一般
投稿論文

GPUプログラミングの深度カメラ画像処理への適用

深井 裕二[†1]　竹沢 恵[†1]　島 貢[†1]　川上 敬[†1]

[†1] 北海道工業大学

近年，五感や手の動きなど，人の自然な行為を用いたコンピューティングである NUI(Natural User Interface) の入力装置に深度カメラが利用されている．ジェスチャによる機器操作や 3 次元情報をもとにした活用を実践するためには，レスポンス性の良い画像処理が不可欠である．一方，高性能化した GPU(Graphics Processing Unit) が普及しており，さらに，GPU の演算能力を，汎用目的に応用する技術である GPGPU(General Purpose computing on GPUs) が活用されている．筆者らは，深度カメラ Kinect（Microsoft 社）を用い，NUI の発展に向けた人物の 3D モデリングや手の認識などを，GPU によってリアルタイムに処理するシステムを試作した．開発フレームワークに DirectX11 を用い，深度情報を活用して，GPU による 3D グラフィクス処理と GPGPU による並列演算を連携させた処理手法について，試作で得られた知見，ノウハウを報告する．

1. はじめに

ナチュラルユーザインタフェースNUI(Natural User Interface)はタッチ，ジェスチャなど，人が自然な行動で直感的に操作できるユーザインタフェースである．ゲームマシンXbox360（Microsoft社）などは，深度カメラ(Depth Camera)を用いたNUIの応用例である．

深度カメラは，深度センサや3Dセンサとも呼ばれ，赤外線反射等により，被写体までの距離を得る装置である．Xbox360用の深度カメラKinect（Microsoft社）は，消費者向けで安価であり，PC利用のための開発ライブラリも提供されている．深度カメラの用途では，人の位置検出によるセキュリティ設備や広告装置，ハンドジェスチャ認識による情報家電機器の操作など，さまざまな活用が期待されている．NUIの目的には，利用者の思考を妨げないストレスのない操作性がある．そのために深度カメラの画像処理や認識処理の高レスポンス性が課題となる．

実際にNUIシステムを構築し，機能を発展させていく場合，処理の積み重ねによるレスポンスの損失が懸念される．性能低下はユーザの満足度を低下させ，NUIのニーズを損なう可能性がある．そこで，本研究は高レスポンス性実現に向け，その基礎構築に対し，ハードウェア技術の活用における検討と試作, 実験に重点を置いた．

まず，深度カメラ画像をもとに，人物等の位置や動きを全体的に把握してNUIを構築する場合，画像処理および認識処理の精度と，人の動きに対する瞬時のレスポンスの両立が重要となる．しかし，プログラミング実践経験から，イメージデータに対する画素ごとの反復演算負荷により，それらの精度とレスポンスの十分な両立は実現が困難と予想された．このような反復演算の高速化には，GPGPU(General Purpose computing on GPUs)の利用が考えられるが，処理対象が空間上の深度情報であることから，さらにGPU(Graphics Processing Unit)のハードウェア3Dグラフィクス処理を活用し，GPGPU処理との有機的な連携を試みるという研究の方向に向かった．

筆者らが試作したシステムは，Kinectを用い，NUIの発展に向けた人物の3Dモデリングや，ジェスチャ入力の基盤となる手の認識処理を行う．この開発ではGPUによるリアルタイムな処理を目的とし，特に深度情報を活用して，3D処理と並列演算の連携による処理手法を検討した．GPUの開発フレームワークには，それらの連携に効果的と考えられるDirectX11を用いた．第2, 3章では，開発指針を得るために，DirectX11で本格活用が可能になった並列演算の手法と性能について述べる．第4，5，6章では，人物と背景の分離，3次元化とポリゴンの詳細化，手の認識と追跡について，それぞれGPUによる処理手法を述べる．第7章では，本システムのレスポンス, 高速化, DirectX11の評価などをまとめる．

2. DirectX11 による GPU 処理

2.1 GPGPU 処理と 3D 処理の連携方式

一般のカメラ画像は輝度値で構成されるが，深度カメラの場合は深度値で構成される．深度値をもとにした3次元情報の生成，加工，表示には，3D処理のライブラ

リおよびハードウェアの活用が効果的である．PCのビデオカードは，高速なGPUおよびVRAM(Video RAM)で構成され，汎用的な並列演算を目的とするGPGPUとしても高い効果がある．GPGPUの処理系としてはNVIDIA社のCUDA，OpenCL，Microsoft社のDirectX11に含まれるコンピュートシェーダ(Compute Shader) [4],[5]などがある．今回の試作システムでは，コンピュートシェーダを採用した．採用理由は，**図1**に示すように，DirectX11が3D処理とGPGPUの両機能を包含しており，VRAMを介し，GPU上で両機能の連携が可能である点や，両機能ともHLSL(High Level Shader Language)で記述できる点，さらに，DirectX11の3D処理が，ポリゴン密度をハードウェアで動的に調整するテッセレーション(Tessellation) [10]に対応する点などである．CUDAやOpenCLは，3D機能を含まず，GPU上での連携に不向きであるが，DirectX11は，単一のフレームワークで開発可能である．

