2008－CVIM－163 (32)
2008／5／9

画像処理とコンピュータビジョンのためのGPU

ノジク ヴァンソン[†] 石川 尋代[†] フランソワ ドゥ ソルビエ[‡]

[†]慶應義塾大学大学院 理工学研究科 〒223-8522 横浜市港北区日吉 3-14-1
[‡]Université Paris-Est LABINFO-IGM UMR CNRS 8049

E-mail: [†]{nozick, hiroyo}@ozawa.ics.keio.ac.jp, [‡]fdesorbi@univ-mlv.fr

あらまし CPUとは異なりGPUでできることは増加し続け，画像処理に対して想定以上の性能を発揮している．実際にGPUは主にコンピュータグラフィックスのアプリケーション用に設計されており，2Dテクスチャや3D処理を非常に効率よく行うことができる．こういった能力はリアルタイム画像処理やコンピュータビジョンのアプリケーションソフトウェアの速度向上に適している．また，初期のシェーダー言語に比べて，最新版のシェーダー言語はよりいっそう簡単に使用できるようになっている．しかしながら，GPUを効率よく使用するためにはいくつかのシェーダー基礎知識が必要とされている．本稿では画像処理とコンピュータビジョンにおけるシェーダーの使い方の概要を記述する．

本稿では，はじめに，コンピュータグラフィックスのレンダリングパイプラインとシェーダーの一般概念を詳細に述べる．この部分は現存するシェーダー言語の概要，特にGLSLの詳細な記述でもある．これらの解説にはシェーダープログラミング，さらに，シェーダーの読み込みとコンパイルに関する部分も含まれる．ここではシェーダーは特別なプログラム設計となっていることを理解することが目的である．次に，画像処理とコンピュータビジョンに関するいくつかの実用的なアプリケーションを提供する．これらの章では特に色操作，幾何的アプリケーション，さらにバーチャルリアリティや一般的な目的のためのGPU使用方法を扱う．そして，最後に，技術的，理論的なアドバイスとともにシェーダーの導入方法を提供する．

GPU for Image Processing and Computer Vision

Vincent Nozick[†], Hiroyo Ishikawa[†] and François de Sorbier[‡]

[†]Graduate School of Science and Technology, Keio University
3-14-1 Hiyoshi Kohoku, Yokohama, 223-8522, Japan
[‡]Université Paris-Est LABINFO-IGM UMR CNRS 8049

E-mail: [†]{nozick, hiroyo}@ozawa.ics.keio.ac.jp, [‡]fdesorbi@univ-mlv.fr

Abstract Contrary to CPU, GPU capability continues to increase and reaches unexpected performances for image processing tasks. Indeed, GPU is mainly designed for computer graphics applications and can deal with 2D texture or 3D operations very efficiently. These abilities are well suited to speed up real-time image processing and computer vision applications. Contrary to the first generations of shaders, the latest shaders languages become more and more easy to use. However, some basic knowledge about shaders is required for an efficient use of the GPU. This paper presents an overview of how to use shaders for image processing and computer vision. The first part details the computer graphics rendering pipeline and shaders generalities. This part also overview the existing shader languages and especially details GLSL. These explanations include shaders programming but also the shader loading and compilation. This part aims to underline that shaders require a specific program design. Then, this paper presents some practical applications related to image processing and computer vision. These sections especially deal with colour manipulation, geometric applications but also with virtual reality and general purpose GPU methods. Finally, this document provides a part on how to start with shaders, including technical and theoretical recommendations.

1. はじめに

コンピュータのあらゆる部分でグラフィックカードは主な位置になる傾向がある．CPU に比べて GPU(Graphic Process Unit)は年々強い力をもってきている．当然の結果として，多くの新しいアプリケーションは処理速度を上げるために GPU を用いて実行することが増えている．

本論文はどうして GPU を使うのか，特に画像処理とコンピュータビジョンでなぜ GPU を使うのかという問いに答えることを目的とする．Shader プログラムによって GPU は主にリアルタイムシステム専用になり，しばしば，アプリケーションの処理速度が上がることを最初に知る．コンピュータグラフィックス(CG)の技術は画像処理やコンピュータビジョンの手法に類似している場合があるという事実を考えると，画像処理やコンピュータビジョンの研究者が GPU に興味を持つということもまた自然なことである．コンピュータサイエンスの技術者のための最初のアプローチはすべてのアプリケーションを CPU から GPU へ移行することである．しかしながら，それはいつでも可能という訳ではないため，CPU で実行するよりも GPU で実行した方が非常に高速であったとしても驚くことではない．もう一つのアプローチはアプリケーションの半分だけ GPU へ移行することである．それゆえ，コンピュータのリソースは活用され，パフォーマンスは向上する．本稿では，最初にいくつかのコンピュータグラフィックスパイプラインの基本知識を紹介し，“Shader”とは何かを説明する．次に Shader プログラムで Shader をどのように利用するか示す．そして，最後に GPU で効果的に実行できるいくつかの画像処理とコンピュータビジョンの応用を紹介する．

2. コンピュータグラフィックスの基礎

GPU は CG アプリケーションのために設計されるため，はじめに CG について概要を説明し，画像処理やコンピュータビジョンでどのように使用するか理解する．

2.1. グラフィックパイプライン

CG の主な目的はデータセット(3 次元幾何，色，...)を色情報，すなわち画像に解釈することである．点や線，三角パッチのようなプリミティブが与えられたら，いくつかの命令ステップで処理された後，画面上の複数ピクセルの色が決定される．これらの三角パッチと画像上のピクセル間のステップはレンダリングパイプラインによって決定される．これらのパイプライン処理は CPU 上や一般に GPU と呼ばれている専用のプロセッサ(3D グラフィックアクセラレーター)を持つグラフィックカード上で行われる．GPU 専用の主な二つのレンダリング API は OpenGL (Khronos Group)[1]と Direct3D (Microsoft)[2]である．これらの API のパイプラインはほぼ同じである(図 1 参照)．

2000 年までレンダリングパイプラインの処理は部分修正することができなかった．しかし，Shader の導入により GPU のいくつかの特定の位置でプログラムすることが可能となった．それゆえ，GLSL[3](OpenGL)や HLSL[4](Direct3D)のような特定の言語を使用し，正しいドライバの拡張機能がインストールされているなら，頂点上やフラグメント上の操作を再定義することができる．これらの操作の再定義ために，Vertex プログラムやフラグメントプログラムは選択された言語で書かれる．そして，これらのプログラムはコンパイル，リンクされグラフィックアプリケーション上で実行される．図 2 はパイプライン上でのこれら 2 つの Shader プロセスの位置を示している．2007 年には Geometry Shader と呼ばれる新しい種類の Shader が紹介された[5]．その目的はプリミティブに適用する操作を再定義することであり，プリミティブのアセンブリ位置で発生する．これらの 3 つの Shader プログラムの特性は以下の節で示す．

2.1.1. Vertex プログラム

最初に取り上げるプログラム可能なステップは Vertex プログラム(または Vertex Shader)によって実行される頂点操作である．このプログラムでユーザは入力されたプリミティブに特定のふるまいを割り当てることができる．図 3 にティーポットを描く例を示す．図 3(a)では通常の状態で示す．一方，図 3(b)では Vertex プログラムを用いて，3 角パッチの頂点の位置と色をその頂点自身の 3 次元位置によって変化させてレンダリングした結果を示す．

2.1.2. Geometry Shader

Geometry Shader は最新のグラフィックカードでのみ利用できる GPU の能力である．このプログラムは図 4 で描くように，いくつかの頂点に対し複製や追加，削除を行うために使用される．

2.1.3. フラグメントプログラム

最後のプログラム可能なステップはフラグメントプログラム(または Pixel Shader)である．フラグメントプログラムはラスタライゼーションで最終的に三角パッチがピクセルに分割された後に発生する Vertex Shader の位置のようにフラグメントの特性によってユーザは特定のふるまいをフラグメント(ピクセル)に割り当てることができる．図 5(a)は通常のパイプラインで描いたティーポットである．図 5(b)はフラグメントプログラムを用いて同じ物体をフラグメントのフレームバッファでの 2 次元位置によってフラグメント自身の色を変えてレンダリングした結果である．特筆すべきことは，上と下のティーポット間で回転はフレームバッファに描かれたティーポットのピクセルの色に影響を及ぼさないことである．

