GPU を用いたビデオ映像の安定化

天谷貴大[†] 藤澤 誠[††] 三浦 憲二郎[†††]

この論文では，ビデオ映像に含まれる振動成分を取り除くための処理の計算を GPU を用いて行う手法を提案する．映像の安定化処理には，グローバルモーションの推定，振動補正，モザイキングの３つの処理を行うが，CPU でこれらの処理を行うと処理時間が長く，その中でもグローバルモーションの推定が処理の大半を占めている．そこで，並列処理が可能な GPU で計算処理を行うことで処理時間の短縮を計った．提案した手法は，ビデオ映像のフレーム画像をテクスチャデータとして GPU に転送し計算を行い，計算結果をオフスクリーンバッファに描画し，ピクセルの値を読み込むことによって結果を得る．ピクセルを読み込む速度は描画速度に比べて時間がかかるため，計算結果を１つのピクセルにまとめることで，読み込み時間を短縮することに成功した．

Video Stabilization with GPU

Takahiro Amaya,[†] Makoto Fujisawa[††] and Kenjiro T. Miura[†††]

This paper proposes a fast computational method of video stabilization using the Graphics Processing Unit (GPU) that removes the unwanted vibrations from videos. The video stabilization is composed by estimation of the global motion, removal of the undesired motion and mosaicking. When these are processed with CPU, the computational cost for the global motion estimation is very high. We improve the speed of this computation with GPU that the parallel processing is possible. Our method can obtain the result by forwarding the frame image of the video to GPU as texture data, and drawing the calculation result to the off-screen buffer. Although the transfer speed from GPU to CPU is very slower than the other way around, the method only has to transfer one pixel data from GPU.

1. 緒 言

災害時において活躍するレスキューロボットは荒れた路面や地震により不安定な状況で走行する．そのため，ロボットに搭載されたカメラから送られてくる映像にはゆれが生じ，即座の状況把握が困難になったり，オペレータが画面酔いを起こして操作に影響が出る可能性がある．したがって，映像のゆれによる影響を抑えるために，リアルタイムでの動画像処理を行い，ゆれを軽減する必要がある．

現在，デジタルカメラで開発，研究されているゆれを軽減する手法として，電子式，光学式手ぶれ補正などがあげられる．しかし，これらはカメラに対する補正であり，そのカメラで撮影した映像だけしか補正できず，またすべてのカメラにこれらの手法を搭載できるとは限らない．そのため，どのような映像でも処理できるようにするには PC を利用した安定化処理が望まれる．しかし，動画像処理はデータ量が多く，それらを処理するには CPU では負荷がかかりすぎリアルタイムでの処理は難しい．そこで，処理時間を短縮するために，並列処理による高速演算が可能な GPU(Graphics Processing Unit) に CPU で負荷が多くかかる計算処理を行わせる．GPU は，元々グラフィックス処理専用のプロセッサだったが，近年ではプログラマブルシェーダの搭載により，Cg 言語[3] などを使用して，これまで CPU で行ってきた汎用計算が GPU でも可能になった．GPU を計算に使用している研究例として，流体などのシミュレーション[4] や画像処理[7]，形状処理[9] 等があげられる．この論文では安定化処理を，専用のハードウェアではなく汎用のハードウェアを用いて行うことで，一般の PC でも簡単に処理を行えるようにし，低コストかつリアルタイムでの映像安定化を実現することを目的とした，GPU を用いた映像の安定化手法を提案する．

映像の安定化には，Litvin らの手法[5] や Matsushita らの手法[6] があり，どちらもカメラの動き (グローバルモーション) を推定し，それを基に振動補正を行う．Litvin らは，グローバルモーションにカルマンフィルタリングを行うことによってゆれを抑えた動きを求め，フレーム画像を変形させる．さらに変形による画像の劣化をモザイキングを行うことで補間する．しかし，モザイキングでは映像の中の物体の動き (ローカルモーション) を補間しきれない．そこで，Matsushita らはローカルモーションを推定し[2]，それを Motion inpainting によって補間することで，より良画質な映像を生成する手法を提案した．また，彼らは，階層的な運動推定[1] を行うことで，グローバルモーション

