GPUを用いた点群データの高速描画

粟野 直之† 西尾 孝治† 小堀 研一†
大阪工業大学†

1 はじめに

近年の3次元スキャナの発達により，高密度な点群データの取得が容易になってきた．それに伴い，ポイントベースグラフィックスが発展し，点群データを直接モデリング及びレンダリングする研究が盛んに行われている[1]．ところが，点群データをレンダリングする際には予め各点に法線ベクトルを付与する，あるいは前処理によってレンダリングするためのデータを作成する必要がある．そこで本研究では，法線ベクトルを入力とせず陰点消去とシェーディングを行い，GPUを用いることで処理コストを抑えて高速にレンダリングできる手法を提案する．

2 提案手法

2.1 処理の概要

提案手法では法線ベクトルを持たない点群データを入力する．まず，陰点消去として形状表面によって隠されるべき点を消去するためのテクスチャを作成する．次にシェーディングとして，拡散反射を表現するためのテクスチャと鏡面反射を表現するためのテクスチャを作成する．最後に，上記の3つのテクスチャを参照し，各点を描画するかどうかを判定することでレンダリング結果を出力する．

2.2 陰点消去

最初に，GPU上にスクリーン解像度以上のテクスチャを作成してスクリーン位置に設置する．そして，図1(a)に示すように作成したテクスチャ上に点群データを投影し，対応するピクセルに最も小さいデプス値を格納しておく．このとき，同図中の数値は投影した点A～Cの各デプス値である．

次に，形状表面によって隠されるべき点を消去するため，テクスチャ上で各点の膨張を行う．同図(b)に示すように，各ピクセルのデプス値を周囲のピクセルに格納し，それぞれ最小のデプス値を保存する．これにより，隠されるべき点のデプス値は描画されるべき点のデプス値で上書きされ，テクスチャ上で消去されることになる．

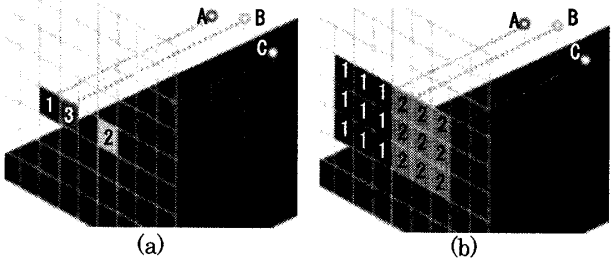


図1 テクスチャへの点の投影

しかし，このような陰点消去バッファでは点が密集している個所で多くの点が消去され，必要以上に点が削除されてしまう問題がある．そこで，周囲にデプス値を格納する際に，一定値大きくしたデプス値を格納することで，図2に示すような結果のテクスチャが得られ，このテクスチャを用いて陰点消去を行う．ここで，同図では一定値を5としている．

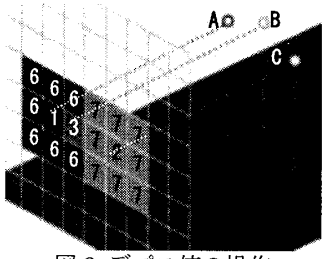


図2 デプス値の操作

2.3 拡散反射

拡散反射は全体的に均等に点を非表示とすることで表現する．そこで，図3に示すようなテクスチャのFull Treeを作成する．以後の説明のため，同図に示すようにFull Treeの根から順にLEVEL 0～nと定義する．次に，陰点消去時と同様に，同図の各ピクセルには最小のデプス値を1つ保存し，保存したデプス値に対応する点を非表示とする．そして，LEVELを1つ指定し，指定したLEVELが保持するデプス値に対応する点をすべて非表示とすることで，拡散反射の割合を変更できるようにする．上記のFull Treeを GPU上で効率よく管理するため，提案手法では図4に示すようにGPU上のテクスチャに各LEVELに対応するテクスチャを保存しておく．

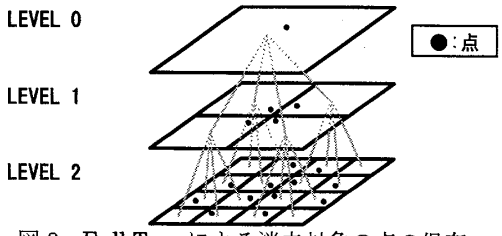


図3 Full Treeによる消去対象の点の保存

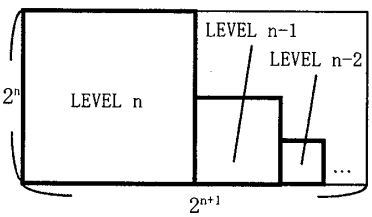


図4 テクスチャ上でのFull Treeの構成

2.4 鏡面反射

鏡面反射はフォンのモデルを参考にする．一般に法線ベクトルが入力された場合，法線ベクトルと光線ベクトルの内積により鏡面反射の強さが決定される．しかし，

A Fast Rendering for Point Cloud using GPU
† Naoyuki Awano, Koji Nishio, Ken-ichi Kobori
Osaka Institute of Technology

提案手法では法線ベクトルを入力しないため，フォンのモデルを適用できない．

ここで，例えば図 2 において点 A をもとに格納したデプス値は 8 ピクセル残されているように，鏡面反射の強さと残されたピクセル数はほぼ比例している特徴がある．この特徴に注目し，ピクセル数をカウントすることで法線ベクトルとして代用する．これに伴い，鏡面反射を表現するためのテクスチャは陰点消去と同様の処理によって実現する．ただし，視点と光源の位置は必ずしも一致しないため，陰点消去と鏡面反射を表現するための各テクスチャは同じ内容になるとは限らない．