図1の各シェーダ(Shader)は，GPUにおける各演算工程に対するプログラムである．GPGPU処理用のコンピュートシェーダは，3D処理との親和性が高い[1],[5]．カメラ画像をVRAMへ転送した後，GPU側での3D処理およびGPGPU処理（並列演算）がシームレスに行える．

2.2　コンピュートシェーダの利点

本システムの並列演算に用いた，コンピュートシェーダについて，以下にその利点をあげる．

- 3D処理に用いるテクスチャに対し，HLSLの[ ]演算子によって，配列として読み書きが可能．
- 複数のスレッド（最大1024個）を1グループとし，x, y, z方向で任意に分割可能なグループ構成．
- グループ内のスレッド間でアクセス可能な共有メモリ（Thread-group-shared memory，最大32KB）が，レジスタと同様に，高速にアクセスできる．
- スレッドの進行を待ち合わせるバリア同期命令．
- 引数により，テクスチャを1あるいは2次元で配列参照でき，添え字換算の手間がかからない[5],[6].

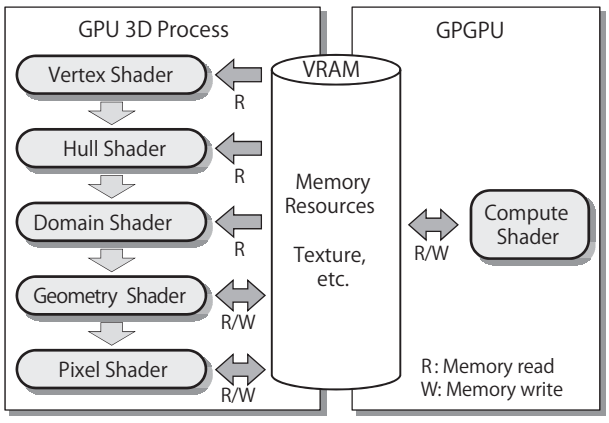


図1　DirectX11のシェーダ構成

2.3　深度画像処理システムの開発環境

本システムの開発環境は，Windows 7マシン（Core i7 2.93GHz，ターボブースト有効：最大3.06GHz，PC3-8500 6GB，PCI Express2.0 x16）上で，Microsoft Visual C++ 2008，DirectX11 SDKを用いた．深度カメラにはKinectを用い，GPUにはAMD Radeon HD6950（コアクロック800MHz，メモリクロック1250MHz，演算ユニット1408基）を選択した．GPU利用率の確認にはAMD System Monitorを用い，HLSLのコンパイルには，DirectX11 SDKのfxc.exeを用いた．

3.　画像の並列処理

3.1　リダクション演算

リダクション演算（縮約演算）[2]は，逐次繰返し処理を並列化する際に用いられる演算方法である．**図2**にHLSLによって，深度最小値を求めるための，リダクション演算を用いたコンピュートシェーダ手続きを示す．groupsharedにより，変数Wを共有メモリとして宣言している．numthreads(16,16,1)により，スレッド数に16×16=256を指定し，2次元の添え字tid.xyは[0,0]～[15,15]，1次元の添え字giは[0]～[255]をとる．この手続きは，16×16ピクセルの画像領域に対して，1ピクセルごとに並列に呼び出される．各スレッド内の逐次ループ部分では，バリア同期命令GroupMemoryBarrierWithGroupSyncで，スレッド間の同期をとりつつ，配列を半分に集約しながら最終的に1つの結果を得る（**図3**）．

最小値算出処理の逐次繰返し回数に着目すると，通常

```
RWTexture2D<uint>  TexIn, Result;
groupshared uint  W[256];      // 共有メモリ

[numthreads(16,16,1)]          // 16x16 スレッド並列処理
void MIN1(uint3 gid:SV_GroupID, uint gi:SV_GroupIndex,
          uint3 tid:SV_DispatchThreadID)
{
      W[gi] =TexIn[tid.xy];
      for (uint n = 128; n > 0; n >>= 1) {
            GroupMemoryBarrierWithGroupSync();
            if (gi < n) W[gi] = min(W[gi], W[gi+n]);
      }
      if (gi == 0) Result[gid.xy] = W[0];
}
```

図2　リダクションによる最小深度の抽出手続き

N回に対し，並列演算方式は$\log_2 N/N$倍の効果となる．本システムでは，合計平均，最小値，カウント，照合などを要する処理に，リダクション演算を適用している．

3.2 コンボリューション演算

コンボリューション演算（畳み込み演算）は，周りの画素をもとにした中心画素の処理を，画像全体に繰り返すときに用いられ，この繰返し部分は並列化の効果が得られる．画像処理における平滑化には，ガウスフィルタがあげられる．なお深度画像において，物体間の深度差の大きい部分では，メディアンフィルタ（エッジ保存型のフィルタ）が適している．また，ノイズや測定不良により発生する穴状部分は，オープニングやクロージング[7] で除去できる．本システムでは，以上のようなコンボリューション演算に並列演算を適用している．

図4は簡易ガウスフィルタの手続きである．16×16ピクセルの領域に対し，並列に呼び出される．ガウスフィルタには画像全域を水平方向に処理（**図5**）した後，垂直方向に処理するといった別手法があり，テクスチャアクセス頻度を減らし，高速な共有メモリが活用できる[6].

