한 것이기 때문에 약간의 뭉개지는 현상은 큰 문제가 되지 않는다. 본 논문에서 제안한 방법을 통하여, 무인차량 시뮬레이터 및 대규모 전투 시뮬레이터 분야에서 적극 사용될 수 있길 희망한다.

2. 관련 연구

LOD 알고리즘은 다양한 목적으로 개발되었다. Jens.외는 대규모 지형 모델을 렌더링하기 위해 GPU 기반 중첩 메시 계층 구조(Nested Mesh Hierarchy)를 이용한 LOD 시스템을 제안하였다[1]. 이 알고리즘은 실시간으로 대규모 지형 모델을 렌더링할 수 있다. Wen.Jiang.외는 쿼드 트리(Quad-Tree)를 이용하여 모바일 장치에서 대규모 지형 모델을 렌더링을 시도하였다[2]. 지형 모델을 여러 타일(Tile)로 분할하고 사용자 뷰 포트에 따라 알맞은 지형 타일 모델을 쿼드 트리로 렌더링하는 방식이다. Zhou,S.외는 대규모 3D 도시 모델을 렌더링하기 위하여, 사용자 뷰 포트와 3D 건물의 위치를 계산하여 정점(Vertex)과 선(Edge)을 축소하는 자동 LOD 시스템을 제안하였다[3].

많은 연구자는 2D 이미지에서 3D 객체를 재구성하는 방법에 대해 연구하였다. Zienkiewicz,J.외는 단안(Monocular) 카메라로 촬영된 2D 이미지에 라플라스 피라미드(Laplace Pyramid)[4-6]를 적용한 LOD 시스템을 제안하였다[4]. 이 LOD 시스템은 서로 다른 다양한 품질의 텍스처를 라플라스 피라미드로 생성하고 사용자 뷰 포트와 3D 모델의 사이에 따라 삼각형(Triangles)을 추가하는 방식이다. He.Y.외는 사용자 뷰 포트와 3D 모델과의 거리에 따라 서로 다른 프래그먼트 셰이더(Fragment Shader)를 적용하는 LOD 시스템을 제안하였다[7]. 플레그먼트 셰이더는 빛, 그림자, 기본색상을

디스플레이 해상도에 따라 연산하기 때문에 버텍스 셰이더(Vertex Shader)보다 느리다[8,9]. 그러므로 이 LOD 시스템은 High-Level GPU 디바이스에서는 높은 성능을 낼 수 있지만, Low-Level GPU 디바이스에서는 적합하지 않다.

볼륨 렌더링(Volumetric Rendering)은 구름, 불, 안개, 나무, 조명, 효과(Effect) 등을 렌더링할 수 있다[10-12].

빌보드 클라우드(Billboard Cloud)[11,12]는 매우 간단한 구조의 볼륨 렌더링 알고리즘이다. Mantler,S.외는 빌보드 직사각형(Billboard Rectangle)을 기반한 빌보드 클라우드를 이용하여 LOD 3D 모델을 구축하였다[12,13]. 이 알고리즘은 적은 수의 정점으로 복잡한 3D 모델을 만들 수 있는 장점이 있지만, 디스플레이의 해상도의 픽셀 수에 따라 렌더링 속도가 저하 될 수 있다. 복셀(Voxel)렌더링은 볼륨 렌더링의 또 다른 방법이다. Lorensen.외는 마칭큐브(Marching Cube)를 제안하였다[14]. Jabłoński,S.외는 SVO[15-17]를 사용하여 자동화 LOD 시스템을 제안하였다. 이 LOD 알고리즘은 사용자 뷰 포트와 3D 모델 사이 거리에 따라 GPU의 기하학 셰이더(Geometry Shader)를 사용하여 실시간으로 렌더링 품질을 조정할 수 있다. 하지만, 기하학 셰이더를 지원하지 않는 Low-Level GPU와 모바일 기기에서는 이 LOD 시스템을 사용할 수 없다[18,19].

3D 그래픽스를 연구하는 많은 연구자들은 PCD를 사용한다[20]. Zhao.외는 PCD를 사용하여 2D 실내 이미지를 3D 모델로 구축하였다[21]. Bo Zheng.외는 재질과 물리가 있는 상황을 이해하기 위해 PCD를 사용하였다[22]. 실 센서를 이용하여 3D 환경을 재구축하기 위해 많은 연구자들은 PCD를 사용한다[23]. 그러나 PCD는 입력 데이터와 잡음(Noise) 데이터를 구분할 수 없다. 게다가, 최신 센서 데이터는 수백만 개의 Point data를 생성하기 때문에 메모리 점유율 및 렌더링 속도를 위해 Point data를 줄일 필요가 있다. Eckart,B.외는 하향식 계측 구조(Top-down Hierarchical)와 GMM(Gaussian Mixture Model)을 이용하여 Point data를 줄이는 방법을 제안하였다[24,25]. 본 논문에서는 6 방향의 직교 광선(6-Direct Orthographic Ray)를 사용하여 원본 객체를 PCD로 재구성한다. 본 논문에서 제안된 LOD 시스템은 PCD를 기반으로 하므로 Low-Level GPU 혹은 모바일 디바이스에서도 사용이 가능하다. 직교 광선의 이미지 크기에 따라 Point data 수를 제어할 수 있다. 이 기능으로 LOD 3D 모델의 품질을 제어할 수 있다.