2.2. 変換

CG に関する操作では GPU は主に三角パッチの変形と 2 次元のテクスチャに関して取り扱う．三角パ

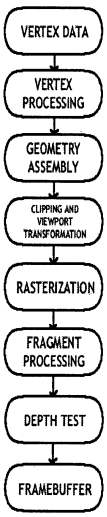


図 1　Rendering pipeline

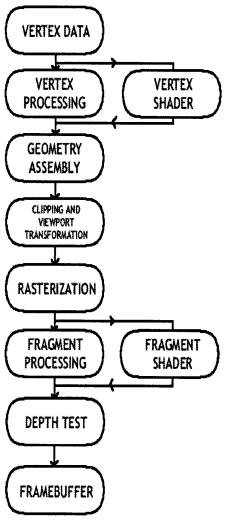
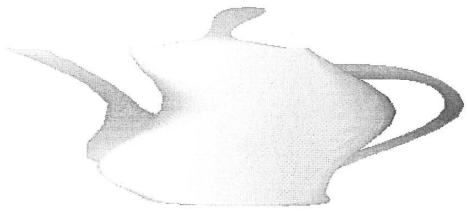


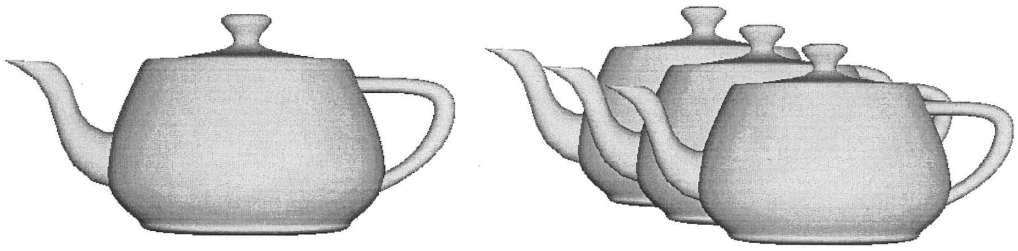
図 2　Rendering pipeline with shader programs

(a) Normal rendering



(b) Example of vertex manipulation (colour and position)

図 3　Vertex プログラムの例



(a)Normal rendering　　(b) Duplication of the incoming primitives

図 4　Geometry Shader の例

(a) Normal rendering　　(b) Example of fragment manipulation

図 5　フラグメントプログラムの例

ッチの変形は P3(4 次元空間)射影幾何で実行される．通常 P2 を用いるコンピュータビジョンアプリケーションはこのようなアーキテクチャを簡単に利用することができる．2 次元テクスチャ操作は GPU で行うと非常に効率がよく，画像処理プログラムに対して画期的な速度向上を提供することができる．

前章で説明したパイプラインはいくつかのパラメタ設定を必要とする．これらのパラメタに対して，CG とコンピュータビジョンの共通点を作成することは簡単である．コンピュータビジョンではカメラの解像度で画像の解像度が決定されるが CG ではユーザ(ビューポート)が決定する．CG ではカメラは投影行列で指定される特定の焦点距離を持つレンズを持つ．コンピュータビジョンでは，内部パラメタ行列が焦点距離とピクセルの特性を記述する．そして，CG ではカメラはシーンの座標空間でモデルビュー行列を用いて配置されなければならない．モデルビュー行列はコンピュータビジョンでは外部パラメタである．主に違う点は CG において現れる．シーンでは回転と並進，拡大縮小を組み合わせたモデリング行列を用いて物体を配置しなければならないが，コンピュータビジョンでは存在する物体の情報が獲得したい．

そして，いくつかのプリミティブは 1 次元や 2 次元，3 次元のテクスチャを用いて生成される．これ

らのテクスチャはテクスチャ行列で定義されるテクスチャ座標を用いて表面にマッピングされる．この行列はGPUを用いたコンピュータビジョンのアプリケーションで非常に役立つ．注意したい点はこれらのテクスチャはカラー画像であるが，整数や実数の配列でもよいということである．そしてこれらのテクスチャはルックアップテーブルのようにテクスチャ以外の目的でも使用できる．

2.3. グラフィックカード

Shaderを使う場合，グラフィックカードは最も重要なハードウェアであり，実行速度はその品質で決まる．実際，異なるグラフィックカードは異なるプロセッサを供給するため，グラフィックカードによって利用できる能力が異なる．

Shaderを利用するには最近のグラフィックカード(少なくともATI Radeon 9XXXまたはNVIDIA GeForce 5)と利用できる最新のドライバが必要である．最新のグラフィックカードと古いものの差は非常に大きい．グラフィックカードがOpenGLのAPIを使用するShaderと互換性を持つかどうかは以下の拡張機能を提供しているかを調べればよい．

- GL_ARB_fragment_program
- GL_ARB_vertex_program
- GL_ARB_shading_language_100

CPUと比較して，GPUのプロセッサはシンプルな(または明確な)操作のために設計されているが，速く命令を処理することができる．さらに，グラフィックカードは複数の並列化されたパイプラインから構成され，Shaderプログラムは完全に並列処理できるように設計されている．しかし，パイプラインの数を増やすことはグラフィックカードの性能を自動的に向上させるわけではない．プロセッサの周波数はそれぞれの処理速度を決定するため非常に重要である．そのため，良いグラフィックカードはパイプラインの数とプロセッサの周波数の比率がよい．現在のグラフィックカードの特性の例を表1に示す．

最近，グラフィックカードの性能を向上させるため，新しい種類のアーキテクチャが開発されている．それは2つまたはそれ以上のグラフィックカードを1つのPCに取り付ける．この種のアーキテクチャはNVIDIAのグラフィックカードでSLIと呼ばれ，ATIのグラフィックカードではCrossfireと呼ばれる．1つのグラフィックカードに2つのプロセッサがあるものがリリースされている．しかし，性能はまだ不確かである．

その他のグラフィックカードの重要な特性としてはCPU/RAMとグラフィックカードの間でデータをやり取りするために使用するBUSがある．最新の効率良いBUSはPCI Expressである．PCI Expressではメインメモリとグラフィックカード上のメモリ間の高速なデータ転送(テクスチャのような)を許可している．実際，PCI Express BUSのセカンドバージョンは最近のマザーボード上にインストールされており，パフォーマンスが向上している．

このデータ転送はGPUプログラミングを行う上で認識しておかなければならない重要な点である．実際，メインメモリからグラフィックカードへのテクスチャの転送はデータ処理の時間に比べて比較的長い時間を費やす．

2.4. Language

グラフィックカードのためのプログラミングは特定の言語を使用しなければならない．表2に主なGPU言語の進化を年代順に示す．これらの言語は低級か中級言語であり，グラフィックパイプラインとGPUの能力に関して若干の知識を含んでいる．その他の言語は高級言語でありアプリケーションのどの部分がGPUへ移行されるかによって選択できる．これらの言語には完全に画像処理やコンピュータビジョンアプリケーション向きのAPIがいくつか存在する．

2.4.1. Medium level

GPUプログラミングで最初に使用されたのはアセンブリ言語であるが，それは乱雑でエラーを起こしやすかった．その後，いくつかの言語(GLSL，HLSL，Cgなど)が簡単なGPUプログラミングのために設計された．これらの言語のコーディングの構文はC言語に近い．

OpenGL の Shader 言語や GLSL[3]は 2003 年にリリースされた．その目的はクロスプラットフォームなミドルレベルの手続き言語を提供することであった．HLSL(High Level Shading languag)[4]は Microsoft と NVIDIA によって開発されており， Microsoft がサポートし続けるため，アプリケーション開発に非常に人気がある．そして，Cg(C for graphics language)[7]は NVIDIA と Microsoft によって開発されたため HLSL に非常に類似している．HLSL との主な相違点として Cg は OpenGL，Direct3D の両方のアプリケーションで使用できることがある．これらすべての言語は非常に強力であるが，レンダリングに関する知識を必要である．GPU で利用できる操作がよりいっそう複雑になったとしても，これらの最新の言語を使用するのは難しくない．

2.4.2. High level

新しい複雑な言語を習得するのが困難な場合は CUDA (C style) [8]や Sh (C++ style) [9]を使用することができる．これらの言語は高級言語でハードウェアの部分をユーザに隠蔽している．

表 1　the properties of graphic cards

Manufacturer	Name	Year	Vertex pipelines	Fragment pipelines	Transistors (millions)	GPU frequency	Vertex shader version	Pixel shader version
ATI	Radeon HD3870	Nov. 2007	320	320	666	775	4.0	4.0
ATI	Radeon HD2900 XT	May 2007	320	320	720	740	4.0	4.0
ATI	Radeon X1900 XT	Jan. 2006	8	48	380	625	3.0	3.0
ATI	Radeon X800 XT	Jun. 2004	6	16	160	500	3.0	3.0
Nvidia	Geforce 9600 GT	Feb. 2008	64	64	505	1625	4.1	4.1
Nvidia	Geforce 8800 GTX	Nov. 2006	128	128	681	1350	4.0	4.0
Nvidia	Geforce 7900 GTX	Mar. 2006	8	24	278	650	3.0	3.0

表 2　GPU language history

1985	TMS34010 is the first programmable graphics processor (TI)
1988	RenderMan
1992	OpenGL 1.0
2000	NV_register_combiners extension
2001	GLSL
2002	HLSL, Cg, ARB_vertex_program and ARB_fragment_program extensions
2004	OpenGL 2.0
2006	unified shaders architecture
2008	OpenGL 3.0?