† 静岡大学大学院工学研究科
Graduate School of Engineering, Shizuoka University
†† 静岡大学大学院理工学研究科
Graduate School of Science and Engineering, Shizuoka University
††† 静岡大学創造科学技術大学院
Graduate School of Science and Technology, Shizuoka University

の推定時間を減少させ，さらにガウスカーネルを用いることで映像の不必要なぶれを取り除くことで振動補正を行った．しかし，ローカルモーションの推定には多くの時間を費やしてしまうためリアルタイム処理が難しくなる．我々は，ガウスカーネルを用いて振動補正を行い，モザイキングによって補間を行う．

この論文の構成は以下である．第 2 章で安定化のアルゴリズム，グローバルモーションの推定，振動補正，モザイキングについて説明し，第 3 章ではグローバルモーションの推定における計算を GPU で行う手法を示す．そして，第 4 章で本手法の結果と CPU との計算速度の比較結果を示し，第 5 章において結言を述べる．

2. 映像の安定化

2.1 グローバルモーションの推定

映像の安定化を行うにはグローバルモーションを知る必要がある．グローバルモーションは隣接するフレーム間での動きを求めることによって推定される．フレーム I^n から I^{n+1} までのピクセル座標 $\mathbf{x}=(x,y)$ の変化は

$$\mathbf{x}_{n+1} = \begin{pmatrix} a_1 & a_2 \\ a_3 & a_4 \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \mathbf{A}_n^{n+1}\mathbf{x}_n + \mathbf{b}_n^{n+1}$$

で表すことができる．アフィン変換 $(\mathbf{A}_n^{n+1}, \mathbf{b}_n^{n+1})$ は

$$E(n,n+1) = \sum_{\mathbf{x}\in\chi} \varphi(I^n(\mathbf{x}_n) - I^{n+1}(\mathbf{A}_n^{n+1}\mathbf{x}_n + \mathbf{b}_n^{n+1})) \tag{1}$$

$$\varphi(x) = \sqrt{x^2+\beta},\ \beta = 0.01$$

の最小値を求めることによって得られる．ここで,χ は画面平面上全ての座標値の集合である．フレーム I^{n+1} を変形させたときの座標 (x', y') のピクセルとフレーム I^n の座標 (x,y) のピクセルが対応している場合，その輝度値の差は 0 になる．輝度値の差の合計をエラー値とし，その最小化によりアフィン変換 $(\mathbf{A}, \mathbf{b})$ を算出する (図 1 参照)．ただし，場面が急激に変わるフレーム間や画面内を動的な物体が多く占める場合には対応点がとれずにグローバルモーション推定に失敗する可能性があることに注意しなければならない． 本論文では，高速化のために映像の動きを平行移動と回転移動のみと仮定してアフィン変換を

$$\mathbf{x}_{n+1} = \begin{pmatrix} cos\theta & -sin\theta \\ sin\theta & cos\theta \end{pmatrix} \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix}$$

とし，求めるパラメータを (θ, b_1, b_2) の 3 つにした．最小値の探索には関数値のみで実装できる Powell 法と勾配値 (導関数) を用いて検索する準ニュートン法 (BGFS 法) を使用する[8]．BGFS 法を使用する際には導関数を求める必要があり，その値は以下の式で求めることができる．

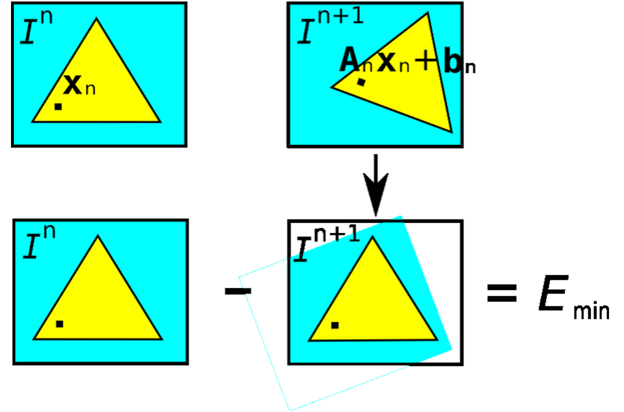


図 1 グローバルモーションの推定

$$\frac{\partial E}{\partial \theta} = -\sum_{\mathbf{x}\in\chi} \sqrt{\frac{\Delta I^2}{\Delta I^2+\beta}} \left(\frac{\partial I^{n+1}}{\partial x_{n+1}} \frac{\partial x_{n+1}}{\partial \theta} + \frac{\partial I^{n+1}}{\partial y_{n+1}} \frac{\partial y_{n+1}}{\partial \theta} \right) x,$$

