GPU による直接的ポイントレンダリング

川田 弘明　　金井　崇
慶應義塾大学 環境情報学部　　理化学研究所
t02282hk@sfc.keio.ac.jp　　kanait@acm.org

概要．本論文では，位置情報のみを持つ点群データを，GPU を利用して直接レンダリングする方法について提案する．本手法におけるレンダリング処理のほとんどは GPU により実行され，高品質なレンダリング結果を高速に得ることができる．我々のアルゴリズムでは，イメージバッファと呼ばれる，フレームバッファより解像度の低い画像を一枚用意し，その画像を利用して法線ベクトルの計算や，測定点群データに含まれるノイズの軽減処理が行われる．実際の描画処理はスプラットにより行われるが，スプラットの描画処理に必要なフィルタリング処理は，画像バッファの中で選択されたピクセルにのみ適用される．このため，点群データを直接フィルタリングする手間を省くことができ，処理を高速化することができる．実際，多くの例題において，法線計算やノイズ軽減処理を含む我々のアルゴリズムは，既存のアルゴリズムよりも高速に処理することが可能である．

1　はじめに

近年の 3D 計測技術の進展と急速な普及により，点群曲面データがコンピュータグラフィックスの分野の中ではより重要かつ魅力的なものとなっている．3D 計測デバイスより大量の点群データを得ることが可能となった反面，これらのデータは，大容量であるが故にその処理や編集，表示を難しくしている．そこで，これらの点群から三角形メッシュや高次の曲面を生成するかわりに，接続情報の計算やサンプリングの分布に制約を課すことなく，点群を直接操作する点群ベースのアプローチが注目されている．

その中でも，点群レンダリングは特に大容量かつ複雑なモデルの表示に有効である．数百万の小さな三角形面から構成されるモデルの表示において，スクリーンへの投影面積は数ピクセル以下であるため，三角形のセットアップによるオーバーヘッドが大きい．このことが，点群レンダリングに代替することへの大きな動機となっている．さらに，最近のグラフィックスハードウェア (GPU) の進展は著しく，ラスター化のプロセスの大部分を制御することが可能である．GPU の利用により，点群レンダリングにおける処理をさらに加速することができる．

本稿では，点群を GPU によって直接レンダリングする手法について提案する．なお，本稿で述べる「直接的ポイントレンダリング」とは，**点群データの位置情報のみを利用した**レンダリング手法であることを意味する．本手法は，特に 3D 計測デバイスから得られる大容量の点群データをレンダリングする際に有効である．これは，計測デバイスから得られるのは，主に点群の位置情報や色情報のみだからである．一方で，レンダリングの際シェーディング処理を行うためには，点の位置情報の他に向きを示す法線ベクトルが定義されている必要がある．この法線ベクトルは，点群の位置情報から直接計算することが可能である (例えば [9, 10]) が，前処理により計算しデータとして保持する必要がある．一つの頂点につき一つの法線ベクトルが定義されているものとすれば，法線ベクトルデータの格納には，点の位置データと同じだけの空間量が必要となる．このことは，特に大容量データの扱いにおいて大きな障害となり得る．

我々の手法の基本的なアイデアは [12] に基づくが，そのレンダリング処理のほとんどは GPU により実行される．さらに，GPU に実装するためのアルゴリズムは [12] とは本質的に異なるものである．本手法の利点として以下の三つが挙げられる:

- GPU 上で法線ベクトルの計算が直接行われる．これより法線ベクトルを前処理によって計算する手間が省ける．
- GPU 上で点群データに含まれるノイズの軽減処理が行われる．一般に 3D 計測デバイスによって得られる点群データにはノイズが含まれるため，このようなデータの描画には有効である．
- 実際の描画処理はスプラットにより行われるが，スプラットの描画処理に必要なテクスチャフィルタリングは，イメージバッファの中で選択されたピクセルにのみ適用される．一般に，その数はオリジナルの点よりも少ないため，描画にかかる処理を高速化することができる．

2　関連研究

点群レンダリングを含む点群ベース手法に関しての詳細は [14, 21, 13] に詳しく述べられている．ここでは，点群レンダリングに絞って関連研究の紹介を行う．点群レンダリングは，Levoy と Whitted [15] によって初めて導入された手法である．点群レンダリングにおける重要な処理の一つに，点と点の隙間を埋める穴埋め処理が挙げられる．その手法は，大別してスクリーン空間における処理とオブジェクト空間における処理の二つに分けることができる．本手法は，本質的に前者の手法として分類できる．

