2004 年度 卒業論文

ＧＰＵを利用した文字認識システム

提出日: 2005 年 2 月 2 日

指導: 筧 捷彦 教授

早稲田大学理工学部情報学科

学籍番号: 1G01P103-3

宮永 直樹

目 次

第1章　はじめに

本研究は文字認識を GPU (Graphics Processing Unit) で行い高速化を図ることを目的としている。画像認識において，ノイズや変形に強い手法は多くの処理時間を要する。こうした手法を高速化し文字認識に適用することで，認識精度の向上が望める。文字認識のアルゴリズムには一般化ハフ変換 (GHT: Generalized Hough Transform) [1] を用いる。この変換はノイズ，重なり，隠れに強いが，処理速度が遅いとされているものである。

高速化には GPU の計算能力を利用する。GPU はコンピュータのグラフィックス計算に用いるプロセッサである。この GPU 上で汎用計算を行う GPGPU (General-Purpose computation on GPUs) [2] という研究がある。GPU は近年，急速に性能を向上させ，高速な並列計算能力を持つようになった。リアルタイム３Ｄグラフィックスの目覚ましい進歩に伴ったものである。以前は映画でしか表現できなかったようなグラフィックスが，リアルタイムで計算できるようになってきた程である。また，GPU 上でプログラムを動作させることが可能になった。プログラマブルパイプラインという概念が導入されたためである。それまでは GPU に用意されている機能しか利用できなかったが，自由なグラフィックス表現を作ることができるようになった。GPGPU はこの高速な並列計算能力とプログラム可能であることを利用し、さまざまなアルゴリズムの高速化を果たすという研究である [3]。この GPGPU の手法を使い，一般化ハフ変換アルゴリズムの高速化を行った。

なお、この研究は吉田光寿氏との共同研究である。私は主に GPGPU 関連資料の調査や，GPU による改良アルゴリズムの実装等を担当した。吉田氏は主に一般化 Hough 変換関連資料の調査，改良アルゴリズムの検討，実装等を担当した。

第2章　一般化ハフ変換

2.1　特徴

一般化ハフ変換は直線検出などに用いられるハフ変換 (Hough Transform) を任意の図形に対して適用できるように拡張したアルゴリズムである [4]。一般化ハフ変換は文字認識の中でパターンマッチング的手法の部類に含まれる。パターンマッチング的手法とは，認識文字パターンと辞書文字パターンを重ね合わせ，一致の度合いで文字を認識する手法である。そのため，辞書文字パターンに用いるテンプレートが必要となる。

まずは，基本となるハフ変換を説明する。その後，一般化ハフ変換とその拡張について述べる。

2.2　ハフ変換

2.2.1　概要

ハフ変換は，Hough が直線線分検出法として 1962 年にアメリカで特許をとった仕事を始めとしている。直線線分に対して作られたものであるが，パラメタで表現できる図形ならば検出することができる。パラメタで表現できる図形としては円，楕円，放物線などがある。ただし，パラメタ数が増大するほど計算コストもかかってしまうため，一般的に直線の検出に対して使われている。ハフ変換は途切れのある図形に対しても検出できるという特徴がある。ここではハフ変換による直線検出についての方法を示す。

2.2.2　直線検出

直線検出のハフ変換として Duda と Hart の方法 [5] を説明する。ある直線に対し，原点から垂線を下ろしたときの垂線の長さを ρ，x 軸と垂線とのなす角を θ とする。このとき，直線は $\rho = x\cos\theta + y\sin\theta$ の式の形で表現することができる。図 2.1(a) はある点を通る直線が取りうる (ρ, θ) を図示したものである。この (ρ, θ) をパラメタとして用いる。直線が画像上の点 (x_0, y_0) を通るとすると，次の等式が成り立つ。

$$\rho = x_0\cos\theta + y_0\sin\theta \tag{2.1}$$

式 2.1 は ρ と θ との関数として見たとき，$\rho - \theta$ 空間上で正弦波を描く (図 2.1(b)) 。また，$x - y$ 空間においては (x_0, y_0) を通る全ての直線を表している。ところで，一

本の直線上にある全ての点は同じパラメタを持っているはずである。これらの点に対応する $\rho - \theta$ 空間上の曲線は，ある一点で交わることになる [4]。

直線検出の手順は以下のようになる。

1. 画像から直線の候補となる点を任意に選ぶ。
2. その点の座標を (x_i, y_i) とし，パラメタ空間 $\rho - \theta$ 上に $\rho = x_i \cos\theta + y_i \sin\theta$ の式が表す曲線を描く。
3. 直線の候補となる点全てについて 2. を行う。
4. $\rho - \theta$ 上で曲線が集中して交わっている点を探す。
5. この点での (ρ, θ) の値をパラメタとして直線を検出する。

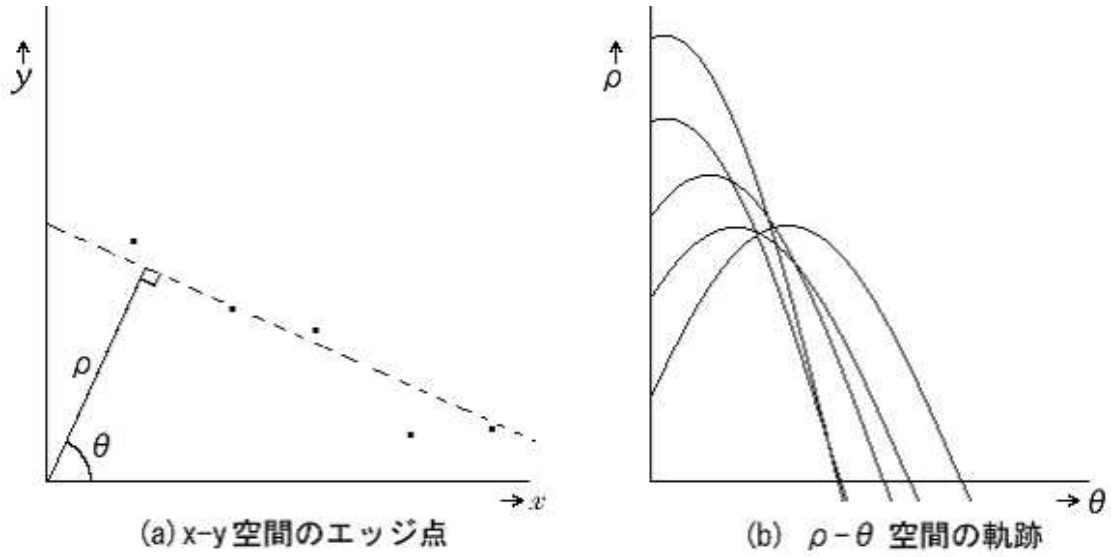


図 2.1: ハフ変換

なお，Hough は直線検出に $y = ax + b$ という式を用いた。これは直線を表す式として一般的なものである。(x_0, y_0) を通る直線を，$y_0 = ax_0 + b$ とすると (a, b) の組み合わせで全ての直線を表すことができる。この場合，(a, b) をパラメタとして使うのであるが，傾き a が 0 に近づいた場合に，y 切片である b が正または負の無限大に発散してしまうという問題があった。そのため，直線検出には欠点を取り除いた Duda と Hart の方法がよく用いられる。ρ と θ を使って極座標で表現しているためパラメタが発散することがない。ρ は原点から直線までの距離であるため，画像の対角線の長さよりも大きくなることはない。θ は直線から原点へ下ろした垂線と水平軸の間の角度であるため，$[0, 2\pi)$ などの区間で十分である。

2.2.3 投票

実際に検出を行う際には，多くの曲線が交わっている点を調べるために投票という手段を用いる。まず，パラメタ空間を細かく分割して離散化する。ρ を区間 $\rho_1, \rho_2, \cdots, \rho_m$ に，θ を区間 $\theta_1, \theta_2, \cdots, \theta_n$ に分ける。分割された各区域に $(\rho_i, \theta_j)(i = 1, 2, \cdots, m, j = 1, 2, \cdots, n)$ をインデックスとするカウンタを設ける。観測点の座標 (x_k, y_k) が与えられたとき，各 θ に対する ρ の値 $\rho(\theta)$ を求める。点 $(\rho(\theta), \theta)$ を含む領域 (ρ_i, θ_j) のカウンタを 1 だけ増やす。この操作を投票と呼ぶ。

すべての観測点について投票が行われると，2 次元上の度数分布が形成される。この極大値を求めてパラメタを特定することで，直線を検出することができるという仕組みである。

2.3 一般化ハフ変換

一般化ハフ変換は，ハフ変換のパラメタの取り方を変え，どんな図形でも検出できるように拡張されたものである。

2.3.1 基本原理

一般化ハフ変換ではテンプレート上の一点に参照点を決め，それの位置座標でテンプレートの位置を表す。テンプレートの形は参照点を原点とした相対的な位置関係で表す。これによって，図形が画像上のどの位置にあったとしても，相対的な位置関係が同じであれば検出ができることになる。

基本的な原理を三角形を例にとって説明する。いま，三角形 abc の b 点が観測点であったとする。辺 ca 上の各点が b 点を通るように平行移動した場合，参照点 O は図 2.2(a) の O_1O_2 の軌跡を描く。同様に，辺 ab，辺 bc の各点を通るようにした場合の参照点の軌跡を書き加える (図 2.2(b))。

同様に a 点，c 点を観測点とした場合の参照点の軌跡を書き加えると図 2.3 の点線のようにになる。

a 点，b 点，c 点を観測点としたそれぞれの軌跡は点 O の場所でだけ重なっていることがわかる。この軌跡を投票された点の集まりと考えると，点 O の場所には 3 票入っており，他の点では 1 票しか入っていないことになる。投票度数が極大になっている点を探せば図形の移動した量がわかる [4]。

2.3.2 アルゴリズム

実際の図形検出に行う処理を説明する。

まず，テンプレートとなる画像上の任意の位置に参照点 O を決める。輪郭上にあるピクセルの各点について，O に向かうベクトル v_i を求める。これは r を v の長さ，α を v と x 軸とのなす角としたとき，$v_i = (r_i \cos\alpha_i, r_i \sin\alpha_i)$ と極座標系で表すことができる。その点での画像の濃淡勾配の方向と x 軸のなす角 ω_i を求め，(r, α) との

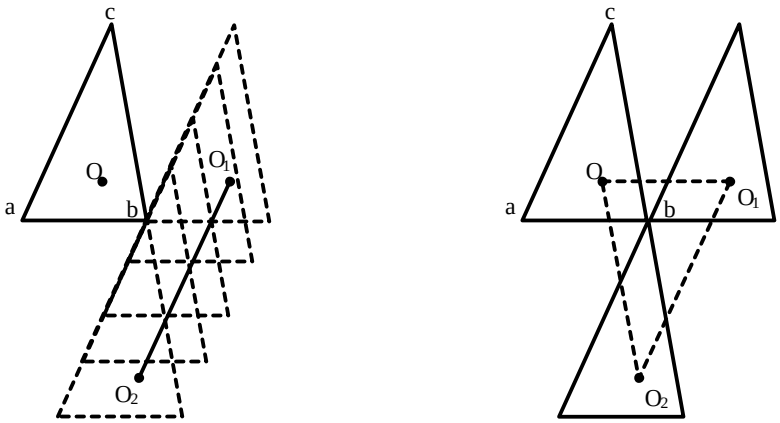


図 2.2: 参照点が描く軌跡

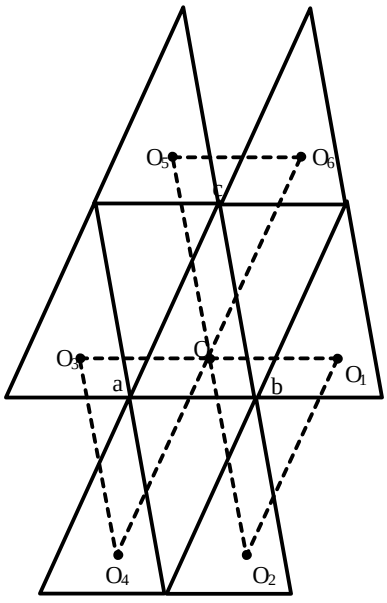
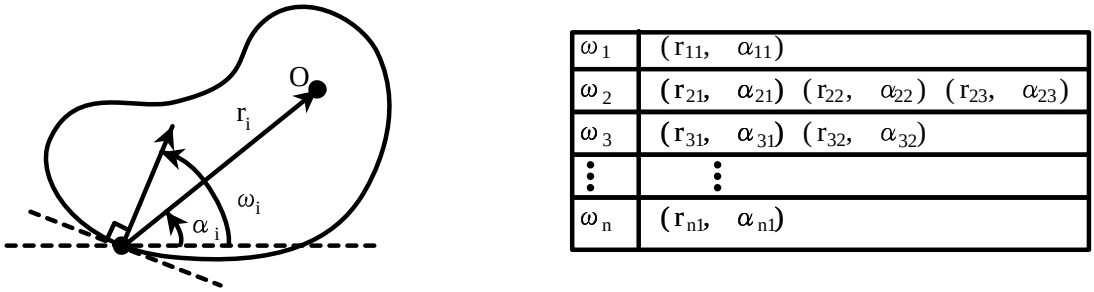