3.3 GPU 処理の性能評価と設計指針

表1にCPUとGPU（並列演算）による，各種処理時間の測定結果を示す．各処理では640×480ピクセルの画像を用いた．リソースコピー（CPU-GPU）は，主メモリからVRAMへの画像コピーである．CPU側に関してC++の最適化コンパイルオプションは，実行速度重視（/O2）を適用した．CPU側の測定には，高分解能カウンタAPI（QueryPerformanceCounter）を用いた．GPU側の測定には，GPUタイムスタンプを取得するAPI（ID3D-11Query）を用いた．なお，CPU側はシングルスレッドである．また，各画像処理にCPU-GPU間のデータ転送時間は含まない．

結果において，深度最小値算出では，リダクション演算量は256スレッド×逐次1,200回＋256スレッド×逐次5回＋8スレッドとなり，逐次要素を多く含む．一方CPU側は，処理が単純で計算量が少ないため，性能比があまり高くない．ガウスフィルタ（水平垂直）では，共有メモリの効果が性能比にみられる．

メディアンフィルタは，整列処理を含み，GPU処理時間がやや長い．整列の並列化手法も考案されているが[2]，順序性のある処理は並列化が困難である．なお，性能比が高い理由の1つには，CPU側ではガウスフィルタに対し4.3倍だが，C++の最適化コンパイルを無効にすると1.9倍になることから，メディアンフィルタへの最適化効果が低いことがあげられる．

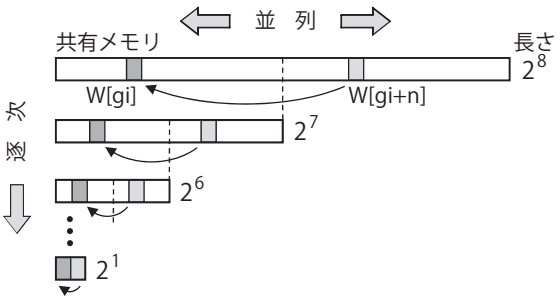


図3 リダクション演算の過程

```
float Weight[9] = {  0.0625,  0.1250,  0.0625,
                     0.1250,  0.2500,  0.1250,
                     0.0625,  0.1250,  0.0625  };

[numthreads(16,16,1)]          // 16x16 スレッド並列処理
void GAUSSIAN(uint3 tid : SV_DispatchThreadID)
{
      float v = 0;
      for (uint i = 0; i < 9; i++)
            v += TexIn[tid.xy + Offset [i]] * Weight[i];
      TexOut [tid.xy] = (uint)v;
}
```

図4 コンボリューションによるガウスフィルタ

```
float Weight[3] = { 0.25,  0.50,  0.25 };
groupshared float W[640];     // 共有メモリ

[numthreads(640,1,1)]          // 640 スレッド並列処理
void GAUSSIAN _H( uint3 gid : SV_GroupID,
                  uint gi : SV_GroupIndex)
{
      uint2 pos = uint2(gi, gid.y);
      W [gi] = TexIn[pos];
      GroupMemoryBarrierWithGroupSync();
      float v = 0;
      for (int i = -1; i <= 1; i++)
            v += W[gi + i] * Weight[i + 1];
      TexTmp[pos] = v;
}
```

図5 共有メモリを用いたガウスフィルタ（水平処理部）

表1 CPU と GPU の性能比較

	CPU [μs]	GPU [μs]	性能比
深度最小値算出	504	144	3.5
ガウスフィルタ	8,095	1,232	6.6
ガウスフィルタ（水平垂直）	3,330	388	8.6
メディアンフィルタ	34,822	1,526	22.8
リソースコピー	105	29	3.6
リソースコピー（CPU-GPU）	—	543	—

最小値やガウスフィルタのような計算量も少なく，また，最適化効果が高い処理でも，並列演算により，数倍の性能向上がみられた．そして，さらに複雑な画像処理でも，並列演算による効果が期待できる．

リソースコピー(CPU-GPU)のスループットは2.2GB/sであり，CPU側と比べて5.2倍遅い．これは，メモリバスよりも狭帯域な拡張バス(8GB/s)経由であることに加え，メモリロックにより相手プロセッサを同期（待機）させるため，深刻なボトルネックとなる可能性がある．

以上をもとに，本システムでの並列演算スタイルの選択にかかわる設計指針を以下に示す．

(a) 逐次繰返し処理に対し，リダクションによる並列演算を，また，画素ごとや3D頂点ごとの処理に対し，コンボリューションによる並列演算を適用する．
(b) (a)の並列演算では，共有メモリを活用する．
(c) CPU-GPU間の画像コピーを，動画1フレームあたり1回に抑え，深刻なボトルネックを回避する．
(d) (c)の実現のためには，並列化が困難で複雑な逐次処理でも，GPU側への処理移植を試みる．

4. 深度画像利用に向けた課題と解決策

4.1 振動する深度値データの平滑化

深度カメラから得られた深度値は，量子化ノイズおよび測定精度に起因すると考えられる激しい微細振動が見られ，一般のカメラと異なる性質を示した．これに対し，ヒステリシス平滑化[8]を用いて，**図6**のように深度値の振動を平滑化し，さらに画素ごとの並列演算で高速化した．

また，主に物体の輪郭左側に測定不可部分が存在した．これは，深度カメラの赤外線発光および受光の位置関係から，死角が生じるためと考えられる．人の頭髪部，鏡面等でも，測定不良と思われる同様の現象が確認された．