2.4.3. Dedicated languages

最後に，画像処理とコンピュータビジョンに対して使用することが目的ならば，GPU アプリケーションをすぐに使えるような OpenVIDIA[10]を試すことができる．OpenCV を使用しているユーザは OpenCV[11]と互換性を持つ GPUCP[12]を見るとよい．この API を使用するにあたって，基本的な OpenCV プログラムを変更することはほとんどない．CPUCV はグラフィックカードでできるタスクを選択する．

これらの最新の専用言語や API は多くの画像処理やコンピュータビジョンのアプリケーションには十分であるが，下級言語は革新的な手法を生成するか特定の問題を解決することを要求される．

3. GLSL

本論文の以降の部分では GLSL を用いたアプリケーションの例を示す．この言語は Cg や HSLS と同じ理論に基づいているため，GLSL の例は簡単に変換できる．GLSL は中級言語であり，マルチプラットホーム言語(Windows，Linux，MacOS X，irix)であるため選択した．さらに，この API はオープンソースである．

3.1. 概要

GLSL はそれ自身の規則を持つ Shader 言語であり，頂点 Shader やフラグメント Shader を記述するのに用いられる．これらの最新バージョンはメインのグラフィックアプリケーションの外に記述される独立したコードで，GLEW ライブラリによって拡張される関数やウィンドウマネージャが使用する gls(Linux)や wgl(Windows)ライブラリの特定の関数を可能にする．

この独立した Shader プログラムの主な長所はグラフィックアプリケーションを再コンパイルすることなく Shader のコードを変更することができる．

3.2. Language description

GLSL は簡単で直観的であるため，C と C++の特徴を使用する．以下の章では GLSL 言語を紹介する．詳細な情報は[3]を参照のこと．

3.2.1. データ型

GLSL ではスカラーに関して，符号付浮動小数点の float(32bit)，符号付整数 int，ブール型 bool を扱う．いくつかの変数は C スタイルの構造で構築できるか，配列として扱うことができる．GLSL ではポインタは扱わない．

ビルトインのベクトル，行列，sampler(テクスチャ)がある．ベクトルは浮動小数点の入力を含み，2 次元(vec2)，3 次元(vec3)，または 4 次元(vec4)が可能である．値へのアクセスは括弧を使うかビルトインアクセスを使う C スタイルの方法で実行できる．4 つの排他的でないアクセスはベクトルに利用できる．

x,y,z,w：P3 における点の 3 次元位置
r,g,b,a：4 次元のカラーベクトル
s,t,p,q：テクスチャ座標

行列のビルトインタイプも浮動小数点の要素を含む．これらの行列は正方行列で 2x2(mat2)，3x3(mat3)，4x4(mat4)でなければならない．OpenGL でのようにこの行列は Column-major order である．

最後に sampler のビルトインは 1 次元(sampler1D)，2 次元(sampler2D)，3 次元(sampler3D)となるテクスチャを記述する．テクスチャへのアクセスはテクスチャ関数で行う．2 次元テクスチャに対しては sampler2D テクスチャを必要とする texture2D 関数を使用しなければならない．そして，2 次元ベクトル座標は 0.0 から 1.0 に正規化されたものとする．結果の色は近傍のピクセル値か近傍のピクセル値から補間した色となる．

3.2.2. 関数とビルトイン関数

関数を作成するためにユーザはパラメタのタイプと qualifier を定義しなければならない．この qualifier は"in"は関数内に値をコピーするが返さない，"out"はパラメタをコピーして戻される，"inout"は両方である．関数は C スタイルの値を返すこともできる．

GLSL は GPU に対して最適化されたビルトイン関数のセットも提供する．提供された関数の中には三角関数(sin, cos, asin, acos, tan, atan, …), 幾何関数(length, distance, dot, cross, nomalize, …), さらに, (pow, exp, log, sqrt, abs, floor, ceil, mod, min, max, clamp, …)がある．

4. Shader Program

以降の部分で異なる Shader プログラムの構造を紹介する．

4.1. Vertex shader

4.1.1. Example

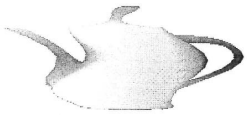
以下の例は Shaer を使わず，デフォルトのふるまい以外は何もしない Vertex プログラムで表している．

```
void main(void)
{
 gl_Position = ftranform();
  // this command is equivalent to the following line
  // gl_Position = gl_modelViewProjectionMatrix * gl_Vertex;
}
```

default program

次の例は最初に頂点の色をその初期位置によって変更した後，頂点の位置を x，y に正弦ノイズを用いて変更した．この頂点は最終的にシーンの座標空間からカメラの座標空間に変換される．

```
void main(void)
{
 gl_FrontColor = vec4(gl_Vertex.x/2+0.2,
                      gl_Vertex.y/2+0.2,
                      gl_Vertex.z/2+0.2,
                      1.0);

 gl_Vertex = vec4(gl_Vertex.xy+0.5*(1.0+sin(gl_Vertex.z*3)),
                  gl_Vertex.z,
                  1.0);

 gl_Position = gl_modelViewProjectionMatrix * gl_Vertex;
}
```



example program

4.1.2. Attribute, uniform and varying

Vertex プログラムにいくつかのパラメタを追加することが可能である．3 種類のパラメタ(attribute, uniform , varying)について説明する．

attribute パラメタは色や法線のような頂点ごとの値である．これらはメインアプリケーション(OpenGL プログラム)で初期化され，Vertex プログラムに送られる．

uniform 変数もメインアプリケーションで定義されるが，1 つの変数が同じ値で，すべての頂点に利用できる．これは閾値やテクスチャなどのどんなデータでもよい．

varying の値は Vertex プログラムからフラグメントプログラムに通信できるように設計されている．それゆえ，varying は Vertex プログラムで定義され，その値はどのフラグメントにも補間できるようにすべての頂点で独立に定義できる．これらの変数は色，位置，番号，どんな頂点情報でもよい．

4.2. Geometry shader

Geometry Shaer の主な目標は追加，複製，入力された三角パッチや線，点などのプリミティブの頂点の変換である．したがって，Geometry Shader は画像処理とコンピュータビジョンにはおそらく適しておらず，本稿では扱わない．また，Geometry Shader は最新の NVIDIA のグラフィックカード(NVIDIA GeForce 8 family)のみで利用可能である．

4.3. Fragment shader

4.3.1. Example

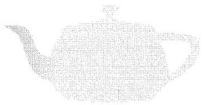
以下の例は Shader なしのデフォルトのふるまいをするフラグメントプログラムの例を示す．

```
void main(void)
{
 gl_FragColor = gl_Color;
}
```

default program

次の例はフラグメントの色はフラグメントバッファで正規化された位置によって変更している．

```
void main(void)
{
 gl_FragColor
     = vec4(gl_Color.xyz + gl_FragCoord.xyz * 0.005, 1.0);
}
```



example program

4.3.2. Attribute, uniform and varying

フラグメントプログラムにおいても Vertex プログラムのように，3 種類のパラメタ(attribute, uniform , varying)をサポートしている．

attribute パラメタは色や depth のようにフラグメントごとの値である．これらの値は Vertex プログラムで初期化され，フラグメントプログラムに送られる．

uniform の値はメインプログラムで定義され，そして，また，1 つの変数がすべてのフラグメントで使用される．この値は閾値やテクスチャなど，どんなデータでもよい．

varying 値は Vertex プログラムからフラグメントプログラムに通信するために設計された．それゆえ，varying は Vertex プログラムで定義され，値は各頂点で独立に定義でき，各フラグメントで頂点を補間する．これらの値は色，位置，番号など，どんな頂点データでもよい．