図 2.3: 三角形の一般化ハフ変換

対応を R テーブルという表に登録する (図 2.4) 。



(a) 形状のモデル化　　(b) Rテーブル

図 2.4: 一般化ハフ変換での形状記述

投票は，パラメタを平行移動 (u, v)，回転角 θ，拡大率 s とした 4 次元空間に行う。入力画像上の特徴点 (X_j, Y_j) と，その点での濃淡勾配の方向 ω_j を求める。すべての (θ, s) の組み合わせについて，R テーブルから $\omega_j - \theta$ に対応する座標値 $(r(\omega_j - \theta), \alpha(\omega_j - \theta))$ を取り出す。回転角 θ， 拡大率 s に対応する平行移動ベクトル (u, v) を次の式から求める。

$$u = X_j + r(\omega_j - \theta) \cdot s \cdot \cos(\alpha(\omega_j - \theta) + \theta) \quad (2.2)$$

$$v = Y_j + r(\omega_j - \theta) \cdot s \cdot \sin(\alpha(\omega_j - \theta) + \theta) \quad (2.3)$$

(u, v, θ, s) をパラメタとして投票を行う。投票数の多いパラメタを用いて入力画像に座標変換を行ったとき，テンプレート画像と一致する部分が高いことになる。

投票の多いパラメタをもとにして検出を行うため，入力画像の品質が悪い場合でも比較的精度の高い検出を行うことができる。輪郭の一部に重なりや，隠れている部分があってもあまり影響を受けず，ノイズに強いという特徴を持っている。

しかし，回転と拡大については値を変化させながら総当りで調べていくため，処理速度が遅いという欠点がある。4 次元のパラメタ空間を使用するため，多くのメモリ領域も消費してしまう [6]。

2.4 改善手法

一般化ハフ変換を改良したものの一つに FGHT (高速一般化ハフ変換) [7] がある。一般化ハフ変換と Chord-Tangent 変換 (CTT) [8] を組み合わせており，処理速度や検出精度が向上している。FGHT では輪郭上の二点を結ぶ線分 (コード) を用いている。水平方向の座標軸とコードの間の角度と，コードの長さを使うことで，回転と拡大の含めた計算を行っている。また，参照点のほかにチェック点というものを導入することで，無駄な投票を抑えている。ただし，線分近似ができる形状のものに限られることや，所要メモリがかなり大きいという問題もあり，改善方法が提案されている [9]。

第3章　エッジ検出

一般化ハフ変換では輪郭上の点から参照点までの相対位置を用いて図形検出を行うため，前処理として輪郭を検出する必要がある。輪郭は，図3.1のような濃淡変化が周囲のピクセルで連続して起こっている場合であると言われている。特に，図3.1(a)のようなステップ状の濃淡変化を一般にエッジと呼ぶが，これは明るさが急激に変化したときに現れる。この章ではエッジを検出する方法として微分フィルタについて述べる。

図 3.1: 輪郭の濃淡パターン

3.1　一次微分フィルタ

エッジは明るさが急激に変化しているところであることから，明るさの変化を微分値から求められることになる。画像を対称にしているため，実際には差分を取って計算する。

$x-y$ 座標上の画像の明るさを $f(x,y)$ で表したとき，x,y に関する $f(x,y)$ の偏微分係数は，次のような差分に近似して表すことができる。

$$f_x(x,y) = f(x+1,y) - f(x,y) \tag{3.1}$$
$$f_y(x,y) = f(x,y+1) - f(x,y) \tag{3.2}$$

しかし，これは厳密には座標 $(x+0.5, y+0.5)$ の値であり，2つの画素の境目での1次微分となってしまう。そこで，1画素をおいて差分を計算し，次のような形で表す。

$$f_x(x,y) = f(x+1,y) - f(x-1,y) \tag{3.3}$$
$$f_y(x,y) = f(x,y+1) - f(x,y-1) \tag{3.4}$$

ここで，x についての偏微分係数は $f(x-1,y), f(x,y), f(x+1,y)$ に $-1, 0, 1$ の重みをかけて足したことになる。そこで，この重みを次のようにベクトルで表すことにする。

$$f_x : \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} f_y : \begin{bmatrix} -1 \\ 0 \\ 1 \end{bmatrix}$$

という形で表現する。安定した値を得るために 3×3 の近傍の平均を取るように拡張すると，次の形になる。

$$f_x : \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} f_y : \begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.5}$$

これは Prewitt フィルタと呼ばれている。係数を微妙に変更したものは多く提案されており，平均を求める際の中心点の重みを大きくしたものが次の係数で表される Sobel フィルタである。

$$f_x : \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} f_y : \begin{bmatrix} -1 & -2 & -1 \\ 0 & 0 & 0 \\ 1 & 2 & 1 \end{bmatrix} \tag{3.6}$$

これらは一次微分によってエッジを検出しているため，一次微分フィルタと呼ばれる。

各画素の持つエッジの強度 I は次の式から求めることができる。

$$I = \sqrt{f_x^2(x,y) + f_y^2(x,y)} \tag{3.7}$$

また，一次微分フィルタでは x と y の偏微分係数が求められるため，エッジの方向を次の式によって求めることができる。

$$\theta = tan^{-1}\left(\frac{f_y}{f_x}\right) \tag{3.8}$$

Sobel フィルタはエッジの方向によってエッジの強度に対する感度がほとんど変化せず安定している。また，エッジの実際の方向と測定した方向がほぼ同じになるという特徴がある [4]。

3.2 ラプラシアンフィルタ

ラプラシアンフィルタ (Lablacian filter) はエッジの方向によらず，エッジの強度のみに敏感に反応するフィルタである。ラプラシアン ∇^2 は空間二次微分を行うオペレータであり，次のような演算を行う。

$$\nabla^2 f(x,y) = f_{xx}(x,y) + f_{yy}(x,y) \tag{3.9}$$

これを差分形式で表すと，

$$f_{xx}(x,y) = f(x-1,y) - 2f(x,y) + f(x+1,y) \tag{3.10}$$
$$f_{yy}(x,y) = f(x,y-1) - 2f(x,y) + f(x,y+1) \tag{3.11}$$

となることから式 (3.9) は次のようになる。

$$\begin{aligned}\nabla^2 f(x,y) &= f_{xx}(x,y) + f_{yy}(x,y) \\ &= f(x-1,y) - 2f(x,y) + f(x+1,y) + f(x,y-1) - 2f(x,y) + f(x,y+1) \\ &= f(x-1,y) + f(x+1,y) + f(x,y-1) + f(x,y+1) - 4f(x,y) \end{aligned} \tag{3.12}$$

これを係数で表現すると，

$$\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \tag{3.13}$$

となる。これをラプラシアンフィルタという。また，45° 方向も含めると，

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{3.14}$$

という形式になり，この形もよく使われる。

ラプラシアンフィルタは二次微分を使っているため，画像の明るさの変化に強く反応する。ただし，雑音まで強調してしまうという欠点もある。そこで画像の平滑化を行ったあとでラプラシアンを使うといった工夫が行われている。

第4章　GPGPUについて

4.1　アーキテクチャ

GPU はグラフィックス計算に適したアーキテクチャとなっている。物体を画面に描画する際には一連の流れで計算が行われる (図 4.1) 。この流れはグラフィックスパイプラインと呼ばれている。

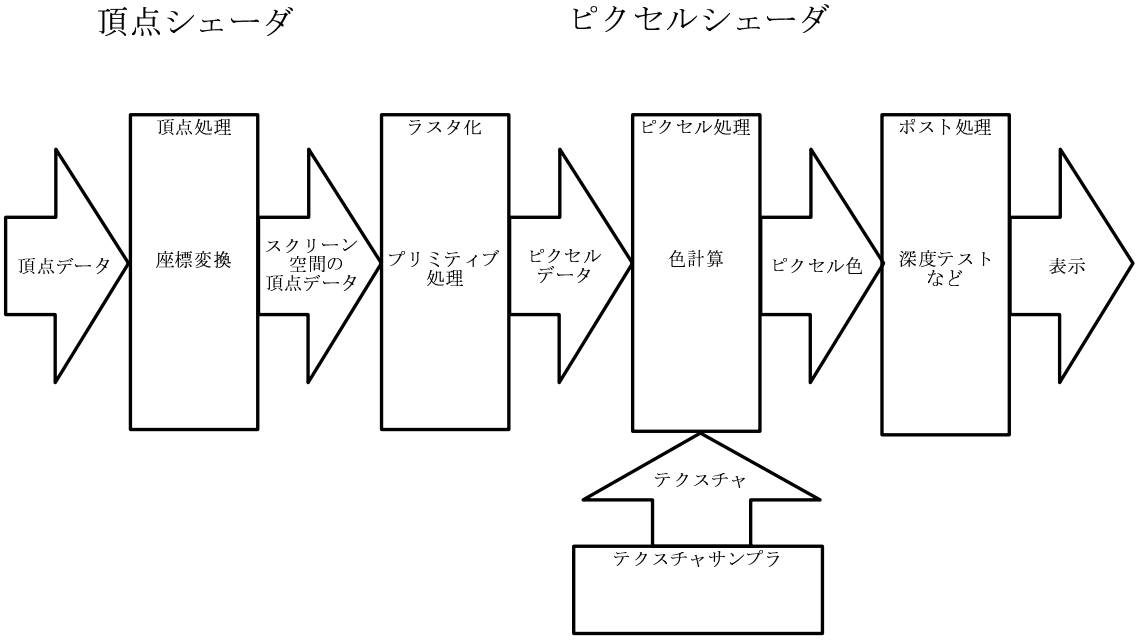


図 4.1: グラフィックスパイプライン

リアルタイム３Ｄグラフィックスにおいて物体は頂点の集合データで表されている。通常，物体はポリゴン (Polygon) と呼ばれる要素で構成される。ポリゴンとは三つ以上の頂点が結ばれた閉じた図形であり，頂点間に面を張ることができる。これら頂点の集合を三次元空間からスクリーンの二次元空間へと変換し，それらの位置にあるピクセルに色を塗りつぶすことによって描画が行われる。ピクセルの色は頂点の色から補完されたものが使われるが，テクスチャを使って色を読み込むこともできる。テクスチャは物体表面に張り付けるためのビットマップ画像である。物体とテクスチャの張り付ける位置とを対応付けるには，頂点データにテクスチャ座標というテクスチャ内の相対位置を入れておく。

物体を画面に描画する際には，まずグラフィックスパイプラインに頂点データが渡される。頂点処理によって座標変換などが行われ，三次元空間から二次元のスク

リーン空間へと変換される。二次元となった頂点にはラスタ化 (Rasterization) という処理が行われ，スクリーン上の塗りつぶされるピクセルが計算される (図 4.2) 。そのピクセルはピクセル処理によって色が計算される。その後，深度テストによって物体の重なりを判断し，最終的に画面への表示が行われる。これらの処理はレンダリングと呼ばれる。

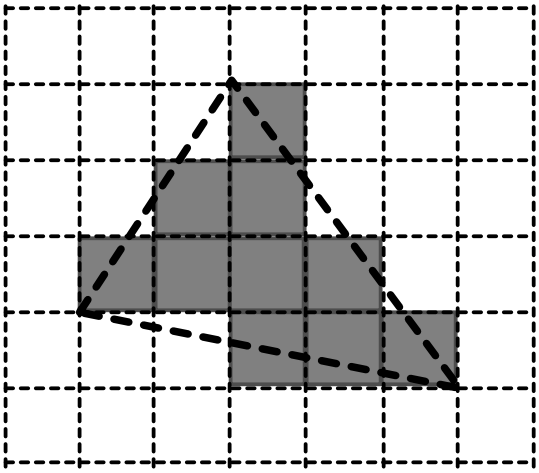


図 4.2: ラスタ化

プログラマブルパイプラインは，グラフィックスパイプラインの頂点処理とピクセル処理をプログラム可能にしたものである。頂点処理を行うものは頂点シェーダ (Vertex Shader) ，ピクセル処理を行うものはピクセルシェーダ (Pixel Shader) と呼ばれる。また，これらは総称してプログラマブルシェーダと呼ばれる。頂点シェーダを Vertex Processor，ピクセルシェーダを Fragment Processor と呼ぶこともある。