$$\frac{\partial E}{\partial b_1} = -\sum_{\mathbf{x}\in\chi} \sqrt{\frac{\Delta I^2}{\Delta I^2+\beta}} \frac{\partial I^{n+1}}{\partial x_{n+1}},$$

$$\frac{\partial E}{\partial b_2} = -\sum_{\mathbf{x}\in\chi} \sqrt{\frac{\Delta I^2}{\Delta I^2+\beta}} \frac{\partial I^{n+1}}{\partial y_{n+1}}.$$

ここで,

$$\Delta I = I^n(\mathbf{x}_n) - I^{n+1}(\mathbf{x}_{n+1}),$$
$$x_{n+1} = x\cos\theta - y\sin\theta + b_1,$$
$$y_{n+1} = x\sin\theta + y\cos\theta + b_2,$$
$$\frac{\partial x_{n+1}}{\partial \theta} = -x\sin\theta - y\cos\theta,$$
$$\frac{\partial y_{n+1}}{\partial \theta} = x\cos\theta - y\sin\theta.$$

2.2 振動補正

推定したグローバルモーションをもとに，Matsushita らの方法[6] を用いて振動を補正する．補正するフレームの前後 k フレームを利用して補正変換 S_n を

$$S_n = \sum_{m=n-k}^{n+k} T_n^m \star G(k)$$

によって求める．ここで，T_n^m はフレーム n から m までのアフィン変換，G はガウスカーネル，そして $\star$ は畳み込み演算子である．得られたアフィン行列を用いて振動補正を行う．

$$\bar{\mathbf{x}}_n = S_n\mathbf{x}_n = \bar{\mathbf{A}}_n\mathbf{x}_n + \bar{\mathbf{b}}_n$$

図 2 に振動補正を行う前後の X 軸方向と Y 軸方向のカメラの動きの変位量を示す．灰色のグラフが補正前の X 軸方向，Y 軸方向のカメラの動きを示しており，振動により上下左右に細かに動いている．黒のグラフが補正後のカメラの動きを示しており．ユーザの意図したカメラの動きを残したまま全体の動きが滑らかなものになっていることが確認できる．また，図 3 に振動補正した際のフレーム画像を

示す．フレーム画像を変形させるため，画像にはピクセルの未定義領域 (黒く塗りつぶした領域) が発生する．

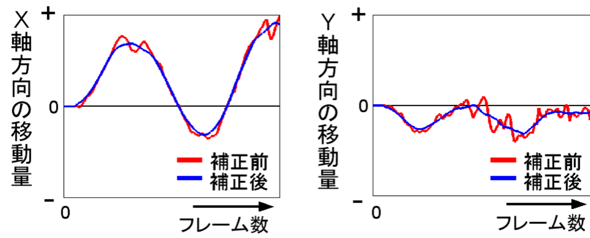


図 2　カメラの移動量の推移

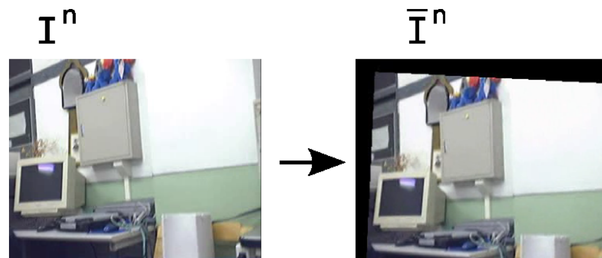


図 3　振動補正後のフレーム画像．黒い部分は未定義領域

2.3　モザイキング

振動補正したフレームに発生したピクセルの未定義領域は，Litvin らのモザイキングを用いた手法[5] で補間する．周囲のフレーム $\bar{I}^{n+m}$ を補間の対象となるフレーム $\bar{I}^{n}$ の位置に変形 ($\bar{I}^{n+m} \rightarrow \bar{\bar{I}}^{n+m}$) させ，式 (2) を用いて補正をかけることによって未定義領域のピクセルを補間することができる．ここで E はフレーム n と $n+m$ のエラー値である．図 4 にモザイキング結果を示す．丸で囲った未定義領域が補間された．

$$\bar{I}^{n} = \frac{1}{\sum E(n,m)} \sum_{-M \leq m \leq M, m \neq 0} E(n,m)\bar{\bar{I}}^{n+m} \quad (2)$$