본 논문에서 제안된 LOD 자동 프레임 워크(FrameWork)는 정점과 정점 사이의 메시를 고려하지 않는다. 수학적 계산 방법으로 정점을 줄이고 다시 면(Mesh)을 만들면 3D 모델의 외형이 왜곡될 수 있기 때문이다. 본 논문에서는 면을 재구성하지 않고 중앙값 필터(Median Filter)를 적용하여 3D 모델의 외형을 최대한 보존하였다. GPU에서는 버텍스 셰이더에서 계산된 정점에 맞추어 해상도에 맞게 Mesh를 생성한다. 이때, 생성된 Mesh의 픽셀에 따라 프래그먼트 셰이더의 연산량이 정해진다. 본 논문에서는 Mesh를 만들지 않기 때문에 버텍스 셰이더에서 생성된 정점의 양과 프래그먼트 셰이더의 픽셀이 양이 동일하여 Mesh 생성에 들어가는 연산 부하를 줄였다. 또한, Median Filter를 통하여 외형을 보존하며 정점 사이를 채웠기 때문에 Low Level GPU에서도 빠른 렌더링 속도와 품질을 제공할 수 있다.

3. 제안된 LOD 생성 FrameWork

본 논문에서 제안된 LOD 생성 FrameWork의 개요는 Fig. 2에 설명되어 있다. 제안된 LOD 생성 FrameWork는 사용자 뷰 포트와 3D 모델 사이의 거리에 따라 LOD Level을 변경한다. LOD Level이 변경될 시, 부드러운 변경을 위하여 페이드 인/아웃(Fad In/Out)을 사용한다. 제안된 LOD 생성 FrameWork에서는 3D 원본 모델의 직교 광선 결과를 각 슬라이드마다 FBO(Frame Buffer Objects) Texture에 저장한다. ‘Point Cloud Generation’ 부분에서는 저장된 FBO Texture를 기반으로 PCD를 생성한다. 6 방향의 PCD를 결합한 후, 중앙값 필터를 적용하여 정점과 정점 사이의 구멍을 채운다.

3.1 6 방향 직교 광선(6-Direct Orthographic Ray)

이 섹션에서는 원본 3D 모델에 대한 6 방향 직교 광선에 대해 설명한다. 직교 광선 결과는 FBO Texture에 정점 데이터를 저장한다. 본 논문에서는 Axially aligned projection 직교 투영을 사용하였다.

Fig. 3의 (a)는 3D 오브젝트의 Z축 직교 투영 결과를 보여준다. Fig. 3의 (b)는 6방향의 직교 투영 결과를 보여준다.

직교 광선 사용 시 각 3D 모델 별로 매개변수를 받아야 한다. Table 1은 직교 광선 PCD를 만들기 위한 입력 매개변수를 보여준다.