これらの部分で見られるNull値（測定不能）との振動に対し，時間軸での平滑化は適さず，空間方向の平滑化であるオープニングやクロージングを適用した．これらにより，輪郭の欠けなどを埋める効果が多少得られた．

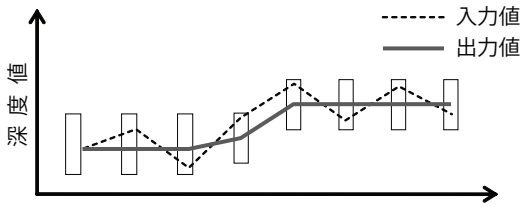


図6　ヒステリシス平滑化

4.2 背景と移動物体の分離

深度画像から人物や背景を3Dモデリングする場合，カメラ視点での人物の影による不可視領域がある（**図7**）．そのため，人物と背景間の距離に比例して，影による背景の欠損が増大する（**図8 (a)**）．これは，背景を必要とする発展を考えた場合に支障となるため，この背景欠損をなくすことが課題である．解決策として，並列演算により，深度値を格納するテクスチャを背景用と人物用に振り分けることで，高速に影を除去する手法を提案する．まず，システムの実行開始時に背景テクスチャを構築する．その後，視野に入る物体に対し，画素ごとの並列演算を用いて，条件（物体深度値＋マージン＜背景テクスチャ値）を満たす場合，画素を人物テクスチャに格納する．背景テクスチャは変更されず，影は発生しない（**図8(b)**）．一般の建造物の壁などは動かないという前提において，この手法は有効である．なお，マージンは振動等によって背景を人物に誤認しないための余裕であり，500[mm]とした．また，この処理で得られた人物テクスチャを，VRAMに保存しておくと，認識処理効率が向上する．

5. 深度画像の 3 次元モデリング

5.1 深度画像からの 3 次元変換

2D深度画像の各画素に対し，並列演算で3D座標を求め，ポリゴンメッシュを構築する．座標算出には，深度

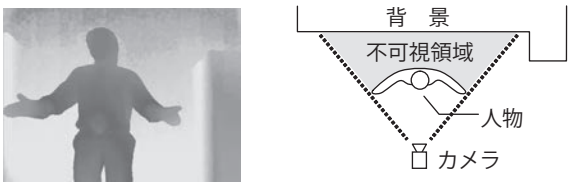


図7　深度画像とカメラ視野による影

(a) 影ができる背景　　**(b) 記憶済み背景による影の除去**

図8　人物と背景のワイヤーフレームモデル

値からz軸実距離への変換が必要であり，事前の測定作業により得た近似式（式(1)）を用いた．

測定作業では，マーカを複数設置し，その実距離の逆数と深度値をサンプルとし，回帰分析（最小二乗法）によって近似式を得た．式(1)において，d_{Raw}は深度カメラから出力される未加工の深度値（非線形），d_{Lin}は線形化されたz軸実距離[mm]である．

$$d_{Lin} = \frac{1000.0}{-0.0028796 \times d_{Raw} + 3.1385797} \quad \cdots \text{式(1)}$$

これらをもとに構築した，3Dモデリング画像を**図9**に示す．ポリゴンは，カメラから見えている物体部分に対して構築され，実空間上の位置やサイズを反映している．得られた3Dモデルをもとに，GPUの高速な3D処理によって，さまざまに投影した2D画像が生成できる．

5.2 法線の動的計算

得られたポリゴンを用い，3Dレンダリングを行う．通常，グーローシェーディングによる滑らかな3D画像を描画するためには，各頂点に設定した頂点法線ベクトルを用いて，ライティング計算および頂点間の線形補間が行われる．しかし，深度画像から構築したポリゴンは常時形状変化するため，動画1フレームごとに，3D処理に先立って，法線を随時動的に得ることが課題である．

解決策として，並列演算により，高速に法線を計算し，ポリゴンに対応させる手法を提案する．**図10**において，ポリゴンを構成する個々の三角形に対し，(1)2辺の外積から面法線ベクトルを求め，(2)1つの頂点について隣接するすべての三角形の面法線ベクトル和を計算し，(3)それを正規化して頂点法線ベクトルを得る．そして，レンダリング時に参照できるように，法線マップとしてテクスチャに保存する．これらを頂点ごとの並列演算で行う．