頂点シェーダでは各頂点のデータを操作することができる。各頂点は位置，法線，色，テクスチャ座標などのデータを持っている。頂点シェーダで出力された値はピクセルシェーダに渡される。頂点とピクセルは一対一対応ではない場合がほとんどであるため，頂点シェーダの出力データを補完したものがピクセルシェーダの入力となる。ピクセルシェーダでは，ピクセルの色を計算できる。ここでは、色やテクスチャ座標などのデータが処理される。テクスチャを参照して色を読み込むこともできる [10, 11]。

4.2 性能

現在の GPU は CPU を凌ぐトランジスタ数を持っている。GeForce 6800 の場合は 2 億 2,200 万のトランジスタがあり，Prescott コアの Pentium4 プロセッサは 1 億 2500 万であるため約 1.78 倍となっている。また，その一年前に発表されたモデルである GeForceFX 5900 のトランジスタ数 1 億 3000 万の約 1.7 倍になっている [12, 13]。

１秒間当たりに塗りつぶすピクセル数を表すフィル速度についても GeForceFX 5900 の 38 億ピクセル/秒から GeForce 6800 の 64 億ピクセル/秒へと約 1.68 倍になっ

ている [13]。

また，浮動小数点計算の性能を見てみると，GeForceFX 5900 は 20GFlops，GeForce 6800 は 40GFlops に達しており，動作周波数が 3GHz の Pentium 4 の理論値 6GFlops を大きく上回る。

表 4.1: GPU 性能比較

	Pentium4	GeForce 5900	GeForce 6800
トランジスタ数	125 M	130 M	222 M
フィル速度	–	3800 Mpixels/sec	6400 Mpixels/sec
浮動小数点計算	6 GFlops	20 GFlops	40 GFlops

GPU の性能は年間成長率が 2 倍を超えている。一方，CPU の性能はムーアの法則に従い 3 年で 4 倍となる。GPU は 3 年で 8 倍になることから，ムーアの法則を大きく超えた速度で成長していることになる (図 4.3)。この成長速度には二つ理由がある。一つは，GPU の構造上トランジスタの追加が容易であり，それが直接計算性能に結びつくことである。二つ目は，コンピュータゲームの市場に支えられ，更なる進化を後押しされているためである [13, 14]。

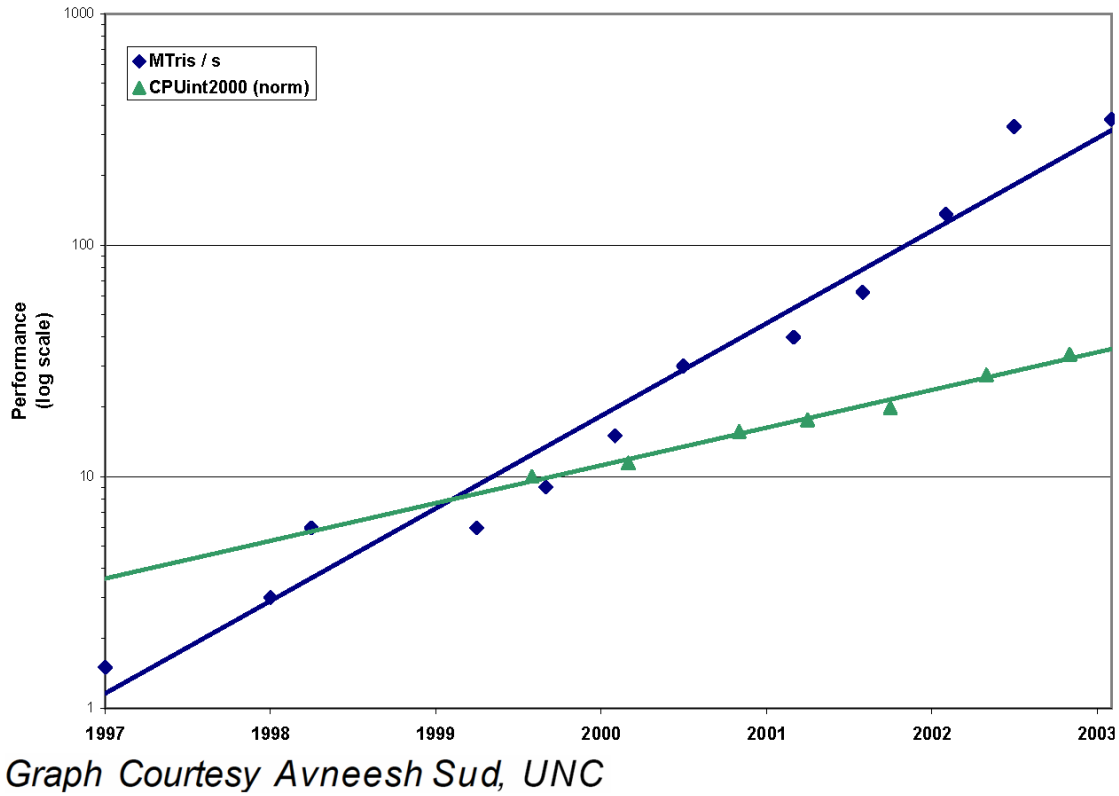


図 4.3: GPU の成長速度

4.3 特徴

GPU はプログラマブルパイプラインを複数持った並列ベクトルプロセッサである (図 4.4)。ピクセルシェーダは各画素について同じ命令を発行する SIMD 型になっている。頂点シェーダも SIMD 型であったが，MIMD 型のものも出てきており，分岐命令を使った場合に他のパイプラインと同期を取る必要がなく，遅延が抑えられている。パイプラインの数は通常ピクセルシェーダの方が頂点シェーダよりも多くなっている。例えば，NVIDIA 社の GeForce6800Ultra や，ATI 社の RadeonX850 は 6 つの頂点パイプラインと 16 個のピクセルパイプラインを持っている。パイプラインの並列性を効率的に利用することで，大量のデータに対し同一の処理を高速に行うことができる。

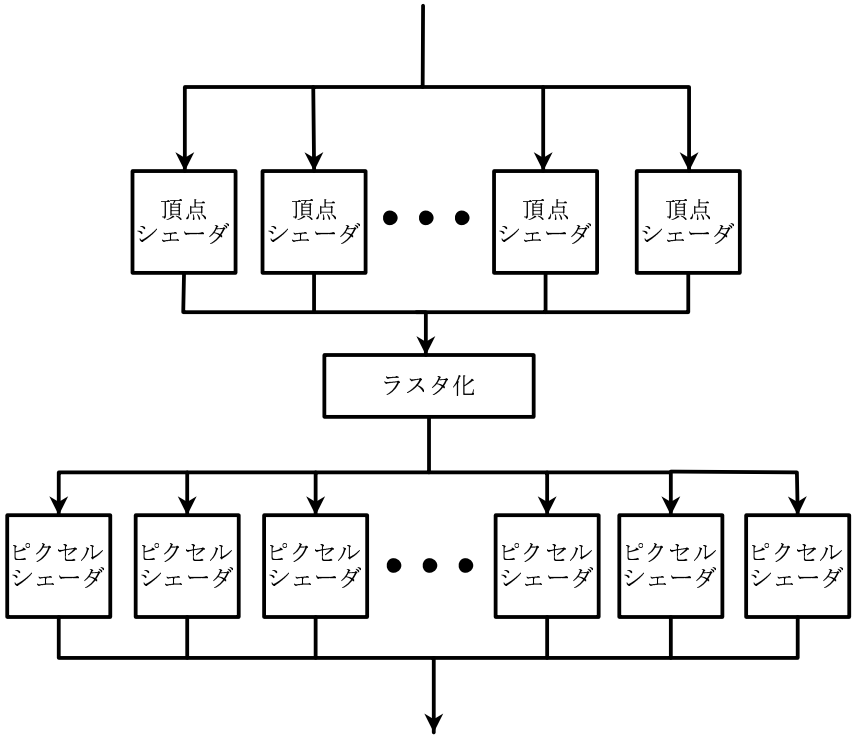


図 4.4: 並列パイプライン

また，4 次元ベクトルの計算に適した命令を持っている。4 成分の浮動小数点ベクトルは頂点や色のデータ表現に用いられるためである。頂点は x，y，z，w の 4 成分で表されている。色は RGBA の 4 成分で表され，それぞれ赤，緑，青，透明度のアルファを示す。多くの命令は，4 次元ベクトルの各要素を使った計算を 1 命令で処理することができる (図 4.5) 。内積計算命令は物体面の表裏判定や色の計算など多くの計算に使われる。行列計算も内積計算の組み合わせであるため高速に処理が行われる。他にも積和演算命令や距離計算命令，正規化などに用いられる逆数平方根などの命令を持つ。加算や乗算といった基本的な命令も，各要素ごとに演算した結果を返すようになっているため効率が良い。これらの命令を効果的に利用することで，実行速度を大きく向上させることができる。

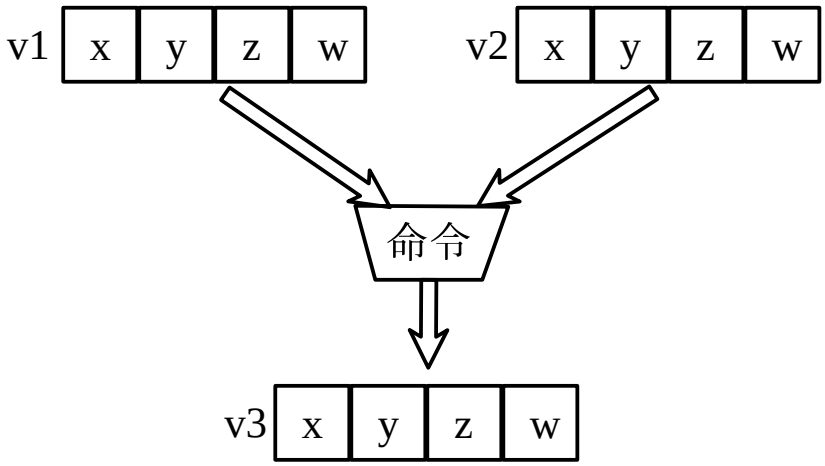


図 4.5: ベクトル命令

GPU は並列ベクトルプロセッサとしては安価である。ベクトルプロセッサは一部のスーパーコンピュータなどに搭載されているが，非常に高価である。他のベクトルプロセッサと比べると，GPU はコストパフォーマンスが高い。また，一般に普及しているため，比較的多くの人が恩恵を受けることができる [10, 11, 14]。

4.4 制約

GPU はグラフィックス目的に作られているため GPGPU の利用に不都合となる制約も多い。

シェーダの命令数には制限があり小さなプログラムしか作ることができない。プログラマブルシェーダのプログラムは，実行前に GPU のビデオメモリに転送しておかなければならない。このとき，プロセッサによってプログラムの命令数の上限が決められているのである。繰り返しなどの動的フロー制御に対しても，反復回数やネスト数に制限がある。また，再帰を使うことはできない。複雑なプログラムを作る場合はシェーダプログラムをいくつも用意し，処理ごとに切り替えるなどの工夫が必要となる。

また，ポインタがなく動的配列を使うことができない。メモリアクセスはできず、記憶領域としてはレジスタのみを使うことができる。大きなデータを参照するには，テクスチャにあらかじめデータを格納しておき，そこからデータを読み込むという形を取る必要がある。

テクスチャのデータフォーマットには整数フォーマットと浮動小数点フォーマットがある。整数フォーマットでは 8bit のものがよく使われ，RGBA のそれぞれの成分を 0 から 255 までの値で表す。浮動小数点フォーマットは後に追加されたものであり，各成分を 16bit や 32bit で表現することができる。しかし，浮動小数点フォーマットはハードウェアによってはサポートされていないものがある。サポートされている場合でも，色の補完処理に用いるフィルタリングや，転送元の色と転送先の色の合成を行うブレンディングに対応していないものが多い。用途にあわせてフォー