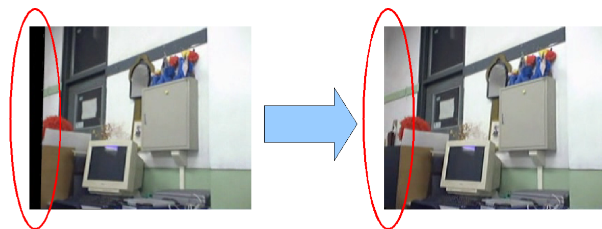


図 4　モザイキング結果

3.　GPU での実装

2 章で提示した安定化処理を CPU 上で行うと非常に時間がかかる．特にグローバルモーションの推定の処理時間が長く，１フレームの推定に数秒要する．これは式 (1) やその導関数の計算において輝度値の差を 1 ピクセルずつ計算し，最小値探索のため何度もその計算を繰り返しているためである．そこで，計算速度を向上させるために GPU を用いてこれらの計算を並列処理する．

3.1　GPU 上の計算

GPU は CPU から送られた各種データを用いて計算を行う．しかし，フレーム画像は GPU に直接渡すことはできないので，テクスチャデータに変換してから渡す．画像の差分はテクスチャ画像をアフィン変換して求める．GPU 側では，テクスチャデータの参照にテクスチャ座標を使っているため，アフィン変換を行う際には，テクスチャ座標 (u, v) を次のようにピクセル座標 (x, y) に変換する必要がある．

$$x = u \times \mathrm{FrameWidth},$$
$$y = v \times \mathrm{FrameHeight}.$$

GPU は本来描画専用なので CPU に値を直接返すことができない．そのため，計算結果をピクセルのカラー値に格納し，描画された画面からピクセルの値を読込むことによって値を取得する．また，計算結果はディスプレイに直接描画せずに，オフスクリーンレンダリングによって pixel buffer(以下 pbuffer) に描画し，そこから値を取得する．しかし，ピクセルの値を読込む処理は CPU から GPU にデータを送る場合と比べて時間がかかるため，計算結果の合計を 1 つのピクセルにまとめることで読込み時間を短縮する．

3.2　計算結果の取得

差分画像が得られたら，次に読込み時間の短縮のに全てのピクセルの値をまとめて 1 つのピクセルに格納する．pbuffer に出力した画像は，そのままテクスチャとして使用可能なのでピクセルの値をまとめるためにまず，計算結果の画像を

$$\mathrm{FrameWidth} <= 2^n$$
$$\mathrm{FrameHeight} <= 2^n$$

となる $2^n \times 2^n$(n は最小の整数) を描画範囲とした pbuffer に出力する．このとき，pbuffer の背景色は後の計算に影響がないように黒 (RGBA=(0.0, 0.0, 0.0, 0.0) にしておく．そして，それを同じサイズの pbuffer に貼り付ける．このとき描画する画像のピクセル座標 (x, y) の値は，テクスチャ画像のピクセル座標

$$(2x, 2y) \qquad (2x+1, 2y)$$
$$(2x, 2y+1) \quad (2x+1, 2y+1)$$

の値を図 5 のように合計することによって $2^{n-1} \times 2^{n-1}$ の範囲に，ピクセルの値をまとめた画像を出力する．さらにこの画像を別の pbuffer に貼り付けて同じ作業を n 回繰り返すことで，画面隅の $2^0 \times 2^0$ つまり 1×1 の範囲に差分値を合計した値を持つピクセルが描画される (図 6)．そのピクセルを CPU に読込むことによって，計算結果の合計を得ることができる．