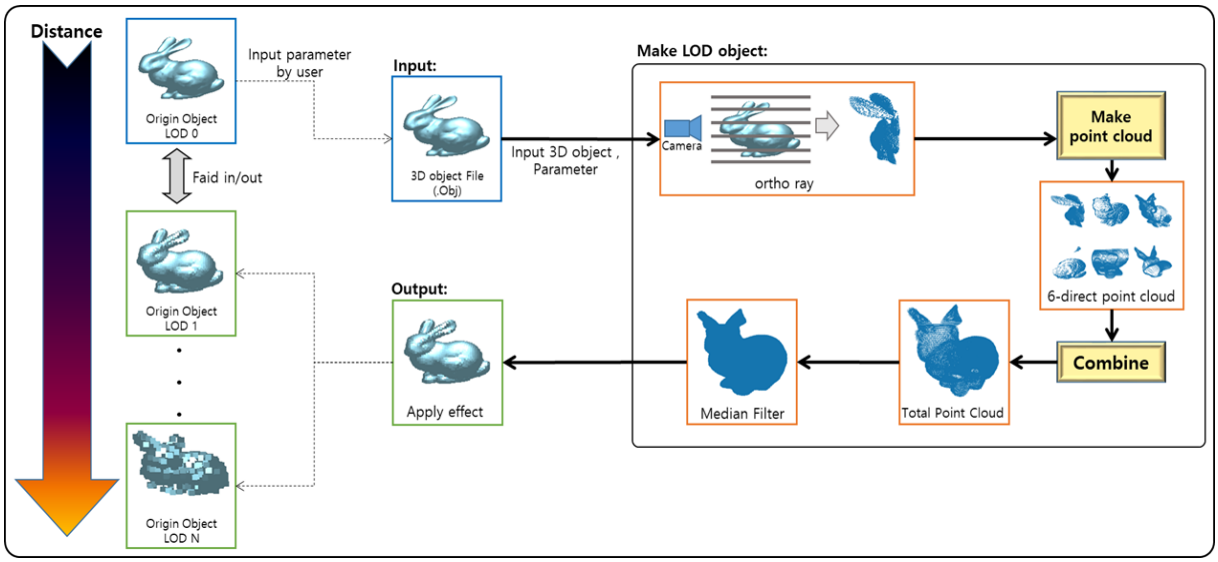

Fig. 2. Create LOD FrameWork

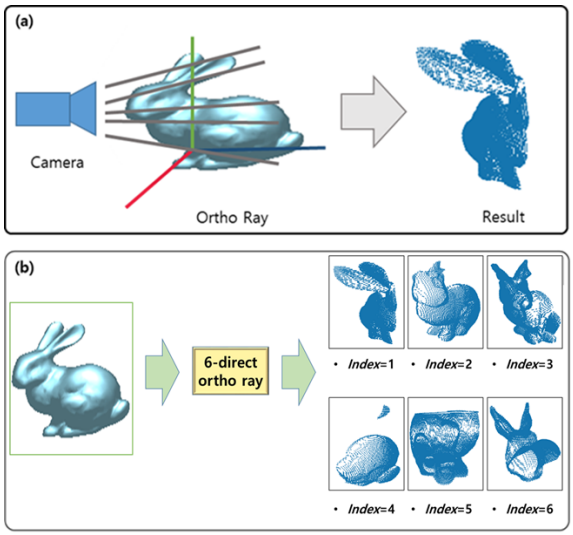

Fig. 3. 6-direct orthographic ray result

'Obj'는 원본 3D 모델을 의미한다. 'CamPos'는 카메라 뷰 포트 위치를 의미한다. 'CamDir'는 카메라 뷰 포트의 방향 벡터를 의미한다. 'Center'는 원본 3D모델의 위치를 의미한다. 'Dw'는 직교 행렬의 좌우, 'Dh'는 직교 행렬의 위아래를 의미한다. 'ObjLen'은 원본 3D 모델의 크기를 의미한다. 'Tdepth'는 슬라이드의 간격을 의미한다. 'LookMat'과 'PreMat'은 각각 3D의 Lookat Matrix와 Projection Matrix를 의미한다.

Table 1. Input parameter

Abbreviation	Details
Obj	원본 3D모델
CamPos	카메라 뷰 포트
CamDir	카메라 뷰 포트 방향
Center	원본 3D모델 위치
Dw	직교 행렬의 좌우
Dh	직교 행렬의 위아래
ObjLen	원본 3D 모델의 크기
Tdepth	슬라이드 간격
LookMat	Lookat matrix.
PreMat	Projection matrix.
Index	방향 Index
T_1	슬라이드 수(사용자입력)
T_2	Point 크기(사용자입력)
T_3	Texture 크기(사용자입력)
PointData	FBO 안의 Point data
Obj.Maxvec	원본 3D모델 크기의 최댓값
Obj.Minvec	원본 3D모델 크기의 최솟값

Table 2. 6-direct orthographic ray pseudocode

Algorithm 1

```
input : Obj ,index , T1 , T2 , T3
output : PointData

Center ← (Obj.Maxvec+Obj.Minvec)*0.5
ObjLen ← Obj.Maxvec-Obj.Minvec)
J← 0

while j ≤ T1 do
  if index = 1 or index = 2 then
    Tdepth ← ObjLen.z*0.5/ T1
    CamPos ← [Center.x,Center.y,Obj.Minvec.z+(Tdepth*j)]
    if index = 1 then
     CamDir ← [0,0,1]
    else if index = 2 then
     CamDir ← [0,0,-1]
    else if
    Dw ← ObjLen.x*0.5
    Dh ← ObjLen.y*0.5

  else if index = 3 or index = 4 then
    Tdepth ← ObjLen.x*0.5/ T1
    CamPos ← [Obj. Minvec.x+(Tdepth*j),Center.y,Center.z]
    if index = 3 then
      CamDir ← [1,0,0]
    else if index = 4 then
      CamDir ← [-1,0,0]
    else if
    Dw ← ObjLen.z*0.5
    Dh ← ObjLen.y*0.5

  else if index = 5 or index = 6 then
    Tdepth ← ObjLen.y*0.5/ T1
    CamPos ← [Center.x, Obj. Minvec.y+(Tdepth*j),Center.z]
    if index = 5 then
      CamDir ← [0,1,0]
    else if index = 6 then
      CamDir ← [0,-1,0]
    else if
    Dw ← ObjLen.x*0.5
    Dh ← ObjLen.z*0.5
  end if

  LookMat ← LookAt(CamPos,CamPos+CamDir,[0,1,0])
  PreMat ← Ortho(-Dw,Dw,-Dh,Dh,0.0,Tdepth*2)
  Obj.SetMatrix(PreMat*LookMat)
  FBORender(j , index , T2)
  ReadPixels(0, T2, 0 T3 ,PointData)
  j ← j+1

end while
```

'Index'는 6 방향 중 어떤 방향인지 정하는 Index이다. 'T_1'은 사용자 입력에 의한 슬라이드 개수이다. 'T_2'는 사용자 입력에 의한 정점의 크기를 의미한다. 'T_3'은 사용자 입력에 의한 FBO Texture 크기를 의미한다. 'T_1', 'T_2', 'T_3'는 사용자 입력에 따른 매개변수들로, 사용자 입력에 따라 LOD 품질을 정하게 된다. 'PointData'는 FBO Texture 내의 Point data를 의미한다. 'Obj.Maxvec', 'Obj.Minvec'는 각각 원본 3D모델 크기의 최댓값과 최소값을 의미한다.

Table 2는 6 방향 직교 광선에 대한 수도 코드를 보여준다. 원본 3D 모델을 기준으로 상,하,좌,우,앞,뒤 6방향의 직교 광선을 사용자 입력 변수 'T_1', 'T_2', 'T_3'에 따라 수행 후 FBO Texture에 저장한다. 6방향을 사용하는 이유는 원본 3D 모델의 구조에 따라 빈 공간이 생기는 것을 방지하기 위해서이다.

3.2 Point Cloud Generation

FBO texture에 Point data를 저장 후, 각 슬라이드 별로 PCD를 생성한다. 6방향의 직교 광선 후, FBO texture에는 각 슬라이드 별로 Point data가 저장된다. FBO texture의 x, y는 Point data의 각 Point 별 x, y 위치를 나타내고, 슬라이드 간격은 Point의 z 위치를 나타낸다. 모든 FBO texture의 Point data를 하나의 Buffer에 저장하면 PCD가 생성된다. 본 논문에서 PCD는 정밀한 소프트웨어에 의한 직교 광선 투영(Projection)으로 생성되었다. 그렇기 때문에 PCD안에 잡음은 존재하지 않는다.