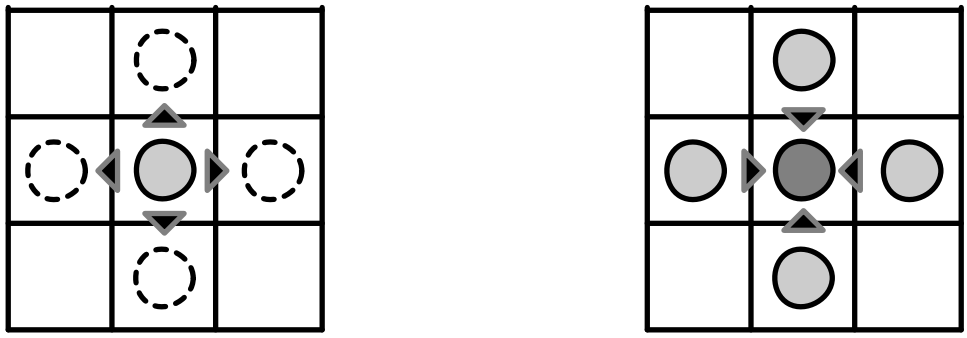
マットを選択する必要がある。

テクスチャのデータを読むときや，スクリーンに出力するときにはデータを色として扱う。色は 0 以上 1 以下の浮動小数点で表される。通常データの読み書きを行うときにはこの区間に正規化しておかなくてはならない。

テクスチャの利用はデータ処理を行うのに有効である。テクスチャは通常ピクセルシェーダからしか読み込むことができない。サンプラとテクスチャ座標を指定することでピクセルの色を参照することが出来る。テクスチャの座標は 0 以上 1 以下の浮動小数点で表され，画像の左上が $(0,0)$，右上が $(1,0)$，左下が $(0,1)$，右下が $(1,1)$ に対応する。テクスチャから複数のピクセルを参照したり，複数枚のテクスチャから色を取り出したりできるため，多くのデータを処理することが出来る。

一方，頂点シェーダで扱うことが出来るデータはその頂点のものに限られてしまう。頂点シェーダにはどの頂点と隣接しているかという情報が与えられない。またテクスチャは基本的に使うことが出来ない。極一部のハードウェアでのみ，頂点シェーダでのテクスチャのロードがサポートされている。

ただし，頂点シェーダは座標変換に用いるものであるため，頂点の位置を自由に移動することが可能である。これにより出力先のピクセルを指定することが出来る (図 4.6(a)) 。逆に，ピクセルシェーダでは出力先のピクセルが指定されている形になっているため，別のピクセルへデータを移動することはできない。必要なデータを処理するには，テクスチャ内のデータのある位置を指定して受け取らなければならないということになる (図 4.6(b)) [14]。



(a) 頂点シェーダ　　(b) ピクセルシェーダ

図 4.6: 各シェーダのデータ処理形式の違い

出力の結果を CPU に転送する際にはオーバーヘッドが発生してしまう。通常の CPU から GPU へのデータ転送は，AGP8X バスの場合 2GB/s に近い転送速度を得ることができる。しかし，逆方向の GPU から CPU へのデータ転送では，数 100MB/s 程度の転送速度しか得られない (図 4.7)。なお，PCI-Express x16 バスの場合は上り，下りともに 4GB/s であるため，転送速度に差は生じない。ただし，GPU から CPU へのデータ転送を行うには，一旦 GPU のパイプラインをフラッシュしなければならないという制約もある。これはバスの種類に関係なく，GPU のアーキテクチャ上の

問題である。そのため，GPU から CPU へのデータ転送が頻繁に行われると，GPU の性能を十分に活かすことができない [2, 3]。

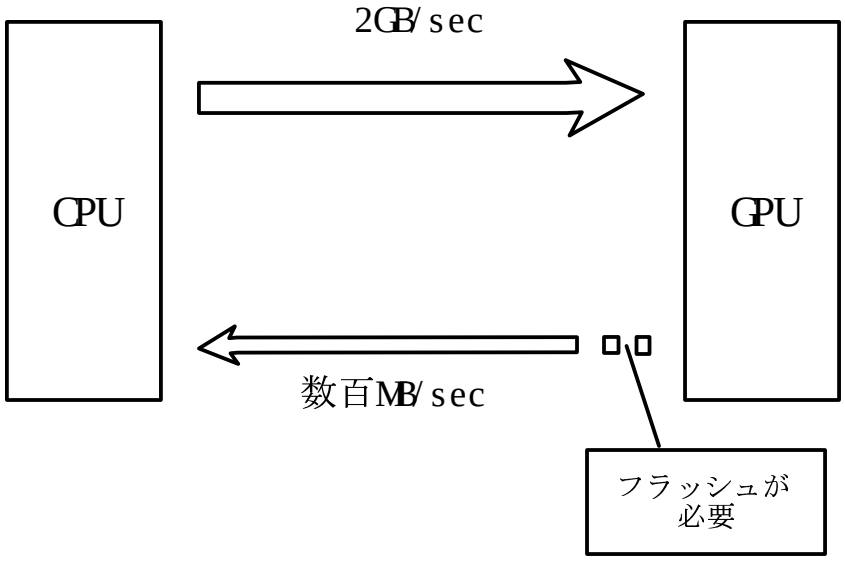


図 4.7: AGP8X バスでのデータ転送

4.5 GPGPU プログラミング

4.5.1 基本手順

シェーダを使って GPGPU プログラミングを行う方法を説明する。入力データは頂点ストリームかテクスチャに格納しておく。

頂点ストリームの場合，頂点の位置，法線，テクスチャ座標などのデータに入力データを入れておく。この頂点ストリームをレンダリングすると頂点シェーダが呼ばれるため，そこで頂点データを処理するという形になる。また，頂点処理が終わるとピクセルシェーダが呼ばれるため，処理を分担することもできる。

テクスチャの場合は，テクスチャの RGBA の色値に入力データを入れておく。これをピクセルシェーダから参照してデータ処理を行うのであるが，シェーダは物体をレンダリングしないと呼び出せない。そこで，長方形の面などを物体として用意しテクスチャを張り付けておく。この面がスクリーンと同じサイズであり位置を合わせてあれば，ラスタ処理でスクリーン上のピクセル全体が選択され，全てのピクセルについて計算を行うことができるようになる。物体の形や位置によって計算を行うピクセルを調節するといったこともできる (図 4.8) 。

シェーダで計算した結果は最終的に色情報のみがスクリーンにレンダリングされる形になる。出力したいデータは色として返す必要がある。また計算結果を得るためには，出力用のテクスチャを用意しておき，あらかじめレンダリングターゲットとして設定しておく必要がある。

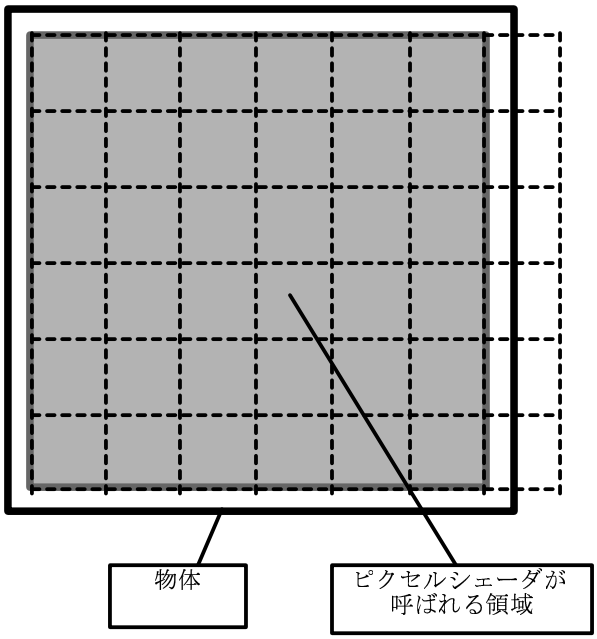


図 4.8: ピクセルシェーダの適用領域

これらをまとめると全体の流れは以下のようになる。

1. 物体の頂点ストリームやテクスチャなどを作成しデータを格納する。
2. レンダリングターゲットを出力用のテクスチャに設定し，使用する頂点シェーダやピクセルシェーダを設定しておく。
3. 頂点のレンダリングを行う。
4. 頂点シェーダ，ピクセルシェーダの順に実行されるので，ここで計算処理を行う。
5. 必要に応じて，別のシェーダについて 2〜4 を繰り返す。
6. 計算結果を持つテクスチャを CPU に返す。

計算結果を CPU に返す必要がない場合は，出力されたテクスチャを別のシェーダの入力として使うことができる。こうすることで，複雑なプログラムを分割して処理することが出来る。ここで注意すべき点はテクスチャを同時に入力と出力に使うことが出来ないことである。つまり，自分のデータを更新するような処理は行うことが出来ない。そのため出力用テクスチャを複数枚用意してテクスチャとレンダリングターゲットを交互に切り替える必要がある。

4.5.2 実装例

GPGPU を利用した計算として物理シミュレーションがある。グラフィックスにおいても，リアルな映像表現を得るために物理シミュレーションが行われ，計算を

GPU で行うことで高速化をしているものがある。たとえば，水面の波の表現や雲の動きに流体シミュレーションや粒子シミュレーションが使われている。

多くの数値計算にも GPU 実装が行われている。FFT を GPU で実装したものや，ナビエストーク方程式を解く流体シミュレーションなどがある。

そのほか，音楽の分野に GPU が使われたものもある。BionicFX 社は音響エフェクトの計算に GPU を用いる技術を開発している。また，室内音響計算における音の反射を GPU で実装したというものもある [2, 14]。

GPGPU では高速な並列計算が可能であることが最大の利点であるが，内部のデータ表現によってはデータの可視化も可能となる。

4.6 シェーダ言語

プログラマブルパイプラインで実行されるシェーダプログラムの作成には，アセンブラ言語やシェーダ言語という GPU 専用の高級言語を使うことができる。シェーダ言語は NVIDIA 社の Cg 言語，Microsoft DirectX の HLSL，OpenGL の GLSL などがある。いずれも C 言語ベースで GPU に特化された言語仕様となっている。これらの言語は文法が似通っており，シェーダプログラムの書き方もほとんど同じである。シェーダは関数の形で定義し，１つの要素についての計算処理を記述する。この関数が全ての要素について並列に処理される。頂点シェーダの場合，頂点ストリームの各頂点ごとに関数が呼ばれ，ピクセルシェーダではスクリーン上の塗りつぶされるピクセルについて関数が呼ばれることになる [10, 11, 15]。

Waterloo 大学の Sh 言語 [16] は，C++言語と全く同じ文法でシェーダを書くことができる。API が C++のクラスライブラリという形で提供されており，これを利用することでシェーダを作成できるためである。Sh を利用して書かれたプログラムを実行すると，GPU の実装に依存しない中間コードを作り出す。これをアセンブラに変換して GPU で利用できるようにするというものになっている。

また，GPGPU を目的として開発された言語もある。Stanford 大学の Brook 言語 [17] は ANSI C に言語拡張を行い，ストリーム処理を取り入れたものである。並列データに対して同一の処理を行う方法を指定でき，局所計算に向いた演算を書きやすいようになっている。コンパイルを行うと，C++コードに翻訳される。この C++は Brook のランタイムライブラリを利用しており，ランタイムライブラリによってアーキテクチャの違いを吸収するという仕組みになっている。

Brook や Sh は GPU のハードウェアを隠蔽するため，GPU を意識することなくプログラムを書くことができる。その反面，GPU のハードウェアに最適化されたプログラムを書くには Cg などのシェーダ言語を用いる必要があると思われる。

4.7 GPGPU による一般化ハフ変換

一般化ハフ変換は GPGPU での処理に向いていると思われる。まず，浮動小数点ベクトルを多用するためベクトル演算ユニットを活用することができる。通常，濃

淡勾配や参照点までのベクトルは極座標に変換して計算を行う。GPU の場合は，ベクトル計算の命令が高速であるため，それらをベクトルのまま扱うことで高速化が行えると考えられる。

また，並列化を行いやすい。入力画像の全ての輪郭点について同一の処理が行われるためである。ハフ変換では並列アーキテクチャで実装されたものもあり，それらは一般化ハフ変換についても応用可能であるものが多い。

文字認識は画像処理の一種であるため GPU で扱いやすいという点もある。画像の参照はテクスチャを使って扱うことができ，フィルタも簡単に実装することができる。

これらの要因から，一般化ハフ変換の計算処理は GPU で高速に行うことができると考えられる。

第5章　実装

5.1　環境

GPGPU のプログラムを作成する環境として，DirectX を用いることにした。DirectX を使うことによって Windows 上でグラフィックスの制御を行うことができる。シェーダ言語には DirectX で提供されている HLSL 言語を使うことにした。

5.2　方法1

5.2.1　特徴

一般化ハフ変換における参照点の座標変換の計算部分のみを GPU で行う。計算結果は CPU に返し，投票空間には配列を使って行う。また，輪郭の濃淡勾配を微分フィルタで求め，濃淡勾配の方向を元に回転も含めた計算を行う。