Grossman と Dally [6] は，点群の間の穴埋め処理を Gortler らの pull-push 法 [5] を応用して行った．pull-push 法とは，階層化された解像度の異なる複数の画像をあらかじめ用意しておき，高解像度の画像における点と点の隙間を，低解像度の画像を用いて埋めるという方法である．後にこの方法は Pfister らによって改良され [16]，オブジェ

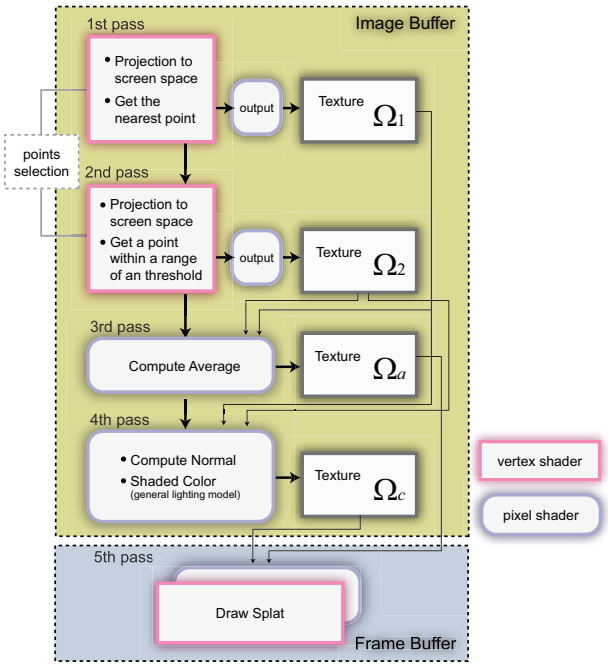


図 1: GPU による直接的ポイントレンダリングの描画手順.

クトの階層的表現とテクスチャフィルタリングを用いて描画品質を向上している． Shade ら [18] は， LDI (*Layered Depth Image*) と呼ばれる，階層的な深さを持つピクセル群を利用して面の再構築を行っている． Kalaiah ら [11] は， DP (*Differential Points*) と呼ばれる，位置情報の他に曲率に関する情報を利用した高品質な点群レンダリング手法を提案している．

ポイントの描画においては，各点に対し，スプラットと呼ばれる四角形や円，楕円を定義し，重複させることで点間の隙間を埋めるとともに，フィルタリングによりアンチエイリアシングをかける，という方法がとられている．Zwicker ら [19] は EWA スプラッティングと呼ばれるガウシアンフィルタカーネルを用いた点群レンダリング手法を提案し，高品質な描画結果を得ている． Botsch ら [3] は，空間分割データ構造に基づくルックアップテーブルを利用することで， EWA スプラッティングの描画速度を改善した．

一方，ポイントレンダリングにおける GPU の利用に関しては，その多くが上記フィルタリング処理の GPU による実装について議論されている． Ren ら [17] は，オブジェクト空間での異方性 EWA テクスチャフィルタリングと GPU による高速化を行ったが，各スプラットを一つの四辺形で描画するため，描画速度がかかる (各スプラットにつき 4 つの頂点を GPU に転送する必要があるため) という問題点があった．その後， Guennebaud ら [8] や Botsch [1] らによりポイントスプライトで描画する手法が提示され，また， Zwicker [20] らによって，正確なオブジェクト空間異方性 EWA テクスチャフィルタリングの GPU による高速化が行われた．また， Botsch ら [2] は，各スプラットに法線場を定義することで，ピクセル単位でのライティング (Phong シェーディング) を GPU により実現した．その他， GPU の利用方法としては，線形のポイントリストを用いた GPU への転送の効率化 [4] や，点選択アルゴリズムの実装 [7] など，描画の効率化のための利用がいくつか見受けられる．

これらの一連の研究に対し，本手法では，法線ベクトルの計算やノイズの軽減処理をも GPU で実装している．テクスチャフィルタリングに関しては他の方法を流用することも可能である．

3 GPU による描画アルゴリズム

図 1に， GPU による直接的ポイントレンダリングアルゴリズムの手順を示す．入力は点群 $\mathcal{P}$: $p_i \in \mathcal{P}$ $(i = 1 \ldots n$, n は点群の数) とする．各点は三次元座標 $\mathbf{p}_i$ のみを持つ必要があるが，場合によっては色情報 $\mathbf{c}_i$ を持つことも可能である．

点間の隙間を埋める処理において， [12] では，イメージバッファ (*Image Buffer*) と呼ばれる，フレームバッファよりも低解像度のバッファにデータを格納することで処理を行っていた．本手法においても， 1 〜 4 パス目の処理はこのイメージバッファの中で行われるが，最後のパスでフレームバッファの解像度に拡大し，スプラットによる描画処理が行われる．また，イメージバッファと同じ解像度の浮動小数点テクスチャ Ω を，各パスにおける入出力として GPU の VRAM メモリ上に複数確保する．