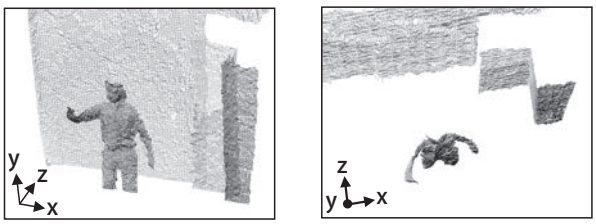


図9 ワイヤーフレームによる3Dモデリング画面

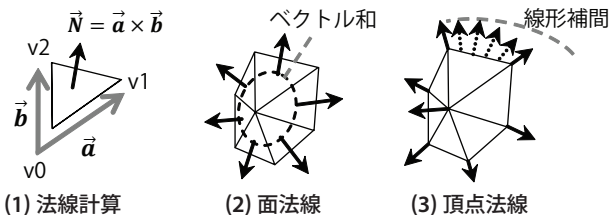


図10 ポリゴンの動的法線計算

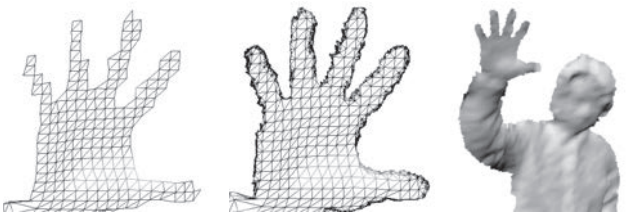
5.3 ポリゴンの詳細化

640×480ピクセルの深度画像に対し，全画素ごとに3D頂点を対応させることは，大量のポリゴンによって高負荷となる．これに対して，人物のエッジ部分のみポリゴンを細かく分割すること（テッセレーション[10]）で，輪郭の詳細性を上げる手法があるが，任意のエッジ形状に対し，最適な分割制御は難しい．随時変化するエッジ形状に対して，輪郭の詳細性を上げつつ，不必要な分割を抑え，より低負荷とすることが課題である．

解決策として，DirectX11のストリーム出力機能[5]を用いた段階式テッセレーション手法を提案する．概要は，まず，8ピクセル間隔で頂点を対応させた基礎ポリゴンを構築する（**図11(a)**）．この時点でポリゴン数は1/64に減少する．次に，段階式テッセレーションによって，エッジに近づくにつれ小さく分割していき，不必要な分割を抑えた輪郭の詳細な結果を得る（**図11(b)**）．

本手法は，テッセレーション制御部と，ストリーム出力制御部に分けられる．DirectX11は，GPUのテッセレーション機能に対応しており，ポリゴンの分割制御では，三角形あるいは四角形単位のポリゴン要素（パッチ）ごとに，各辺に対する分割数(factor) Nを指定する（**図12**）[9]．

本手法のテッセレーション制御部の制御ルールを，以



(a) 基礎ポリゴン　(b) テッセレーション　(c)3D レンダリング

図11 テッセレーションによる3D構築

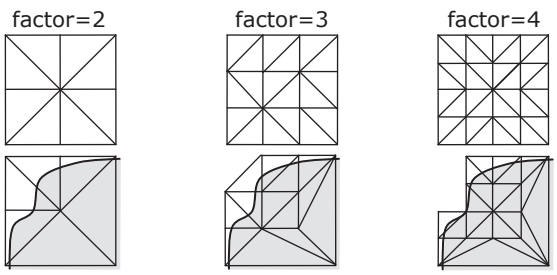


図12 テッセレーションのポリゴン分割

下に示す．これにより，Nが増すにつれ，図12下のように輪郭に近づく．

規則1：パッチのz座標がすべてNull値の場合（背景部分を意味する），そのポリゴンを削除する．

規則2：パッチのz座標のいずれかがNull値の場合（エッジ部分を意味する），属する辺をN等分割する．

規則3：パッチのz座標の中で落差が一定閾値を超える辺をN等分割する．

規則4：分割後の小パッチについて規則1があてはまれば同様に削除する．

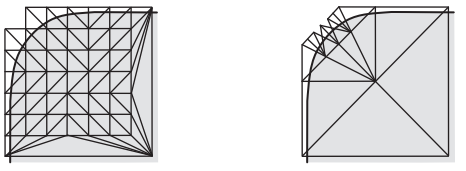
この処理単体では，不必要な部分でポリゴン数が増加する欠点がある．規則2において，たとえば四角形パッチの1頂点が，わずかに背景部分に突出していたとしても，突出の度合いにかかわらず辺がN等分割される．よって，エッジに隣接するパッチ内では，エッジから離れた部分でも無意味な細分化が起こる（**図13 (a)**）．

5.4　段階的なテッセレーション制御

DirectX11のストリーム出力機能は，図1のGeometry Shaderの頂点出力をVertex Shaderに入力しなおすことで，繰り返しテッセレーション処理を通過させることができる．本手法のストリーム出力制御部では，テッセレーション制御部を複数回実行して，段階的な分割を制御する．

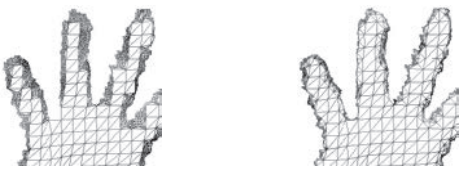
本システムは，ストリーム出力による処理時間のオーバヘッドも考慮し，エッジ部分を2分割するストリーム出力を2回繰り返し，最後に細かく分割するという，合計で3回のテッセレーションを行った．図13(a)では，エッジ近辺の四角形パッチが1/Nの均等分割に対し，(b)は1/2－1/4－1/Mと，エッジ外側へ向かうにつれ詳細度を増す．ここで，N，Mは末端の分割数である．**図14**に示す，手を拡大した比較では，(b)がエッジ近辺の余計な細分化を抑え，全体的に少ないポリゴン数で，(a)と同程度の輪郭を実現しているのが分かる．このような，ポリゴン数の最適化は負荷の低減以外にも，認識処理への応用を考えた場合，詳細な輪郭形状が得られるので望ましい．

図15にエッジを詳細化した人物のレンダリングを示す．この実行例のポリゴン数は，テッセレーションのみのとき42,372に対し，ストリーム出力併用時は22,912となり，本手法の効果により0.54倍に減少した（N=M=16）．