5.2.2　アルゴリズム

入力画像に Sobel フィルタをかけて輪郭を検出する。輪郭の強度が閾値より高いところについて，座標を配列 P^I に，濃淡勾配ベクトルを配列 V^I に格納する。Sobel フィルタを使えば x 軸方向の微分 f_x と y 軸方向の微分 f_y がわかるため，これを濃淡勾配のベクトルとする。テンプレート画像でも同様に輪郭の座標を配列 P^T，濃淡勾配ベクトルを配列 V^T に格納する。

P^T から位置ベクトル p を取り出し，参照点 O までの参照ベクトル r を求める。P^I から位置ベクトル q，V^T から濃淡勾配 t，V^I から濃淡勾配 s をそれぞれ取り出す。t から s への回転行列 M，回転角 θ を次のように求める。

$$\cos\theta = \frac{t_x s_x + t_y s_y}{|t||s|} \tag{5.1}$$

$$\sin\theta = \frac{t_x s_y - t_y s_x}{|t||s|} \tag{5.2}$$

$$M = \begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix} \tag{5.3}$$

$$\theta = \begin{cases} \cos^{-1}\cos\theta & (\sin\theta > 0) \\ -\cos^{-1}\cos\theta & (\sin\theta \leq 0) \end{cases} \tag{5.4}$$

回転行列 M を参照ベクトル r に掛け，入力画像の輪郭点の位置ベクトル q に加算

する。この座標を O' とする

$$O' = q + Mr \tag{5.5}$$

図 5.1 にこの変換を示した。

投票は，回転角を含めた三次元の投票空間の (q'_x, q'_y, θ) に行う。投票空間から最大の投票度数を探し，適合率を求める。

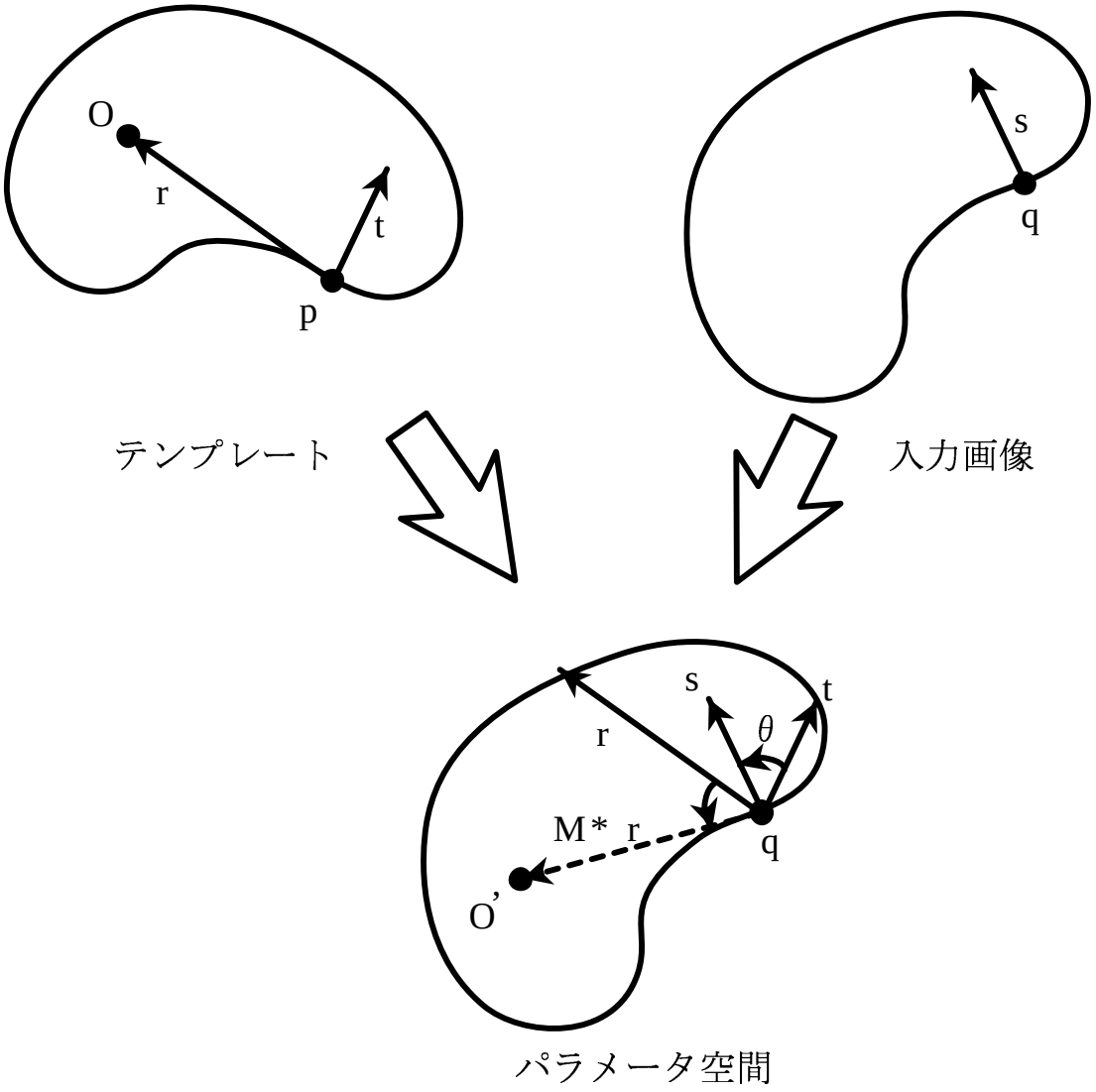


図 5.1: 回転を含めた一般化ハフ変換

5.2.3 GPGPU での実装

入力画像の輪郭データを一次元テクスチャに格納する。輪郭は Sobel フィルタで検出し，輪郭点の座標と濃淡勾配とを調べる。ただし文字が太字の場合，Sobel フィルタは 3×3 のフィルタであるため，実際には隣のピクセルに輪郭がある場合にも

輪郭として検出されてしまい輪郭が太くなってしまう。細線化処理を行えばよいのだが，処理時間がかかってしまう。そこで，ピクセルの色が背景色の場合にのみ輪郭として扱うようにした。

検出した輪郭の座標の x 成分をテクスチャの色の R 成分に，y 成分をテクスチャの G 成分に入れる。また同様に，テクスチャの B 成分，A 成分に輪郭点上の濃淡勾配の x 成分，y 成分をそれぞれ入れる。テンプレート画像についても輪郭点の座標と濃淡勾配とを出し，別の一次元テクスチャに格納する。

計算結果を格納するテクスチャを用意する。幅を入力画像の輪郭テクスチャの長さとし，高さをテンプレート画像の輪郭テクスチャの長さとする。このテクスチャをレンダリングターゲットに設定する。画面全体と同じ大きさの正方形の物体をレンダリングして，スクリーン上の全てのピクセルについてピクセルシェーダが呼ばれるようにする。このとき，ピクセルシェーダに一般化ハフ変換のシェーダプログラムを設定しておく。

シェーダプログラムでは，まず輪郭の組み合わせを選択する。計算されるピクセルのテクスチャ座標を調べ，テクスチャ座標の x 成分を使って入力の輪郭テクスチャを参照する。この輪郭テクスチャから座標と濃淡勾配を取り出す。同様にテクスチャ座標の y 成分でテンプレートの輪郭テクスチャを参照し，輪郭点の座標と濃淡勾配を取り出す。この輪郭の組み合わせについて一般化ハフ変換の計算を行う (図 5.2) 。

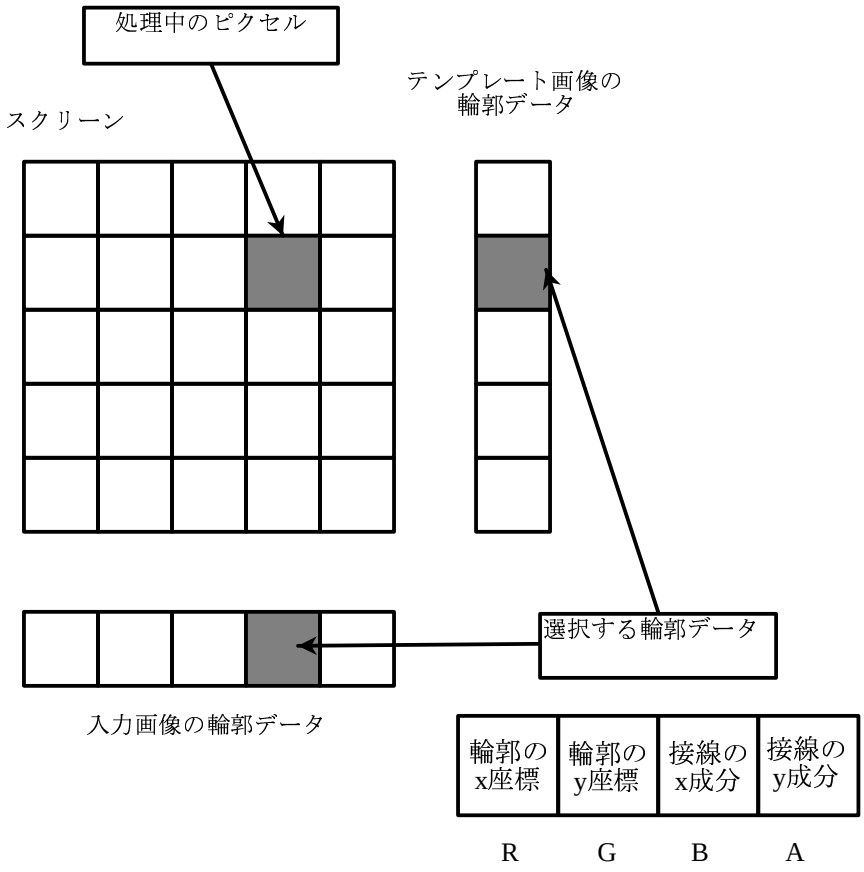


図 5.2: 輪郭点の組み合わせの選択

次に，投票先の座標を求める。テンプレート画像の輪郭の座標から参照点までのベクトルを計算する。テンプレート画像と入力画像のそれぞれの輪郭の濃淡勾配から回転行列と回転角を求める。参照点までのベクトルに回転行列を掛けて入力画像の輪郭の座標に加算し，その座標を投票先とする。この座標と回転角とを結果の色として出力する。

投票が終わるとテクスチャの各ピクセルに座標と回転角が入った状態になっている。このテクスチャを CPU に返し，各ピクセルデータを取り出す。また，投票空間を三次元配列として確保しておく。ピクセルから投票先の座標と回転角を取り出し，これらの値に対応する配列の要素に投票を行う。その要素の投票数がそれまでの最大投票数より多い場合，最大値を更新し配列の添字の値を記憶しておく。すべての投票が終わったときの最大値が適合率となり，添字の値から移動量と回転角を調べることができる。

5.3 方法2

5.3.1 特徴

この方法では，投票先のブロックを中心に考えている。通常の一般化ハフ変換では，投票先のブロックを計算し値を加算する。そのため，投票が行われないブロックについては計算は行われない。しかし，この方法ではすべてのブロックについて処理を行う。そのブロックに投票される票数をカウントする形で計算される (図 5.3)。計算量は多くなってしまうが並列化がしやすい。同一処理をすべてのブロックについて行えばよいためである [18]。

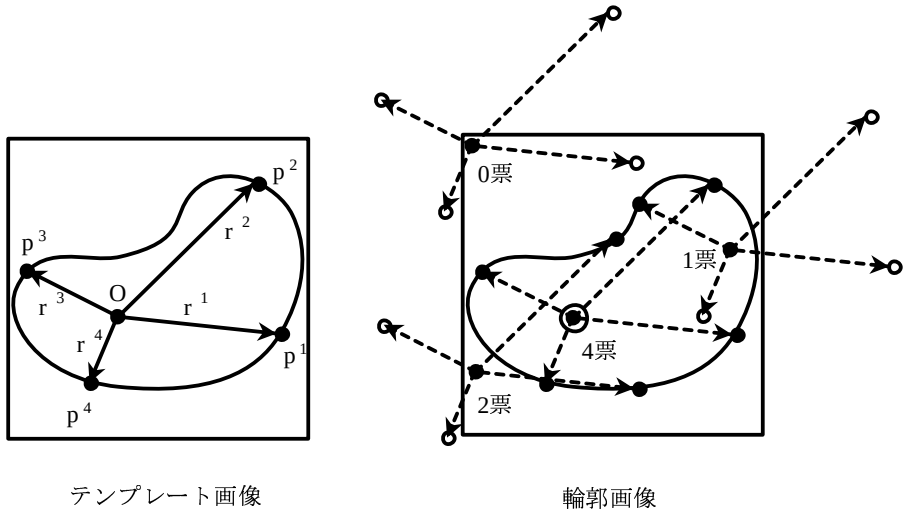


図 5.3: 出力先ピクセルからの逆参照による投票

5.3.2 アルゴリズム

テンプレート画像の輪郭をラプラシアンフィルタを用いて検出し，0 と 1 に二値化する。輪郭点の座標を配列 l に格納する。入力画像 I にも同様に輪郭検出をする。このとき，輪郭に隣接しているピクセルには 0 と 1 の間の値を入れるようにする。これは参照する座標が誤差によってずれた場合を許容するためである。そのため投票は整数ではなく浮動小数点数で行う。この画像を輪郭画像 I^{edge} とする。