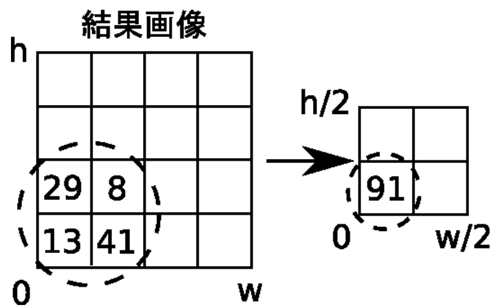


図 5　値の合計

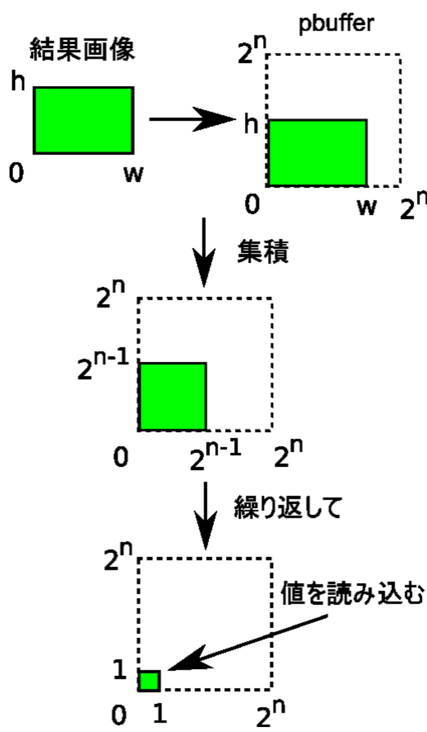


図 6　ピクセルの集積

各カラー値 RGBA が出力できる値はそれぞれ 8bit(0〜255) であり，それ以上の値は出力できない．我々は，各カラー値のビット数を図 7 のように繋げるように扱い 32bit までの値を扱えるようにした．　また，導関数の計算のように，値のとりうる範囲が広い場合や，符号付の計算の場合には pbuffer を複数枚使用して計算を行う．例として pbuffer を 2 枚使用する場合，各 pbuffer で同じ計算を行い片方の pbuffer には 32bit までの計算結果をいれてもう片方の pbuffer には 32bit の値を超えた 56bit までの値を rgba の内 3 つを用いて格納し，余ったカラー値に符号を表

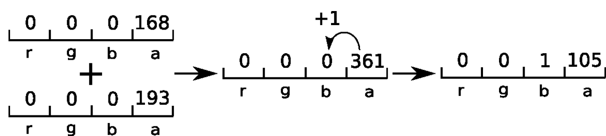


図 7　カラー値の扱い

す値をいれる (＋なら 0，－なら 1)．そして，ピクセルの値をまとめる際には，2 枚の pbuffer に描画した画像をそれぞれ読込んで値を 10 進数に戻し，値を合計してから再び 2 枚の pbuffer に結果を分割させ，最終的に 2 つのピクセルを読込むことで結果を得ることができる．

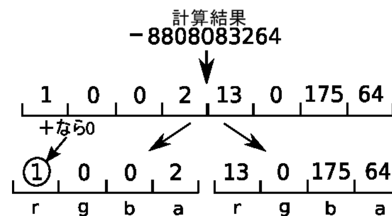


図 8　計算結果の分割

この論文では，1 チャンネル 8bit のテクスチャを使用しているが本来なら 1 チャンネル 32bit のテクスチャを使用したかった．しかし，float 型の pbuffer を使用して差分画像の出力を行ったところ，ピクセルの値自体は読込むことができたが，結果画像が出力されず，ピクセルの値の結合が不可能だった．そのため，1 チャンネル 8bit のテクスチャを代わりに用いた．

4. 結　　果

CPU と GPU のグローバルモーション推定時間の比較結果を表 1 に，各最小値検索法において値が収束までの平均反復回数を表 2 に，そして安定化結果を図 9 に示す．図 9 の左の列は補正前のフレーム画像を示し，中心の列は振動補正を行った後のフレーム画像を示す．そして，右の列は補正後のフレーム画像にモザイキングを行った結果を示す。また，推定開始直後はメモリ確保処理により CPU に負担がかかるため正確な時間を計れないので，開始から数フレームの結果は使用しないものとする．

使用した映像は，サイズが 320×240 で，フレームレートは 30fps(frame per second) の 6 秒の動画である．．また，計算に用いた PC 環境は CPU：Pentium D 3.4GHz，メモリ：2048Mbyte，GPU：GeForce 8800 GTX である．