4.3.3. FrameBufferObject と Multiple Render Target

OpenGL のレンダリング処理に新しい機能を加えるために 2 つの役に立つ拡張が開発された．最初のひとつはフレームバッファでレンダリングする代わりにテクスチャ上でシーンのレンダリングをすることを許可する FrameBufferObject (FBO)である．この拡張は Shader を使っているアルゴリズムがマルチパスプロセスを必要とするときにしばしば使われる．パスの結果はテクスチャに格納され，次のパスの Shader に送ることができる．ラジアルディストーションの補正は FBO の通常利用の典型的な例である．FBO はオフスクリーンレンダリングにも用いられる．

2 番目の拡張は Multiple Render Target (MRT)であり，ユーザがいくつかのバッファまたはテクスチャ(FBO を使う)に Shader で同じピクセルを使用することを許可する．

4.4. Creating a shader

Vertex プログラムとフラグメントプログラムを使うために一組のステップを GLSL コードからアセンブリコードに変換しなければならない．このスキームは C や C++などのコードベースのアプリケー

ションを作成するときに用いる処理に類似している．最初に OpenGL の glCreateShaderObjectARB 関数 と glShaderSourceARB 関数を用いてソースコードを Shader object と呼ばれる構造に格納する．次に glCompileShaderARB 関数で各読み込まれたソースコードを個別にコンパイルする．Shader のコンパイルに失敗した場合，Shader object 情報ログにコンパイルに関する情報が出力される．Shader を作る最後のステップではコンパイルされたコードを単に Shader と呼ばれる他のオブジェクトにリンクする．必要なものは glAttachObjectARB 関数と glLinkProgramARB 関数である．glUseProgramObjectARB 関数を使うレンダリング処理に使われるのは Shader object である．

最終的に glGetUniformLocationARB 関数を使用している uniform の値を定義し，glUniform1iARB 関数でこの変数に設定することが可能である．この変数は Shader プログラムとこの関数で同じ名前にする必要がある．

```
GLhandleARB vertexShader;
GLhandleARB fragmentShader;
GLhandleARB programShader;

...

vertexShader = glCreateShaderObjectARB(GL_VERTEX_SHADER_ARB);
GLcharARB *l_Source = loadShaderFile("plop.vert");
const GLcharARB **l_SourcePtr = const_cast<const GLcharARB**>(&l_Source);
glShaderSourceARB(vertexShader, 1, l_SourcePtr, 0);
delete [] l_Source;
glCompileShaderARB(vertexShader);


fragmentShader = glCreateShaderObjectARB(GL_FRAGMENT_SHADER_ARB);
l_Source = loadShaderFile("plop.frag");
const GLcharARB **l_SourcePtr2 = const_cast<const GLcharARB**>(&l_Source);
glShaderSourceARB(fragmentShader, 1, l_SourcePtr2, 0);
delete [] l_Source;
glCompileShaderARB(fragmentShader);


programShader = glCreateProgramObjectARB();
glAttachObjectARB(programShader, vertexShader);
glAttachObjectARB(programShader, fragmentShader);
glLinkProgramARB(programShader);

...

glUseProgramObjectARB(programShader);
GLint loc = glGetUniformLocationARB(programShader,"imageWidth");

int inputImageWith = 800;
glUniform1iARB(loc, inputImageWidth);
```

openGL.cpp

4.5. Debugging

Shader をデバッグすることは簡単ではない．OpenGL API は変数やバッファの状態をチェックするツ

ールを提供していない．Shader が正しいかどうかをチェックする唯一の方法はレンダリング処理の結果を視覚化することである．

しかし，チェックを容易にして，速度を上げるためにデバッガがリリースされた．GLSL で使用できる主な 2 つのデバッガは glslDevil [13,14] と gDEBugger [15]である．これらのデバッガはユーザインタフェースが使いやすく，頂点やフラグメントの特性を示し，結果のインタラクティブな可視化を許可する．どちらのデバッガも Windows と Linux 上で利用できる．

5. Display an image

前章で説明したように，GPU は特定のタスクを CPU より速く実行するように設計されている．さらにそのタスクを並列で処理することができる．CG アプリケーションは画像処理やコンピュータビジョンのタスクに類似しているものもあるため，Shader を利用することでこれらのアプリケーションのスピードアップにもつながる．

しかし，GPU を利用には，メインメモリから GPU に，または GPU からメインメモリにデータ(画像)を転送するというボトルネックも存在する．さらに，GPU の使用はたまにアルゴリズムを GPU のアーキテクチャに合うように適用させることが必要になる．幸いなことに多くの単純な画像処理では GPU を使うことはアプリケーションのスピードアップにつながる．このようなアプリケーションを示す前に，GPU を用いた画像の表示についての概要を示す．

基本的に画像の描画は画像を用いて四角形(2 つの三角形)にテクスチャを貼ることである．この目的を達成する方法には多少速いものや，直感的なもの，使い方が簡単なものがある．最初に描画バッファの解像度を画像解像度(ビューポート)上に設定しなければならない．それから，正射影を選択しなければならない．透視投影は要求されない．直交カメラの次元は表示する画像の解像度と一致させなければならない．図 6 のようにカメラの前にあるスクリーンサイズの長方形に完全に描く．もし，Shader を使用しなかったら，テクスチャ座標を指定しなければならない．もし，Shader を使うなら．フラグメントプログラムにおいて座標を決定する．

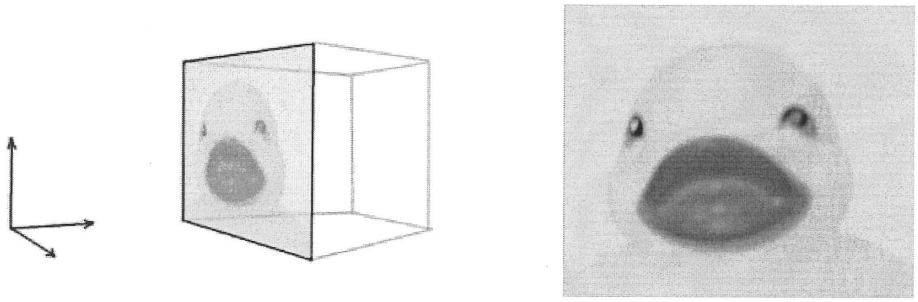
Vertex プログラムではほとんどすることがない．それは頂点の位置によって初期化される varying 変数 pos を定義するだけある．varying 変数はそれらの位置を知っているためすべてのフラグメントを補間する．

```
varying vec2 pos; // pixel position for the fragment

void main(void)
{
 // 2D position to be interpolated for the fragment program
 pos = vec2(gl_Vertex.xy);

 // normal transformation
 gl_Position = ftranform();
```

フラグメントプログラムは多少長くなる．フラグメントプログラムには表示されるテクスチャ(画像)や大きさのようないくつかのグローバル uniform 変数が含まれる．これらの大きさはピクセルの位置を知る必要がない描画手法では省略可能である．すなわち，色情報だけで十分である．しかしながら，すべての例で使用できるアプローチをとる．

Shader で定義されるさまざまな varying 変数はこのとき補間値でリカバーされる．

メインプログラムでは，最初に varying 変数を用いて現在のフラグメントのテクスチャ座標をテクスチャ上から見つけなければならない(我々の手法ではテクスチャの大きさも使っている)．テクスチャが GL_LINEAR オプションで初期化された場合，ピクセルの色は周囲のピクセルから補間する．



(a) Orthogonal projection　　(b) Final image

図 6

テクスチャが GL_NEAREST オプションで初期化された場合，要求されたピクセルからもっとも近いピクセルの値が返される．最後にこの色はフラグメントに塗られる．

```
uniform sampler2D image;       // input image
uniform int       imageWidth;  // input image width
uniform int       imageHeight; // input image height
varying vec2      pos;         // pixel position of the fragment

void main(void)
{
 // find the texture coordinate corresponding to this fragment
 vec2 texCoord = vec2(pos.x/imageWidth, pos.y/imageHeight);

 // get the corresponding color
 vec4 color = texture2D(image, texCoord);

 // final color
 gl_FragColor = color;
}
```

6. Colour operation

カラー処理はどのピクセルも通常，基本的な操作で並列処理されるため，最も簡単な GPU を使用する画像処理アプリケーションだろう．

6.1. Basic colour operations

画像処理タスクには最初のカラー空間よりも他のカラー空間で実行した方が有効な場合がある．GPU は以下の例で記述されるようにカラー空間を変換させることがある．

colour space conversion (ex: RGB to YUV)

```
uniform sampler2D image;       // input image
uniform int       imageWidth;  // input image width
uniform int       imageHeight; // input image height
varying vec2      pos;         // pixel position of the fragment

void main(void)
{
```

colour space conversion (ex: RGB to YUV)

```
 // find the texture coordinate corresponding to this fragment
 vec2 texCoord = vec2(pos.x/imageWidth, pos.y/imageHeight);

 // get the corresponding color
 vec4 color = texture2D(image, texCoord);

 // colour conversion
 color.r =  0.257*inputColor.r + 0.504*inputColor.g + 0.098*inputColor.b +  16.0/255.0;
 color.g = -0.148*inputColor.r - 0.291*inputColor.g + 0.439*inputColor.b + 128.0/255.0;
 color.b =  0.439*inputColor.r - 0.368*inputColor.g - 0.071*inputColor.b + 128.0/255.0;

 // final color
 gl_FragColor = color;
}
```

他の処理としては閾値計算のようなパラメタを追加する必要がある.