(a) テッセレーションのみ　(b) ストリーム出力併用

図13　ポリゴン分割方法の改善



(a) テッセレーションのみ　(b) ストリーム出力併用

図14　エッジ詳細化の結果

6. 手のパターン認識

6.1　深度値を活用したマッチング

3D処理で構築した人物のポリゴンを活用して，並列演算によるテンプレートマッチング[7]で，手の認識処理を行う．通常のカメラでは，人物の距離によって画像上の手のサイズが異なる．その場合，複数サイズのテンプレートを用意する対処法があるが，処理の負荷，あるいは距離による認識の不安定さが問題となる．

解決策として，3D処理側でサイズを正規化した探索画像を生成し，それを並列演算側の認識処理へ受け渡す手法を提案する．**図16**において，3Dモデル化で得られたポリゴンは空間上の実際の人物のサイズを反映している．これに，平行投影(Parallel projection)によるレンダリングを行うことで，カメラ距離にかかわらず，物体が投影面に写されるサイズが一定となる．また，テッセレーションで詳細化しているので，写される形状の正確性

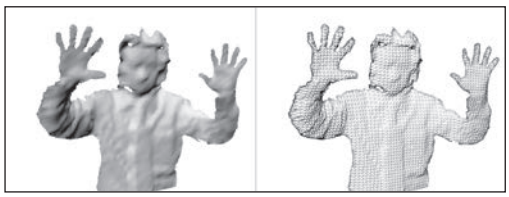


図15　3D レンダリング画面

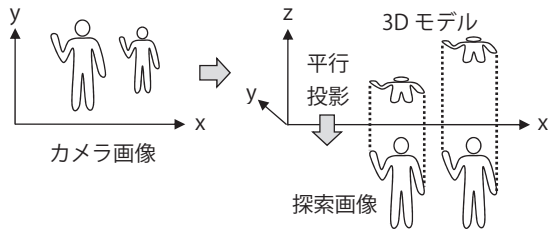


図16　平行投影による探索画像の生成

も失われない．この投影結果を探索画像にすれば，人物の距離にかかわらず，1種類のサイズのテンプレートで済む．

さらに，カメラに向かってかざした手を効果的に認識するために，遠い部分をあらかじめ除去した．カメラ前に手をかざした場合，それよりも前に物体がなければ，画像全体の深度最小値は，おおよそ手のz座標を示すことになる．人物テクスチャの深度値dにおける最小値d_{min}をリダクション演算で求め，式(2)により，新たな深度値として，接近度pを並列演算によって求めた．

$$t = d_{min} + d_{th} - d(x,y),$$
$$p(x,y) = \begin{cases} t, & t \geq 0 \\ Null, & t < 0 \end{cases} \quad \cdots \text{式(2)}$$

この条件処理により，d_{min}（手の位置）を基準に，閾値d_{th}を超える遠方部分（体や顔など）は除去される（**図17**下）．また，図17右のように手と顔が重なるとき，通常のカラーカメラ画像の場合，認識効率を上げるために，肌色のマスク処理を施すと顔と同化してしまう．こういったケースでも，手より後ろにあるものは除去されており，影響を及ぼさない．認識対象より前に物体がないという条件のもとで，深度値による除去を用い，注目すべき領域に限定することで，認識効率が上げられる．本システムでは，主に手首から先のみを認識対象にするために，d_{th}を200[mm]に設定した．

6.2 動きの追跡

かざした手を動かしたとき，手の角度変化や，ぶれから認識が無効になっても，位置を追跡する処理を検討する．物体追跡の処理方法では，特徴点の抽出とオプティカルフロー推定[3]があげられる．しかし，深度画像では，通常のカメラ画像のように被写体の輝度（色）による特徴点の抽出には向かない．また，LK法(Lucas-Kanade)[3]を用いても，速い動きには向かないなどの問題がある．

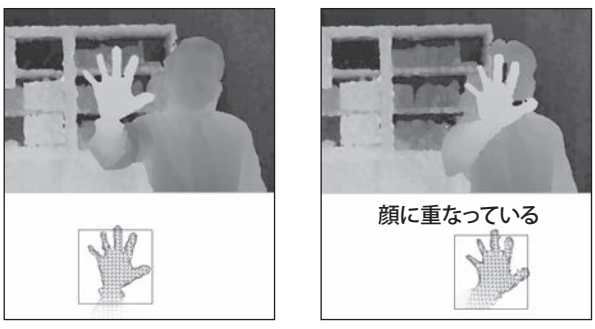


図 17　遠方を除去した認識処理

解決策として，重心移動をもとに，速い動きに対応した追跡手法を提案する．この手法では，手を認識した矩形領域（認識エリア）の動画フレーム間の重心変動から移動ベクトルを得る．これにより複数の手を個別かつ同時に追跡することができる．重心g_x, g_yは，認識エリア内の接近度p（手の部分以外は値が0）をもとに，式(3)により求めた．この式の合計計算部分は，リダクションによる並列演算で高速化している．

$$M = \sum_x \sum_y p(x,y),$$
$$\begin{pmatrix} g_x \\ g_y \end{pmatrix} = \frac{1}{M} \sum_x \sum_y p(x,y) \begin{pmatrix} x \\ y \end{pmatrix} \quad \cdots \text{式(3)}$$