3.1 イメージバッファの解像度の決定

まず 1, 2 パス目で，点群をイメージバッファ[1]に投影し格納する (詳細は 3.2 節に後述) ． この際，点間の隙間を埋めるために，イメージバッファの解像度を適切に決定する必要がある．イメージバッファの解像度は，点群の解像度やカメラ情報 (視点の位置，視野角，視錐の大きさ等) によって動的に決定される．

フレームバッファの幅と高さをそれぞれ w_s, h_s，イメージバッファの幅と高さをそれぞれ w_i, h_i とすると， w_i, h_i は透視投影の式より導かれる次式をもって計算する．

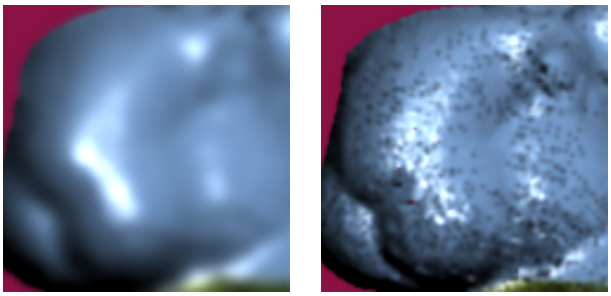
$$w_i = w_s/\rho, \quad h_i = h_s/\rho, \tag{1}$$

$$\rho = \frac{\sigma}{\tan(f/2)} \cdot \frac{1}{\lambda} \cdot w_s, \tag{2}$$

$$\lambda = \frac{z_f \cdot z_n}{z_f - z_n} \cdot |\mathbf{v} - \tilde{\mathbf{p}}| + 1. \tag{3}$$

ここで， f は視野角， z_f, z_n はそれぞれ視点から視錐台の後方クリップ面，前方クリップ面への距離， $\mathbf{v}$ は視点座標， $\tilde{\mathbf{p}}$ は点群 $\mathcal{P}$ の重心座標を示す． σ は点群の解像度，すなわち隣接点間の間隔を見積るためのパラメータである． σ は次のように計算できる: 点群データ全体を $k-$ 隣接グラフとして定義し，各点に対して最も近い点との距離を計算してそれらの最小値とする． $k-$ 隣接グラフの定義と近傍点の計算は視点に依存することがないため， σ の

[1]GPU でのイメージバッファは，浮動小数点バッファを利用する．DirectX では，テクスチャのフォーマットを IEEE フォーマット `D3DFMT_A32B32G32R32F` と指定することで利用できる． OpenGL では pbuffer を利用できる．



(a) (b)

図 2: イメージバッファの解像度による描画結果への影響. (a) 解像度が低い場合の描画結果. (b) 解像度が高い場合の描画結果.

計算は前処理（点群データの入力時）により行うことが可能である．上記の式より，ρ は必ず 1 よりも大きい値が設定され，その結果，w_i, h_i は，それぞれ w_s, h_s よりも小さい値をとる．

イメージバッファの解像度は，最終的な描画結果に大きな影響を及ぼす．図 2 は，解像度を変化させた場合の描画結果に及ぼす影響を調べるために，ρ の値を，上記の計算式を用いずに意図的に大小させた場合の結果を示している．解像度を低くした場合 (図 2(a))，隙間は埋めることができるものの描画結果にぼけが現れる．これは，点群の情報がイメージバッファに記録される際，各ピクセルにより多くの点が含まれてしまうためである．また，解像度を高くした場合 (図 2(b))，点が格納されないピクセルが出てくるため，隙間ができてしまう結果となる．

3.2 ノイズ軽減のための点群の選択的格納と平均化

図 1 の 1, 2 パス目では，頂点シェーダを用いたノイズの軽減のためのイメージバッファへの点群の選択的格納処理が行われる．GPU では，1 回のパスにつき，イメージバッファの各ピクセルに 1 点ずつしか格納できないため，パスの回数を増やすことで，複数の点をパスの回数毎に格納していく．

各回のパスにおいて，点それぞれを投影変換することによりイメージバッファに格納していく．各回のパスにおける出力は，あらかじめ用意しておいたテクスチャ Ω に設定しておく．この処理は，GPU の頂点シェーダとピクセルシェーダを通して行われる．頂点シェーダでは，入力点群のカメラ座標系への投影変換が行われ，ピクセルシェーダでは単に値がバッファに書き込まれる．ピクセルシェーダにおける処理の後，バッファの内容はテクスチャに書き出される．この際，各頂点を投影することにより，イメージバッファの対応するピクセルに，点座標が浮動小数点として格納される．

図 3 に，イメージバッファへの選択的点群格納処理の手順を示す．一つのピクセルに対応する点が複数存在する場合，1 回目のパスでは，視点に最も近い点のみが選択され，これらのデータをテクスチャ Ω_1 として格納する (図 3左).