Threshold

```
uniform sampler2D image;       // input image
uniform int       imageWidth;  // input image width
uniform int       imageHeight; // input image height
varying vec2      pos;         // pixel position of the fragment
uniform float     threshold;   // threshold value

void main(void)
{
 // find the texture coordinate corresponding to this fragment
 vec2 texCoord = vec2(pos.x/imageWidth, pos.y/imageHeight);

 // get the corresponding color
 vec4 color = texture2D(image, texCoord);

 // apply threshold
 color = step(threshold, color);

 // final color
 gl_FragColor = color;
}
```

この例から分かるように，このような操作はビルトイン関数によって行われる．同様にして，フラグメント Shader はコントラストや明るさ，またはその他の非線形のアプリケーションを変えることがで

きる．これ以上の基本技術の情報は[3]を参照のこと．

6.2. Lookup tables

ルックアップテーブルは画像処理に重要なツールである．まず，ユーザはいくつかのルックアップテーブルを定義する．以下の例では各チャンネルに 1 つで 3 つ定義している．このルックアップテーブルはフラグメント Shader に 1 次元のテクスチャとして転送される．

look up table

```
uniform sampler2D image;       // input image
uniform int       imageWidth;  // input image width
uniform int       imageHeight; // input image height
varying vec2      pos;         // pixel position of the fragment
uniform sampler1D lutR;        // lookup table for red component
uniform sampler1D lutG;        // lookup table for green component
uniform sampler1D lutB;        // lookup table for blue component

void main(void)
{
 // find the texture coordinate corresponding to this fragment
 vec2 texCoord = vec2(pos.x/imageWidth, pos.y/imageHeight);

 // get the corresponding color
 vec4 color = texture2D(image, texCoord);

 // apply look up tables
 color.r = texture1D(lutR,color.r).r;
 color.g = texture1D(lutG,color.g).g;
 color.b = texture1D(lutB,color.b).b;

 // final color
 gl_FragColor = color;
}
```

ここで，ルックアップテーブルを含んでいるテクスチャは GL_NEAREST オプションで初期化されなければならないことに注意する．また，ルックアップテーブルへアクセスすると隣接する 2 つの値から補間した結果を得る．

6.3. Convolution

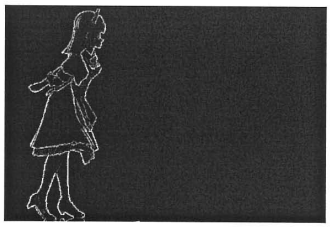
GPU で Convolution 操作が利用可能である．並列アプローチにはいくつかの最適化を使うことができない．それゆえ，標準的アプローチでは 2 つのステップに分割する．ステップ 1 ではテクスチャ画像上での新しいフラグメントの位置を見つける．ステップ 2 では，近傍ピクセルの値を読み込み，Convolution カーネルを適用する．

Convolution

```
uniform sampler2D inputImage;   // input image
uniform int       imageWidth;   // input video width
uniform int       imageHeight;  // input video height
varying vec2      pos;          // pixel position of the fragment

void main(void)
{
  // find the texture coordinates corresponding to this fragment
```

Convolution

```
  vec2 textureCoord = vec2(pos.x/float(imageWidth), pos.y/float(imageHeight));

  // find the corresponding color on the input texture
  vec3 color = vec3(0.0);

  // note : matrices are column major
  float gaussian[25] = {0.00366300, 0.01465201, 0.02564102, 0.01465201, 0.00366300,
                        0.01465201, 0.05860805, 0.09523809, 0.05860805, 0.01465201,
                        0.02564102, 0.09523809, 0.15018315, 0.09523809, 0.02564102,
                        0.01465201, 0.05860805, 0.09523809, 0.05860805, 0.01465201,
                        0.00366300, 0.01465201, 0.02564102, 0.01465201, 0.00366300};

  // constants to speed up the application
  float widthStep      = 1.0/float(imageWidth-1);
  float heightStep     = 1.0/float(imageHeight-1);
  float pixelPosWidth  = (pos.x-2.0)/float(imageWidth);
  float pixelPosHeight = (pos.y-2.0)/float(imageHeight);

  for(int i=0; i<5; i++)
    for(int j=0; j<5; j++)
      {
        // find the neighbourhood pixel color
        vec2 textureCoord = vec2(pixelPosWidth  + float(i)*widthStep,
                                 pixelPosHeight + float(j)*heightStep);
        vec3 colorTmp = texture2D(inputImage,textureCoord).rgb;

        // apply the kernel
        color += colorTmp*gaussian[i*5+j];
      }

  // set the final color of the fragment
  gl_FragColor = vec4(color, 1.0);
}
```

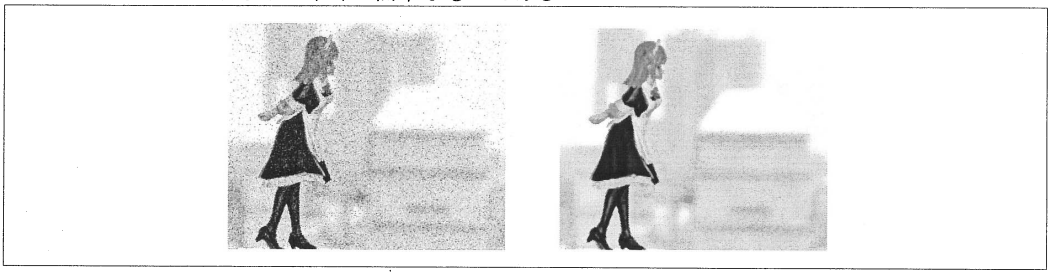


上記の例のために選ばれるアプローチはフラグメントプログラムに Convolution カーネルが直接定義する静的な手法である．当然このカーネルはメインアプリケーションにからも定義でき，uniform の float 配列としてフラグメントプログラムに送られる．

6.4. メディアンフィルタ

メディアンフィルタのようにより複雑な処理も GPU で実行できる．フラグメント Shader はまず，フ

ラグメント付近の画像のカラーを読み込んで配列に格納する．同じアプローチはConvolution処理でも行っている．そして，この配列はソートされる．命令数が規制されているため，ソーティングアルゴリズムはバブルソートなどの早くて簡単なものである．



6.5. Blending

画像処理において，ブレンディング関数は既に多く使われているため，多くのビルトイン関数が存在する．追加，差分，平均，クリア，暗くする，掛け算などの基本の決定論的なブレンディング関数は1行の最適化されたビルトイン関数で実装できる．より高度な非決定論的ブレンディング関数は簡単に定義できる．ユーザに定義すれば，いくつかの基準によって複数の画像がブレンドできるようになる．

6.6. 高速フーリエ変換

高速フーリエ変換(FFT)は GPU 上で実行することができるが，この方法は難しい．簡単な解決法はCUDA[8]を使うことだろう．CUDA は FFT と逆 FFT の実装を提案している．

6.7. ヒストグラム: very difficult!