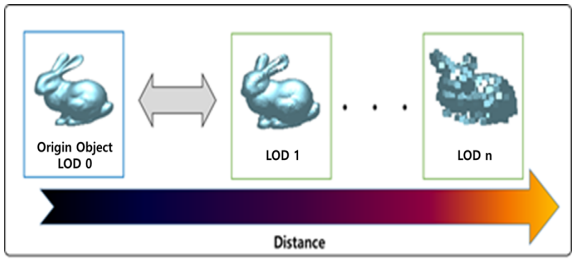

Fig. 4. Change of LOD level by distance

Fig. 4는 거리에 따른 LOD Level 변경 결과를 보여준다. 본 논문에서 제안된 LOD 생성 FrameWork는 한 매개변수당 하나의 LOD 3D 모델을 생성한다. 그러므로 사용자의 입력에 따라 거리별로 LOD를 몇 개 생성할 것인지 선택이 가능하다.

4. 실험 및 결과

이 섹션에서는 제안된 LOD 생성 FrameWork의 검증을 위하여 다양한 실험을 진행하였다. FBO Texture 크기에 대한 실험, 슬라이드 개수에 대한 실험, 정점 크기에 대한 실험, Smooth function에 대한 실험, 렌더링 속도에 대한 실험 그리고 Recall and Precision and F-measure을 이용한 지형 모델의 유사도 검증 실험. 모든 실험에 필요한 프로그램은 C/C++, OPENGL 4.0, OPENCV 3.0으로 개발하였고 실험 환경은 Intel-i5 CPU, 8GB RAM, 2개의 서로 다른 GPU(Nvidia GTX-1080Ti, Nvidia GTX-750)을 사용하였다. 본 논문은 Low-Level GPU에서 효율적인 LOD 생성 FrameWork를 제안하므로 Low-Level GPU인 모바일 디바이스에서도 실험을 진행하였다. 모바일 디바이스의 프로그램은 Android NDK, C/C++, OPENGL ES 2.0으로 개발하였고 두 개의 서로 다른 디바이스(LG Q7, ASUS Nexus 7 Tab)을 사용하였다.

4.1 FBO Texture 크기에 대한 실험

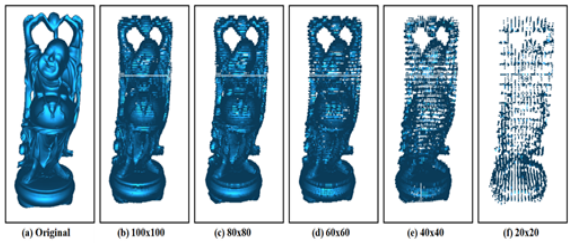

Fig. 5. results based on FBO Texture size

Fig. 5는 FBO Texture 크기에 따른 실험 결과를 보여준다. Fig. 5(a)는 원본 3D모델을 의미한다. Fig. 5(b)~(f)는 FBO Texture 크기를 각각 100×100, 80×80, 60×60, 40×40, 20×20으로 설정하고 실험한 결과이다. Fig. 5(b)와 같이 FBO Texture 크기가 클 경우, 정밀하게 LOD 3D모델이 생성되는 것을 확인할 수 있다. 반면, Fig. 5(f)와 같이 FBO Texture 크기가 작을 경우, 필요 이상으로 정점의 수가 줄어들어, LOD 3D모델에 많은 오류와 구멍이 생기는 것을 볼 수 있다. 이는 원본 3D모델 대비 왜곡이 심해진 것을 의미한다. 만약 모델이 단순한 정육면체인 경우, FBO Texture 크기를 2×2로 줄이면 정점의 개수가 8개로 축소된다. 더 심하게 FBO Texture의 크기를 1×1로 축소 할 경우, 모든 3D 모델의 정점은 1로 수렴하게 된다. 포인트의 기본 도형은 정사각형이기 때문에 3D모델이 정육면체일 경우 원형을 어느 정도 보존한 LOD가 될 수 있지만, 복잡한 3D 모델에서는 심한 왜곡으로 작용한다.

4.2 슬라이드 개수에 대한 실험

Fig. 6은 슬라이드 개수에 따른 실험 결과를 보여준다. Fig. 6(a)는 원본 3D모델을 의미한다. Fig. 6(b)~(f)는 슬라이드 개수를 각각 100, 80, 60, 40, 20으로 설정하고 실험한 결과이다.

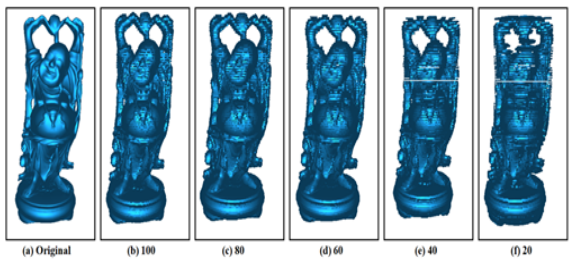

Fig. 6. Results by number of slides

Fig. 6(b)와 같이 슬라이드 개수가 많을 경우, 정밀하게 LOD 3D모델이 생성되는 것을 확인할 수 있다. 반면, Fig. 6(f)와 같이 슬라이드 개수가 적을 경우, 필요 이상으로 정점의 수가 줄어들어, LOD 3D모델에 많은 오류와 구멍이 생기는 것을 볼 수 있다. 이는 원본 3D모델 대비 왜곡이 심해진 것을 의미한다.