2 回目のパスにおいても，1 回目と同様投影変換によるイメージバッファへの点の格納が行われるが，その際，1

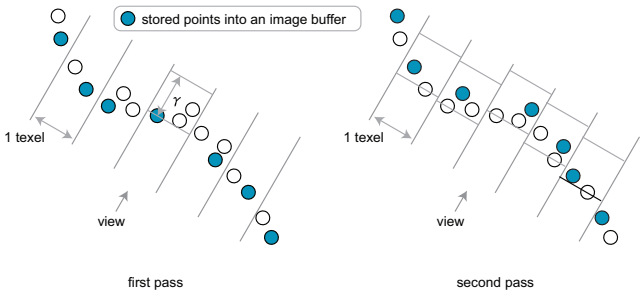


図 3: イメージバッファへの選択的点群格納処理の手順.

回目に選択された点より，視点から見た深さが閾値 γ の範囲にあり，かつ最も遠い点を選択する (図 3右). この操作は，ノイズにより大きく隔離した点を排除することと，ノイズを平均化して削減することを目的としたものである．この選択的格納処理には，まず 2 パス目の頂点シェーダ内で Ω_1 の点を参照し，深さが閾値 γ の範囲にあるかどうかを調べ，範囲外の点に関しては排除する．γ を設定することにより，例えば，裏の面を構成する点や，誤差により大きく外れた点に関する処理を省くことができる．この γ は視点に依存しないパラメータのため，前処理によって一定の値を設定することができる．我々の実験の結果，物体を囲むバウンディングボックスの対角線の長さの 0.5 〜 1 % 程度をとると，良好な結果が得られることがわかった．

次に，GPU のデプステスト処理において「既に書き込まれている値より大きい場合に書き込む」と設定する[2]ことで，γ の範囲内にある最も遠い点が選択される．このように選択された点をテクスチャ Ω_2 として格納する．

1 パス目と 2 パス目において，点の位置情報の他に色情報を持つ場合，MRT (*Multiple Renger Targets*) を利用することによって，単一のパスでその両方を別々のテクスチャに書き出すことが可能である．MRT は，ピクセルシェーダの処理の結果を同時に複数のテクスチャにおいて書き出すことのできる機能である．さらに 2 パス目では，1 パス目の結果が書き込まれているテクスチャを頂点シェーダから参照する必要があるが，これはシェーダモデル 3.0 よりサポートされた頂点テクスチャ (*Vertex Texture*) 機能を利用することで実現できる．いずれも，*n*VIDIA GeForce 6 以降の GPU でサポートされている機能である．

3 パス目では，ピクセルシェーダを用いて Ω_1, Ω_2 の各ピクセルを参照し，その平均値を求めてテクスチャ Ω_a に格納する (この値はスプラットの描画の時にのみ利用). この Ω_a は，最後のパスでのスプラットによる描画処理における，点の位置情報として利用される．

図 4 は，1, 2 パス目で行われている選択的格納処理によるノイズ削減効果を実証すべく，スタンフォード兎の測定機からの点群データ (362,272 点) をレンダリングした結果である．図の真中と図右は，それぞれ 2 パス目を行わなかった場合と行った場合の拡大図である．選択的格納処理によって若干ノイズが平滑化され，削減されているのが確認できる．

[2]DirectX では `D3DCMP_GREATER` というフラグで設定できる.

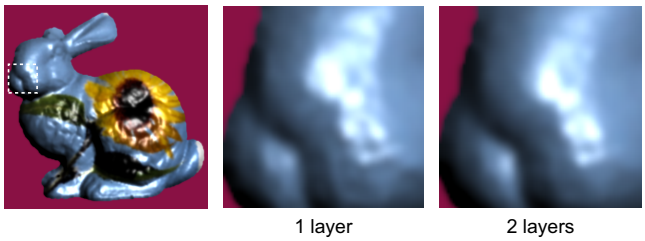


図 4: ノイズの軽減処理の効果.

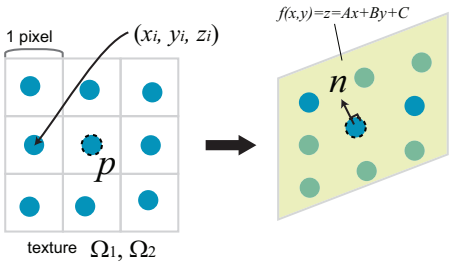


図 5: 法線ベクトルの計算.