大規模並列処理の限界がヒストグラムである．画像の色のヒストグラム計算のように非常に単純で連続した問題だとしても GPU で解決するのは非常に難しい．実際はヒストグラム計算ではすべてのピクセルを 1 度読み込んで配列にその色が発生した数を格納しなければならない．フラグメントプログラムの立場では，異なる2つのフラグメントを一般の配列に(すなわち同じピクセル上に)書き込むことは不可能である．したがって，これは Shader で並列化することができない．この問題を解決するマルチパス技術[18][19]が存在する．しかし，それらを実行するのは難しい．

7. Geometric operation

色操作の章では Shader プログラムを通してどのようにピクセルの色を修正するかを示した．本章では現在の位置から他の位置にピクセルをどのように動かすか説明する．

7.1. Zoom

ズームや切り出し処理をグラフィックカードで実行するのは非常に簡単である．実際にこれらの処理は Shader プログラムを明確に使うことなく実行でき，Shader を使っても差は生じない．GPU を使用する主な利点はこれまでのようにアプリケーションの実行速度を上げることである．ズームの方法ではアンチエイリアスの部分はよりいっそう速くするハードウェアで実行される．

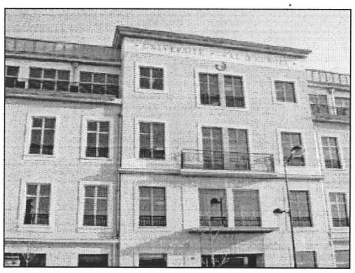
7.2. Homography

Homograph または一次変換は GPU がコンピュータビジョンに対してできることの良い例である．Homograph は完全に並列化でき，グラフィックカードはハードウェアでアンチエイリアス処理を行う．GPU で Homograph を実行するために 2 つの異なる方法がある．一つは uniform の float 配列に Homography を格納してすべてのフラグメントに送る方法．もう 1 つは，3x3 の Homography 行列を利用できるテクスチャ単位の 4x4 のテクスチャ行列に格納する方法である．この処理の間，第 1 行と第 2 行，第 1 列と第 2 列の順列をシステム座標の互換性のために作成する必要があるかもしれない．以下の例に示すように，この第 2 の状態では uniform の float 配列を行列に変換する必要はない．

Homography
`uniform sampler2D inputImage; // input image`

Homography

```
uniform int       imageWidth; // input video width
uniform int       imageHeight;// input video height
varying vec4      pos;        // pixel position of the fragment (homogenous coordinates)

void main()
{
  // apply the homography
  vec4 pos2 = gl_TextureMatrix[0]*pos;
  pos2 /= pos2.z; // homogenous coordinates

  // find the texture coordinates corresponding to this fragment
  vec2 textureCoord = vec2(float(pos2.x)/float(imageWidth),
                           float(pos2.y)/float(imageHeight));

  // find the corresponding color on the input texture
  gl_FragColor = texture2D(inputImage,textureCoord);
}
```



Homography の計算をしない手法は長方形でない四角形にテクスチャを描くことである．実際，この四角形は 2 つの三角に分割される．したがって，この場合，透視変換をせず，4 点でなく 3 点の補間を用いてテクスチャを描く．

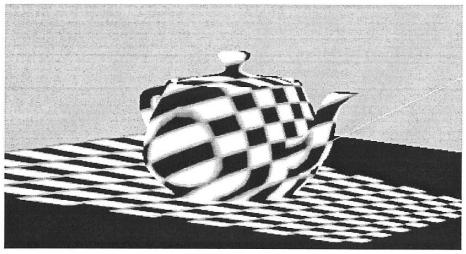
7.3. Projected texture

3 次元再構築はコンピュータビジョンの分野では必須のアプリケーションである．3 次元データは通常，三角形によって定義された 3 次元形状に変換される．一般的なアプローチではこれらの三角にテクスチャを投影して描く．この処理は投影されたテクスチャマッピングを使用する GPU によって簡単に実行できる．カメラの画像テクスチャが結び付けられているとき，カメラの射影行列は対応するテクスチャ行列に読み込まれなければならない．そして，3 角形を描いている間，以下のフラグメントプログラムを呼ばなければならい．

Projected texture

```
uniform sampler2D inputImage; // input image
varying vec4 pos;             // pixel position of the fragment

void main(void)
{
  // set the final color of the fragment
  gl_FragColor = texture2DProj(inputImage,gl_TextureMatrix[0]*pos);
}
```

Projected texture



この手法は，いくつかのテクスチャをバインドして 6.5 章で細述したブレンディング関数を使うことで表示に依存するテクスチャマッピングの扱いを変化させることが簡単である．

8. Computer vision tools

本章ではコンピュータビジョンのためにより特定のツールを紹介する．

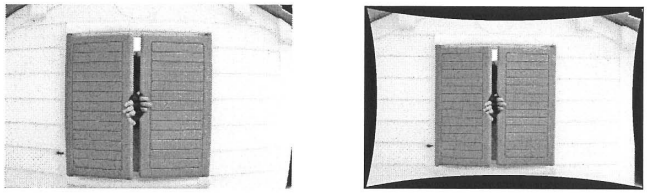
8.1. レンズ歪補正

レンズ歪補正はコンピュータビジョンの処理において必要な処理である．この操作は完全に並列化することができるため，GPU で処理できる．また，GPU は早いアンチエイリアシング関数を提供する．ユーザは最初にひずみパラメタを計算し，それを uniform 変数としてフラグメントプログラムへ送る必要がある．以下の例は OpenCV のパラメタを使用してラジアルディストーションを取り除く処理である．

Radial distortion correction

```
uniform sampler2D inputImage;   // input image
uniform int       imageWidth;   // input video width
uniform int       imageHeight;  // input video height
uniform float     k[2];         // radial distorsion parameters (openCV style)
uniform float     C[2];         // radial distorsion center (openCV style)
varying vec2      pos;          // pixel position of the fragment

void main(void)
{
  // find the texture coordinates corresponding to this fragment
  vec2 textureCoord = vec2(pos.x/float(imageWidth), pos.y/float(imageHeight));

  // correct radial distorsion (openCV style)
  float du = pos.x - C[1];
  float dv = pos.y - C[0];
  float r2 = du*du + dv*dv;

  float u2 = pos.x + du*(r2*(k[0] + r2*k[1]));
  float v2 = pos.y + dv*(r2*(k[0] + r2*k[1]));

  textureCoord.x = u2 / imageWidth;
  textureCoord.y = 1.0- (v2 / imageHeight);

  // find the corresponding color on the input texture
  gl_FragColor = texture2D(inputImage,textureCoord);
}
```

Radial distortion correction



2.2.13 で説明したフレームバッファオブジェクト(FBO)を使ってオフスクリーンで処理するときに役に立つ.

8.2. 背景差分とモザイキング

背景差分のアルゴリズムは背景画像と与えられた画像の非決定論的ブレンディング関数とみなすことができる. これは基本的な背景差分プログラムは 6.5 で示した手法を用いて実行できる. より正確な方法のために参考文献[20]を参照のこと.

モザイキングはブレンディングと幾何変換を含んでいる. モザイキングの場合, 幾何変換は 7.2 章で記述してあるように実行する簡単な Homography である. それゆえ, 重なっている画像間でブレンディング関数は画像上のフラグ面の位置によってバランスを取らなければならない.

8.3. トラッキング

ほとんどのトラッキング手法は速度のパフォーマンス向上のために GPU を利用することができる. 最初にエッジと角検出器が Convolution 操作やその他のツールを用いて実行される. それから, 抽出された特徴は異なる手法を用いてトラッキングされる.

[21]では GPU 上での SIFT/KLT の実相を提案している. オプティカルフロー技術は GPU によって処理されるのに適している.

注意する点は GPU によるトラッキングの問題は時々トラッキング情報がグラフィックカードからメインメモリに送られることである. 必要な場合, 追跡したフラグメントのパラメタのトラッキング情報を"書く"ことが含まれる. このテクスチャの結果は完全に CPU によって情報を抽出しなければならない.

8.4. ビデオベースレンダリングと奥行きマップ

ビデオベースレンダリングの手法は新しいビューポイントからシーンの画像を作成しようとする. 既存の方法のほとんどは時間がかかり, リアルタイムレンダリングに達することができない. その他の方法はリアルタイムに新しいビューを生成することができるが, 前処理段階が必要となる. これらの手法はオフラインと呼ばれる. そして, オンライン手法では実況ででもリアルタイムに新しいビューが生成できる. このあとの集団はリアルタイムレンダリングに達するために広く GPU を使っている. 最も一般的な手法はおそらく, 3 次元モデルを生成するためにオブジェクトのシルエットを抽出する Visual Hull の手法である. 多くの実装は[22]で記述される. 他の手法では[23]のように背景を取り扱うことができる Plane-sweep アプローチを用いている(図 7). この手法は GPU を大規模に使用する. 当然これらのビデオベースのレンダリング手法はリアルタイムの奥行きマップを提供することができる(図 8).