4.3 정점 크기에 대한 실험

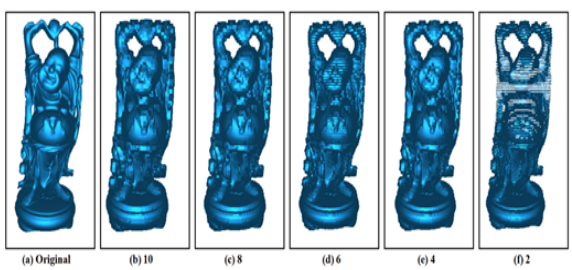

Fig. 7. Results by point size

Fig. 7은 정점 크기에 따른 실험 결과를 보여준다. Fig. 7(a)는 원본 3D모델을 의미한다. Fig. 7(b)~(f)는 정점 크기를 각각 10, 8, 6, 4, 2로 설정하고 실험한 결과이다. Fig. 7(b)와 같이 정점의 크기가 매우 클 경

우, 원본 3D모델의 크기보다 더 커져 구멍은 존재하지 않지만, 원본 3D 모델 대비 왜곡이 발생하게 된다. Fig. 7(d)의 경우, 약간의 구멍은 발생하더라도 크기가 원본 3D모델과 같은 것을 볼 수 있다. Fig. 7(f)와 같이 정점 크기가 매우 작을 경우, 구멍이 많이 발생하게 되고 크기도 작아져서 원본 3D 모델 대비 왜곡되는 것을 볼 수 있다. 사용자 입력 매개변수인 'T_1', 'T_2', 'T_3'는 정해진 값들이 아니며, 사용자가 상황에 맞게 알맞은 수치를 입력해 주어야 올바른 LOD 3D모델을 생성할 수 있다.

4.4 Smooth function에 대한 실험

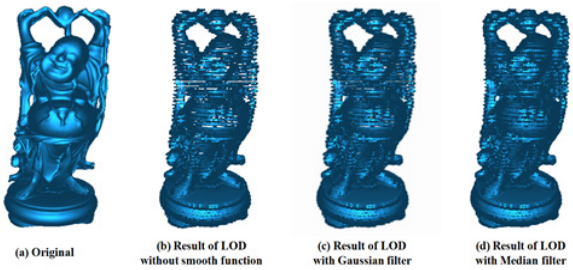

Fig. 8. Results according to Smooth function

Fig. 8는 Smooth function에 따른 실험 결과를 보여준다. Fig. 8(a)는 원본 3D모델을 의미한다. Fig. 8(b)는 LOD 3D모델에 Smooth function을 적용하지 않은 결과이다. Fig. 8(c)~(d)는 각각 Gaussian Filter, Median Filter를 적용한 결과이다. 이미지프로세싱에서 Median Filter가 Gaussian Filter보다 구멍을 더 정밀하게 채운다고 한다[26,27]. 본 논문에서도 Fig. 8(c), Fig. 8(d)와 같이 Median Filter는 Gaussian Filter 보다 구멍을 더 정밀하게 채우는 것을 볼 수 있다. 그러므로, 본 논문에서는 정점과 정점 사이를 채우는데 Median Filter를 사용하였다.

4.5 렌더링 속도에 대한 실험

Table 3은 LOD Level에 따른 3D 모델의 정점 개수를 나타낸다. Happy Buddha의 원본 정점 개수는 300,000개이지만, LOD 4에서는 22,301개로 LOD Level이 증가할수록 정점의 개수가 줄어드는 것을 볼 수 있다. Dragon 모델은 Happy Buddha와 정점의 개수는 같지만, Dragon 모델의 외형이 Happy Buddha 모델의 외형보다 단순하여 각 LOD Level 별로 Happy Buddha 보다 더 많은 정점이 줄어든 것을 확인할 수 있다. 본 논문에서 제안한 LOD 생성 FrameWork는 6방향의 직교 광선 투영을 사용하기 때문에 원본 3D 모델의 외형에 따라 생성된 LOD 3D모델의 정점 수가 결정된다.

Table 3. Number of vertices according to LOD level

	LOD 0 (Original) Number of Vertex	LOD 1 Number of Vertex	LOD 2 Number of Vertex	LOD 3 Number of Vertex	LOD 4 Number of Vertex
Happy Buddha	300,000	138,967	89,022	35,492	22,301
Dragon	300,000	109,486	69,952	27,251	17,478
Stanford Bunny	14,904	12,654	7,092	3,191	800
Stanford Lucy	1,346,640	81,566	52,162	39,897	20,354
Armadillo	1,037,832	64,200	41,211	31,621	16,102

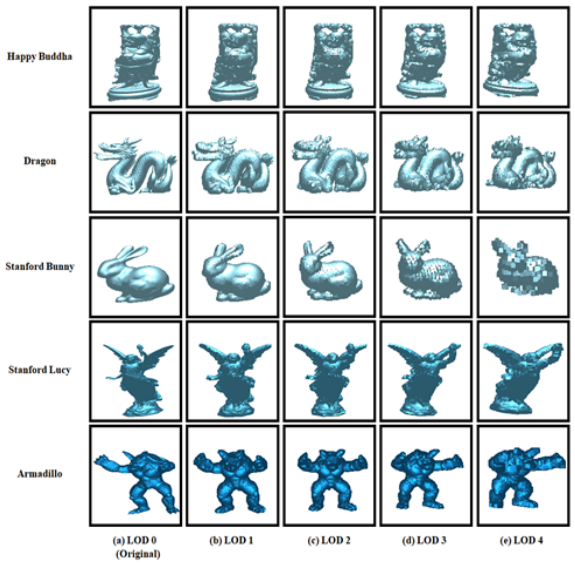

Fig. 9. Changes in LOD 3D models by LOD level

Fig. 9는 각 원본 3D모델 별 LOD Level에 따른 LOD 3D 모델 결과를 보여준다. Fig. 9(a)는 LOD 0 Level로 원본 3D모델을 의미한다. Fig. 9(b)~(e)는 각 3D 모델 별 LOD Level에 따른 LOD 3D 모델 렌더링 결과이다.