3.3 法線ベクトルの計算

本節では，GPU を利用した法線ベクトルの計算手法について説明する．法線ベクトルの計算は，図 1 の描画アルゴリズムの中の 4 パス目において，ピクセルシェーダを利用して行われる．

法線ベクトルの計算では，ある点に対する隣接点情報が必要となる．隣接点の迅速な取得のために，前節の選択的格納処理により得られるテクスチャ Ω_1, Ω_2 を利用する．図 5 に法線ベクトル計算の原理を示す．図 5左において，テクスチャ内のあるピクセルに格納される点 p とその隣接点 (最大で 18 点) を用いて，p における法線ベクトルを計算する．

ここでは，カメラ座標系により定義された点 p と，その隣接点より張られる近似的な平面

$$f(x, y) = z = Ax + By + C, \tag{4}$$

を，最小自乗法により求める (図 5右)．まず，ピクセルシェーダ内で点 p とその隣接点の座標をテクスチャ Ω_1, Ω_2 より参照する．これらの座標を $p_i = \{x_i, y_i, z_i\}$ $(i = 1 \ldots N)$ とすると，点 p に対し，

$$\sum_{i=1}^{N} \{z_i - (Ax_i + By_i + C)\}^2 \rightarrow min, \tag{5}$$

を満たすような係数 A, B, C を計算する．この計算は，A, B, C を変数とする三元連立一次方程式となる．連立方程式の解法のうち，我々はガウスの消去法およびクラーメルの公式をピクセルシェーダ上で実装してみたが，速度的にはほぼ同じであり，またシェーダの命令数もほとんど変わらなかった (ガウスの消去法が 336, クラーメルの公式が 340)．ガウスの消去法は数値解法であるが，そのコードを展開して記述することで，どちらの方法でも解を解析的に求めることができる．計算された係数より，法線ベクトルは $(A, B, -1)$ として求められる．求めた平面の法線ベクトルと，平均化された点座標を用いてシェーディングを行い，色情報をテクスチャ Ω_c に格納する．

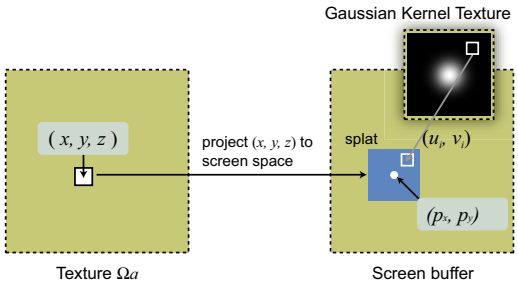


図 6: ガウシアンカーネルテクスチャによるスプラットの描画手順.

法線ベクトルの計算では，カメラ座標系で定義された点群に対し，式 (4) の平面式を当てはめている．しかし，この平面では，視点の向きと垂直となるような平面を定義することはできない．ただし，法線ベクトルが視点と完全に垂直になるような状況は極めて起こりにくいため，実用的には特に問題ないように思われる．近似平面式に関しては，より一般的な式，例えば $Ax + By + Cz + D = 0$ を用いることも考えられる．この場合，係数を計算するためには，ニュートン法等の数値解法を利用する必要があり，GPU での計算を行うには計算負荷が高くなる．

3.4 スプラットによる描画

図 1 の 5 パス目では，浮動小数点テクスチャ Ω_a を位置情報とし，テクスチャ Ω_c を色情報として，スプラットを利用した描画を行う．

スプラットの描画手法として，[1] の方法にならい，ガウシアンカーネルテクスチャ (*Gaussian Kernel Texture*) を付加したポイントスプライト (*Point Sprite*) により，透明度を考慮したブレンディングを行う方法を用いる．ガウシアンカーネルテクスチャとは，2 次元のテクスチャサンプルにガウシアン関数の値を記録したものである．スプラットの描画処理は頂点シェーダとピクセルシェーダの中で行われ，最終的なフレームバッファの色は，ガウシアン関数によって重み付けされた重み付き平均和となる．

このとき，点の座標や色情報が格納されたテクスチャ Ω_a, Ω_c は，フレームバッファよりも解像度が低い．このため，このテクスチャに記録されている情報を，フレームバッファの解像度にあわせる必要がある．