認識処理は以下のように行う。

1. テンプレート画像の任意の位置に参照点 O を決める (式 5.6) 。
2. ある輪郭点 p^i を任意に選び，参照点 O から p^i までのベクトル r^i を求める。
3. 投票空間 C の任意の座標 (x, y) について，r^i を加算した座標での I^{edge} の値を調べる。
4. テンプレート画像の輪郭点の数を n としたとき，輪郭画像の値 $I^{edge}(x + r^i_x, y + r^i_y)$ を $|l|$ と n との大きい方で割る。この値を $v^i(x, y)$ とする (式 5.7) 。
5. $v^i(x, y)$ を投票空間の座標 $C(x, y)$ に加算する。
6. これを全ての p^i と全ての (x, y) との組み合わせについて繰り返す。
7. 最大の投票度数をテンプレート画像との適合率とする。
8. すべてのテンプレート画像について適合率を求め，最大の適合率を持つものを一致したテンプレートとする。

$$r^i := p^i - O \tag{5.6}$$

$$v^i(x, y) := \frac{I^{edge}(x + r^i_x, y + r^i_y)}{max(|l|, n)} \tag{5.7}$$

$$C(x, y) := \sum_{i=1}^{|l|} v(x, y) \tag{5.8}$$

5.3.3 GPGPU での実装

前処理として入力画像の輪郭検出を行う。あらかじめ画面全体と同じ大きさの正方形の物体を作っておく。入力画像をテクスチャとして張り付ける。これをレンダリングして，ピクセルシェーダを実行させる。ピクセルシェーダでラプラシアンフィルタをかけて輪郭を強調する。その後，ガウスフィルタで輪郭をぼかして隣接ピクセルに広げる。出力されたテクスチャ画像 I は輪郭の参照用に，GPU 内に置いたままにしておく。

また，輪郭画像の平均色を求めておく。平均の計算にも並列アルゴリズムを用いる。輪郭画像の幅と高さがそれぞれ半分の大きさの画像を用意する。面積は 1/4 に

なる。これをレンダリングターゲットに設定する。ピクセルシェーダでは，$[0, 1]$ 区間のテクスチャ座標 (x, y) について次のように 4 ピクセルの平均色を計算する。

$$\frac{I^{edge}(x, y) + I^{edge}(x + dx, y) + I^{edge}(x, y + dx) + I^{edge}(x + dx, y + dx)}{4}$$

ここで，w を輪郭画像の幅，h を輪郭画像の高さとしたとき，dx は $1/w$，dy は $1/h$ である。出力されるピクセルの色は，輪郭画像の 4 ピクセルのブロックの平均色となる。同様に，この幅と高さがそれぞれ半分の大きさの画像を用意し，4 ピクセルの平均を計算する。これを画像の大きさが 1×1 になるまで繰り返す (図 5.4)。最後の画像の 1 ピクセルの色は輪郭画像の平均色となる。この方法では輪郭画像の幅と

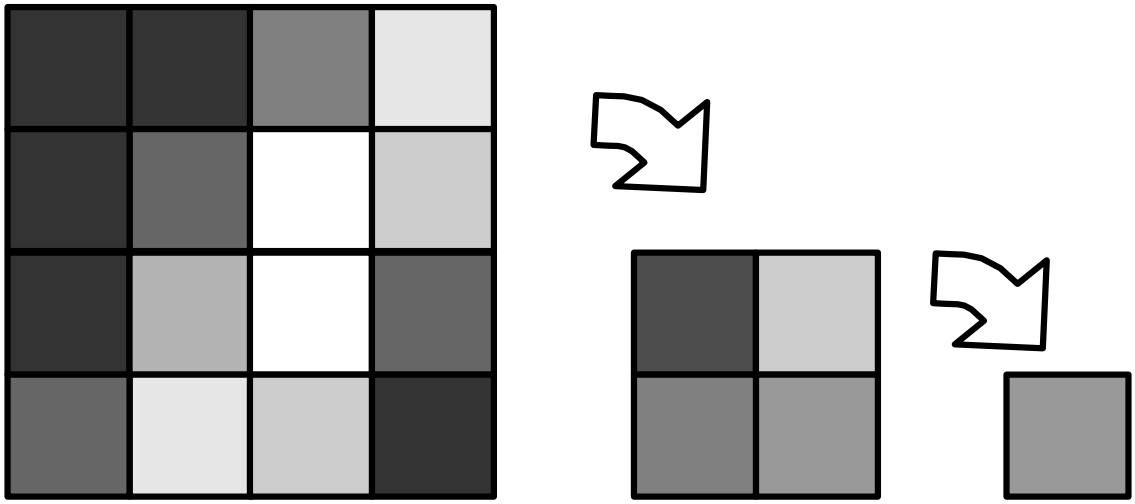


図 5.4: 平均化

高さが 2 の累乗でなければ使えない。それ以外の場合を考慮するには，出力用画像の大きさを ⌈ 輪郭画像の幅/2⌉ × ⌈ 輪郭画像の高さ/2⌉ にしておく。座標に元画像と出力画像との大きさの比率を掛けるようにすれば，どんな大きさの画像にも用いることができるようになる。

次に，テンプレート画像の参照ベクトルを求める。テンプレート画像に対しラプラシアンフィルタをかけて輪郭強調を行う。輪郭の値に対して，任意に設定した閾値を境に二値化する。輪郭として判断された全ての位置ベクトルを CPU 側で配列に格納する。

一般化ハフ変換の計算処理は次のように行われる。配列から位置ベクトルを取り出し，シェーダの定数にセットしておく。レンダリングターゲットに投票用画像を設定する。シェーダでは参照点を任意の場所に決めておき，位置ベクトルまでの逆参照ベクトルを求める。逆参照ベクトルに回転変換の行列をかける。現在のピクセルの座標に逆参照ベクトルを足した位置について，輪郭画像の値を参照する。入力画像の輪郭点の数を，入力画像の平均色 × 入力画像の面積 として計算する。これとテンプレート画像の輪郭点の数の大きい方をエッジ数として選ぶ。輪郭画像の値をエッジ数で割った数を出力ピクセルの色に足し合わせる。同様に，配列の全ての位置ベクトルについてこれらの処理を繰り返す。こうして投票画像に投票が行われる。

最後に出力された画像の最大の値を持つ色を調べる。これは平均色を求める処理と同様に行う。幅，高さがそれぞれ半分の大きさの画像を用意する。4 ピクセルのブロック内で最大の値を持つ色を求め，この色を出力する。画像の大きさが 1×1 になるまで処理を繰り返す。最後のピクセルの色が投票画像の最大の値を持つ色となる。この画像を CPU 側に送信し，色を読み取って適合率とする (図 5.5)。また，最大値を取ったピクセルの座標から画像の移動量を求められる。

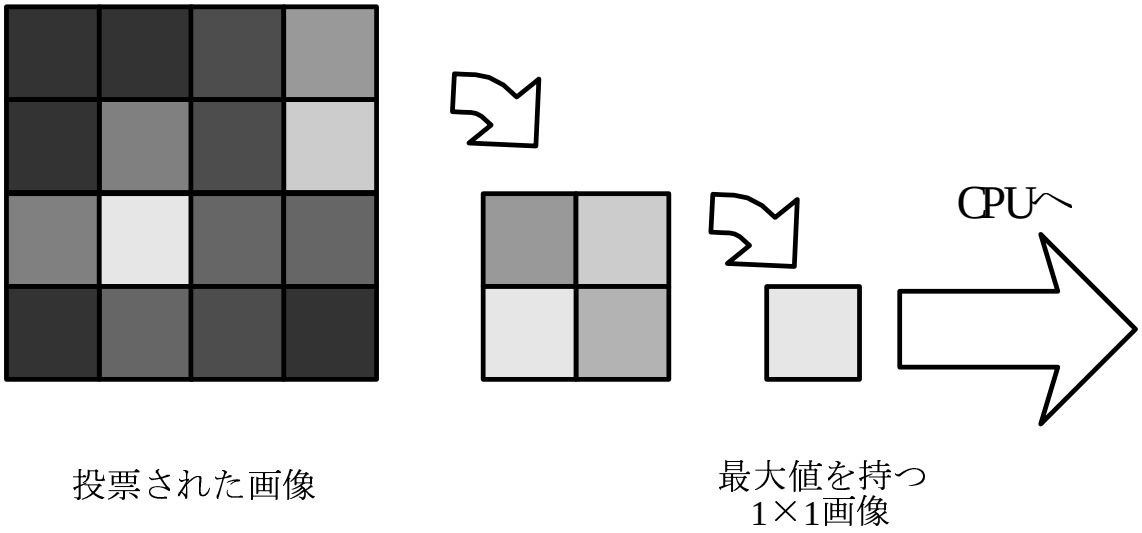


図 5.5: 最大値の取得

回転を含める場合，投票画像を複数枚用意しておく。これらの投票画像にそれぞれ別の回転角を設定して投票を行う。ピクセルごとに，同じ座標上で最大値を持つ投票画像を探し，色値を別の画像に記録する。記録された画像には全てのピクセルについて，最も投票の多かった回転角での投票度数が入っていることになる。このあと画像全体の最大値を求めれば回転を含んだ最大値を求めることができる。

5.4 方法 3

5.4.1 特徴

方法 1 を改良し，投票も GPU で行えるようにした。一般化ハフ変換での参照点の変換処理を，グラフィックスにおける頂点の位置の座標変換として捉えた。輪郭点の位置を頂点の座標に設定し，頂点シェーダで頂点座標を変換させる。変換した頂点座標に対応するスクリーン上のピクセルに色を加算していくことで，GPU 上での投票処理を可能とした。また，最大値を求めるのに方法 2 で用いた処理を用いた。これらによって，GPU 内で一般化ハフ変換の全ての処理を実行できるようになった。

5.4.2 GPGPU での実装

入力画像の輪郭データをテクスチャでなく頂点ストリームに格納する。輪郭を Sobel フィルタで検出し，輪郭の座標と濃淡勾配を調べる。頂点の位置に輪郭の座標を，頂点のテクスチャ座標に濃淡勾配をそれぞれ入れる。テンプレート画像は Sobel フィルタで輪郭の座標と濃淡勾配を出し，CPU の配列に入れておく。

輪郭データを入れた頂点ストリームをレンダリングする。このときテンプレート画像の輪郭の配列から 1 つの輪郭の座標と濃淡勾配とを取り出し，シェーダの定数に渡して GPU からアクセスできるようにする。また，頂点シェーダに一般化ハフ変換のシェーダプログラムを設定しておく。レンダリングターゲットには入力画像の幅と高さそれぞれを数倍にした大きさの画像を設定し，これを投票空間として扱う。通常のレンダリングでは頂点ストリームをポリゴンとして扱い頂点間で作られる面が塗りつぶされる。しかしこの場合は，座標変換された頂点の位置のみに投票を行いたい。そこで，頂点ストリームの頂点の位置に対応するピクセルだけに色が塗られるように，ポイントリストとしてレンダリングする。ポイントリストを用いれば，頂点を点として塗ることができ，点の大きさは設定することが出来る。投票位置のずれを防ぐために点の大きさは 2×2 のピクセルにまたがるようにした。

一般化ハフ変換のシェーダプログラムは次のように処理を行う。テンプレート画像の輪郭の座標から参照点までのベクトルを計算する。頂点データから位置と，テクスチャ座標に入れておいた濃淡勾配ベクトルとを取り出す。テンプレート画像と入力画像のそれぞれの輪郭の濃淡勾配から回転行列と回転角を求める。参照点までのベクトルに回転行列を掛けて輪郭の座標に加算し，投票先の位置ベクトルを求める。