Table 4. Render FPS results without LOD on Nvidia GTX-750

	Happy Buddha	Dragon	Stanford Bunny	Stanford Lucy	Armadillo
121 objects	44	44	120	10	14
441 objects	14	14	120	4	4
961 objects	6	6	120	2	2
1681 objects	4	4	74	2	2
5041 objects	2	2	26	2	2
6561 objects	2	2	20	2	2

Table 4는 LOD 없이 Nvidia GTX-750에서 실험한 렌더링 속도 결과이다. Table 4의 첫 번째 열은 렌더링 될 3D 모델의 개수를 의미한다. 모든 FPS(Frame Per Second)는 렌더링 될 3D 모델의 개수에 따라 줄어들게 된다. Happy Buddha와 Dragon의 정점 개수는 같으므로 렌더링 FPS가 똑같이 측정된다. Happy Buddha, Dragon, Stanford Lucy, Armadillo는 1681개 이상부터는 4~2 FPS를 유지한다. 반면, 가장 정점이 적은 Stanford Bunny의 경우 6561개까지 20 FPS를 유지한다.

Table 5. Rendering FPS results after LOD application on Nvidia GTX-750

	Happy Buddha	Dragon	Stanford Bunny	Stanford Lucy	Armadillo
121 objects	56	66	120	24	32
441 objects	26	32	120	20	24
961 objects	18	22	120	16	20
1681 objects	14	18	120	14	18
5041 objects	8	12	120	10	12
6561 objects	6	8	100	8	10

Table 5는 LOD 적용 후 Nvidia GTX-750에서 실험한 렌더링 속도 결과이다. 모든 모델의 FPS는 Table. 4에 비해 증가한 것을 확인할 수 있다. Happy Buddha, Dragon, Stanford Lucy, Armadillo는 1681개 이상부터는 14~6 FPS를 유지하며, Stanford Bunny는 6561개에서도 100 FPS를 유지한다.

Table 6. Render FPS results without LOD on Nvidia GTX-1080Ti

	Happy Buddha	Dragon	Stanford Bunny	Stanford Lucy	Armadillo
121 objects	120	120	120	78	100
441 objects	104	104	120	22	30
961 objects	48	48	120	10	14
1681 objects	26	26	120	6	8
5041 objects	10	10	120	2	2
6561 objects	8	8	115	2	2

Table 6은 LOD 없이 Nvidia GTX-1080Ti에서 실험한 렌더링 속도 결과이다. 모든 모델의 FPS는 Table 4에 비해 증가하지만 6561개에서는 대부분(Happy Buddha, Dragon, Stanford Lucy, Armadillo) 모델이 8~2 FPS를 유지하는 것을 볼 수 있다. Stanford Bunny는 6561개에서도 115 FPS를 유지한다.

Table 7. Rendering FPS results after LOD application on Nvidia GTX-1080Ti

	Happy Buddha	Dragon	Stanford Bunny	Stanford Lucy	Armadillo
121 objects	120	120	120	120	120
441 objects	105	110	120	120	120
961 objects	50	62	120	106	112
1681 objects	44	50	120	74	80
5041 objects	22	32	120	38	46
6561 objects	18	22	120	19	28

Table 7은 LOD 적용 후 Nvidia GTX-1080Ti에서 실험한 렌더링 속도 결과이다. 모든 모델의 FPS는 Table 6에 비해 증가하지만 6561개에서는 대부분(Happy Buddha, Dragon, Stanford Lucy, Armadillo) 모델이 28~18 FPS를 유지하는 것을 볼 수 있다. Stanford Bunny는 6561개에서도 120 FPS를 유지한다.

Table 8은 LOD 없이 ASUS Nexus 7 Tab에서 실험한 렌더링 속도 결과이다. 169개 이상부터는 대부분(Happy Buddha, Dragon, Stanford Lucy, Armadillo) 모델

이 8~1 FPS를 유지하는 것을 볼 수 있다. Stanford Bunny는 289개에서도 28 FPS를 유지한다.

Table 8. Render FPS results without LOD in ASUS Nexus7-Tab

	Happy Buddha	Dragon	Stanford Bunny	Stanford Lucy	Armadillo
25 objects	24	24	40	10	12
81 objects	12	12	36	4	4
121 objects	10	10	34	1	2
169 objects	8	8	32	1	1
225 objects	6	6	30	1	1
289 objects	4	4	28	1	1

Table 9. Rendering FPS results after LOD application in ASUS Nexus7-Tab

	Happy Buddha	Dragon	Stanford Bunny	Stanford Lucy	Armadillo
25 objects	26	26	44	26	28
81 objects	18	22	40	20	24
121 objects	16	20	40	18	22
169 objects	14	18	38	18	20
225 objects	14	14	38	16	20
289 objects	12	13	36	13	16

Table 9는 LOD 적용 후 ASUS Nexus 7 Tab에서 실험한 렌더링 속도 결과이다. 모든 모델의 FPS는 Table 8에 비해 증가한 것을 확인할 수 있다.