本手法におけるスプラットによる描画処理の概要を図 6 に示す．まず，スプラットを描画するための点の座標と色を，テクスチャ Ω_a, Ω_c を参照することで得る (図 6左)．この参照は頂点シェーダの中で行われるが，これは，前述の頂点テクスチャの機能を使うことができる．Ω_a より取得した点は，座標変換によりフレームバッファ上に投影される (図 6右)．各フラグメントにおいてガウシアンカーネルテクスチャを参照するためのテクスチャ座標 (u_i, v_i) は，スプラット描画の際，Ω_a より取得した点を中心とする，テクスチャ座標を付加した小四角形を，頂点シェーダに渡すことで得られる．この小四角形がポイントスプライトの

プリミティブとなる．

各ピクセルの色 $c = (r, g, b, a)$ は，隣接点の色のガウシアン関数による重み付き平均和となる．色の計算は，[1] において述べられている手法を用いており，GPU 上で 2 パスで処理することができる．まず 1 パス目では，色の重み付け和の計算を行う．テクスチャ Ω_c に格納されている，ある点とその隣接点を含む 9 つの点のうち，各点における色座標を (r_i, g_i, b_i)，ガウシアンカーネルテクスチャを参照するためのテクスチャ座標を (u_i, v_i) とすると，c は以下の式で計算される:

$$c = (r, g, b, a) = \sum_i g(u_i, v_i)(r_i, g_i, b_i, 1). \qquad (6)$$

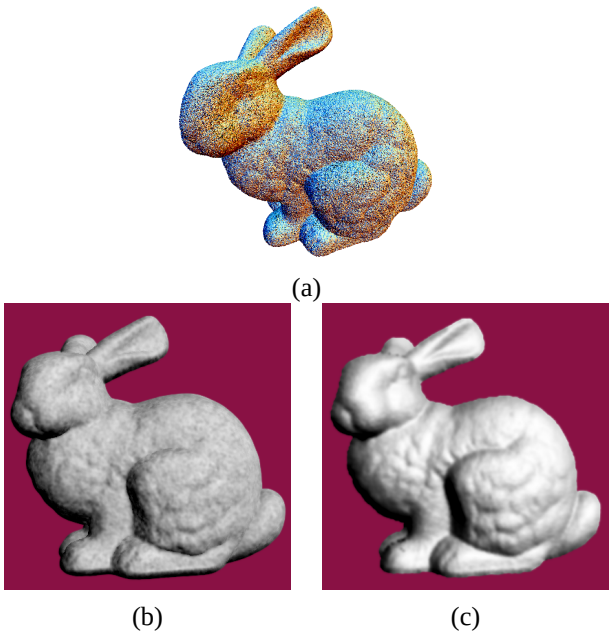
ここで，$g(u_i, v_i)$ はテクスチャ座標 (u_i, v_i) におけるガウシアン関数の関数値を示す．この処理は，GPU によるテクスチャブレンディングの際，フレームバッファの設定を「加算する」とすることによって実現することができる[3]．2 パス目では，各ピクセルに対しピクセルシェーダによって正規化を行う．このステップでは，各ピクセルの色は，c の 4 番目の要素 a で各 r, g, b の値を除算することで計算できる．

実際に加算合成を行う際，出力先のテクスチャフォーマットとして，16 ビットの浮動小数点値を利用した．これは，近年の GPU をもってしても，32 ビットの浮動小数点値のフォーマットでは加算合成を行うことができないためである．

4 結果と議論

まず，本手法の有効性を検証するための実験として，Stanford Bunny のメッシュの各頂点に対し，ランダムな微小ノイズを付加した点群データ (556,051 点) (図 7(a)) を生成した．図 7(b) に本手法による描画結果を，図 7(c) に [1] による描画結果をそれぞれ示す．なお図 7(c) では，点群データの他に法線ベクトルデータを事前に計算して利用している．図 7(c) では，すべての頂点の法線を考慮してスプラットによるブレンディング操作を行っているため，正しい描画結果が得られない反面，図 7(b) の本手法では，このようなノイズを持つ点群データに対しても，正しい描画結果が得られていることが確認できる．また，計算時間に関しては，1 〜 3 パス目で 37.5ms, 4 パス目で 32.8ms, 5 パス目で 26.5ms ほどかかった (10 回のレンダリング時間の平均時間．Athlon XP 2600+，GeForce 6800 Ultra により計測)．本手法の特徴である，選択的格納と法線ベクトルの計算で，全体の計算時間の 2/3 程度を占める．それでも，本手法は，描画する点数が少ないため，おおよそ 10 fps 程度で描画を行うことができ，スプラットによる手法 (4 fps) の 2 倍以上の速度で計算できることが確認できた．