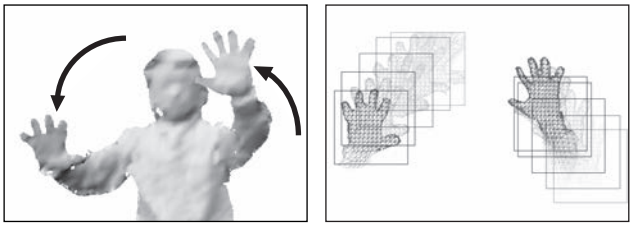
一度認識すると，重心の移動ベクトルが連続的に得られ，もし認識が無効になっても位置の推定を一定時間続ける．ただし，認識エリアから急激に離れると追跡が困難になる．認識エリアの128×128ピクセルに対し，重心を求める対象エリアを256×256と広げることで，急激に手を動かしても追跡が容易になった．これによって，速い動きや断続的な認識に対しても，連続した追跡が単純な手法で実現できた．このような素早い動きに対するスムーズな追跡は，NUIに重要である．**図18**は，両手を動かした際の追跡状況を軌跡表示したものである．

7. まとめと今後の課題

本章では，今回のGPUプログラム試作におけるプラクティスから得られた知見をまとめる．

7.1 設計指針の適用と評価

前章までに述べたように，我々は高い効率を達成するため以下の3点を同時に考慮する必要があった：並列演算の高速性，GPU側での3D処理と並列演算の連携から得た効率性，ポリゴン数の低減や深度値の活用による認識や追跡の効率化．これら3点を同時に達成する試行錯



(a) 3D レンダリング　　**(b) 認識と追跡の表示**

図 18　追跡処理

誤の過程から得られたプラクティスは3.3節に設計指針として示した．以下その適用と評価を述べる．

本システムの実行画面を**図19**に示す．①は第3，4章で説明した2D深度画像処理，②は図9の3Dモデリング，③は図15の3Dレンダリング，④は図18の認識と追跡処理である．また，人物のエッジ画像表示や，③とシンクロナイズして，マウス操作で拡大，回転可能なワイヤーフレーム画像も表示し，各種処理効果の確認に用いた．

本システムの応答速度を**図20**に示す．a ～ dは画面①～④の各処理を累積した場合のフレームレートである．dの段階で40.4[fps]（GPU利用率88%）となった．また4つの画面のマルチスレッド化により46.0[fps](GPU利用率98%)と，約14%速度が向上した．

今回の試作では，応答速度の目標値を，一般的な動画カメラ速度の30[fps]とした．この処理間隔33.3[ms]は，並列演算未使用時には，表1のメディアンフィルタの処理だけで飽和してしまう．設計指針適用の結果，1フレームあたりの所要時間は21.7[ms]となった．カメラ入力から，手の認識と追跡の結果表示までの応答時間が，この所要時間内に収まることになる．実際に，認識された状態で素早く手を上下に動かしても，追跡表示に目立った遅延や不連続さは感じられず，良好なレスポンスが得られることが確認できた．

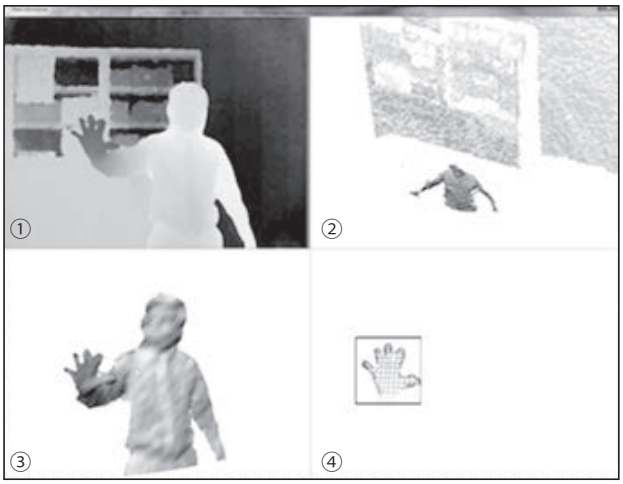


図 19　深度画像処理システムの画面

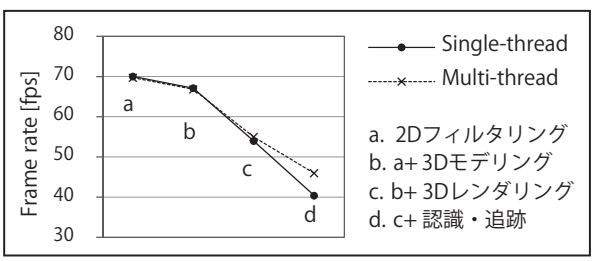


図 20　システムの応答速度

本システムでの投影やテッセレーションなどの3D処理の活用は，通常のカメラ画像処理では見られない特徴である．また，本システムはGPU利用率がきわめて高く，GPU性能への依存が高い．GPUの今後の進歩により，本システムはさらなる性能向上が期待できる．