Table 10. Render FPS results without LOD in LG Q7

	Happy Buddha	Dragon	Stanford Bunny	Stanford Lucy	Armadillo
25 objects	18	18	37	8	9
81 objects	10	10	34	4	3
121 objects	7	7	32	1	1
169 objects	6	6	31	1	1
225 objects	4	4	27	1	1
289 objects	3	3	25	1	1

Table 10은 LOD 없이 LG Q7에서 실험한 렌더링 속도 결과이다. 121개 이상부터는 대부분(Happy Buddha, Dragon, Stanford Lucy, Armadillo) 모델이 7~1 FPS를 유지하는 것을 볼 수 있다. Stanford Bunny는 289개에서도 25 FPS를 유지한다.

Table 11. Rendering FPS results after LOD application in LG Q7

	Happy Buddha	Dragon	Stanford Bunny	Stanford Lucy	Armadillo
25 objects	26	26	44	26	28
81 objects	18	22	40	20	24
121 objects	16	20	40	18	22
169 objects	14	18	38	18	20
225 objects	14	14	38	16	20
289 objects	12	13	36	13	16

Table 11은 LOD 적용 후 LG Q7에서 실험한 렌더링 속도 결과이다. 모든 모델의 FPS는 Table 10에 비해 증가한 것을 확인할 수 있다.

4.6 Recall and Precision and F-Measure을 이용한 유사도 분석

본 논문에서는 제안된 LOD 생성 Framework의 품질을 검증하기 위하여 'Recall and Precision'[3]을 사용하였다. 3D 오브젝트는 바라보는 방향에 따라 형태가 달라지기 때문에 일반적인 1:1 매칭으로 품질을 비교할 수 없다. 그래서 본 논문에서는 카메라의 움직임을 구면 좌표계로 설정하고 γ는 30~360, θ는 0~360, φ는 0~360만큼 이동하며 렌더링 화면을 저장하였다.

LOD 적용 없이 원본 3D 모델을 1,331개 렌더링한 영상을 정답으로 설정하였고, 본 논문의 LOD 생성 FrameWork로 5단계의 LOD를 생성하여 원본 물체와 거리에 따른 LOD 물체를 1,331개 렌더링한 영상을 예측으로 설정하였다.

정답 영상과 예측 영상의 픽셀을 1:1 비교하여, 정답 영상과 예측 영상 둘 다 픽셀이 있으면 TP, 예측 영상에만 픽셀이 있을 경우 FP, 정답 영상에만 픽셀이 있으면 FN으로 설정하였다. 마지막으로 정답 영상과 픽셀 영상 둘 다 픽셀이 없을 경우 TN으로 설정하였다.

$$ReCall = \frac{TP}{TP+FN} \quad (1)$$

$$Precision = \frac{TP}{TP+FP} \quad (2)$$

식 (1)과 식 (2)는 본 논문에서 사용된 Recall과 Precision의 계산 방법을 보여준다.

Table 12. Result of recall and precision and F-measure

Object Name	Recall	Precision	F-measure
Happy Buddha	0.949	0.969	0.959
Dragon	0.938	0.951	0.944
Stanford Bunny	0.950	0.981	0.965
Stanford Lucy	0.949	0.889	0.918
Armadillo	0.971	0.890	0.928
Total Average	0.952	0.944	0.947

Table 12는 본 논문에서의 'Recall and Precision'을 사용하였고 F-Measure로 최종 유사도를 측정한 결과이다. 평균 Recall의 평균 수치는 0.952고 Precision의 평균 수치는 0.944이다. F-Measure의 평균 수치는 0.947로, 제안된 LOD 생성 FrameWork는 평균 94.7 %의 유사도를 가진다.

4.7 대규모 지형 렌더링 결과

Fig. 10은 GTX-1080Ti에서 제안된 LOD 생성 FrameWork를 적용한 대규모 지형 렌더링 결과이다. 실험에 쓰인 지형 3D모델은 하나당 5백만 개의 정점을 가지고 5개의 서로 다른 LOD를 가지게 설정하였다.

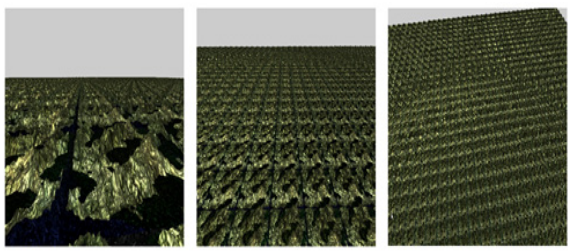

Fig. 10. Large terrain rendering results on GTX-1080Ti

총 1,681개의 지형을 실시간으로 렌더링하였으며 40 FPS를 유지하였다. F-Measure 값은 0.972로 평균 97.2 %의 유사도를 보였다.

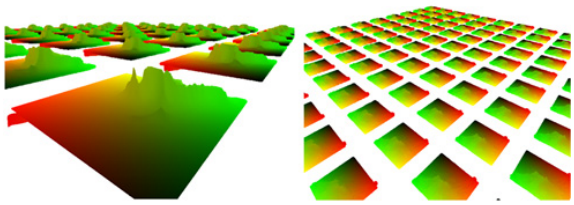

Fig. 11. Terrain rendering results on mobile devices

Fig. 11은 모바일 디바이스(ASUS Nexus 7 Tab, LG Q7)에서 제안된 LOD 생성 FrameWork를 적용한 지형 렌더링 결과이다. 실험에 쓰인 지형 3D모델은 하나당 280,488개의 정점을 가지고 5개의 서로 다른 LOD를 가지게 설정하였다. 총 225개의 지형을 실시간으로 렌더링하였으며 24~30 FPS를 유지하였다. 본 실험을 통하여 제안된 LOD 생성 FrameWork는 모바일 디바이스에서도 효율적으로 쓸 수 있음을 확인할 수 있다.