図 8 には，実際に測定された点群データに対する，本手法と [1] の比較結果を示している．なお，[1] の手法で用いる法線ベクトルの計算は，[9] の手法により行っている．図 8 上段は，Stanford Asian Dragon の測定データ (3,609,600 点) に関する結果を示している．このデータは，測定データではあるものの，測定誤差のほとんどない「綺麗な」データである．このようなデータの場合，法線ベクトルが正しく計算できるため，[1] の手法の方がレンダリングの質は高い．本手法では，少々ぼけが現れる結果となる．これは，点の選択的格納によるデータ損出が原因であることが考えられる．一方で，図 8 下段の Beetle の測定データ (559,327 点) では，測定誤差が非常に大きいため，法線ベクトルを正しく計算することが難しい．そのため，[1] の手法では所々に斑点模様が見られる．これに対し，本手法による結果の方が，誤差削減の効果が現れるためレンダリングの質が高いと言える．



(a)

(b) (c)

図 7: (a) ノイズを含む点群データ．(b) [1] による描画結果．(c) 本手法による描画結果．

5 結論と展望

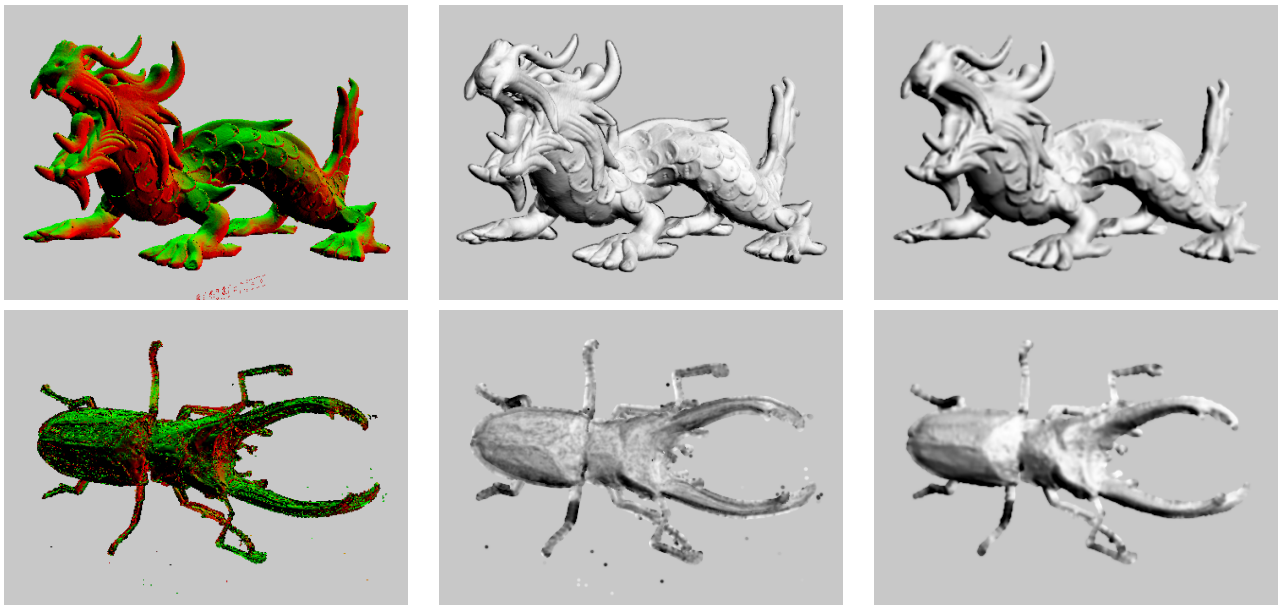
本手法では，位置情報のみを持つ点群データを，GPU を利用して直接レンダリングする方法について提案した．法線ベクトルデータを GPU により動的に計算するものの，従来手法と大差ないレンダリングの質を確保することができること，また，測定点群データに含まれるノイズの軽減処理を行うことにより，特に 3D 測定デバイスにより取得した点群データに対し威力を発揮すること，さらに，点群の選択的格納により，大容量データに対しても高速にレンダリングできることを実験により実証した．

今後の展望としては，インタラクティブなレンダリングの際に，フレーム間に見られるちらつきの対処や，不均一なデータに関する対応が挙げられる．

謝辞

Bunny と Asian Dragon は，スタンフォード大のコンピュータグラフィックス研究室により計測されたデータである．Beetle は，慶應義塾大学小檜山研究室で計測した

[3] DirectX では，レンダリングの状態設定で D3DRS_SRCBLEND と D3DRS_DESTBLEND を D3DBLEND_ONE にしておけば良い．



(a) (b) (c)

図 8: 測定データに対する描画結果．上段: Stanford Asian Dragon．下段: Beetle．(a) 測定点群データ． (b) [1] による描画結果． (c) 本手法による描画結果．

データである．また， [9] による法線ベクトルの生成において，理化学研究所の大竹豊氏のコードを利用させて頂いた．ここに感謝の意を表する．

参考文献

[1] M. Botsch and L. P. Kobbelt. High-quality point-based rendering on modern GPUs. In *Proc. Pacific Graphics 2003*, pp. 335–343. IEEE CS Press, Los Alamitos CA, 2003.