ここで投票は x 軸，y 軸，回転角 θ の三次元空間に行わなければならない。しかし，結果はテクスチャに描画しなければならないため投票空間は二次元である。そこで，投票用テクスチャをブロックに分割し回転角によって投票先のブロックを決定するようにする。角度はブロックの行を左から右へ進むごとに一段階ずつ大きくなり，右端のブロックの次は下の行の左端のブロックへ進むようにする。配列の行優先順や，画像のラスタ走査と同じ要領である。角度が 0 度から 10 度までのブロック，10 度から 20 度までのブロックといったように分割し，角度があてはまったところを投票先のブロックとする。各ブロックは入力画像の大きさと同じにする。あらかじめ出力用画像は入力画像の数倍の大きさにしてあるため，幅の比率 × 高さの比率 の数のブロックを用意することができる。あてはまったブロックの左上端の座標に，投票位置ベクトルの xy 成分を加算した座標に投票する (図 5.6) 。また，出力先となるデータは色であるため四つの成分を持っている。この四成分に対しても回転角で振り分けて色を加算する。これらによって，角度成分の投票空間の段階数は，ブロックの行の数 × ブロックの列の数 × 色成分 4 つ となる。

投票先のブロックの決定は次のように行う。回転角 θ の値を $[-\pi, \pi)$ の範囲から $[0, 1)$ の範囲に変換する。この値をブロックの行の全体数に対する比率として考え，ブロックの行の数を掛けたものを I'_y とする。I'_y の整数部をとったものを I_y とし，これがブロックの行のインデックスとなる。I'_y の小数部はその行でのブロック数に対する比率と見ることができるので，ブロックの列の数を掛けたものを I'_x とする。I'_x の整数部を I_x とし，これはブロックの列のインデックスである。さらに，値を色成

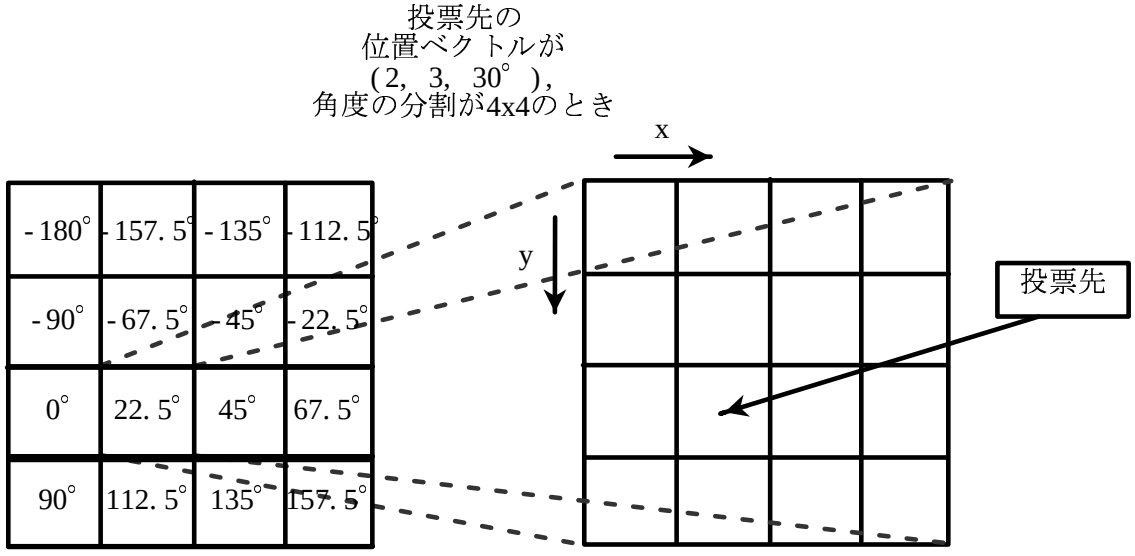


図 5.6: 投票用画像のブロック化

分に割り当てるため，I'_x の小数部を 4 で掛ける。この値の整数部を c とし，0 が赤，1 が緑，2 が青，3 がアルファとする色成分のインデックスとする。

$$f(t) := t - \lfloor t \rfloor (\forall t \geq 0) \tag{5.9}$$

$$I'_y := \left(\frac{\theta}{2\pi} + 0.5 \right) N_y \tag{5.10}$$

$$I'_x := f(I'_y) N_x \tag{5.11}$$

$$I_x := \lfloor I'_x \rfloor \tag{5.12}$$

$$I_y := \lfloor I'_y \rfloor \tag{5.13}$$

$$c := \lfloor 4 f(I'_x) \rfloor \tag{5.14}$$

$$V := 2\frac{I + O''}{N} - 1 \tag{5.15}$$

ここで N_y はブロックの行数，N_x はブロックの列数，f は浮動小数点の小数部を取る関数である。O'' は式 5.5 の O' の座標を $[0, 1]$ 範囲で表したもの，V が実際に投票する点の座標である。

求めた投票先の座標に色を塗ることで投票をする。一度のレンダリングで，テンプレートの一つの輪郭情報と，全ての入力画像の輪郭との組み合わせで投票が行われる。テンプレートの全ての輪郭情報について投票処理を行い，最終的な投票度数を計算する。ここで，投票画像上での同じピクセルに色が塗られることは，同じパラメタでの投票を加算することに相当する。ただし同じピクセルに色を塗る場合，通常の設定では色が上書きされてしまう。そのため，加算合成を用いて投票する色と投票画像の色を足し合わせるようにしておかなければならない。加算合成などのブレンディング処理を使えば，転送元のピクセルの色と転送先の色の値を任意の係数を掛けて足し合わせることが出来る。この係数を転送元と転送先との両方について 1 に設定し，投票の加算を行うようにする。また，投票は累積されていくため 8bit 整数では精度が足りない。投票用のテクスチャは浮動小数点フォーマットにしてお

く。浮動小数点テクスチャフォーマットで加算合成をしなければならないが，浮動小数点フォーマットでは加算合成を含むブレンディング処理に対応しているものが少ないため，ハードウェアが限られてしまうという欠点がある。

投票が終わると二次元の投票画像から最大値を探す。まず，各ピクセルの色の 4 成分から最大値を求め，出力する色の R 成分に設定する。G 成分，B 成分にはブロック内での xy 座標を入れ，A 成分には回転角を格納する。座標や回転角は，投票画像のテクスチャ座標から逆算する。テクスチャ座標を T，ブロックの列数と行数をそれぞれ N_x, N_y，c を最大値をもつ色成分を 0 から 3 まで整数値で表したインデックスとすると，座標 (x, y)，回転角 θ は式 5.17，5.18，5.19 のようになる。なお，色の範囲は $[0, 1]$ 区間であるため，θ はこの範囲での角度である。

$$
\begin{aligned}
f(t) &:= t - \lfloor t \rfloor (\forall t \geq 0) && (5.16)\\
x &:= f\left(T_x N_x\right) && (5.17)\\
y &:= f\left(T_y N_y\right) && (5.18)\\
\theta &:= \frac{c/4 + \lfloor T_y N_y \rfloor N_x + \lfloor T_x N_x \rfloor}{N_x N_y} && (5.19)
\end{aligned}
$$

この処理によって各ピクセルには，4 つの回転の中での投票の最大値と，その変換パラメタの値が格納されることになる。

投票画像全体の最大値の計算には，実装方法 2 で使った手順と同様に行う。周辺のピクセルの最大値を求めて，画像のサイズを小さくしていく。画像全体の最大値をテンプレート画像との適合率とし，最大値を持っていたピクセルから移動量と回転角を取り出す。

第6章　実験結果

6.1　実験方法

プログラムの実験は4種類について行った。CPUで方法1を実装したものと，方法1，方法2，方法3をそれぞれGPUで実装したものとを合わせた4種類を比較する。

それぞれのプログラムは次の条件下で実験した。

- 入力画像は32ptのフォントを一文字ごとに画像に書き出したものを使う。
- 最も投票の多かったテンプレートと入力文字が一致した場合にのみ正解とする。
- 文字はランダムに決めた角度で回転させ，回転を考慮した検出を行う。
- 文字の種別は英数字，カナ，漢字を用意する。
- テンプレートとなる文字の輪郭情報はあらかじめファイルとして作成したものを用いる。
- 入力画像とテンプレートについてはそれぞれ3種類のフォントのものを用意する。
- 処理時間は1文字ごとに3回実験したものの平均を取る。

ここで回転は角度10度ずつの検出を行うようにした。方法1，方法3では投票空間の角度成分を36段階にし，方法2では角度を10度ずつ変化させながら比較を行う。拡大縮小については含めない。

文字は種別ごとに分けて認識を行う。カナと英数字との組み合わせといった，別の種別との検出は行わない。英数字は大文字，小文字，記号を含めた94文字を対象とする。カナは全角ひらがな83文字，カタカナは全角カタカナ86文字を含めた169文字とする。漢字については文字数が多く，処理に時間がかかってしまうため，JIS X 0208規格の16区のものだけを対象とした94文字とする。

フォントはＭＳ Ｐゴシック，ＭＳ Ｐ明朝，HG正楷書体-PROを用いる。入力画像，テンプレートそれぞれついて3種類ずつ用意し，9通りの組み合わせで比較を行う。

プログラムの実行環境を表6.1に示す。

表 6.1: 実行環境

CPU	Pentium4 2.6GHz
GPU	GeForce 6800
メモリ	1GB
OS	WindowsXP Professional

6.2 結果

6.2.1 CPU 実装

表 6.2 は CPU で実装したプログラムの実験結果である。認識率は，フォントが同一の場合と異なる場合とのデータを集計した。処理時間については，1 文字にかかった処理時間と，1 つのテンプレートに対しての比較にかかった時間とを集計した。

表 6.2: CPU 実装の実験結果

	文字の種別	カナ	英数	漢字 16 区	平均
認識率 (%)	同一フォント	78.4	69.3	99.4	82.3
	異なるフォント	20.4	40.5	46.7	35.9
処理時間 (ms)	1 文字あたり	3186	1026	4455	2889
	1 テンプレートあたり	16.9	10.9	47.4	25.1

認識率について見てみると，同じフォントの組み合わせの場合は認識率が高い。フォントが同じ場合には平均約 82% の認識が行われている。フォントが異なる場合には，平均約 36% とかなり低い結果となってしまった。

また，文字の種別で比較をすると，漢字 16 区が最も高く平均 99.4%もの認識率となった。カナは平均 78.4%，英数字は平均 69.3%とやや低くなった。

一方，処理時間については，フォントの違いによって大差は出ていない。文字の種別ごとの処理時間に違いが出ている。平均時間を比較すると，英数字が 1026ms と最も速く，次いでカナが 3186ms，漢字 16 区が最も遅い 4455ms となっている。このデータは 1 文字当たりの処理時間であり，カナの場合はテンプレート数が多くなっている。1 テンプレートあたりの平均処理時間を比較すると，英数字は 10.9ms，カナは 16.9ms，漢字 16 区は 47.4ms となる。英数字とカナの処理時間にそれほど大きな差は開いていないが，漢字 16 区は他に比べて非常に遅いことが分かる。

これらのデータから処理時間の長いものほど認識率が高くなっていることが分かる。これはエッジ数の違いによるものだと考えられる。しかし，最も速い英数字でも約 1 秒かかり，漢字については 1 つの区のテンプレートとの比較だけで約 4.5 秒もかかってしまう。一般化ハフ変換の処理速度の遅さを確認する結果となった。

6.2.2 GPU 実装 1

方法 1 を GPU で実装したプログラムの結果を表 6.3 にまとめる。同一フォントでの認識率は，平均が 73.6%，英数字については 59.9%と低くなってしまった。フォントが異なる場合は，30.3%とやはり低い結果となっている。処理時間についても平均 2211ms とあまり高速化がされていない。漢字 16 区は 2180ms と CPU より速い結果となったが，英数字は 1503ms と CPU より遅くなってしまった。これはエッジ数の多い漢字については並列化の効果が大きいためだと思われる。逆に，エッジ数の少ない英数字では，GPU を使ったことによるオーバーヘッドの方が，並列化の効果より大きくなってしまったと推測することが出来る。

表 6.3: GPU 実装 1 の実験結果

	文字の種別	カナ	英数	漢字 16 区	平均
認識率 (%)	同一フォント	62.3	59.9	98.6	73.6
	異なるフォント	15.7	33.2	42.0	30.3
処理時間 (ms)	1 文字あたり	2951	1503	2180	2211
	1 テンプレートあたり	15.6	16.0	23.2	18.26