2.5 描画の判定

前節までに作成したすべてのテクスチャを用いて，点群データの各点を描画するかどうかを判定する．

陰点消去では図 2 に示すように，点群データの各点をテクスチャ上に投影する．例えば，同図中の点 A を投影した場合，投影されたピクセルとその周囲のピクセルに格納されているデプス値(1,6,6,6,6,6,6,6,3)を参照する．このとき，点 A のデプス値は 1 であり，参照したデプス値以下の値であることがわかる．このような場合は描画するとし，反対に参照したデプス値以下の値が 1 つもない場合は消去する．

拡散反射では非表示の対象として LEVEL l を指定した場合を例に説明する．まず点群データを図 4 中の LEVEL n の位置に投影する．このとき，投影されたピクセルを (x,y) とし，式(1)を用いてピクセル (x',y') を参照する．投影した点と参照したピクセルが保持するデプス値が等しいかを調べ，デプス値が等しい場合はその点を非表示とすることで拡散反射を表現する．

$$(x',y')=\begin{cases}(x,y) & (l=n)\\ \left(\left(x/2^{n-l}\right)+\sum_{k=l+1}^{n}2^k,\ y/2^{n-l}\right) & (l<n)\end{cases} \qquad (1)$$

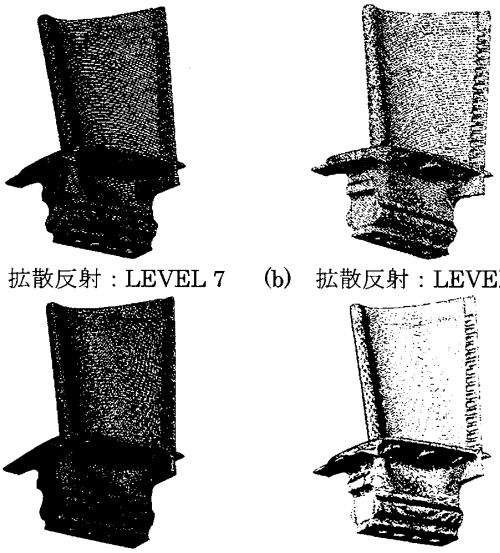
鏡面反射では前節で述べたようにピクセル数をカウントすることで表現する．まず，陰点消去と同様に点群データの各点をテクスチャ上に投影する．そして，投影されたピクセルが保持するデプス値とその周囲のピクセルが保持するデプス値を参照し，投影した点のデプス値との比較を行う．例えば，図 2 中の点 B の場合，参照するピクセルのデプス値は(3,6,6,7,7,7,6,6,1)となっており，点 B のデプス値は 3 となっている．従って，参照するピクセルのデプス値の中で，点 B のデプス値 3 以下のデプス値は 2 つ存在している．ここで，テクスチャ上には 1 点あたり最大 9 ピクセル保存されていることから，点 B には 2/9 とする鏡面反射率を定義して付加する．

次に，定義した鏡面反射率に対して閾値を設定することで鏡面反射を表現する．例えば閾値を 1/9 とした場合，1/9 以上の鏡面反射率を保持する点をすべて非表示とする．従って，閾値を大きくすれば非表示となる点が少ないため，弱い鏡面反射の表現となり，反対に閾値を小さくすれば非表示となる点が多くなるため，強い鏡面反射の表現となる．

3 実験と考察

提案手法の有効性を検証するために実験を行った．実験環境は Intel Core2 Duo E6700 2.66GHz，2.0GB RAM，GeForce 8800GTX 768MB とし，陰点消去と鏡面反射のテクスチャサイズは 1024×1024，拡散反射のテクスチャサイズは 2048×1024 とした．

まず，図 5(a)と(b)に拡散反射のみを表現した結果を示す．ここで，同図中の LEVEL は非表示の対象としている LEVEL である．同図より光が拡散するような表現ができ，LEVEL を変更することでその強さを変更できることが分かる．次に，同図(c)と(d)に鏡面反射のみを表現し，閾値を変更してその強さを変更した結果を示す．同図より，光沢があるような表現ができ，閾値を変更することでその強さを変更できることが分かる．



(a) 拡散反射：LEVEL 7　(b) 拡散反射：LEVEL 9

(c) 鏡面反射：閾値 大　(d) 鏡面反射：閾値 小

図 5 レンダリング結果

次に，CPU での実装と GPU での実装の処理時間を比較した．比較実験の結果を図 6 に示す．ここで，同図はすべての処理の合計時間を表し，実験形状名の上の数値は点数を表している．同図より，GPU を用いることで約 8~13 倍高速にレンダリングできることが分かる．

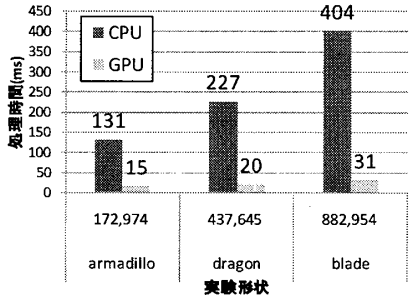


図 6 処理速度

4 おわりに

本研究では法線ベクトルを持たない点群データの新しいレンダリング手法を提案した．また，GPU を用いることで処理コストを抑えることができることを確認した．

今後の課題として，現在は点の色を表 2 値で表現しているが，多値にすることでより表現の幅が広がると考えられる．

参考文献

[1] M.Gross & H.PFister：Point-Based Graphics, Morgan Kaufmann Publishers (2007)