[2] M. Botsch, M. Spernat, and L. P. Kobbelt. Phong splatting. In *Proc. Symposium on Point-Based Graphics 2004*, pp. 25–32. Eurographics Association, 2004.

[3] M. Botsch, A. Wiratanaya, and L. P. Kobbelt. Efficient high quality rendering of point sampled geometry. In *Proc. 13th Eurographics Workshop on Rendering*, pp. 53–64. Eurographics Association, 2002.

[4] C. Dachsbacher, C. Vogelgsang, and M. Stamminger. Sequential point trees. *ACM Transactions on Graphics (Proc. SIGGRAPH 2003)*, 22(3):657–662, 2003.

[5] S. J. Gortler, R. Grzeszczuk, R. Szeliski, and M. F. Cohen. The lumigraph. In *Computer Graphics (Proc. SIGGRAPH 96)*, pp. 43–54. ACM Press, New York, 1996.

[6] J. P. Grossman and W. J. Dally. Point sample rendering. In *Proc. 9th Eurographics Workshop on Rendering*, pp. 181–192. Eurographics Association, 1998.

[7] G. Guennebaud, L. Barthe, and M. Paulin. Deferred splatting. *Computer Graphics Forum (Proc. Eurographics 2004)*, 23(3):653–660, 2004.

[8] G. Guennebaud and M. Paulin. Efficient screen space approach for hardware accelerated surfel rendering. In *Proc. 8th International Fall Workshop on Vision, Modeling and Visualization (VMV 2003)*, pp. 1–10. IOS Press, Amsterdam, 2003.

[9] H. Hoppe, T. DeRose, T. Duchampy, J. McDonald, and W. Stuetzle. Surface reconstruction from unorganized points. *Computer Graphics (Proc. SIGGRAPH 92)*, 26(2):71–78, 1992.

[10] T. Jones, F. Durand, and M. Zwicker. Normal improvement for point rendering. *IEEE Computer Graphics & Applications*, 24(4):53–56, July/August 2004.

[11] A. Kalaiah and A. Varshney. Modeling and rendering points with local geometry. *IEEE Transactions on Visualization and Computer Graphics*, 9(1):30–42, 2003.

[12] H. Kawata and T. Kanai. Image-based point rendering for multiple-range images. In *Proc. 2nd International Conference on Information Technology & Applications (ICITA 2004)*, pp. 478–483. Macquarie Scientific Publishing, Sydney, 2004.

[13] L. P. Kobbelt and M. Botsch. A survey of point-based techniques in computer graphics. *J. Computers & Graphics*, 28(6):801–814, 2004.

[14] J. Krivanek. Representing and rendering surfaces with points. Technical Report DC-PSR-2003-03, Department of Computer Science and Engineering, Czech Technical University in Prague, 2003.

[15] M. Levoy and T. Whitted. The use of points as a display primitive. Technical Report 85-022, Computer Science Department, University of North Carolina at Chapel Hill, Jan. 1985.

[16] H. Pfister, M. Zwicker, J. van Baar, and M. Gross. Surfels: surface elements as rendering primitives. In *Computer Graphics (Proc. SIGGRAPH 2000)*, pp. 335–342. ACM Press, New York, 2000.

[17] L. Ren, H. Pfister, and M. Zwicker. Object space EWA surface splatting: A hardware accelerated approach to high quality point rendering. *Computer Graphics Forum (Proc. Eurographics 2002)*, 21(3):461–470, 2002.

[18] J. Shade, S. Gortler, L. wei He, and R. Szeliski. Layered depth images. In *Computer Graphics (Proc. SIGGRAPH 98)*, pp. 231–242. ACM Press, New York, 1998.

[19] M. Zwicker, H. Pfister, J. van Beer, and M. Gross. Surface splatting. In *Computer Graphics (Proc. SIGGRAPH 2001)*, pp. 371–378. ACM Press, New York, 2001.

[20] M. Zwicker, J. Rasanen, M. Botsch, C. Dachsbacher, and M. Pauly. Perspective accurate splatting. In *Proc. Graphics Interface 2004*, pp. 247–254. Morgan Kaufmann Publishers, San Francisco, CA, 2004.

[21] 藤本, 今野, 千葉. ポイントグラフィックス概説. 芸術科学会論文誌, 3(1):8–21, 2004.