5. 결 론

본 논문에서는 대규모 크기의 지형과 수많은 3D모델을 렌더링하기 위하여 PCD를 이용한 자동 LOD 생성 FrameWork를 제안하였다. 6방향의 직교 광선 투영을 FBO Texture에 저장하여 LOD의 품질을 제어할 수 있게 하였다. 또한, 줄어든 정점을 다시 면(Mesh)으로 변환하면 원본 3D모델 대비 왜곡되는 현상을 막기 위하여 중앙값 필터(Median Filter)를 사용하여 정점과 정점 사이의 구멍을 채웠다. 제안된 LOD 생성 FrameWork는 평균 94.7 %의 유사도를 보인다. 본 논문이 대규모 훈련환경에서 지형 및 모델을 표현함에 더욱더 빠른 렌더링 속도 및 원거리 모델의 사실감 표현에 적극적으로 활용되길 기대한다.

References

[1] Schneider et al., "GPU-Friendly High-Quality Terrain Rendering," 2006.

[2] WEN, et al., Real-Time Rendering of Large Terrain

on Mobile Device, The International Archives of the Photogrammetry, Remote Sensing and Spatial Information Sciences, p. 37, 2008.

[3] Zhou et al., A Hybrid Level-of-Detail Representation for Large-Scale Urban Scenes Rendering, Computer Animation and Virtual Worlds, 25(3-4), pp. 243-253, 2014.

[4] Zienkiewicz et al., Monocular, Real-Time Surface Reconstruction using Dynamic Level of Detail, In 2016 Fourth International Conference on 3D Vision (3DV), IEEE, pp. 37-46, 2016.

[5] Burt and Adelson, The Laplacian PYRAMID as a Compact Image Code, IEEE Transactions on Communications, 31(4), pp. 532-540, 1983.

[6] Ghiasi and Fowlkes, Laplacian Pyramid Reconstruction and Refinement for Semantic Segmentation, In European Conference on Computer Vision, pp. 519-534, 2016.

[7] He et al. A System for Rapid, Automatic Shader Level-of-Detail, ACM Transactions on Graphics (TOG), 34(6), p. 187, 2015.

[8] Shreiner et al., OpenGL Programming Guide: The Official Guide to Learning OpenGL, Versions 3.0 and 3.1. Pearson Education, 2009.

[9] Rost et al., OpenGL Shading Language, Pearson Education, 2009.

[10] Sanz-Pastor et al., Volumetric Three-Dimensional Fog Rendering Technique, U.S. Patent 6,268,861, 2001.

[11] Decaudin and Neyret, Volumetric Billboards, In Computer Graphics Forum, Oxford, UK: Blackwell Publishing Ltd., Vol. 28, No. 8, pp. 2079-2089, December 2009.

[12] Mantler et al., Displacement Mapped Billboard Clouds, In Proceedings of Symposium on Interactive 3D Graphics and Games, Citeseer, January 2007.

[13] Vichitvejpaisal and Kanongchaiyos, Enhanced Billboards for Model Simplification, 2006.

[14] Lorensen et al., Marching Cubes: A High Resolution 3D Surface Construction Algorithm, In ACM Siggraph Computer Graphics, ACM, Vol. 21, No. 4, pp. 163-169, August 1987.

[15] Jabłoński, S. and Martyn, Real-Time Voxel Rendering Algorithm based on Screen Space Billboard Voxel Buffer with Sparse Lookup Textures, 2016.

[16] Baert et al., Out-of-Core Construction of Sparse Voxel Octrees, In Proceedings of the 5th High-Performance Graphics Conference, ACM, pp. 27-32, July 2013.

[17] Laine and Karras, Efficient Sparse Voxel Octrees, IEEE Transactions on Visualization and Computer Graphics, 17(8), pp. 1048-1059, 2011.

[18] Ginsburg et al., OpenGL ES 3.0 Programming Guide, Addison-Wesley Professional, 2014.

[19] Brothaler, OpenGL ES 2 for Android: A Quick-Start Guide, Pragmatic Bookshelf, 2013.

[20] Linsen, Point Cloud Representation, Technical Report, Faculty of Computer Science, University of Karlsruhe: Univ., Fak. für Informatik, Bibliothek, 2001.

[21] Zhao and Zhu, Image Parsing with Stochastic Scene Grammar, In Advances in Neural Information Processing Systems, pp. 73-81, 2011.

[22] Zheng et al., Beyond Point Clouds: Scene Understanding by Reasoning Geometry and Physics, In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 3127 -3134, 2013.

[23] Yan et al., 3D Point Cloud Map Construction based on Line Segments with Two Mutually Perpendicular Laser Sensors, In 2013 13th International Conference on Control, Automation and Systems(ICCAS 2013), IEEE, pp. 1114-1116, October 2013.

[24] Eckart et al., Accelerated Generative Models for 3D Point Cloud Data, In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, pp. 5497-5505, 2016.

[25] Belton et al., Processing Tree Point Clouds using Gaussian Mixture Models, Proceedings of the ISPRS Annals of the Photogrammetry, Remote Sensing and Spatial Information Sciences, Antalya, Turkey, pp. 11-13, 2013.

[26] Arce and Gonzalo, Nonlinear Signal Processing: A Statistical Approach, John Wiley & Sons, 2005.

[27] Camplani and Salgado, Efficient Spatio-Temporal Hole Filling Strategy for Kinect Depth Maps, In

Three-Dimensional Image Processing(3DIP) and Applications Ii(Vol. 8290, p. 82900E), International Society for Optics and Photonics, January 2012.

[28] Touma et al., 3D Mouse and Game Controller based on Spherical Coordinates System and System for use, U.S. Patent 7,683,883, 2010.