7.2　DirectX11 活用におけるプラクティス

今回DirectX11を用いて，本システムの多くの処理をGPUで行った．そこで得られたプラクティスを述べる．

3D処理と並列演算の関係については，以下の観点から，親和性の高さが確認できた．

- HLSLによる文法の統一性．
- C++ソースコードにおける処理の柔軟な混在．
- GPU上でのリソース共有および相互の結果利用．

ホスト言語側のDirectX11は，性能重視のためAPIが複雑である．テクスチャの作成やコピー処理は，種類や制約が多く，使い分けを要する[4],[5]ため，各種機能を実験しつつ開発していくことが必要である．

一方HLSLは，C言語に類似しており，さらにテクスチャ座標，ベクトル（uint2, float3型）や行列（float4×4型）演算はC言語より簡素かつ便利であり，また，組込み関数の多くは，スカラ型にもベクトル型にも適用でき，一貫性と利便性が考慮されているというように，記述容易性が特徴である．

そこで両者の得失を見極めたプログラミングを実施することで性能と開発効率を両立させることが可能になると考え，本試作では，3D処理と並列演算を連携させて，性能および開発能率向上を実現した．

ここで得られたプログラミングレベルのプラクティスは次の通りである．

- ポリゴンデータの演算，画像処理などのデータ処理をGPU側に集中させ，3D処理と並列演算のGPU上での相互結果参照を活用する（RWTexture2D型の使用）．これらはCPU-GPU間の転送頻度軽減にもつながる．
- HLSLでは関数はインライン展開されるので，オーバヘッドを気にせず可読性を重視する．なお，ループ構造のインライン展開指示は速度を実測し選択する．
- CPU側のDirectX11には準備と制御を担当させ，処理の複雑さをC++によりラッピング，隠蔽する．また，使用頻度の多いテクスチャ，ポリゴンメッシュ，シェーダ等をオブジェクト化し，初期化や参照，

実行などが1行程度で呼び出せるようにする．

このようなプラクティスを得るためには，実験的なプログラミングの実践によって，各種機能およびその組合せが実際どのような性能を示すか確認し開発を進めることが重要である．たとえば，今回の場合も，テッセレーション機能を適用したとき，カメラ視野に複数の人物が入る場合は，顕著に処理速度が低下した．これは複数の人物エッジによる頂点数増加が原因で，図1のVertex ～ Geometry ShaderのGPU負荷が大きくなるためであるが，解決策としてDirectX11のストリーム出力機能の有効性が実験から得られた．

7.3 今後の課題と発展性

今後の課題として，深度値のNull値との振動問題については，物体の移動状態をもとに，時間軸での平滑化を検討したい．また，追跡処理では，手の角度変化や手首部分が影響して，移動ベクトルにずれが生じるため，手の角度検出や，手首部分の除去による対応を検討したい．

今後の発展としては，さまざまな角度の手の認識や，3次元上の軌跡認識や指示方向の認識など，本システムの上位レイヤとなる部分の研究があげられる．それにより，GPUを活用した高レスポンスなジェスチャ操作や，3Dモデルのフィードバック表示を活用した，NUIの応用システムの考案や開発に取り組んでいきたい．

参考文献

1）Boyd, C.: DirectX 11 DirectCompute, Gamefest2010, http://www.microsoft.com/download/en/details.aspx?id=16995

2）Breshears, C.: 並行コンピューティング技法，オライリージャパン，pp.116-120, 131-181 (2009).

3）Gray, B. and Adrian, K: 詳解 OpenCV, オライリージャパン，pp.321-334 (2009).

4）Yang, J.: Basics of DirectCompute Application Development, http://channel9.msdn.com/Blogs/gclassy/

5）MSDN ライブラリ : Windows DirectX Graphics Documentation (Aug. 2009), http://msdn.microsoft.com/ja-jp/library/

6）Thibieroz, N.: Shader Model 5.0 and Compute Shader, GDC2009, http://developer.amd.com/documentation/presentations/pages/

7）高木幹雄，下田陽久 : 新編画像解析ハンドブック，東京大学出版会，p.869, 1669 (2004).

8）田村秀行 : コンピュータ画像処理，オーム社，pp.115-116 (2002).

9）Valdetaro, A., Nunes, G., Raposo, A. and Feijó, B.: Understanding Shader Model 5.0 with DirectX 11, SBGames 2010 - IX Simpósio Brasileiro de Jogos e Entretenimento Digital, pp.1-18, http://sbgames.org/sbgames2010/proceedings/tutorials/tuto_comp1.pdf

10）Engel, W.: GPU Pro 2, A K Peters/CRC Press, pp.3-14, 387-471 (2011).

深井 裕二（非会員）fukai@hit.ac.jp
北海道工業大学創生工学部情報フロンティア工学科講師，情報工学分野，プログラミングの研究に従事．

竹沢 恵（正会員）mtakezaw@hit.ac.jp
北海道工業大学創生工学部情報フロンティア工学科准教授，画像処理分野の研究に従事．

島 貢（非会員）shima@hit.ac.jp
北海道工業大学創生工学部情報フロンティア工学科教授，品質工学，情報ネットワーク分野の研究に従事．

川上 敬（正会員）kawakami@hit.ac.jp
北海道工業大学創生工学部情報フロンティア工学科教授，知的システム工学，知能情報工学，最適化分野の研究に従事．

投稿受付：2011 年 8 月 2 日
採録決定：2013 年 4 月 23 日
編集担当：茂木　強（(独）科学技術振興機構）