9. Virtual Reality

バーチャルリアリティのツールは CG のビデオとカメラからのビデオの両方から複数のグラフィックアプリケーションが関わる. 最も重要なものはおそらく, ステレオ画像の計算だろう. 当然, 以下のコードで解説されるようにShaderプログラムはブレンディング関数を用いて立体写真(赤/シアン-画像)の計算に用いられる.

```
uniform sampler2D image1;      // input image1
uniform sampler2D image2;      // input image2
uniform int       imageWidth;  // input image width
uniform int       imageHeight; // input image height
varying vec2      pos;         // pixel position of the fragment

void main(void)
{
  // find the texture coordinates corresponding to this fragment
  vec2 textureCoord = vec2(pos.x/float(imageWidth), pos.y/float(imageHeight));

  // get the 2 color to blend
  vec4 color1 = texture2D(inputImage1,textureCoord);
  vec4 color2 = texture2D(inputImage2,textureCoord);

  // set the final color of the fragment
  gl_FragColor = vec4(color1.r, color2.g, color2.b, 1.0);
}
```

自動ステレオディスプレイなどのより精巧なデバイスのためにGPUは最終的な画像の計算にも用いられる．それは[24]で示すようにレンダリングのスピードアップに用いられることもできる．この手法は幾何Shaderを用いて1つのパスでいくつかのステレオ画像を描くことができる(図9)．主な利点は各ビューにおける頂点操作を共有できるところである．

最後に，自動立体視ステレオスクリーンにはリアルタイムレンダリングするためにコンピュータに取

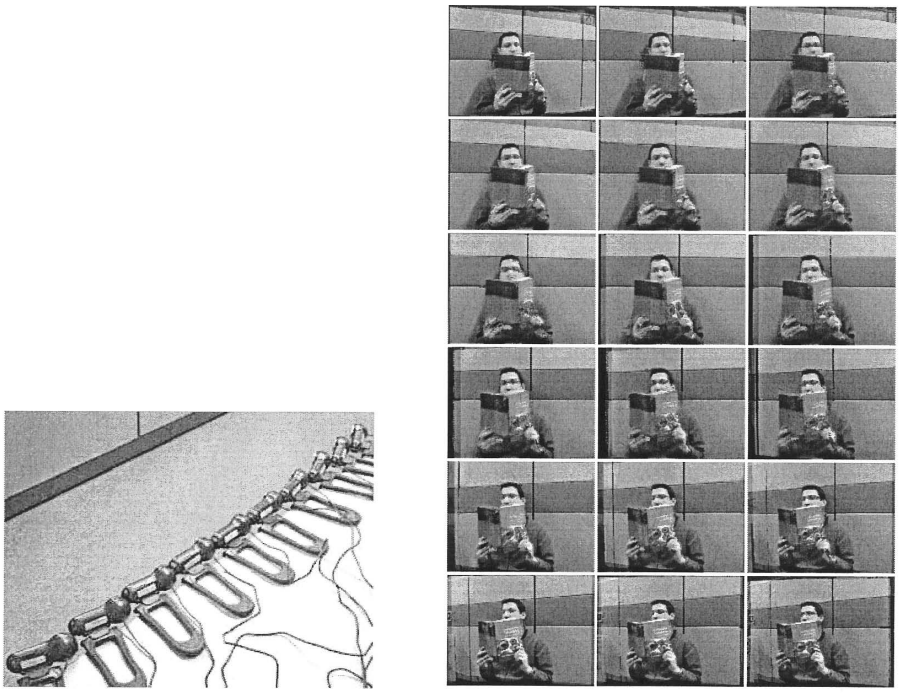


図7

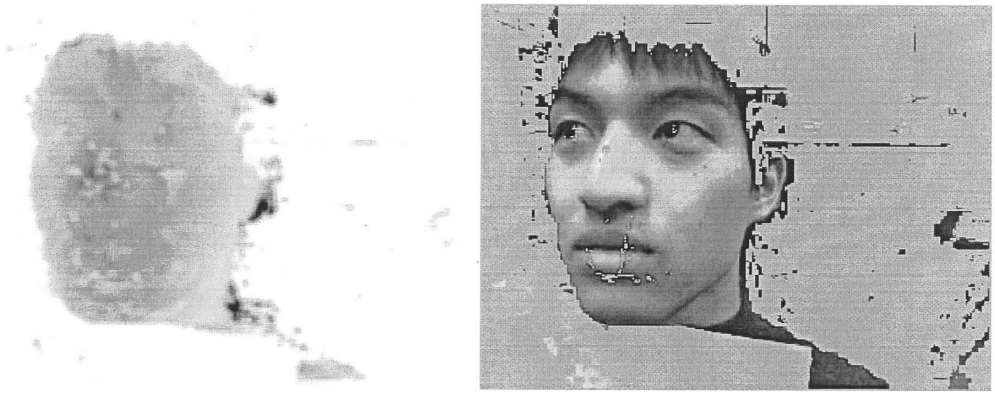


図 8

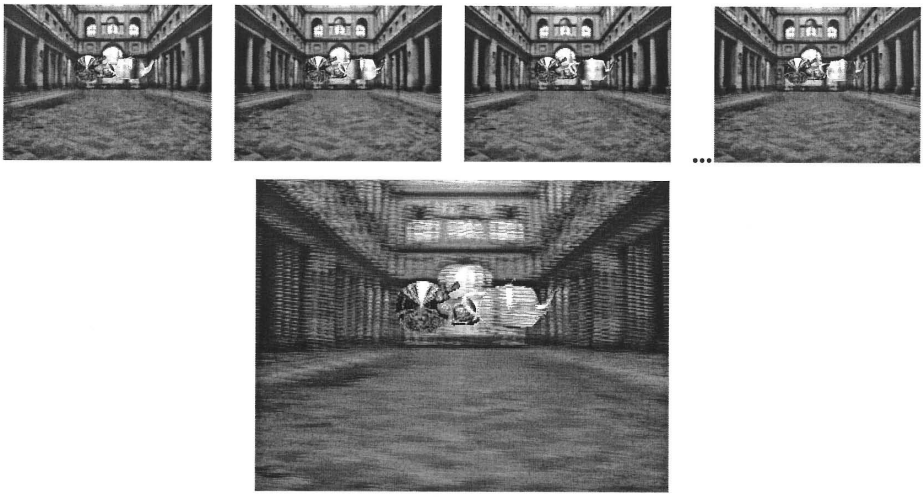


図 9

り付けられるカメラの数に制限がある．あまり多くのカメラを PC につないだ場合，ビデオのストリームが飽和してしまう．この問題は[25]で示されるように入力カメラの間を補間した画像によってビデオベースのレンダリング手法を用いることで解決する(図 10).

10. General Purpose GPU (GPGPU)

前章では画像処理とコンピュータビジョンのためのいくつかの GPU アプリケーションの問題を示した．ここでは GPU を利用することができる他の領域について記述する．

実際，GPU は画像に関係のないアプリケーションのために利用することができる．たとえば，[26]で記述されるように GPU はデータベースシステムに利用できる．[27]で示されるように GPU はデータベースにおけるデータのソートに利用できる．

GPU は各層がテクスチャとして定義されるようなニューラルネットワークに対しても利用できる．いくつかの実行は[28]で行われている．

GPU 上での物理操作は商業テレビゲームに広く使われている．特に，流体シミュレーション[29]や衝突検知[30]に利用されている．

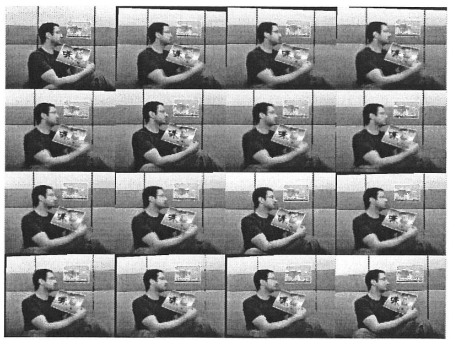


図 10

そして，GPU は小規模な逆行列や，LU 分解[31]，その他の行列演算[32]のような数学的な目的に利用できる．

11. How to start

11.1. インストール

現在の主要な問題は始め方である．最初に Shader を処理できる正しいグラフィックカードを持っているか確認しなければならない．そして，ドライバもチェックしなければならない．これはとても重要である．