6.2.3 GPU 実装 2

方法 2 を GPU で実装したものは予想以上に遅くなってしまった。表 6.4 が結果データを集計したものである。一文字あたりに平均 15 秒もかかっており，CPU の 4，5 倍の処理時間を費やしたことになる。回転に対応していないために，回転角を 36 回変化させて変換処理を行ったことが処理時間の大幅な増加を引き起こしたと思われる。回転を考慮せず，位置だけを検出する場合には CPU の 7，8 倍の速度がでるのではないだろうか。

認識の精度を CPU 実装のものと比較すると，カナ，英数字がやや精度が高く，漢字は若干低いという結果になった。カナ，英数字については輪郭をぼかして広げたことにより，投票のずれが抑えられたのであると思われる。逆に漢字に対しては，それが誤った候補にも票を集中させる要因になったものと思われる。

表 6.4: GPU 実装 2 の実験結果

	文字の種別	カナ	英数	漢字 16 区	平均
認識率 (%)	同一フォント	80.4	83.2	96.0	86.5
	異なるフォント	25.9	45.3	24.1	31.8
処理時間 (ms)	1 文字あたり	19319	7333	18704	15118
	1 テンプレートあたり	102.2	78.0	199.0	126.40

6.2.4 GPU 実装 3

GPU 実装 3 では他の実装に比べてかなり処理速度が速くなった。処理時間の平均は 279.6ms と，CPU 実装の約 10 倍の性能となった (表 6.5) 。頂点の座標変換を利用した投票や，最大値の集計処理に，GPU の並列計算能力を発揮することができたのだと思われる。また，結果は最大値のみとなり，CPU へ返すデータの転送量を削減できた。これによって，GPU を使うことによるオーバーヘッドを抑えることができたのだと考えられる。

認識率も CPU より若干良い結果を得ることが出来た。投票を行う際に，ポイントリストの点の大きさを 2×2 に設定したことで，投票のずれを抑えることができたと思われる。

また，処理時間が長いものほど認識率は高くなるという，CPU の実験結果と同じ傾向が見られた。

表 6.5: GPU 実装 3 の実験結果

	文字の種別	カナ	英数	漢字 16 区	平均
認識率 (%)	同一フォント	77.4	74.2	99.6	83.8
	異なるフォント	23.3	41.6	50.0	38.3
処理時間 (ms)	1 文字あたり	324.7	126.4	387.9	279.6
	1 テンプレートあたり	1.72	1.34	4.13	2.40

6.2.5 総合比較

それぞれの実装の認識率と処理速度を比較する。処理速度は処理時間の逆数を取り，1 秒当たりに処理できる文字数またはテンプレートの数とする。表 6.6 はそれぞれの実装での平均認識率と，平均処理速度を比較したものである。また，図 6.1 は処理速度を比較した棒グラフである。

表 6.6: 総合データ比較

	文字の種別	CPU	GPU1	GPU2	GPU3
認識率 (%)	同一フォント	82.3	73.6	86.5	83.8
	異なるフォント	35.9	30.3	31.8	38.3
処理速度 (ms)	文字数	0.346	0.452	0.066	3.576
	テンプレート数	39.9	54.8	7.9	417.3

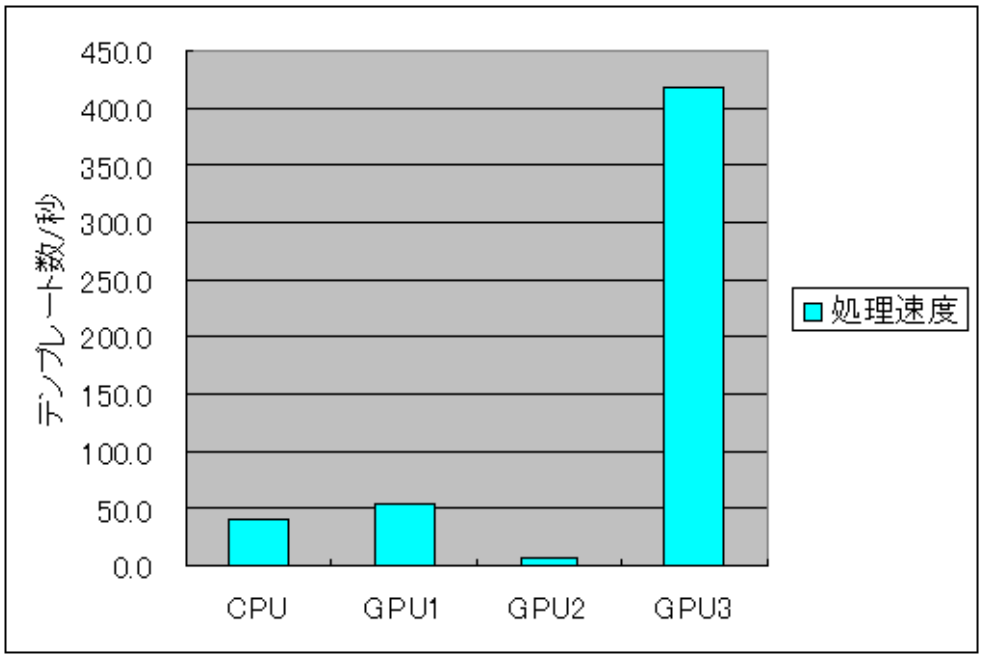


図 6.1: 処理速度の比較

認識率はGPU実装1がやや劣っているが，他の実装は85%前後であまり差がない。

一方，処理速度ではGPU実装3が圧倒的に速い結果となっている。1秒間あたりに処理できる文字数，テンプレート数ともにCPUの10倍強の速度であり，GPU実装1の約8倍の速度となった。

6.2.6 考察

最も速かったGPU実装3とCPUとを比較すると，10倍以上の速度性能が得られた。今回実験に使用したCPUは2.6GHzのPentium4のものであった。しかし，現時点で最新のPentium4のCPUでも3.8GHzが限度である。最新のものと比較したとしても，約7倍の速度が期待される。

認識率についてはGPU実装3は平均83%と，まずまずの結果となった。結果データを調べたところいくつか気づいた点があった。

まず，回転を含めたアルゴリズムであったため，文字ごとに全ての回転について調べていた。しかし，回転を含めたことによる誤認識が発生してしまった。例えば，”q”を”b”と認識したり，”く”を”へ”と認識したりするなどである。なお，今回は最も投票の多かったものを候補とする，1位認識率でデータを集計していた。1位だけではなく5位以内を正解とすれば少し認識率が高くなると思われる。

また，漢字などエッジの多いものが認識率が高いことがデータに表れていた。エッジが多いほど投票が分散しやすくなると考えられ，誤認識が減ったために認識率が向上したと思われる。文字が同じでもフォントが異なると認識率が格段に落ちるという結果も出ていた。形が微妙に異なるだけでも票が分散してしまうようである。エッジ数の少ないものや，エッジ情報がやや異なるものについても，認識率を向上

させる工夫が必要である。

ただし，高速化をすることができたものの，まだ 1 秒間に 3 文字程度の検出しか行うことが出来ない。認識率についてもそれほど高くはなく，処理時間に見合わないものとなっている。実用的な利用にはさらに速度を向上させ，同時に認識率を増加させる必要がある。

第7章　今後の課題

本研究の目的である文字認識の高速化は行うことが出来た。今回作成したシステムでは，1 文字だけを含む画像から認識を行うというものだった。今後は実用に向けた改良を行う必要がある。

実用化の 1 つの方法として OCR との統合がある。フォントが同じ場合の認識率は高かったため，印字された文字の認識に効果的であると思われる。そのために必要な処理を挙げる。

- ページ単位，行単位での回転検出
- 拡大縮小や他の変換を含めたアルゴリズムへの拡張
- さらなる高速化

回転を 1 文字ごとに検出したため，認識率を下げる要因となっていた。しかし，1 ページ中や 1 行中で文字の回転角が大きく変化することはないと思われる。そこで，角度の大きい回転については別の方法で前処理を行いページ単位や行単位で検出しておく。文字単位ではゆがみなどによる小さい回転のみを考慮し，範囲を狭めて検出を行うようにする。これにより認識率の向上と処理時間の短縮が望める。

また，拡大縮小を考慮したアルゴリズムになっていなかった。Chord-Tangent 法などの手法を取り入れて拡大縮小を含んだ計算を行えるようにすることが挙げられる。射影などの他の変換も含めることも考えられる。グラフィックスの分野では座標変換にアフィン変換を用いている。アフィン変換は平行移動と一次変換の合成であり，一次変換であれば高速計算が可能である。この変換についてもページや行，ブロック化した範囲などの大きい単位で求めておき，文字単位では小さい範囲での変換のみ行うことが効果的であると思われる。

高速化としては，根本的なアルゴリズムの高速化やプログラムの最適化がある。アルゴリズムの改良としては FGHT などを参考にして認識処理そのものを高速にすることがある。また OCR の場合，文章の流れから文字の種類や単語内の次の文字などが推測されるため，ある程度候補を特定することが出来る。テンプレートの探索を枝切りすることで大幅に高速化することができると思われる。他の高速な文字認識アルゴリズムで先に認識を行い，認識が難しかった文字についてのみ処理を行うという方法もある。

画像認識に使うことも考えられる。今回作成した GPGPU のプログラムのアルゴリズムは，画像認識に対してもほぼ同じものが使えるはずである。投票空間となる 2 次元テクスチャを擬似的に 3 次元として扱ったため，画像サイズには制限がある。GPU で扱うことのできる最大テクスチャサイズは 4096 であるので，回転角を 36 段

階で検出する場合は 1024×1024 までとなる。これはあまり厳しい制限ではないと思われる。ただし，拡大縮小などの他の変換を含めた拡張を行う場合には，パラメタ数が増大し投票の次元が増える。画像認識に利用するにはこの問題を解決しなくてはならない。

このように，実用化に向けては多くの課題があるが，一般化ハフ変換は画像認識において適用範囲が広いと思われる。GPU を利用したことによって，問題となっていた処理速度については高速化をすることができた。また，消費メモリについてはGPU のメモリリソースを用いることで分散させることができる。そのため，GPGPU を利用することによる効果は大きいと思われる。

GPGPU は並列計算やリソースの再利用という点で非常に有効であると感じた。現時点ではまだ制約が多く，実装は難しいものとなっているが，命令数などの制限は徐々に緩和されている。今後，これらの制約が少なくなっていけば，GPGPU の技術は広まることが期待される。

関連図書

[1] D. H. Ballard, ”Generalizing the Hough transform to detect arbitrary shapes, ” Pattern Recognit., vol.13, no.2, pp.111-1222, 1981.

[2] GPGPU.org, http://www.gpgpu.org/

[3] GP2, http://www.cs.unc.edu/Events/Conferences/GP2/

[4] 森俊二， 板倉栂子,「画像認識の基礎 2」オーム社, 1990.

[5] R. O. Duda and P. E. Hart : ”Use of the Hough transformation to detect lines and curves in pictures, ” Common. A. C. M., Vol. 15, pp. 11-15, Jan. 1972.

[6] 松山隆司，和田俊和， ”Hough 変換：投票と多数決に基づく幾何学的対象の検出と識別，” コンピュータビジョン：技術評論と将来展望，松山隆司，久野義徳，井上淳 (編)，1998， pp. 149-165.

[7] 木村彰男，渡辺孝志，”高速一般化ハフ変換 —相似変換不変な任意図形検出法，” 電子情報通信学会論文誌 (D-II)， vol.J81-D-II，no.4， pp.726-734， April 1998.

[8] T.E. Dufresne and A.P. Dhawan, ”Chord tangent transformation for the polar and pole definition of conics,” Pattern Recognit., vol.28, no.9, pp.1321-1331, 1995.

[9] 木村彰男，渡辺孝志，”図形検出力を向上させた高速一般化ハフ変換，” 電子情報通信学会論文誌 (D-II)， vol.J83-D-II，no.5， pp.1256-1265， May 2000.

[10] Microsoft Corporation: DirectX 9.0 Programmer’s Reference.

[11] NVIDIA Corporation: Cg Toolkit User’s Manual Release 1.2 (2004).

[12] Intel, http://www.intel.co.jp/

[13] NVIDIA, http://jp.nvidia.com/

[14] Eurographics 2004 web site, http://eg04.inrialpes.fr/index.en.html.

[15] OpenGL -The Industry’s Foundation for High Performance Graphics-, http://www.opengl.org/

[16] Sh, http://libsh.sourceforge.net/

[17] BrookGPU, http://graphics.stanford.edu/projects/brookgpu/

[18] Robert Strzodka, "A Graphics Hardware Implementation of the Generalized Hough Transform for fast Object Recognition, Scale, and 3D Pose Detection," International Conference on Image Analysis and Processing (ICIAP 2003), pp 188-193, 2003.