安定化を行ったビデオ映像には，フレームが不自然に飛んだり，歪んでいるものが存在している．これらは，グローバルモーション推定に失敗しているである．また，推定時間にはバラつきがみられ，例えば，GPU を用いた BGFS 法の計算では，1 秒以内で終わるフレームもあれば，10 秒以上かかるフレームも存在した．

表 1　推定時間の比較結果

	CPU	GPU
Powell 法	40.99	1.93
BGFS 法	7.81	2.27

(sec/frame)

表 2　最小値探索の平均反復回数

	CPU	GPU
Powell 法	5.74	6.51
BGFS 法	11.43	42.87

(回/frame)

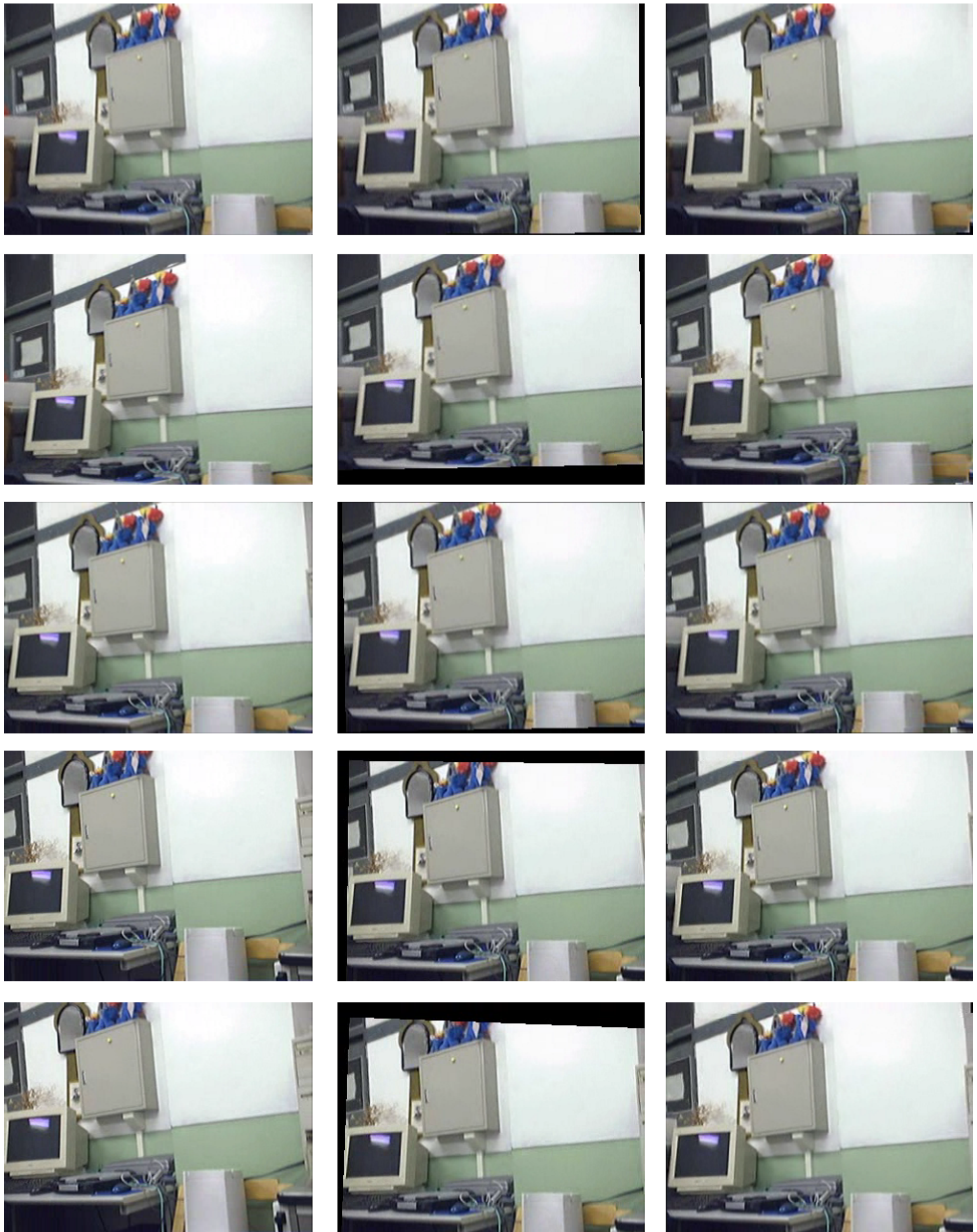
5. 結　　言

GPU を用いて計算を行うことによって，CPU のみを使った場合の処理速度を上回ることができたが, 現状では BGFS 法において推定速度が最も速いフレームでも 2fps 程度なので，リアルタイムは実現できていない．Powell 法と BGFS 法を比較してみると，CPU 側では BGFS 法の方が反復回数が多いが実際には１回の反復処理の中で小さな反復計算が行われており，全ての計算回数を合わせると，BGFS 法の方が計算回数が少なくなり，結果として推定時間が短縮されている．一方，GPU 側では BGFS 法の推定時間の方が長くなっていることがわかる．これは，GPU での計算の精度が低いためだと思われる．Powell 法に関しては計算精度がそれほど高くなくても値が収束するのに対し，BGFS 法では精度が低いためか値の収束に時間がかかり，結果として反復回数が多くなり，推定時間が長くなる．実際に BGFS 法において，CPU での計算を倍精度から単精度に変えて計算したところ，値が収束するまでの反復回数が増えることを確認した。

今後の課題としては，GPU での計算精度を CPU に近づけることと，計算の最適化による速度向上があげられる．GPU での計算精度を CPU での精度と同じにできれば BGFS 法において推定時間が現在の 1/4 以下になるものと考えられる．また，グローバルモーション推定の処理を階層化を行うアルゴリズム[1] を実装することや，float 型の pbuffer を使用できるようにすることによって，処理速度が向上する可能性がある．本論文では，エラー関数とその導関数を求める計算だけに GPU を使用したが，GPU 上での振動補正やモザイキングの実装も今後の課題である．

参 考 文 献

1) Bergen, J.R., Anandan, P., Hanna, K.J. and Hingorani, R.: Hierarchical Model-Based Motion Estimation, *ECCV'92 : Proceeding of the Second European Conference on Computer Vision*, pp. 237–252 (1992).