新しい技術を学ぶのに時間をかけたくない場合は CUDA や GPUCV，OpenVidia を使うことができる．これらのツールは少ない努力でアプリケーションの実行速度を上げることができる．GLSL を使った Shader を試してみたい場合は以下の URL からサンプルコードをダウンロードすることができる．

http://www-igm.univ-mlv.fr/~vnozick

このサンプルコードは主に本論文で提示される課題と一致している．これらのコードは Windows，Linux の両方で使用することができる．

11.2. Shader design

自分で Shader を使ったアプリケーションを実装したい場合，テクスチャの転送に時間がかかることを心に留めておかなければならない．それによってアプリケーションの設計を変更することになるかもしれない．たとえば，アプリケーションがたくさんの画像について異なる処理をする場合，すべての画像に同じ処理を行った後，次のプロセスを続ける代わりにすべての処理を一つの画像に行う方がよい．

さらに GPU は大規模並列マシンなので，可能ならばあなたのアルゴリズムは並列処理と互換性を持つようにする方法を見つけなければならない．

基本的な画像処理操作から始めることがよいだろう．うまく行きそうなら，マルチパスを実行してみることができる．ラジアルディストーション補正の場合であるが，ラジアルディストーションはフレームバッファオブジェクトを使う最初のパスで修正され，結果はテクスチャに保存される．次のパスで，このテクスチャはあなたのアプリケーションで処理されることができる．

12. まとめ

本稿では，GPU を用いた CG のレンダリングパイプラインと Shader について，Shader の読み込みとコンパイルに関する部分も含めて概説した．また，画像処理とコンピュータビジョンに関するいくつかの実用的なアプリケーションを示した．さらに，技術的，理論的なアドバイスとともに Shader の導入方法を提供した．

参考文献

[1] OpenGL(R) Programming Guide: The Official Guide to Learning OpenGL(R), Version 2 (5th Edition) by OpenGL Architecture Review Board (Author), Dave Shreiner (Author), Mason Woo (Author), Jackie Neider (Author), Tom Davis (Author), Addison-Wesley Professional; 5 edition (August 11, 2005), ISBN-10: 0321335732

[2] Games for Windows, http://www.gamesforwindows.com/en-US/AboutGFW/Pages/DirectX10.aspx (last access april 2008)

[3]OpenGL(R) Shading Language (2nd Edition) (OpenGL), by Randi J. Rost (Author), Publisher: Addison-Wesley Professional; 2 edition (February 4, 2006), ISBN-10: 0321334892

[4] The complete effect and HLSL guide, by Sebastien St-Laurent (Author), Publisher: Paradoxal Press; 1 edition (Jully 1,2005), ISBN-10: 0976613212

[5] Geometry shader,http://developer.download.nvidia.com/opengl/specs/GL_EXT_geometry_shader4.txt (last access april 2008)

[6] GeForce 9800 GX2 Review, http://www.nvnews.net/reviews/geforce_9800_gx2/ (last access april 2008)

[7] The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics, by Randima Fernando (Author), Mark J. Kilgard (Author); Addison-Wesley Professional; 1 edition (March 1, 2008), ISBN-10: 0321194969

[8] CUDA, http://www.nvidia.co.uk/object/cuda_learn_uk.html (last access april 2008)

[9] Sh, http://libsh.org/ (last access april 2008)

[10] openVidia, http://openvidia.sourceforge.net/ (last access april 2008)

[11] OpenCV, http://www.intel.com/technology/computing/opencv/overview.htm

[12] J.P. Farrugia, P. Horain, E. Guehenneux and Y. Alusse, "GPUCV: A framework for image processing acceleration", International Conference on Multimedia and Expo, Juillet 2006, Toronto.

[13] M. Strengert, T. Klein and T. Ertl, "A Hardware-Aware Debugger for the OpenGL Shading Language", ACM Siggraph/Eurographics Workshop on Graphics Hardware 2007 (GH'07), San Diego, California, USA, pp.81-88, 2007.

[14] glslDevil, http://www.vis.uni-stuttgart.de/glsldevil/ (last access april 2008)

[15] gDEBugger, http://www.gremedy.com/products.php (last access april 2008)

[16] Kenneth Moreland and Edward Angel, "The FFT on a GPU", Graphics Hardware (2003) M. Doggett, W. Heidrich, W. Mark, A. Schilling (Editors)

[17] Ondˇrej Fialka and Martin Cˇ adík, FFT and Convolution Performance in Image Filtering on GPU, Proceedings of the Information Visualization (IV'06), 0-7695-2602-0/06 $20.00 © 2006 IEEE

[18] Oliver Fluck, Shmuel Aharon, Daniel Cremers, Mikael Rousson, GPU histogram computation, International Conference on Computer Graphics and Interactive Techniques, ACM SIGGRAPH 2006 Research posters, Boston, Massachusetts, SESSION: Research posters: GPU techniques, Year of Publication: 2006, ISBN:1-59593-364-6

[19] Thorsten Scheuermann and Justin Hensley, Efficient Histogram Generation Using Scattering on GPUs, Symposium on Interactive 3D Graphics,Proceedings of the 2007 symposium on Interactive 3D graphics and games, Seattle, Washington, Pages: 33 - 37, Year of Publication: 2007, ISBN:978-1-59593-628-8

[20] Andreas Griesser, Stefaan De Roeck, Alexander Neubeck and Luc Van Gool. GPU-Based Foreground-Background Segmentation using an Extended Colinearity Criterion. Vision, Modeling, and Visualization (VMV), 2005

[21] Sudipta N Sinha, Jan-Michael Frahm, Marc Pollefeys and Yakup Genc, "GPU-Based Video Feature Tracking and Matching", EDGE 2006, workshop on Edge Computing Using New Commodity Architectures, Chapel Hill, May 2006.

[22] Marcus A. Magnor, Video-Based Rendering, Éditor : A K Peters Ltd, ISBN : 1568812442, 2005.

[23] Vincent Nozick and Hideo Saito, Real-Time Video-Based Rendering from Multiple Cameras, ACCV'07 Workshop on Multi-dimensional and Multi-view Image Processing, pages 17-23, Tokyo, Japan, November 19, 2007.

[24] François de Sorbier, Vincent Nozick and Venceslas Biri, Accelerated Stereoscopic Rendering using GPU, in proc. of The 16-th International Conference in Central Europe on Computer Graphics, Visualization and Computer Vision (WSCG'2008), Plzen, Czech Republic, February 4-7, 2008.

[25] Vincent Nozick and Hideo Saito, Multiple View Computation for Multi-Stereoscopic Display, in proc. of 2007 IEEE Pacific-Rim Symposium on image amd video Technology (PSIVT 2007), Vol. 4872, pages 399-412, Santiago, Chile, December 17-19, 2007.

[26] Naga Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin and Dinesh Manocha, Fast Computation of Database Operations using Graphics Processors, in Proc. of ACM SIGMOD 2004 (international conference on Management of data), Pages: 215 - 226 , Year of Publication: 2004, ISBN:1-58113-859-8 , Paris, France

[27] Alexander Greß and Gabriel Zachmann, GPU-ABiSort: Optimal Parallel Sorting on Stream Architectures, Proc. 20th IEEE International Parallel and Distributed Processing Symposium (IPDPS), Rhodes Island, Greece, 25-29 April 2006.

[28] Bernhard, F. and Keriven, R. Spiking neurons on GPUs. International Conference on Computational Science. Workshop General purpose computation on graphics hardware (GPGPU): Methods, algorithms and applications, Readings, UK, May 2006.

[29] Keenan Crane, Ignacio Llamas anf Sarah Tariq, Real Time Simulation and Rendering of 3D Fluids, GPU Gems 3, Addison-Wesley Professional; 1 Har/Cdr edition (August 12, 2007), pages 633-676, ISBN-10: 0321515269

[30] Scott Le Grand, Broad-Phase Collision Detection with CUDA, GPU Gems 3, Addison-Wesley Professional; 1 Har/Cdr edition (August 12, 2007), pages 697-722, ISBN-10: 0321515269

[31] Nico Galoppo, Naga K. Govindaraju, Michael Henson and Dinesh Manocha, LU-GPU: Efficient Algorithms for Solving Dense Linear Systems on Graphics Hardware, In Proceedings of the ACM/IEEE SC|05 Conference. page 3, November 12-18, 2005.

[32] Kayvon Fatahalian, Jeremy Sugerman andPat Hanrahan, Understanding the Efficiency of GPU Algorithms for Matrix-Matrix Multiplication, SIGGRAPH/EUROGRAPHICS Conference On Graphics Hardware, Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware table of contents, Grenoble, France, Pages: 133 - 137, Year of Publication: 2004, ISBN ~ ISSN:1727-3471 , 3-905673-15-0