2) Bouguet, J.: Pyramidal Implementation of the Lucas Kanade Feature Tracker: Description of the Algorithm (2000). OpenCV Document, Intel, Microprocessor Research Labs.
3) Fernando, R. and Kilgard, M.J.: *The Cg Tutorial: The Definitive Guide to Programmable Real-Time Graphics*, Addison-Wesley Pub (2003).
4) Harris, M.: *GPU Gems:Programming Techniques, Tips and Tricks for Real-Time Graphics*, chapter38: Fast Fluid Dynamics Simulation on the GPU, pp. 637–665, Addison-Wesley Pub. (2004).
5) Litvin, A., Konrad, J. and Karl, W. C.: Probabilistic video stabilization using Kalman filtering and mosaicking, *IS&T/SPIE Symposium on Electronic Imaging, Image and Video Communications*, pp.663–674 (2003).
6) Matsushita, Y., Ofek, E., Ge, W., Tang, X. and Shum, H.-Y.: Full-Frame Video Stabilization with Motion Inpainting, *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Vol.28, No.7, pp. 1150–1163 (2006).
7) Mitchell, J. L., Ansari, M. Y. and Hart, E.: *ShaderX2: Shader Programming Tips and Tricks with DirectX 9*, chapterAdvanced Image Processing with DirectX9 Pixel Shaders, Wordware Publishing (2004).
8) Teukolsky, S. A., Vetterling, W. T. and Flannery, B.P.: *Numerical Recipes in C++: The Art of Scientific Computing*, Cambridge University Press (2002).
9) 金井　崇，安井悠介：GPU による細分割曲面の意匠形状評価，グラフィックスと CAD/Visual Computing 合同シンポジウム 2004 予稿集，pp.85–90 (2004).



(a) 補正前　(b) 補正後　(c) モザイキング

図 9　安定化結果. 左の列が補正前のフレーム，中央の列が補正後のフレーム，右の列が補正後のフレームにモザイキングを行ったフレーム.